

Cyrille Rossant

IPython Interactive Computing and Visualization Cookbook

Second Edition

Over 100 hands-on recipes to sharpen your skills in high-performance numerical computing and data science in the Jupyter Notebook



Packt

IPython Interactive Computing and Visualization Cookbook

Second Edition

Over 100 hands-on recipes to sharpen your skills in
high-performance numerical computing and data science
in the Jupyter Notebook

Cyrille Rossant

Packt

BIRMINGHAM - MUMBAI

IPython Interactive Computing and Visualization Cookbook

Second Edition

Copyright © 2018 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 or its dealers and distributors, will be held liable for any damages caused or alleged to have been 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.

Commissioning Editor: Veena Pagare
Acquisition Editor: Dominic Shakeshaft
Project Editor: Suzanne Coutinho
Technical Editors: Bhagyashree Rai, Nidhisha Shetty
Proofreader: Safis Editing
Indexer: Aishwarya Gangawane
Graphics: Tom Scaria
Production Coordinator: Shantanu Zagade

First published: September 2014

Second Edition: January 2018

Production reference: 1290118

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

ISBN 978-1-78588-863-2

WWW.PACKTPUB.COM



mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- ▶ Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- ▶ Learn better with Skill Plans built especially for you
- ▶ Get a free eBook or video every month
- ▶ Mapt is fully searchable
- ▶ Copy and paste, print, and bookmark content

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.

Contributors

About the author

Cyrille Rossant, PhD, is a neuroscience researcher and software engineer at University College London. He is a graduate of École Normale Supérieure, Paris, where he studied mathematics and computer science. He has also worked at Princeton University and Collège de France. While working on data science and software engineering projects, he has gained experience in numerical computing, parallel computing, and high-performance data visualization.

He is the author of *Learning IPython for Interactive Computing and Data Visualization, Second Edition*, Packt Publishing, the prequel of this cookbook.

I'm grateful to everyone who gave their feedback on this book, including Matthias Bussonnier, Thomas Caswell, Guillaume Gay, Brian Granger, Matthew Rocklin, Steven Silvester, and Jake VanderPlas. I'd also like to thank my family for their support.

Packt is Searching for Authors Like You

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

| | |
|---|-----|
| Preface | vii |
| Chapter 1: A Tour of Interactive Computing with Jupyter and IPython | 1 |
| Introduction | 1 |
| Introducing IPython and the Jupyter Notebook | 6 |
| Getting started with exploratory data analysis in the Jupyter Notebook | 13 |
| Introducing the multidimensional array in NumPy for fast array computations | 19 |
| Creating an IPython extension with custom magic commands | 24 |
| Mastering IPython's configuration system | 29 |
| Creating a simple kernel for Jupyter | 33 |
| Chapter 2: Best Practices in Interactive Computing | 41 |
| Introduction | 41 |
| Learning the basics of the Unix shell | 42 |
| Using the latest features of Python 3 | 46 |
| Learning the basics of the distributed version control system Git | 52 |
| A typical workflow with Git branching | 58 |
| Efficient interactive computing workflows with IPython | 63 |
| Ten tips for conducting reproducible interactive computing experiments | 66 |
| Writing high-quality Python code | 70 |
| Writing unit tests with pytest | 73 |
| Debugging code with IPython | 79 |
| Chapter 3: Mastering the Jupyter Notebook | 83 |
| Introduction | 83 |
| Teaching programming in the Notebook with IPython Blocks | 86 |
| Converting a Jupyter notebook to other formats with nbconvert | 91 |
| Mastering widgets in the Jupyter Notebook | 97 |
| Creating custom Jupyter Notebook widgets in Python, HTML, and JavaScript | 104 |
| Configuring the Jupyter Notebook | 107 |
| Introducing JupyterLab | 111 |

| | |
|---|------------|
| Chapter 4: Profiling and Optimization | 125 |
| Introduction | 125 |
| Evaluating the time taken by a command in IPython | 126 |
| Profiling your code easily with cProfile and IPython | 127 |
| Profiling your code line-by-line with line_profiler | 131 |
| Profiling the memory usage of your code with memory_profiler | 134 |
| Understanding the internals of NumPy to avoid unnecessary array copying | 135 |
| Using stride tricks with NumPy | 142 |
| Implementing an efficient rolling average algorithm with stride tricks | 145 |
| Processing large NumPy arrays with memory mapping | 148 |
| Manipulating large arrays with HDF5 | 150 |
| Chapter 5: High-Performance Computing | 153 |
| Introduction | 153 |
| Using Python to write faster code | 156 |
| Accelerating pure Python code with Numba and Just-In-Time compilation | 161 |
| Accelerating array computations with NumExpr | 165 |
| Wrapping a C library in Python with ctypes | 167 |
| Accelerating Python code with Cython | 171 |
| Optimizing Cython code by writing less Python and more C | 175 |
| Releasing the GIL to take advantage of multi-core processors with Cython and OpenMP | 182 |
| Writing massively parallel code for NVIDIA graphics cards (GPUs) with CUDA | 184 |
| Distributing Python code across multiple cores with IPython | 190 |
| Interacting with asynchronous parallel tasks in IPython | 194 |
| Performing out-of-core computations on large arrays with Dask | 197 |
| Trying the Julia programming language in the Jupyter Notebook | 202 |
| Chapter 6: Data Visualization | 209 |
| Introduction | 209 |
| Using Matplotlib styles | 209 |
| Creating statistical plots easily with seaborn | 214 |
| Creating interactive web visualizations with Bokeh and HoloViews | 218 |
| Visualizing a NetworkX graph in the Notebook with D3.js | 224 |
| Discovering interactive visualization libraries in the Notebook | 229 |
| Creating plots with Altair and the Vega-Lite specification | 234 |
| Chapter 7: Statistical Data Analysis | 241 |
| Introduction | 241 |
| Exploring a dataset with pandas and Matplotlib | 245 |
| Getting started with statistical hypothesis testing – a simple z-test | 249 |

Table of Contents

| | |
|--|------------|
| Getting started with Bayesian methods | 253 |
| Estimating the correlation between two variables with a contingency table and a chi-squared test | 258 |
| Fitting a probability distribution to data with the maximum likelihood method | 262 |
| Estimating a probability distribution nonparametrically with a kernel density estimation | 268 |
| Fitting a Bayesian model by sampling from a posterior distribution with a Markov chain Monte Carlo method | 273 |
| Analyzing data with the R programming language in the Jupyter Notebook | 278 |
| Chapter 8: Machine Learning | 285 |
| Introduction | 285 |
| Getting started with scikit-learn | 291 |
| Predicting who will survive on the Titanic with logistic regression | 299 |
| Learning to recognize handwritten digits with a K-nearest neighbors classifier | 305 |
| Learning from text – Naive Bayes for Natural Language Processing | 309 |
| Using support vector machines for classification tasks | 313 |
| Using a random forest to select important features for regression | 319 |
| Reducing the dimensionality of a dataset with a principal component analysis | 324 |
| Detecting hidden structures in a dataset with clustering | 328 |
| Chapter 9: Numerical Optimization | 335 |
| Introduction | 335 |
| Finding the root of a mathematical function | 338 |
| Minimizing a mathematical function | 341 |
| Fitting a function to data with nonlinear least squares | 349 |
| Finding the equilibrium state of a physical system by minimizing its potential energy | 352 |
| Chapter 10: Signal Processing | 359 |
| Introduction | 359 |
| Analyzing the frequency components of a signal with a Fast Fourier Transform | 363 |
| Applying a linear filter to a digital signal | 370 |
| Computing the autocorrelation of a time series | 376 |

| | |
|---|------------|
| Chapter 11: Image and Audio Processing | 381 |
| Introduction | 381 |
| Manipulating the exposure of an image | 383 |
| Applying filters on an image | 386 |
| Segmenting an image | 391 |
| Finding points of interest in an image | 397 |
| Detecting faces in an image with OpenCV | 401 |
| Applying digital filters to speech sounds | 404 |
| Creating a sound synthesizer in the Notebook | 408 |
| Chapter 12: Deterministic Dynamical Systems | 411 |
| Introduction | 411 |
| Plotting the bifurcation diagram of a chaotic dynamical system | 413 |
| Simulating an elementary cellular automaton | 419 |
| Simulating an ordinary differential equation with SciPy | 422 |
| Simulating a partial differential equation — reaction-diffusion systems and Turing patterns | 427 |
| Chapter 13: Stochastic Dynamical Systems | 433 |
| Introduction | 433 |
| Simulating a discrete-time Markov chain | 434 |
| Simulating a Poisson process | 438 |
| Simulating a Brownian motion | 442 |
| Simulating a stochastic differential equation | 444 |
| Chapter 14: Graphs, Geometry, and Geographic Information Systems | 449 |
| Introduction | 449 |
| Manipulating and visualizing graphs with NetworkX | 453 |
| Drawing flight routes with NetworkX | 457 |
| Resolving dependencies in a directed acyclic graph with a topological sort | 463 |
| Computing connected components in an image | 467 |
| Computing the Voronoi diagram of a set of points | 471 |
| Manipulating geospatial data with Cartopy | 477 |
| Creating a route planner for a road network | 481 |
| Chapter 15: Symbolic and Numerical Mathematics | 487 |
| Introduction | 487 |
| Diving into symbolic computing with SymPy | 488 |
| Solving equations and inequalities | 491 |
| Analyzing real-valued functions | 493 |
| Computing exact probabilities and manipulating random variables | 495 |
| A bit of number theory with SymPy | 498 |

Table of Contents

| | |
|---|------------|
| Finding a Boolean propositional formula from a truth table | 501 |
| Analyzing a nonlinear differential system — Lotka-Volterra (predator-prey) equations | 504 |
| Getting started with Sage | 507 |
| <u>Another Book You May Enjoy</u> | 511 |
| Index | 513 |

Preface

We are becoming awash in the flood of digital data from scientific research, engineering, economics, politics, journalism, business, and many other domains. As a result, analyzing, visualizing, and harnessing data is the occupation of an increasingly large and diverse set of people. Quantitative skills such as programming, numerical computing, mathematics, statistics, and data mining, which form the core of data science, are more and more appreciated in a seemingly endless plethora of fields.

Python, a widely-known programming language, is also one of the leading open platforms for data science. IPython is a mature Python project that provides scientist-friendly interactive access to Python. It is part of the broader Project Jupyter, which aims to provide high-quality environments for interactive computing, data analysis, visualization, and the authoring of interactive scientific documents. Jupyter is estimated to have several million users today.

The prequel of this book, *Learning IPython for Interactive Computing and Data Visualization Second Edition*, Packt Publishing was published in 2015, two years after the first edition. It is a beginner-level introduction to data science and numerical computing with Python, IPython, and Jupyter.

This book, the first edition of which was published in 2014, continues that journey by presenting more than 100 recipes for interactive scientific computing and data science. These recipes not only cover programming topics such as numerical computing, high-performance computing, parallel computing, and interactive visualization, but also data analysis topics such as statistics, data mining, machine learning, signal processing, graph theory, numerical optimization, and many others.

This second edition is fully compatible with the latest versions of the platform and its libraries. It includes new recipes to better leverage the latest features of Python 3, and it introduces promising new projects such as JupyterLab, Altair, and Dask.

By design, this book privileges breadth over depth. A particularly wide range of libraries and techniques are covered in this book, but not comprehensively. We give many references that let you deepen your knowledge of individual methods. The goal of this book is not to make you an expert of the subjects covered, but to give you a glimpse of the extremely diverse set of applications that you can tackle with the platform.



All the recipes in this book, which cover a specific techniques, are available online as a Jupyter notebook. This interactive document lets you read, execute, and modify the code interactively, which makes the learning process more engaging and dynamic.

Almost all of this book's content is available online on the GitHub platform (<http://ipython-books.github.io/>). Updates and corrections will be regularly published there, so you should make sure you check out the latest version of the book online.

Who this book is for

This book targets researchers, engineers, data scientists, teachers, students, analysts, journalists, economists, and hobbyists interested in data analysis and numerical computing.

Readers familiar with the scientific Python ecosystem will find many resources to sharpen their skills in high-performance interactive computing with IPython and Jupyter.

Readers who need to implement algorithms for domain-specific applications will appreciate the introductions to a wide variety of topics in data analysis and applied mathematics.

Readers who are new to numerical computing with Python should start with the prequel of this book, *Learning IPython for Interactive Computing and Data Visualization Second Edition*, Packt Publishing published in 2015.

What this book covers

This book is split into two parts:

Part 1 (chapters 1 to 6) covers relatively advanced methods in interactive numerical computing, high-performance computing, and data visualization.

Part 2 (chapters 7 to 15) introduces standard methods in data science and mathematical modeling. Many of these methods are applied to real-world data.

Part 1 – Interactive Computing with Jupyter

Chapter 1, A Tour of Interactive Computing with Jupyter and IPython, contains a brief introduction to data analysis and numerical computing with IPython and Jupyter. It not only covers common packages such as Python, NumPy, pandas, and Matplotlib, but also advanced IPython/Jupyter topics such as interactive widgets in the Notebook, custom magic commands, configurable IPython extensions, and custom Jupyter kernels.

Chapter 2, Best Practices in Interactive Computing, details best practices to write reproducible, high-quality code: task automation, version control with Git, workflows with IPython and Jupyter, unit testing, continuous integration, debugging, and other related topics. The importance of these subjects in computational research and data analysis cannot be overstated.

Chapter 3, Mastering the Jupyter Notebook, covers topics related to the Jupyter Notebook, notably the Notebook format, notebook conversions, and interactive widgets.

Chapter 4, Profiling and Optimization, covers methods to make your code faster and more efficient: CPU and memory profiling in Python, advanced optimization techniques with NumPy (including large array manipulations), and memory mapping of huge arrays. These techniques are essential for big data analysis.

Chapter 5, High-Performance Computing, covers techniques to make your code much faster: code acceleration with Numba and Cython, wrapping C libraries in Python with ctypes, parallel computing with IPython and Dask, OpenMP, and **General-Purpose Computing on Graphics Processing Units (GPGPU)** with CUDA. The chapter ends with an introduction to the Julia language, a high-performance numerical computing programming language that can be used in the Jupyter Notebook.

Chapter 6, Data Visualization, introduces several visualization or interactive visualization libraries, such as `matplotlib`, `seaborn`, `bokeh`, `D3`, `Altair`, and others.

Part 2 – Standard Methods in Data Science and Applied Mathematics

Chapter 7, Statistical Data Analysis, covers methods for getting insights into data. It introduces classic frequentist and Bayesian methods for hypothesis testing, parametric and nonparametric estimation, and model inference. The chapter leverages Python libraries such as pandas, SciPy, statsmodels, and PyMC. The last recipe introduces the statistical language R, which can be easily used in the Jupyter Notebook.

Chapter 8, Machine Learning, covers methods to learn and make predictions from data. Using the scikit-learn Python package, this chapter illustrates fundamental data mining and machine learning concepts such as supervised and unsupervised learning, classification, regression, feature selection, feature extraction, overfitting, regularization, cross-validation, and grid search. Algorithms addressed in this chapter include logistic regression, Naive Bayes, K-nearest neighbors, support vector machines, random forests, and others. These methods are applied to various types of datasets: numerical data, images, and text.

Chapter 9, Numerical Optimization, covers minimizing and maximizing mathematical functions. This topic is pervasive in data science, notably in statistics, machine learning, and signal processing. This chapter illustrates a few root-finding, minimization, and curve-fitting routines with SciPy.

Chapter 10, Signal Processing, covers extracting relevant information from complex and noisy data. These steps are sometimes required prior to running statistical and data mining algorithms. This chapter introduces basic signal processing methods such as Fourier transforms and digital filters.

Chapter 11, Image and Audio Processing, covers signal processing methods for images and sounds. It introduces image filtering, segmentation, computer vision, and face detection with scikit-image and OpenCV. It also presents methods for audio processing and synthesis.

Chapter 12, Deterministic Dynamical Systems, describes the dynamical processes underlying particular types of data. It illustrates simulation techniques for discrete-time dynamical systems, as well as for ordinary differential equations and partial differential equations.

Chapter 13, Stochastic Dynamical Systems, describes the dynamical random processes underlying particular types of data. It illustrates simulation techniques for discrete-time Markov chains, point processes, and stochastic differential equations.

Chapter 14, Graphs, Geometry, and Geographic Information Systems, covers analysis and visualization methods for graphs, flight networks, road networks, maps, and geographic data.

Chapter 15, Symbolic and Numerical Mathematics, introduces SymPy, a computer algebra system that brings symbolic computing to Python. The chapter ends with an introduction to Sage, another Python-based system for computational mathematics.

To get the most out of this book

This book is accessible to beginners. However, it may be easier for you if you are familiar with the contents of *Learning IPython for Interactive Computing and Data Visualization, Second Edition*, Packt Publishing (also called the "IPython minibook"), the prequel of this book. The minibook introduces Python programming, the IPython console, the Jupyter Notebook, numerical computing with NumPy, basic data analysis with pandas, and plotting with Matplotlib. This book tackles scientific programming topics that rely on all of these tools.

Part 2 is a bit more theoretical. It is easier to read if you know the basics of calculus, linear algebra, and probability theory (real-valued functions, integrals and derivatives, differential equations, matrices, vector spaces, probabilities, random variables, and so on). These chapters introduce different topics in data science and applied mathematics, and how to apply them with Python: statistics, machine learning, numerical optimization, signal processing, dynamical systems, graph theory, and others.



Installing Python

This book uses the free Anaconda distribution (<https://www.anaconda.com/download/>). It includes Python 3, IPython, Jupyter, and almost all of the packages that we will be using in this book. Anaconda also includes a powerful packaging system named Conda. The introduction of this book's first chapter gives you more details.

The code of this book has been written for Python 3 and is incompatible with older versions of Python, Python 2 (although minimal to no changes would be required to make it compatible).

GitHub repositories

This book has a website: <http://ipython-books.github.io>. The text, the code, and the data from the book are available on several GitHub repositories at <https://github.com/ipython-books/>. You can also run the code interactively in your web browser without installing anything on your computer, thanks to the Binder project.

Be sure to check out <http://ipython-books.github.io> and the repositories to get the latest updates and corrections. You can also propose your own corrections and suggestions on GitHub by opening issues or pull requests.

You can also follow the author online (<http://cyrille.rossant.net>) and on Twitter (@cyrillerossant).

Download the example code files

You can download the example code files for this book from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at <http://www.packtpub.com>.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the on-screen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- ▶ WinRAR / 7-Zip for Windows
- ▶ Zipeg / iZip / UnRarX for Mac
- ▶ 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/IPython-Interactive-Computing-and-Visualization-Cookbook-Second-Edition>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://www.packtpub.com/sites/default/files/downloads/IPythonInteractiveComputingandVisualizationCookbookSecondEdition_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: « The new `np.block()` function lets one define block matrices. »

A block of code is set as follows:

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

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

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

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

```
# cp /usr/src/asterisk-addons/configs/cdr_mysql.conf.sample
/etc/asterisk/cdr_mysql.conf
```

Bold: Indicates a new term, an important word, or words that you see on the screen, for example, in menus or dialog boxes, also appear in the text like this. Here is an example: "Select **System info** from the **Administration** panel."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Sections

In this book, you will find several headings that appear frequently (*Getting ready*, *How to do it...*, *How it works...*, *There's more...*, and *See also*).

To give clear instructions on how to complete a recipe, use these sections as follows:

Getting ready

This section tells you what to expect in the recipe and describes how to set up any software or any preliminary settings required for the recipe.

How to do it...

This section contains the steps required to follow the recipe.

How it works...

This section usually consists of a detailed explanation of what happened in the previous section.

There's more...

This section consists of additional information about the recipe in order to make you more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email feedback@packtpub.com and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at questions@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book we would be grateful if you would report this to us. Please visit, <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packtpub.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit <http://authors.packtpub.com>.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packtpub.com.

1

A Tour of Interactive Computing with Jupyter and IPython

In this chapter, we will cover the following topics:

- ▶ Introducing IPython and the Jupyter Notebook
- ▶ Getting started with exploratory data analysis in the Jupyter Notebook
- ▶ Introducing the multidimensional array in NumPy for fast array computations
- ▶ Creating an IPython extension with custom magic commands
- ▶ Mastering IPython's configuration system
- ▶ Creating a simple kernel for Jupyter

Introduction

In this introduction, we will give a broad overview of Python, IPython, Jupyter, and the scientific Python ecosystem.

What is Python?

Python is a high-level, open-source, general-purpose programming language originally conceived by Guido van Rossum in the late 1980s (the name was inspired by the British comedy *Monty Python's Flying Circus*). This easy-to-use language is commonly used by system administrators as a glue language, linking various system components together. It is also a robust language for large-scale software development. In addition, Python comes with an extremely rich standard library (the batteries included philosophy), which covers string processing, internet protocols, operating system interfaces, and many other domains.

In the last twenty years, Python has been increasingly used for scientific computing and data analysis as well. Other competing platforms include commercial software such as MATLAB, Maple, Mathematica, Excel, SPSS, SAS, and others. Competing open-source platforms include Julia, R, Octave, and Scilab. These tools are dedicated to scientific computing, whereas Python is a general-purpose programming language that was not initially designed for scientific computing.

However, a wide ecosystem of tools has been developed to bring Python to the level of these other scientific computing systems. Today, the main advantage of Python, and one of the main reasons why it is so popular, is that it brings scientific computing features to a general-purpose language that is used in many research areas and industries. This makes the transition from research to production much easier.

What is IPython?

IPython is a Python library that was originally meant to improve the default interactive console provided by Python, and to make it scientist-friendly. In 2011, ten years after the first release of IPython, the **IPython Notebook** was introduced. This web-based interface to IPython combines code, text, mathematical expressions, inline plots, interactive figures, widgets, graphical interfaces, and other rich media within a standalone sharable web document. This platform provides an ideal gateway to interactive scientific computing and data analysis. IPython has become essential to researchers, engineers, data scientists, and teachers and their students.

What is Jupyter?

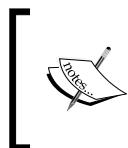
Within a few years, IPython gained an incredible popularity among the scientific and engineering communities. The Notebook started to support more and more programming languages beyond Python. In 2014, the IPython developers announced the **Jupyter** project, an initiative created to improve the implementation of the Notebook and make it language-agnostic by design. The name of the project reflects the importance of three of the main scientific computing languages supported by the Notebook: Julia, Python, and R.

Today, Jupyter is an ecosystem by itself that comprehends several alternative Notebook interfaces (JupyterLab, nteract, Hydrogen, and others), interactive visualization libraries, and authoring tools compatible with notebooks. Jupyter has its own conference named JupyterCon. The project received funding from several companies as well as the Alfred P. Sloan Foundation and the Gordon and Betty Moore Foundation.

What is the SciPy ecosystem?

SciPy is the name of a Python package for scientific computing, but it refers also, more generally, to the collection of all Python tools that have been developed to bring scientific computing features to Python.

In the late 1990s, Travis Oliphant and others started to build efficient tools to deal with numerical data in Python: Numeric, Numarray, and finally, **NumPy**. **SciPy**, which implements many numerical computing algorithms, was also created on top of NumPy. In the early 2000s, John Hunter created **Matplotlib** to bring scientific graphics to Python. At the same time, Fernando Perez created **IPython** to improve interactivity and productivity in Python. In the late 2000s, Wes McKinney created **pandas** for the manipulation and analysis of numerical tables and time series. Since then, hundreds of engineers and researchers collaboratively worked on this platform to make SciPy one of the leading open source platforms for scientific computing and data science.



Many of the SciPy tools are supported by NumFOCUS, a nonprofit that was created as a legal structure to promote the sustainable development of the ecosystem. NumFOCUS is supported by several large companies including Microsoft, IBM, and Intel.



SciPy has its own conferences, too: SciPy (in the US) and EuroSciPy (in Europe) (see [HTTPS : // CONFERENCE.SCIPY.ORG/](https://CONFERENCE.SCIPY.ORG/)).

What's new in the SciPy ecosystem?

What are some of the main changes in the SciPy ecosystem since the first edition of this book, published in 2014? We give here a very brief selection.



Feel free to skip this section if you are new to the platform.



The last version of IPython at the time of writing is IPython 6.0, released in April 2017. It is the first version of IPython that is no longer compatible with Python 2. This decision allowed the developers to make the internal code simpler and to make better use of the new features of the language.

IPython now has a web-based Terminal interface that can be used along with notebooks. Keyboard shortcuts can be edited directly from the Notebook interface. Multiple cells can be selected and copy/pasted between notebooks. There is a new restart-and-run-all button and a find-and-replace option in the Notebook. See <http://ipython.readthedocs.io/en/stable/whatsnew/version6.html> for more details.

NumPy, which last version 1.13 was released in June 2017, now supports the `@` matrix multiplication operator between matrices (it was previously accessible via the `np.dot()` function). Operations such as `a + b + c` use less memory and are faster on some systems (temporary elision). The new `np.block()` function lets one define block matrices. The new `np.stack()` function joins a sequence of arrays along a new axis. See <https://docs.scipy.org/doc/numpy-1.13.0/release.html> for more details.

SciPy 1.0 was released in October 2017. For the developers, the 1.0 version means that the library has reached some stability and maturity after 16 years of development. See <https://docs.scipy.org/doc/scipy/reference/release.html> for more details.

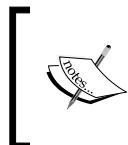
Matplotlib, of which version 2.1 was released in October 2017, has an improved styling and a much better default color palette with the *viridis* colormap instead of *jet*. See <https://github.com/matplotlib/matplotlib/releases> for more details.

pandas 0.21 was released in October 2017. pandas now supports categorical data. Several deprecations were done in the past years, with the deprecation of the `.ix` syntax and Panels (which may be replaced via the `xarray` library). See <https://pandas.pydata.org/pandas-docs/stable/release.html> for more details.

How to install Python

In this book, we use the **Anaconda** distribution, which is available at <https://www.anaconda.com/download/>. Anaconda works on Linux, macOS, and Windows. You should install the latest version of Anaconda (5.0.1 at the time of writing) with the latest 64-bit version of Python (3.6 at the time of writing). Python 2.7 is an old version that will be officially unsupported in 2020.

Anaconda comes with Python, IPython, Jupyter, NumPy, SciPy, pandas, Matplotlib, and almost all of the other scientific packages we will be using in this book. The list of all packages is available at <https://docs.anaconda.com/anaconda/packages/pkg-docs>.



Miniconda is a light version of Anaconda with only Python and a few other essential packages. You can install only the packages you need one by one using the `conda` package manager of Anaconda.



We won't cover in this book the various other ways of installing a scientific Python distribution.

The Anaconda website should give you all the instructions to install Anaconda on your system. To install new packages, you can use the conda package manager that comes with Anaconda. For example, to install the `ipyparallel` package (which is currently not installed by default in Anaconda), type `conda install ipyparallel` in a system shell.



A short introduction to system shells is given in the *Learning the basics of the Unix shell* section of Chapter 2, *Best Practices in Interactive Computing*.



Another way of installing packages is with conda-forge, available at <https://conda-forge.org/>. This is a community-driven effort to automatically build the latest versions of packages available on GitHub, and make them available with conda. If a package is not available with `conda install somepackage`, one may use instead `conda install --channel conda-forge somepackage` if the package is supported by conda-forge.



GitHub is a commercial service that provides free and paid hosting for software repositories. It is one of the most popular platforms for open source collaborative development.



pip is the Python system manager. Contrary to conda, pip works with any Python distribution, not just with Anaconda. Packages installable by pip are stored on the **Python Package Index (PyPI)** available at <https://pypi.python.org/pypi>.

Almost all Python packages available in conda are also available in pip, but the inverse is not true. In practice, if a package is not available in conda or conda-forge, it should be available with `pip install somepackage`. conda packages typically include binaries compiled for the most common platforms, whereas that is not necessarily the case with pip packages. pip packages may contain source code that has to be compiled locally (which requires that a compatible compiler is installed and configured), but they may also contain compiled binaries.

References

Here are a few references:

- ▶ The Python web page at <https://www.python.org>
- ▶ Python on Wikipedia at https://en.wikipedia.org/wiki/Python_%28programming_language%29
- ▶ Python's standard library at <https://docs.python.org/3/library/>
- ▶ Conversation with Guido van Rossum on the birth of Python available at <http://www.artima.com/intv/pythonP.html>
- ▶ History of scientific Python available at <http://fr.slideshare.net/shoheihibo/sci-pyhistory>

- ▶ History of the Jupyter Notebook at <http://blog.fperez.org/2012/01/ipython-notebook-historical.html>
- ▶ JupyterCon at <https://conferences.oreilly.com/jupyter/jup-ny>

Here are a few resources on scientific Python:

- ▶ *Introduction to Python for Computational Science and Engineering*, at <https://github.com/fangohr/introduction-to-python-for-computational-science-and-engineering>
- ▶ Statistical Computing and Computation, at <http://people.duke.edu/~ccc14/sta-663-2017/>
- ▶ SciPy 2017 videos at <https://www.youtube.com/playlist?list=PLYx7XA2nY5GfdAFycPLBdUDOUTdQIVoMf>

Introducing IPython and the Jupyter Notebook

The **Jupyter Notebook** is a web-based interactive environment that combines code, rich text, images, videos, animations, mathematical equations, plots, maps, interactive figures and widgets, and graphical user interfaces, into a single document. This tool is an ideal gateway to high-performance numerical computing and data science in Python, R, Julia, or other languages. In this book, we will mostly use the Python language, although there are recipes introducing R and Julia.

In this recipe, we give an introduction to IPython and the Jupyter Notebook.

Getting ready

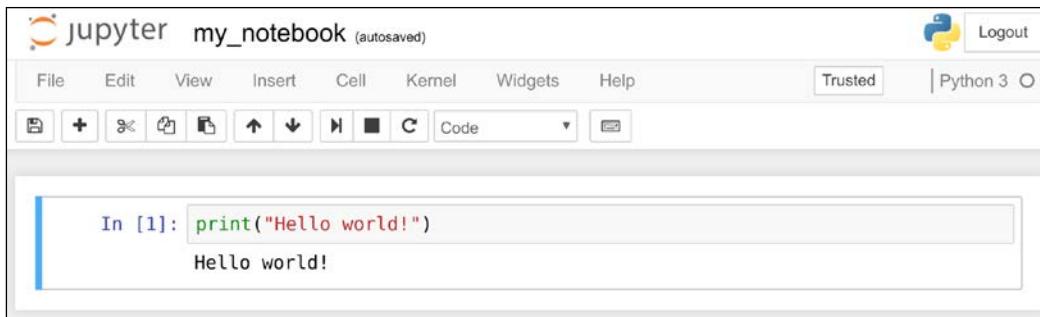
This chapter's introduction gives the instructions to install the Anaconda distribution, which comes with Jupyter and almost all Python libraries we will be using in this book.

Once Anaconda is installed, download the code from the book's website and open a Terminal in that folder. In the Terminal, type `jupyter notebook`. Your default web browser should open automatically and load the address `http://localhost:8888` (a server that runs on your computer). You're ready to get started!

How to do it...

1. Let's create a new Jupyter notebook using an IPython kernel. We type the following command in a cell, and press *Shift + Enter* to evaluate it:

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



A notebook contains a linear succession of **cells** and **output areas**. A cell contains Python code, in one or multiple lines. The output of the code is shown in the corresponding output area.



In this book, the prompt >>> means that you need to type everything that starts after it. The >>> characters themselves should not be typed.

2. Now, we do a simple arithmetic operation:

```
>>> 2 + 2  
4
```

The result of the operation is shown in the output area. More precisely, the output area not only displays text that is printed by any command in the cell, but it also displays a text representation of the last returned object. Here, the last returned object is the result of $2 + 2$, that is, 4.

3. In the next cell, we can recover the value of the last returned object with the `_` (underscore) special variable. In practice, it might be more convenient to assign objects to named variables such as in `myresult = 2 + 2`.

```
>>> _ * 3  
12
```

4. IPython not only accepts Python code, but also shell commands. These commands are provided by the operating system. We first type `!` in a cell before typing the shell command. Here, assuming a Linux or macOS system, we get the list of all the notebooks in the current directory:

```
>>> !ls  
my_notebook.ipynb
```

On Windows, one may replace `ls` by `dir`.

5. IPython comes with a library of magic commands. These commands are convenient shortcuts to common actions. They all start with % (the percent character). We can get the list of all magic commands with %lsmagic:

```
>>> %lsmagic
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 %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 %%js %%latex %%markdown %%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.

Cell magics have a %% prefix; they target entire code cells.

6. For example, the %%writefile cell magic lets us create a text file. This magic command accepts a filename as an argument. All the remaining lines in the cell are directly written to this text file. Here, we create a test.txt file and write Hello world! into it:

```
>>> %%writefile test.txt
Hello world!
Writing test.txt
>>> # Let's check what this file contains.
      with open('test.txt', 'r') as f:
          print(f.read())
Hello world!
```

7. As we can see in the output of %lsmagic, there are many magic commands in IPython. We can find more information about any command by adding ? after it. For example, to get some help about the %run magic command, we type %run? as shown here:

```
>>> %run?
```

```

Docstring:
Run the named file inside IPython as a program.

Usage::

%run [-n -i -e -G]
  [(-t [-N<N>] | -d [-b<N>] | -p [profile options] )]
  (-m mod | file ) [args]

Parameters after the filename are passed as command-line arguments to
the program (put in sys.argv). Then, control returns to IPython's
prompt.

This is similar to running at a system prompt ``python file args``.

```

The pager (a text area at the bottom of the screen) opens and shows the help of the `%run` magic command.

- We covered the basics of IPython and the Notebook. Let's now turn to the rich display and interactive features of the Notebook. Until now, we have only created **code cells** (containing code). Jupyter supports other types of cells. In the Notebook toolbar, there is a drop-down menu to select the cell's type. The most common cell type after the code cell is the **Markdown cell**.

Markdown cells contain rich text formatted with **Markdown**, a popular plain text-formatting syntax. This format supports normal text, headers, bold, italics, hypertext links, images, mathematical equations in **LaTeX** (a typesetting system adapted to mathematics), code, HTML elements, and other features, as shown here:

```

### New paragraph

This is a *rich* **text** with links(http://jupyter.org), equations:
$$\hat{f}(\xi) = \int_{-\infty}^{+\infty} f(x) \exp(-2i\pi x \xi) dx$$

code with syntax highlighting:

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

and images:


```

New paragraph

This is a rich text with [links](#), equations:

$$\hat{f}(\xi) = \int_{-\infty}^{+\infty} f(x) \exp(-2i\pi x \xi) dx$$

code with syntax highlighting:

```
print("hello world!")
```

and images:



Markdown cell

Running a Markdown cell (by pressing *Shift + Enter*, for example) displays the output, as shown in the bottom panel of the preceding screenshot.

By combining code cells and Markdown cells, we create a standalone interactive document that combines computations (code), text, and graphics.

9. The Jupyter Notebook also comes with a sophisticated display system that lets us insert rich web elements in the Notebook. Here, we show how to add HTML, **Scalable Vector Graphics (SVG)**, and even YouTube videos in a notebook. First, we need to import some classes:

```
>>> from IPython.display import HTML, SVG, YouTubeVideo
```

10. We create an HTML table dynamically with Python, and we display it in the (HTML-based) notebook.

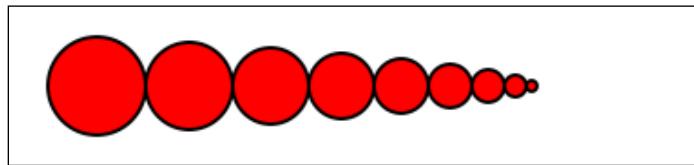
```
>>> HTML('''
      <table style="border: 2px solid black;">
      ''' +
      ''.join(['<tr>' +
               ''.join([f'<td>{row},{col}</td>' +
                       for col in range(5)]) +
               '</tr>' for row in range(5)]) +
      '''
      </table>
      ''')
```

| | | | | |
|-----|-----|-----|-----|-----|
| 0,0 | 0,1 | 0,2 | 0,3 | 0,4 |
| 1,0 | 1,1 | 1,2 | 1,3 | 1,4 |
| 2,0 | 2,1 | 2,2 | 2,3 | 2,4 |
| 3,0 | 3,1 | 3,2 | 3,3 | 3,4 |
| 4,0 | 4,1 | 4,2 | 4,3 | 4,4 |

11. Similarly, we create an SVG image dynamically:

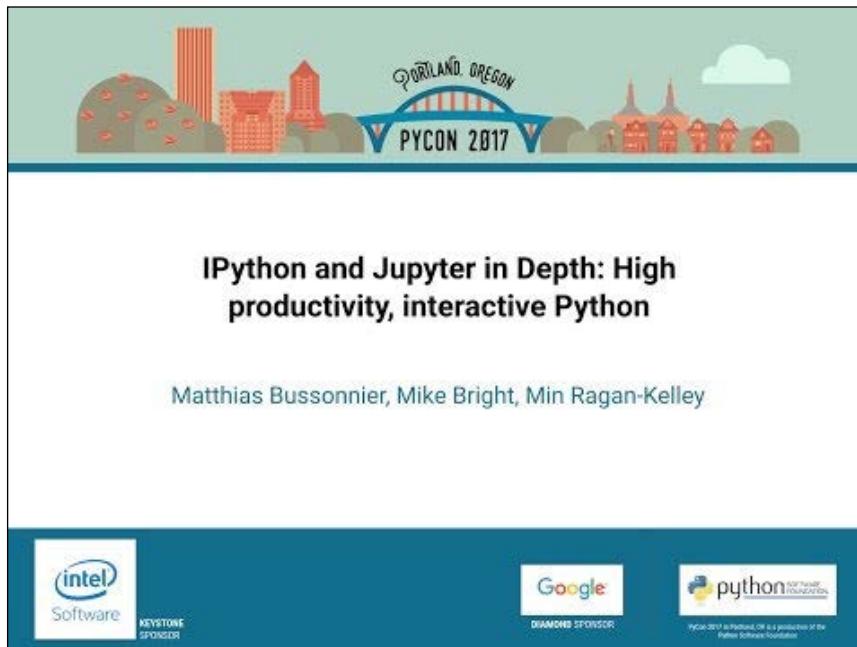
```
>>> SVG('''<svg width="600" height="80">''' +
      ''.join([f'''<circle
                  cx="{(30 + 3*i) * (10 - i)}"
                  cy="30"
                  r="{3. * float(i)}"
                  fill="red"
                  stroke-width="2"
                  stroke="black">''' for i in range(10)]))
```

```
</circle>''' for i in range(10)]) +  
'''</svg>'')
```



12. We display a YouTube video by giving its identifier to `YoutubeVideo`:

```
>>> YouTubeVideo('VQBZ2MqWBZI')
```



There's more...

Notebooks are saved as structured text files (JSON format), which makes them easily shareable. Here are the contents of a simple notebook:

```
{  
  "cells": [  
    {  
      "cell_type": "code",  
      "execution_count": 1,  
      "outputs": []  
    }  
  ]  
}
```

```
    "metadata": {},
    "outputs": [
        {
            "name": "stdout",
            "output_type": "stream",
            "text": [
                "Hello world!\n"
            ]
        }
    ],
    "source": [
        "print(\"Hello world!\")"
    ]
},
],
"metadata": {},
"nbformat": 4,
"nbformat_minor": 2
}
```

Jupyter comes with a special tool, **nbconvert**, which converts notebooks to other formats such as HTML and PDF (<https://nbconvert.readthedocs.io/en/stable/>).

Another online tool, **nbviewer** (<http://nbviewer.jupyter.org>), allows us to render a publicly-available notebook directly in the browser.

We will cover many of these possibilities in subsequent chapters, notably in *Chapter 3, Mastering the Jupyter Notebook*.

There are other implementations of Jupyter Notebook frontends that offer different ways of interacting with the same notebook documents. JupyterLab, an IDE for interactive computing and data science, is the future of the Jupyter Notebook. It is introduced in *Chapter 3, Mastering the Jupyter Notebook*. **nteract** is a desktop application that lets the user open a notebook file by double-clicking on it, without using the Terminal and using a web browser. Hydrogen is a plugin of the Atom text editor that provides rich interactive capabilities when opening notebook files. Juno is a Jupyter Notebook client for iPad.

Here are a few references about the Notebook:

- ▶ *Installing Jupyter*, available at <http://jupyter.org/install.html>
- ▶ Documentation of the Notebook available at <http://jupyter.readthedocs.io/en/latest/index.html>
- ▶ Security in Jupyter notebooks, at <https://jupyter-notebook.readthedocs.io/en/stable/security.html#Security-in-notebook-documents>

- ▶ User-curated gallery of interesting notebooks available at <https://github.com/jupyter/jupyter/wiki/A-gallery-of-interesting-Jupyter-Notebooks>
- ▶ JupyterLab at <https://github.com/jupyterlab/jupyterlab>
- ▶ nteract at <https://nteract.io>
- ▶ Hydrogen at <https://nteract.io/atom>
- ▶ Juno at <https://juno.sh/>

See also

- ▶ The *Getting started with exploratory data analysis in the Jupyter Notebook* recipe
- ▶ The *Introducing JupyterLab* recipe in Chapter 3, *Mastering the Jupyter Notebook*

Getting started with exploratory data analysis in the Jupyter Notebook

In this recipe, we will give an introduction to IPython and Jupyter for data analysis. Most of the subject has been covered in the prequel of this book, *Learning IPython for Interactive Computing and Data Visualization, Second Edition*, Packt Publishing, but we will review the basics here.

We will download and analyze a dataset about attendance on Montreal's bicycle tracks. This example is largely inspired by a presentation from Julia Evans (available at <https://github.com/jvns/talks/blob/master/2013-04-mtlipy/pistes-cyclables.ipynb>). Specifically, we will introduce the following:

- ▶ Data manipulation with pandas
- ▶ Data visualization with Matplotlib
- ▶ Interactive widgets

How to do it...

1. The very first step is to import the scientific packages we will be using in this recipe, namely NumPy, pandas, and Matplotlib. We also instruct Matplotlib to render the figures as inline images in the Notebook:

```
>>> import numpy as np
      import pandas as pd
      import matplotlib.pyplot as plt
      %matplotlib inline
```



We can enable high-resolution Matplotlib figures on Retina display systems with the following commands:

```
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('retina')
```

- Now, we create a new Python variable called `url` that contains the address to a **Comma-separated Values (CSV)** data file. This standard text-based file format is used to store tabular data:

```
>>> url = ("https://raw.githubusercontent.com/"
           "ipython-books/cookbook-2nd-data/"
           "master/bikes.csv")
```

- pandas defines a `read_csv()` function that can read any CSV file. Here, we pass the URL to the file. pandas will automatically download the file, parse it, and return a DataFrame object. We need to specify a few options to make sure that the dates are parsed correctly:

```
>>> df = pd.read_csv(url, index_col='Date',
                     parse_dates=True, dayfirst=True)
```

- The `df` variable contains a DataFrame object, a specific pandas data structure that contains 2D tabular data. The `head(n)` method displays the first `n` rows of this table. In the Notebook, pandas displays a DataFrame object in an HTML table, as shown in the following screenshot:

```
>>> df.head(2)
```

| Date | Unnamed: 1 | Berri1 | CSC | Mais1 | Mais2 | Parc | PierDup | Rachel1 | Totem_Laurier |
|------------|------------|--------|-----|-------|-------|------|---------|---------|---------------|
| 2013-01-01 | | 00:00 | 0 | 0 | 1 | 0 | 6 | 0 | 1 |
| 2013-01-02 | | 00:00 | 69 | 0 | 13 | 0 | 18 | 0 | 2 |

Here, every row contains the number of bicycles on every track of the city, for every day of the year.

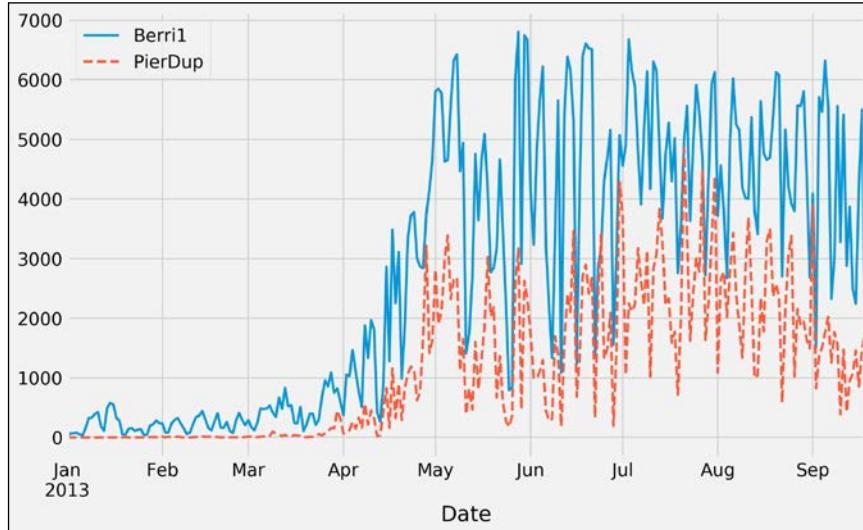
- We can get some summary statistics of the table with the `describe()` method:

```
>>> df.describe()
```

| | Berri1 | CSC | Mais1 | Mais2 | Parc | PierDup | Rachel1 | Totem_Laurier |
|-------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|---------------|
| count | 261.000000 | 261.000000 | 261.000000 | 261.000000 | 261.000000 | 261.000000 | 261.000000 | 261.000000 |
| mean | 2743.390805 | 1221.858238 | 1757.590038 | 3224.130268 | 1669.425287 | 1152.885057 | 3084.425287 | 1858.793103 |
| std | 2247.957848 | 1070.037364 | 1458.793882 | 2589.514354 | 1363.738862 | 1208.848429 | 2380.255540 | 1434.899574 |
| min | 0.000000 | 0.000000 | 1.000000 | 0.000000 | 6.000000 | 0.000000 | 0.000000 | 0.000000 |
| 25% | 392.000000 | 12.000000 | 236.000000 | 516.000000 | 222.000000 | 12.000000 | 451.000000 | 340.000000 |
| 50% | 2771.000000 | 1184.000000 | 1706.000000 | 3178.000000 | 1584.000000 | 818.000000 | 3111.000000 | 2087.000000 |
| 75% | 4767.000000 | 2168.000000 | 3158.000000 | 5812.000000 | 3068.000000 | 2104.000000 | 5338.000000 | 3168.000000 |
| max | 6803.000000 | 3330.000000 | 4716.000000 | 7684.000000 | 4103.000000 | 4841.000000 | 8555.000000 | 4293.000000 |

6. Let's display some figures. We will plot the daily attendance of two tracks. First, we select the two columns, Berri1 and PierDup. Then, we call the `plot()` method:

```
>>> df[['Berri1', 'PierDup']].plot(figsize=(10, 6),
                                style=['-', '--'],
                                lw=2)
```



7. Now, we move to a slightly more advanced analysis. We will look at the attendance of all tracks as a function of the weekday. We can get the weekday easily with pandas: the `index` attribute of the DataFrame object contains the dates of all rows in the table. This index has a few date-related attributes, including `weekday_name`:

```
>>> df.index.weekday_name
Index(['Tuesday', 'Wednesday', 'Thursday', 'Friday',
       'Saturday', 'Sunday', 'Monday', 'Tuesday',
       ...]
```

```
'Friday', 'Saturday', 'Sunday', 'Monday',
'Tuesday', 'Wednesday'],
dtype='object', name='Date', length=261)
```

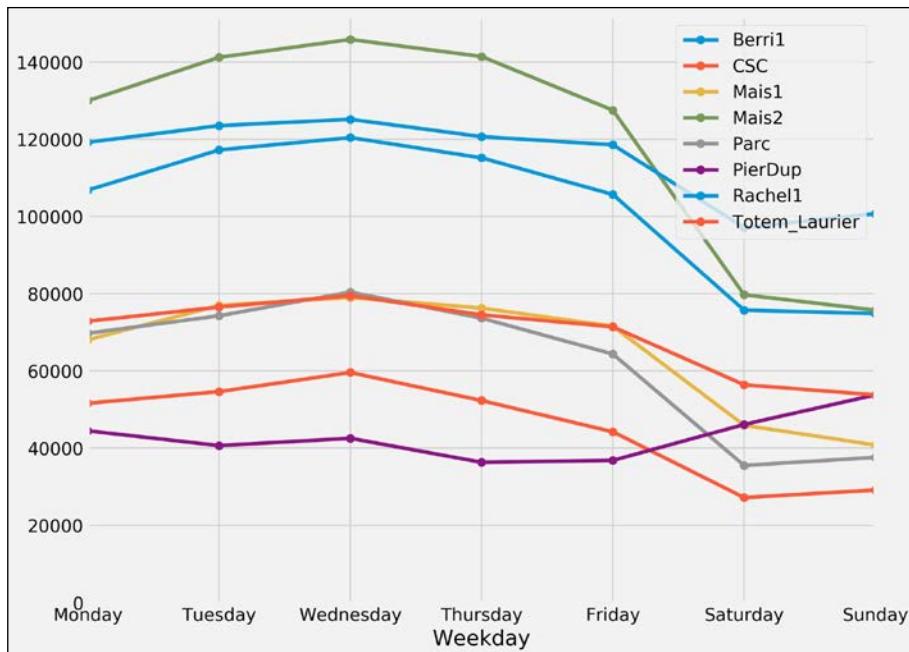
8. To get the attendance as a function of the weekday, we need to group the table elements by the weekday. The `groupby()` method lets us do just that. We use `weekday` instead of `weekday_name` to keep the weekday order (Monday is 0, Tuesday is 1, and so on). Once grouped, we can sum all rows in every group:

```
>>> df_week = df.groupby(df.index.weekday).sum()
>>> df_week
```

| | Berri1 | CSC | Mais1 | Mais2 | Parc | PierDup | Rachel1 | Totem_Laurier |
|------|--------|-------|-------|--------|-------|---------|---------|---------------|
| Date | | | | | | | | |
| 0 | 106826 | 51646 | 68087 | 129982 | 69767 | 44500 | 119211 | 72883 |
| 1 | 117244 | 54656 | 76974 | 141217 | 74299 | 40679 | 123533 | 76559 |
| 2 | 120434 | 59604 | 79033 | 145860 | 80437 | 42564 | 125173 | 79501 |
| 3 | 115193 | 52340 | 76273 | 141424 | 73668 | 36349 | 120684 | 74540 |
| 4 | 105701 | 44252 | 71605 | 127526 | 64385 | 36850 | 118556 | 71426 |
| 5 | 75754 | 27226 | 45947 | 79743 | 35544 | 46149 | 97143 | 56438 |
| 6 | 74873 | 29181 | 40812 | 75746 | 37620 | 53812 | 100735 | 53798 |

9. We can now display this information in a figure. We create a Matplotlib figure, and we use the `plot()` method of DataFrame to create our plot:

```
>>> fig, ax = plt.subplots(1, 1, figsize=(10, 8))
df_week.plot(style='-o', lw=3, ax=ax)
ax.set_xlabel('Weekday')
# We replace the labels 0, 1, 2... by the weekday
# names.
ax.set_xticklabels(
    ('Monday,Tuesday,Wednesday,Thursday,'
     'Friday,Saturday,Sunday').split(','))
ax.set_xlim(0) # Set the bottom axis to 0.
```

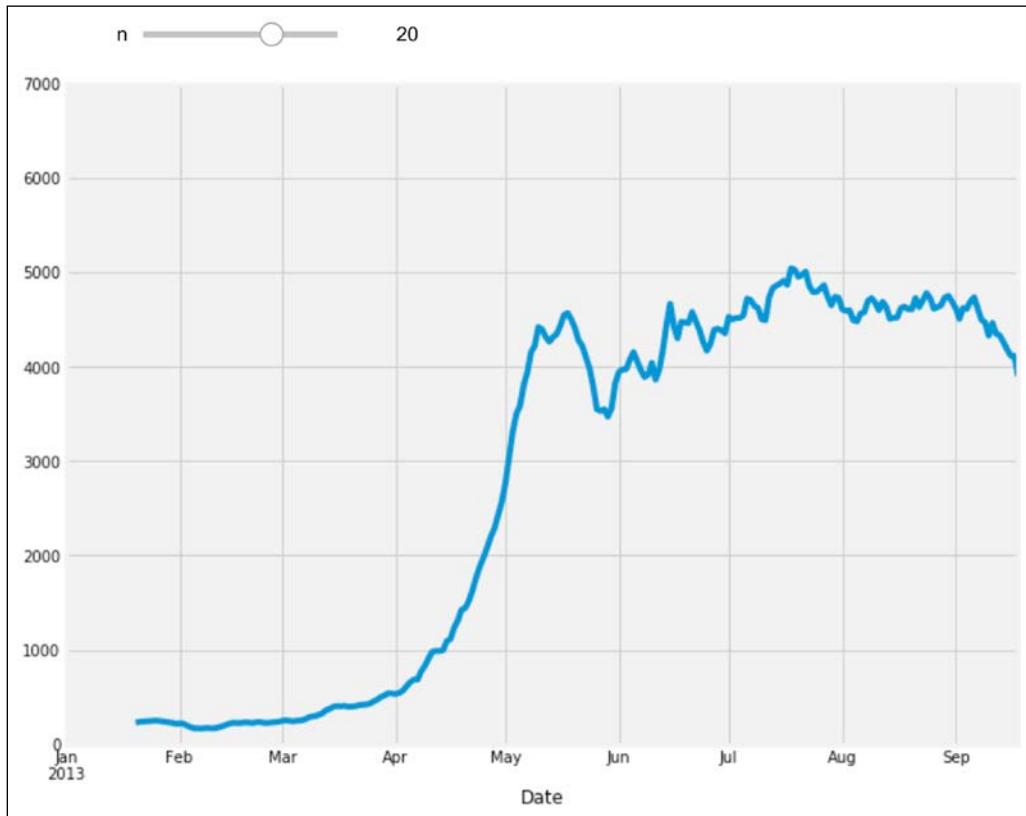


10. Finally, let's illustrate the interactive capabilities of the Notebook. We will plot a smoothed version of the track attendance as a function of time (**rolling mean**). The idea is to compute the mean value in the neighborhood of any day. The larger the neighborhood, the smoother the curve. We will create an interactive slider in the Notebook to vary this parameter in real time in the plot. All we have to do is add the `@interact` decorator above our plotting function:

```
>>> from ipywidgets import interact

@interact
def plot(n=(1, 30)):
    fig, ax = plt.subplots(1, 1, figsize=(10, 8))
    df['Berri1'].rolling(window=n).mean().plot(ax=ax)
```

```
ax.set_ylim(0, 7000)
plt.show()
```



How it works...

To create Matplotlib figures, it is good practice to create a Figure (`fig`) and one or several Axes (subplots, `ax` object) objects with the `plt.subplots()` command. The `figsize` keyword argument lets us specify the size of the figure, in inches. Then, we call plotting methods directly on the Axes instances. Here, for example, we set the y limits of the axis with the `set_ylim()` method. If there are existing plotting commands, like the `plot()` method provided by pandas on DataFrame instances, we can pass the relevant Axis instance with the `ax` keyword argument.

There's more...

pandas is the main data wrangling library in Python. Other tools and methods are generally required for more advanced analyses (signal processing, statistics, and mathematical modeling). We will cover these steps in the second part of this book, starting with *Chapter 7, Statistical Data Analysis*.

Here are some more references about data manipulation with pandas:

- ▶ *Learning IPython for Interactive Computing and Data Visualization, Second Edition*, Packt Publishing, the prequel of this book
- ▶ *Python for Data Analysis*, O'Reilly Media, by Wes McKinney, the creator of pandas, at <http://shop.oreilly.com/product/0636920023784.do>
- ▶ *Python Data Science Handbook*, O'Reilly Media, by Jake VanderPlas, at <http://shop.oreilly.com/product/0636920034919.do>
- ▶ The documentation of pandas available at <http://pandas.pydata.org/pandas-docs/stable/>
- ▶ Usage guide of Matplotlib, at <https://matplotlib.org/tutorials/introductory/usage.html>

See also

- ▶ The *Introducing the multidimensional array in NumPy for fast array computations* recipe

Introducing the multidimensional array in NumPy for fast array computations

NumPy is the main foundation of the scientific Python ecosystem. This library offers a specific data structure for high-performance numerical computing: the **multidimensional array**. The rationale behind NumPy is the following: Python being a high-level dynamic language, it is easier to use but slower than a low-level language such as C. NumPy implements the multidimensional array structure in C and provides a convenient Python interface, thus bringing together high performance and ease of use. NumPy is used by many Python libraries. For example, pandas is built on top of NumPy.

In this recipe, we will illustrate the basic concepts of the multidimensional array. A more comprehensive coverage of the topic can be found in the book, *Learning IPython for Interactive Computing and Data Visualization, Second Edition*, Packt Publishing.

How to do it...

1. Let's import the built-in `random` Python module and NumPy:

```
>>> import random  
import numpy as np
```

2. We generate two Python lists, `x` and `y`, each one containing 1 million random numbers between 0 and 1:

```
>>> n = 1000000  
x = [random.random() for _ in range(n)]  
y = [random.random() for _ in range(n)]  
>>> x[:3], y[:3]  
([0.926, 0.722, 0.962], [0.291, 0.339, 0.819])
```

3. Let's compute the element-wise sum of all of these numbers: the first element of `x` plus the first element of `y`, and so on. We use a `for` loop in a list comprehension:

```
>>> z = [x[i] + y[i] for i in range(n)]  
z[:3]  
[1.217, 1.061, 1.781]
```

4. How long does this computation take? IPython defines a handy `%timeit` magic command to quickly evaluate the time taken by a single statement:

```
>>> %timeit [x[i] + y[i] for i in range(n)]  
101 ms ± 5.12 ms per loop (mean ± std. dev. of 7 runs,  
10 loops each)
```

5. Now we will perform the same operation with NumPy. NumPy works on multidimensional arrays, so we need to convert our lists to arrays. The `np.array()` function does just that:

```
>>> xa = np.array(x)  
ya = np.array(y)  
>>> xa[:3]  
array([ 0.926, 0.722, 0.962])
```

The `xa` and `ya` arrays contain the exact same numbers that our original lists, `x` and `y`, contained. Those lists were instances of the list built-in class, while our arrays are instances of the `ndarray` NumPy class. These types are implemented very differently in Python and NumPy. In this example, we will see that using arrays instead of lists leads to drastic performance improvements.

6. To compute the element-wise sum of these arrays, we don't need to do a `for` loop anymore. In NumPy, adding two arrays means adding the elements of the arrays component-by-component. This is the standard mathematical notation in linear algebra (operations on vectors and matrices):

```
>>> za = xa + ya  
za[:3]  
array([ 1.217,  1.061,  1.781])
```

We see that the `z` list and the `za` array contain the same elements (the sum of the numbers in `x` and `y`).



Be careful not to use the `+` operator between vectors when they are represented as Python lists! This operator is valid between lists, so it would not raise an error and it could lead to subtle and silent bugs. In fact, `list1 + list2` is the concatenation of two lists, not the element-wise addition.

7. Let's compare the performance of this NumPy operation with the native Python loop:

```
>>> %timeit xa + ya  
1.09 ms ± 37.3 µs per loop (mean ± std. dev. of 7 runs,  
1000 loops each)
```

With NumPy, we went from 100 ms down to 1 ms to compute one million additions!

8. Now we will compute something else: the sum of all elements in `x` or `xa`. Although this is not an element-wise operation, NumPy is still highly efficient here. The pure Python version uses the built-in `sum()` function on an iterable. The NumPy version uses the `np.sum()` function on a NumPy array:

```
>>> %timeit sum(x) # pure Python  
3.94 ms ± 4.44 µs per loop (mean ± std. dev. of 7 runs  
100 loops each)  
>>> %timeit np.sum(xa) # NumPy  
298 µs ± 4.62 µs per loop (mean ± std. dev. of 7 runs,  
1000 loops each)
```

We also observe a significant speedup here.

9. Let's perform one last operation: computing the arithmetic distance between any pair of numbers in our two lists (we only consider the first 1000 elements to keep computing times reasonable). First, we implement this in pure Python with two nested `for` loops:

```
>>> d = [abs(x[i] - y[j])
        for i in range(1000)
        for j in range(1000)]
>>> d[:3]
[0.635, 0.587, 0.106]
```

10. Now, we use a NumPy implementation, bringing out two slightly more advanced notions. First, we consider a **two-dimensional array** (or matrix). This is how we deal with the two indices, `i` and `j`. Second, we use **broadcasting** to perform an operation between a 2D array and 1D array. We will give more details in the *How it works...* section.

```
>>> da = np.abs(xa[:1000, np.newaxis] - ya[:1000])
>>> da
array([[ 0.635,  0.587, ...,  0.849,  0.046],
       [ 0.431,  0.383, ...,  0.646,  0.158],
       ...,
       [ 0.024,  0.024, ...,  0.238,  0.566],
       [ 0.081,  0.033, ...,  0.295,  0.509]])
>>> %timeit [abs(x[i] - y[j]) \
              for i in range(1000) \
              for j in range(1000)]
134 ms ± 1.79 ms per loop (mean ± std. dev. of 7 runs,
 1000 loops each)
>>> %timeit np.abs(xa[:1000, np.newaxis] - ya[:1000])
1.54 ms ± 48.9 µs per loop (mean ± std. dev. of 7 runs
 1000 loops each)
```

Here again, we observe a significant speedup.

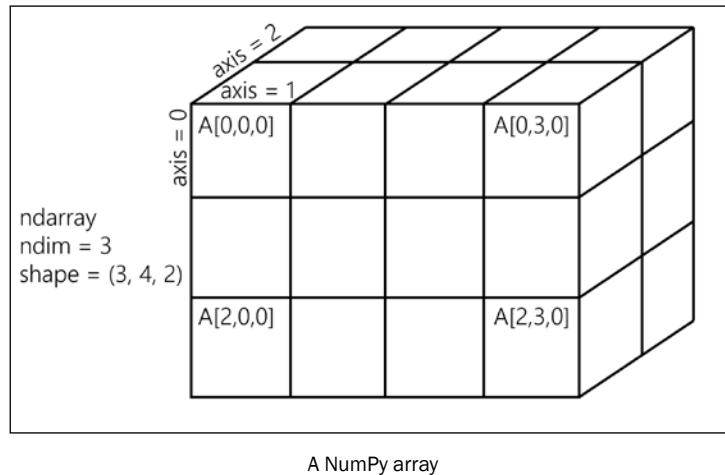
How it works...

A NumPy array is a homogeneous block of data organized in a multidimensional finite grid. All elements of the array share the same **data type**, also called **dtype** (integer, floating-point number, and so on). The shape of the array is an n-tuple that gives the size of each axis.

A 1D array is a **vector**; its shape is just the number of components.

A 2D array is a **matrix**; its shape is (number of rows, number of columns).

The following diagram illustrates the structure of a 3D (3, 4, 2) array that contains 24 elements:



The slicing syntax in Python translates nicely to array indexing in NumPy. Also, we can add an extra dimension to an existing array, using `np.newaxis` in the index.

Element-wise arithmetic operations can be performed on NumPy arrays that have the same shape. However, broadcasting relaxes this condition by allowing operations on arrays with different shapes in certain conditions. Notably, when one array has fewer dimensions than the other, it can be virtually stretched to match the other array's dimension. This is how we computed the pairwise distance between any pair of elements in `xa` and `ya`.

How can array operations be so much faster than Python loops? There are several reasons, and we will review them in detail in *Chapter 4, Profiling and Optimization*. We can already say here that:

- ▶ In NumPy, array operations are implemented internally with C loops rather than Python loops. Python is typically slower than C because of its interpreted and dynamically-typed nature.
- ▶ The data in a NumPy array is stored in a **contiguous** block of memory in RAM. This property leads to more efficient use of CPU cycles and cache.

There's more...

There's obviously much more to say about this subject. The prequel of this book, *Learning IPython for Interactive Computing and Data Visualization, Second Edition, Packt Publishing*, contains more details about basic array operations. We will use the array data structure routinely throughout this book. Notably, *Chapter 4, Profiling and Optimization*, covers advanced techniques of using NumPy arrays.

Here are some more references:

- ▶ Introduction to the ndarray on NumPy's documentation available at <http://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html>
- ▶ Tutorial on the NumPy array available at <https://docs.scipy.org/doc/numpy-dev/user/quickstart.html>
- ▶ The NumPy array in the SciPy lectures notes, at http://scipy-lectures.github.io/intro/numpy/array_object.html
- ▶ NumPy for MATLAB users, at [https://docs.scipy.org/doc/numpy-dev/user\(numpy-for-matlab-users.html](https://docs.scipy.org/doc/numpy-dev/user(numpy-for-matlab-users.html)

See also

- ▶ The *Getting started with exploratory data analysis in the Jupyter Notebook* recipe
- ▶ The *Understanding the internals of NumPy to avoid unnecessary array copying* recipe in *Chapter 4, Profiling and Optimization*

Creating an IPython extension with custom magic commands

Although IPython comes with a wide variety of magic commands, there are cases where we need to implement custom functionality in new magic commands. In this recipe, we will show how to create line and magic cells, and how to integrate them in an IPython extension.

How to do it...

1. Let's import a few functions from the IPython magic system:

```
>>> from IPython.core.magic import (register_line_magic,  
register_cell_magic)
```

2. Defining a new line magic is particularly simple. First, we create a function that accepts the contents of the line (except the initial %-prefixed name). The name of this function is the name of the magic. Then, we decorate this function with `@register_line_magic`:

```
>>> @register_line_magic  
def hello(line):  
    if line == 'french':
```

```
        print("Salut tout le monde!")
    else:
        print("Hello world!")
>>> %hello
Hello world!
>>> %hello French
Salut tout le monde!
```

3. Let's create a slightly more useful `%%csv` cell magic that parses a CSV string and returns a pandas DataFrame object. This time, the arguments of the function are the command's options and the contents of the cell:

```
>>> import pandas as pd
      from io import StringIO

@register_cell_magic
def csv(line, cell):
    # We create a string buffer containing the
    # contents of the cell.
    sio = StringIO(cell)
    # We use Pandas' read_csv function to parse
    # the CSV string.
    return pd.read_csv(sio)
>>> %%csv
      col1,col2,col3
      0,1,2
      3,4,5
      7,8,9
```

| | col1 | col2 | col3 |
|---|------|------|------|
| 0 | 0 | 1 | 2 |
| 1 | 3 | 4 | 5 |
| 2 | 7 | 8 | 9 |

We can access the returned object with `_`:

```
>>> df = _  
df.describe()
```

| | col1 | col2 | col3 |
|--------------|----------|----------|----------|
| count | 3.000000 | 3.000000 | 3.000000 |
| mean | 3.333333 | 4.333333 | 5.333333 |
| std | 3.511885 | 3.511885 | 3.511885 |
| min | 0.000000 | 1.000000 | 2.000000 |
| 25% | 1.500000 | 2.500000 | 3.500000 |
| 50% | 3.000000 | 4.000000 | 5.000000 |
| 75% | 5.000000 | 6.000000 | 7.000000 |
| max | 7.000000 | 8.000000 | 9.000000 |

4. The method we described is useful in an interactive session. If we want to use the same magic in multiple notebooks or if we want to distribute it, then we need to create an IPython extension that implements our custom magic command. The first step is to create a Python script (`csvmagic.py` here) that implements the magic. We also need to define a special function `load_ipython_extension(ipython)`:

```
>>> %%writefile csvmagic.py  
import pandas as pd  
from io import StringIO  
  
def csv(line, cell):  
    sio = StringIO(cell)  
    return pd.read_csv(sio)  
  
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(csv, 'cell')  
Writing csvmagic.py
```

5. Once the extension module is created, we need to import it into the IPython session. We can do this with the `%load_ext` magic command. Here, loading our extension immediately registers our `%%csv` magic function in the interactive shell:

```
>>> %load_ext csvmagic
>>> %%csv
    col1,col2,col3
    0,1,2
    3,4,5
    7,8,9
```

| | col1 | col2 | col3 |
|---|------|------|------|
| 0 | 0 | 1 | 2 |
| 1 | 3 | 4 | 5 |
| 2 | 7 | 8 | 9 |

How it works...

An IPython extension is a Python module that implements the top-level `load_ipython_extension(ipython)` function. When the `%load_ext` magic command is called, the module is loaded and the `load_ipython_extension(ipython)` function is called. This function is passed the current `InteractiveShell` instance as an argument. This object implements several methods we can use to interact with the current IPython session.

The `InteractiveShell` class

An interactive IPython session is represented by a (singleton) instance of the `InteractiveShell` class. This object handles the history, interactive namespace, and most features available in the session.

Within an interactive shell, we can get the current `InteractiveShell` instance with the `get_ipython()` function.

The list of all methods of `InteractiveShell` can be found in the reference API (see the reference at the end of this recipe). The following are the most important attributes and methods:

- ▶ `user_ns`: The user namespace (a dictionary).
- ▶ `push()`: Push (or inject) Python variables in the interactive namespace.
- ▶ `ev()`: Evaluate a Python expression in the user namespace.
- ▶ `ex()`: Execute a Python statement in the user namespace.
- ▶ `run_cell()`: Run a cell (given as a string), possibly containing IPython magic commands.
- ▶ `safe_execfile()`: Safely execute a Python script.
- ▶ `system()`: Execute a system command.
- ▶ `write()`: Write a string to the default output.
- ▶ `write_err()`: Write a string to the default error output.
- ▶ `register_magic_function()`: Register a standalone function as an IPython magic function. We used this method in this recipe.

Loading an extension

The Python extension module needs to be importable when using `%load_ext`. Here, our module is in the current directory. In other situations, it has to be in the Python path. It can also be stored in `~/.ipython/extensions`, which is automatically put in the Python path.

To ensure that our magic is automatically defined in our IPython profile, we can instruct IPython to load our extension automatically when a new interactive shell is launched. To do this, we have to open the `~/.ipython/profile_default/ipython_config.py` file and put `'csvmagic'` in the `c.InteractiveShellApp.extensions` list. The `csvmagic` module needs to be importable. It is common to create a Python package that implements an IPython extension, which itself defines custom magic commands.

There's more...

Many third-party extensions and magic commands exist, for example the `%%cython` magic that allows one to write Cython code directly in a notebook.

Here are a few references:

- ▶ Documentation of IPython's extension system available at <http://ipython.readthedocs.io/en/stable/config/extensions/index.html>
- ▶ Defining new magic commands explained at <http://ipython.readthedocs.io/en/stable/config/custommagics.html>

- ▶ Index of IPython extensions at <https://github.com/ipython/ipython/wiki/Extensions-Index>
- ▶ API reference of InteractiveShell available at <http://ipython.readthedocs.io/en/stable/api/generated/IPython.core.interactiveshell.html>

See also

- ▶ The *Mastering IPython's configuration system* recipe

Mastering IPython's configuration system

The `traitlets` package (<https://traitlets.readthedocs.io/en/stable/>), originated from IPython, implements a powerful configuration system. This system is used throughout the project, but it can also be used by IPython extensions. It could even be used in entirely new applications.

In this recipe, we show how to use this system to write a configurable IPython extension. We will create a simple magic command that displays random numbers. This magic command comes with configurable parameters that can be set by the user in their IPython configuration file.

How to do it...

1. We create an IPython extension in a `random_magics.py` file. Let's start by importing a few objects:

```
>>> %%writefile random_magics.py

from traitlets import Int, Float, Unicode, Bool
from IPython.core.magic import (Magics, magics_class,
                                 line_magic)
import numpy as np
Writing random_magics.py
```

2. We create a `RandomMagics` class deriving from `Magics`. This class contains a few configurable parameters:

```
>>> %%writefile random_magics.py -a

@magics_class
class RandomMagics(Magics):
    text = Unicode(u'{n}', config=True)
    max = Int(1000, config=True)
    seed = Int(0, config=True)
Appending to random_magics.py
```

3. We need to call the parent's constructor. Then, we initialize a random number generator with a seed:

```
>>> %%writefile random_magics.py -a

    def __init__(self, shell):
        super(RandomMagics, self).__init__(shell)
        self._rng = np.random.RandomState(
            self.seed or None)

Appending to random_magics.py
```

4. We create a %random line magic that displays a random number:

```
>>> %%writefile random_magics.py -a

    @line_magic
    def random(self, line):
        return self.text.format(
            n=self._rng.randint(self.max))

Appending to random_magics.py
```

5. Finally, we register that magic when the extension is loaded:

```
>>> %%writefile random_magics.py -a

    def load_ipython_extension(ipython):
        ipython.register_magics(RandomMagics)

Appending to random_magics.py
```

6. Let's test our extension in the Notebook:

```
>>> %load_ext random_magics
>>> %random
'430'
>>> %random
'305'
```

7. Our magic command has a few configurable parameters. These variables are meant to be configured by the user in the IPython configuration file or in the console when starting IPython. To configure these variables in the Terminal, we can type the following command in a system shell:

```
ipython --RandomMagics.text='Your number is {n}.' \
--RandomMagics.max=10 \
--RandomMagics.seed=1
```

In that session, %random displays a string like 'Your number is 5.'

8. To configure the variables in the IPython configuration file, we open the `~/.ipython/profile_default/ipython_config.py` file and add the following line:

```
c.RandomMagics.text = 'random {n}'
```

After launching IPython, `%random` prints a string like `random 652`.

How it works...

IPython's configuration system defines several concepts:

- ▶ An **user profile** is a set of parameters, logs, and command history, which are specific to a user. A user can have different profiles when working on different projects. An `xxx` profile is stored in `~/.ipython/profile_xxx`, where `~` is the user's home directory.
 - On Linux, the path should be `/home/yourname/.ipython/profile_xxx`
 - On macOS, the path should be `/Users/yourname/.ipython/profile_xxx`
 - On Windows, the path should be `C:\Users\YourName\.ipython\profile_xxx`
- ▶ A **configuration object**, or `Config`, is a special Python dictionary that contains key-value pairs. The `Config` class derives from Python's `dict`.
- ▶ The `HasTraits` class is a class that can have special trait attributes. **Traits** are sophisticated Python attributes that have a specific type and a default value. Additionally, when a trait's value changes, a callback function is automatically and transparently called. This mechanism allows a class to be notified whenever a trait attribute is changed.
- ▶ A `Configurable` class is the base class of all classes that want to benefit from the configuration system. A `Configurable` class can have configurable attributes. These attributes have default values specified directly in the class definition. The main feature of `Configurable` classes is that the default values of the traits can be overridden by configuration files on a class-by-class basis. Then, instances of the `Configurable` classes can change these values at leisure.
- ▶ A **configuration file** is a Python or JSON file that contains the parameters of the `Configurable` classes.

The `Configurable` classes and configuration files support an inheritance model. A `Configurable` class can derive from another `Configurable` class and override its parameters. Similarly, a configuration file can be included in another file.

Configurables

Here is a simple example of a Configurable class:

```
from traitlets.config import Configurable
from traitlets import Float

class MyConfigurable(Configurable):
    myvariable = Float(100.0, config=True)
```

By default, an instance of the `MyConfigurable` class will have its `myvariable` attribute equal to `100.0`. Now, let's assume that our IPython configuration file contains the following lines:

```
c = get_config()
c.MyConfigurable.myvariable = 123.
```

Then, the `myvariable` attribute will default to `123`. Instances are free to change this default value after they are instantiated.

The `get_config()` function is a special function that is available in any configuration file.

Additionally, Configurable parameters can be specified in the command-line interface, as we saw in this recipe.

This configuration system is used by all IPython applications (notably console, Qt console, and notebook). These applications have many configurable attributes. You will find the list of these attributes in your profile's configuration files.

Magics

The `Magics` class derives from `Configurable` and can contain configurable attributes. Moreover, magic commands can be defined by methods decorated by `@line_magic` or `@cell_magic`. The advantage of defining class magics instead of function magics (as in the preceding recipe) is that we can keep a state between multiple magic calls (because we are using a class instead of a function).

There's more...

Here are a few references:

- ▶ Configuring and customizing IPython, at <http://ipython.readthedocs.io/en/stable/config/>
- ▶ Defining custom magics, available at <http://ipython.readthedocs.io/en/stable/config/custommagics.html>
- ▶ Detailed overview of the configuration system, at <https://traitlets.readthedocs.io/en/stable/config.html>

See also

- ▶ The *Creating an IPython extension with custom magic commands* recipe

Creating a simple kernel for Jupyter

The architecture of Jupyter is language independent. The decoupling between the client and kernel makes it possible to write kernels in any language. The client communicates with the kernel via socket-based messaging protocols.

However, the messaging protocols are complex. Writing a new kernel from scratch is not straightforward. Fortunately, Jupyter brings a lightweight interface for kernel languages that can be wrapped in Python.

This interface can also be used to create an entirely customized experience in the Jupyter Notebook (or another client application, such as the console). Normally, Python code has to be written in every code cell; however, we can write a kernel for any domain-specific language. We just have to write a Python function that accepts a code string as input (the contents of the code cell), and sends text or rich data as output. We can also easily implement code completion and code inspection.

We can imagine many interesting interactive applications that go far beyond the original use cases of Jupyter. These applications might be particularly useful to nonprogrammer end users such as high school students.

In this recipe, we will create a simple graphing calculator. The calculator is transparently backed by NumPy and Matplotlib. We just have to write functions as $y = f(x)$ in a code cell to get a graph of these functions.

How to do it...

1. First, we create a `plotkernel.py` file. This file will contain the implementation of our custom kernel. Let's import a few modules:

```
>>> %%writefile plotkernel.py
```

```
from ipykernel.kernelbase import Kernel
import numpy as np
import matplotlib.pyplot as plt
from io import BytesIO
import urllib, base64
Writing plotkernel.py
```

2. We write a function that returns a base64-encoded PNG representation of a Matplotlib figure:

```
>>> %%writefile plotkernel.py -a

def _to_png(fig):
    """Return a base64-encoded PNG from a
    matplotlib figure."""
    imgdata = BytesIO()
    fig.savefig(imgdata, format='png')
    imgdata.seek(0)
    return urllib.parse.quote(
        base64.b64encode(imgdata.getvalue())))

Appending to plotkernel.py
```

3. Now, we write a function that parses a code string, which has the form $y=f(x)$, and returns a NumPy function. Here, f is an arbitrary Python expression that can use NumPy functions:

```
>>> %%writefile plotkernel.py -a

_numpy_namespace = {n: getattr(np, n)
                     for n in dir(np)}
def _parse_function(code):
    """Return a NumPy function from a
    string 'y=f(x)'."""
    return lambda x: eval(code.split('=')[1].strip(),
                          _numpy_namespace, {'x': x})

Appending to plotkernel.py
```

4. For our new wrapper kernel, we create a class that derives from Kernel. There are a few metadata fields we need to provide:

```
>>> %%writefile plotkernel.py -a

class PlotKernel(Kernel):
    implementation = 'Plot'
    implementation_version = '1.0'
    language = 'python' # will be used for
                       # syntax highlighting
    language_version = '3.6'
    language_info = {'name': 'plotter',
                    'mimetype': 'text/plain',
                    'extension': '.py'}
    banner = "Simple plotting"

Appending to plotkernel.py
```

5. In this class, we implement a `do_execute()` method that takes code as input and sends responses to the client:

```
>>> %%writefile plotkernel.py -a

def do_execute(self, code, silent,
               store_history=True,
               user_expressions=None,
               allow_stdin=False):

    # We create the plot with matplotlib.
    fig, ax = plt.subplots(1, 1, figsize=(6,4),
                          dpi=100)
    x = np.linspace(-5., 5., 200)
    functions = code.split('\n')
    for fun in functions:
        f = _parse_function(fun)
        y = f(x)
        ax.plot(x, y)
    ax.set_xlim(-5, 5)

    # We create a PNG out of this plot.
    png = _to_png(fig)

    if not silent:
        # We send the standard output to the
        # client.
        self.send_response(
            self.iopub_socket,
            'stream', {
                'name': 'stdout',
                'data': ('Plotting {n} '
                         'function(s)'). \
                         format(n=len(functions)))}

    # We prepare the response with our rich
    # data (the plot).
    content = {
        'source': 'kernel',

        # This dictionary may contain
        # different MIME representations of
        # the output.
        'data': {
            'image/png': png
```

```
        },  
  
        # We can specify the image size  
        # in the metadata field.  
        'metadata' : {  
            'image/png' : {  
                'width': 600,  
                'height': 400  
            }  
        }  
    }  
  
    # We send the display_data message with  
    # the contents.  
    self.send_response(self.iopub_socket,  
                      'display_data', content)  
  
    # We return the execution results.  
    return {'status': 'ok',  
            'execution_count':  
                self.execution_count,  
            'payload': [],  
            'user_expressions': {}}  
        }  
Appending to plotkernel.py
```

6. Finally, we add the following lines at the end of the file:

```
>>> %%writefile plotkernel.py -a  
  
if __name__ == '__main__':  
    from ipykernel.kernelapp import IPKernelApp  
    IPKernelApp.launch_instance(  
        kernel_class=PlotKernel)  
Appending to plotkernel.py
```

7. Our kernel is ready! The next step is to indicate to Jupyter that this new kernel is available. To do this, we need to create a kernel spec `kernel.json` file in a subdirectory as follows:

```
>>> %mkdir -p plotter/  
>>> %%writefile plotter/kernel.json  
{  
    "argv": ["python", "-m",  
             "plotkernel", "-f",  
             "{connection_file}"],
```

```
"display_name": "Plotter",
"name": "Plotter",
"language": "python"
}
Writing plotter/kernel.json
```

8. We install the kernel:

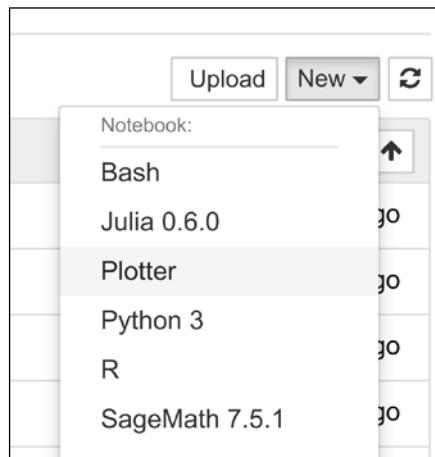
```
>>> !jupyter kernelspec install --user plotter
[InstallKernelSpec] Installed kernelspec plotter in
~/.local/share/jupyter/kernels/plotter
```

9. The **plotter** kernel now appears in the list of kernels:

```
>>> !jupyter kernelspec list
Available kernels:
bash            ~/.local/share/jupyter/kernels/bash
ir              ~/.local/share/jupyter/kernels/ir
plotter         ~/.local/share/jupyter/kernels/plotter
sagemath        ~/.local/share/jupyter/kernels/sagemath
...
...
```

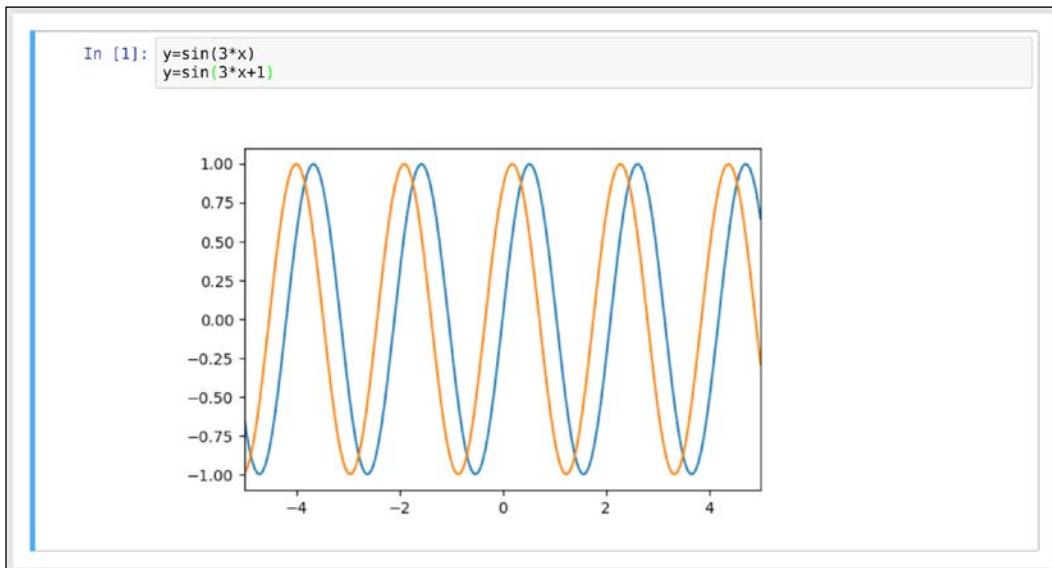
The `plotkernel.py` file needs to be importable by Python. For example, we could simply put it in the current directory.

10. Now, if we refresh the main Jupyter Notebook page (or after a restart of the Jupyter Notebook server if needed), we see that our **Plotter** kernel appears in the list of kernels:



Kernel list

11. Let's create a new notebook with the Plotter kernel. There, we can simply write mathematical equations under the form $y=f(x)$. The corresponding graph appears in the output area. Here is an example:



Wrapper kernel

How it works...

The kernel and client live in different processes. They communicate via messaging protocols implemented on top of network sockets. Currently, these messages are encoded in JSON, a structured, text-based document format.

Our kernel receives code from the client (the notebook, for example). The `do_execute()` function is called whenever the user sends a cell's code.

The kernel can send messages back to the client with the `self.send_response()` method:

- ▶ The first argument is the socket—here, the IOPub socket
- ▶ The second argument is the message type—here, `stream` to send back standard output or a standard error, or `display_data` to send back rich data
- ▶ The third argument is the contents of the message, represented as a Python dictionary

The data can contain multiple MIME representations: text, HTML, SVG, images, and others. It is up to the client to handle these data types. In particular, the Notebook client knows how to represent all these types in the browser.

The function returns execution results in a dictionary.

In this toy example, we always return an `ok` status. In production code, it would be a good idea to detect errors (syntax errors in the function definitions, for example) and return an error status instead.

All messaging protocol details can be found in the following section.

There's more...

Wrapper kernels can implement optional methods, notably for code completion and code inspection. For example, to implement code completion, we need to write the following method:

```
def do_complete(self, code, cursor_pos):
    return {'status': 'ok',
            'cursor_start': ...,
            'cursor_end': ...,
            'matches': [...]}
```

This method is called whenever the user requests code completion when the cursor is at a given `cursor_pos` location in the code cell. In the method's response, the `cursor_start` and `cursor_end` fields represent the interval that code completion should overwrite in the output. The `matches` field contains the list of suggestions.

Here are a few references:

- ▶ Wrapper kernel example https://github.com/jupyter/echo_kernel
- ▶ Wrapper kernels, available at <http://jupyter-client.readthedocs.io/en/latest/wrapperkernels.html>
- ▶ Messaging protocol in Jupyter, at <https://jupyter-client.readthedocs.io/en/latest/messaging.html#execution-results>
- ▶ Making kernels for Jupyter, at <http://jupyter-client.readthedocs.io/en/latest/kernels.html>
- ▶ Using C++ in Jupyter, at <https://blog.jupyter.org/interactive-workflows-for-c-with-jupyter-fe9b54227d92>

2

Best Practices in Interactive Computing

In this chapter, we will cover the following topics:

- ▶ Learning the basics of the Unix shell
- ▶ Using the latest features of Python 3
- ▶ Learning the basics of the distributed version control system Git
- ▶ A typical workflow with Git branching
- ▶ Efficient interactive computing workflows with IPython
- ▶ Ten tips for conducting reproducible interactive computing experiments
- ▶ Writing high-quality Python code
- ▶ Writing unit tests with pytest
- ▶ Debugging code with IPython

Introduction

This is a special chapter about good practices in interactive computing. It describes how to work efficiently and properly with the tools this book is about. We will introduce common tools such as the Unix shell, the latest features of Python 3, and Git, before tackling reproducible computing experiments (notably with the Jupyter Notebook).

We will also cover more general topics in software development, such as code quality, debugging, and testing. Attention to these subjects can greatly improve the quality of our end products (for example, software, research, and publications). We will only scratch the surface here, but you will find many references to learn more about these important topics.

Learning the basics of the Unix shell

Learning how to interact with the operating system using a command-line interface (or Terminal) is a required skill in interactive computing and data analysis. We will use a command-line interface in most of the recipes in this book. IPython and the Jupyter Notebook are typically launched from a Terminal. Installing Python packages is typically done from a Terminal.

In this recipe, we will show the very basics of the Unix shell, which is natively available in Linux distributions (such as Debian, Ubuntu, and so on) and macOS. On Windows 10, one can install the **Windows Subsystem for Linux**, a command-line interface to a Unix subsystem integrated with the Windows operating system (see <https://docs.microsoft.com/windows/wsl/about>).

Getting ready

Here are the instructions to open a Unix shell on macOS, Linux, and Windows. Bash is the most common Unix shell and this is what we will use in this recipe.

On macOS, bring up the **Spotlight Search**, type `terminal`, and press *Enter*.

On Windows, follow the instructions at <https://docs.microsoft.com/en-us/windows/wsl/install-win10>. Then, open the Windows menu, type `bash`, and press *Enter*.

On Ubuntu, open the Dash by clicking on the top-left icon on the desktop, type `terminal`, and open the Terminal application.

If you want to run this notebook in Jupyter, you need to install `bash_kernel`, available at https://github.com/takluyver/bash_kernel. Open a Terminal and type `pip install bash_kernel && python -m bash_kernel.install`.

This will install a bash kernel in Jupyter, and it will allow you to run this recipe's code directly in the Notebook.

How to do it...

The Unix shell comes with hundreds of commands. We will see the most common ones in this recipe:

1. The Terminal lets us write text commands with the keyboard. We execute them by pressing *Enter*, and the output is displayed below the command. The **working directory** is the directory of our filesystem that is currently *active* in the Terminal. We can get the absolute path of the working directory as follows:

```
$ pwd  
~/git/cookbook-2nd/chapter02_best_practices
```



The dollar \$ sign must not be typed: it is typically used by the shell to indicate where the user can start typing. The information written before it may show the username, the computer name, and part of the working directory. Here, only the three characters `pwd` should be typed before pressing *Enter*.

2. We can list all files and subdirectories in the working directory as follows:

```
$ ls
00_intro.md  03_git.md          07_high_quality.md
01_shell.md   04_git_advanced.md 08_test.md
02_py3        05_workflows.md    09_debugging.md
02_py3.md     06_tips.md         images
$ ls -l
total 100
-rw-rw-r-- 1 owner    769 Dec 12 10:23 00_intro.md
-rw-rw-r-- 1 owner   2473 Dec 12 14:21 01_shell.md
...
-rw-rw-r-- 1 owner  9390 Dec 12 11:46 08_test.md
-rw-rw-r-- 1 owner  5032 Dec 12 10:23 09_debugging.md
drwxrwxr-x 2 owner  4096 Aug  1 16:49 images
```

The `-l` option displays the directory contents as a detailed list, showing the permissions and owner of the files, the file sizes, and the last modified dates. Most shell commands come with many options that alter their behavior and that can be arbitrarily combined.

3. We use the `cd` command to navigate between subdirectories. The current directory is named `.` (single dot), and the parent directory is named `..` (double dot):

```
$ cd images
$ pwd
~/git/cookbook-2nd/chapter02_best_practices/images
$ ls
folder.png  github_new.png
$ cd ..
$ pwd
~/git/cookbook-2nd/chapter02_best_practices
```

4. Paths can be specified as relative (depending on a reference directory, generally the working directory) or absolute. The **home directory**, specified as `~`, contains the user's personal files. Configuration files are often stored in a directory such as `~/program_name`. For example, `~/ipython` contains configuration files of IPython:

```
$ ls -la ~/ipython
total 20
drwxr-xr-x  5 cyrille 4096 Nov 14 16:16 .
drwxr-xr-x 93 cyrille 4096 Dec 12 10:50 ..
drwxr-xr-x  2 cyrille 4096 Nov 14 16:16 extensions
drwxr-xr-x  2 cyrille 4096 Nov 14 16:16 nbextensions
drwxr-xr-x  7 cyrille 4096 Dec 12 14:18 profile_default
```

 In most terminals, we can use the arrow keys on the keyboard to navigate in the history of past commands. Also, the `Tab` key enables tab completion, which automatically completes the first characters of a command or a file. For example, typing `ls -la ~/ipy` and pressing `Tab` would automatically complete to `ls -la ~/ipython`, or it would present the list of possible options if there are several files or directories that begin with `~/ipy`.

5. We can create, move, rename, copy, and delete files and directories from the Terminal:

```
$ # We create an empty directory:
$ mkdir md_files
$ # We copy all Markdown files into the new directory:
$ cp *.md md_files
$ # We rename the directory:
$ mv md_files markdown_files
$ ls markdown_files
00_intro.md      05_workflows.md
01_shell.md      06_tips.md
02_py3.md        07_high_quality.md
03_git.md        08_test.md
04_git_advanced.md 09_debugging.md
$ rmdir markdown_files
rmdir: failed to remove 'markdown_files':
    Directory not empty
$ rm markdown_files/*
$ rmdir markdown_files
```



The `rm` command lets us delete files and directories. The `rm -rf` path command deletes the given path recursively, even if subdirectories are not empty. It is an extremely dangerous command as it cannot be undone: the files are immediately and permanently deleted, they do not go into a trash directory first. See <https://github.com/sindresorhus/guides/blob/master/how-not-to-rm-yourself.md> for more details.

6. There are several useful commands to deal with text files:

```
$ # Show the first three lines of a text file:  
$ head -n 3 01_shell.md  
# Learning the basics of the Unix shell  
Learning how to interact with the operating system (...)  
$ # Show the last line of a text file:  
$ tail -n 1 00_intro.md  
We will also cover more general topics (...)  
$ # We display some text:  
$ echo "Hello world!"  
Hello world!  
$ # We redirect the output of a command to  
$ # a text file with '>':  
$ echo "Hello world!" > myfile.txt  
$ # We display the entire contents of the file:  
$ cat myfile.txt  
Hello world!
```

Several command-line text editors are available, such as pico, nano, or vi. Learning these text editors requires time and effort, especially vi.

7. The grep command lets us search substrings in text. In the following example, we find all instances of Unix followed by a word (using regular expressions):

```
$ grep -Eo "Unix \w+" 01_shell.md  
Unix shell  
Unix shell  
Unix subsystem  
Unix shell  
(...)  
Unix shell  
Unix shell
```

8. A major strength of the Unix shell is that commands can be combined with **pipes**: the output of one command can be directly transferred to the input of another command:

```
$ echo "This is a Unix shell" | grep -Eo "Unix \w+"
Unix shell
```

There's more...

We only scratched the surface of the Unix shell in this recipe. There are many other commands that can be combined in an infinite number of ways. Many repetitive tasks that would take hours of manual work can be done in a few minutes by writing the appropriate commands. Mastering the Unix shell may take a lot of effort, but it leads to dramatic time gains in the long term.

Here are a few references:

- ▶ *Linux Tutorial* at <https://ryanstutorials.net/linuxtutorial/>
- ▶ *Bash commands* at <https://ss64.com/bash/>
- ▶ *Learn Bash in Y minutes*, at <https://learnxinyminutes.com/docs/bash/>
- ▶ *Learn the shell interactively*, at <http://www.learnshell.org/>
- ▶ *The fish shell*, at <https://fishshell.com/>
- ▶ *xonsh*, a Python-powered shell, at <http://xon.sh/>
- ▶ *Windows Subsystem for Linux*, at <https://docs.microsoft.com/windows/wsl/about>

See also

- ▶ The *Ten tips for conducting reproducible interactive computing experiments* recipe

Using the latest features of Python 3

The latest version of the Python 2.x branch, Python 2.7, was released in 2010. It will reach its end of life in 2020. On the other hand, the first version of the Python 3.x branch, Python 3.0, was released in 2008. The decade-long transition period between Python 2 and Python 3, which are slightly incompatible, has been somewhat chaotic.

Choosing between Python 2 (also known as *Legacy Python*) and Python 3 used to be tricky since many Python users had not transitioned to Python 3 yet, and many libraries were only compatible with Python 2. Those times are gone and it is now safe to stick with Python 3 in virtually all cases. The only exceptions are when you have to support old unmaintained libraries, or when your users cannot transition to Python 3 for whatever reason.

In addition to fixing the bugs and annoyances of Python 2 (for example, related to Unicode support), Python 3 brings many interesting features in terms of syntax, capabilities of the language, and new built-in libraries.



We use the latest stable version of Python in this recipe, which is Python 3.6.

How to do it...

1. In Python 3, `print()` is a function, whereas it was a statement in Python 2 (it was a historical oddity). This function may accept multiple arguments as well as a few options. Let's create a list:

```
>>> my_list = list(range(10))
```

We can print the `my_list` object:

```
>>> print(my_list)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

But we can also print the ten numbers in the list:

```
>>> print(*my_list)
0 1 2 3 4 5 6 7 8 9
```

Finally we can customize the separator and the end of the string to print:

```
>>> print(*my_list, sep=" + ", end=" = %d" % sum(my_list))
0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45
```

2. Python 3 supports more advanced variable unpacking features:

```
>>> first, second, *rest, last = my_list
>>> print(first, second, last)
0 1 9
>>> rest
[2, 3, 4, 5, 6, 7, 8]
```

3. In Python 3, variable names can contain Unicode characters. This technique may be useful when writing mathematical code. To type mathematical symbols in the Jupyter Notebook, write LaTeX code and press *Tab*. For example, to create a variable π , type `\pi` and then *Tab*:

```
>>> from math import pi, cos
      α = 2
      π = pi
      cos(α * π)
1.000
```

4. Python 3.6 brings new string literals called **f-strings**. They offer a convenient syntax to define strings depending on existing variables:

```
>>> a, b = 1, 2
      f"The sum of {a} and {b} is {a + b}"
'The sum of 1 and 2 is 3'
```

5. We can add custom annotations to function arguments and output. These function annotations convey no semantics, but they can be used in the code as we like. Here is an example coming from <https://stackoverflow.com/a/7811344/1595060>:

```
>>> def kinetic_energy(mass: 'kg',
      velocity: 'm/s') -> 'J':
      """The annotations serve here as documentation."""
      return .5 * mass * velocity ** 2
```

These annotations are stored in the `__annotations__` attribute of the function, and they can be used as follows:

```
>>> annotations = kinetic_energy.__annotations__
      print(*{f"{key} is in {value}"}
            for key, value in annotations.items(),
            sep=", ")
```

mass is in kg, velocity is in m/s, return is in J

The `typing` module, which has been included in Python 3.5 on a provisional basis, implements several annotations that can be used to specify typing information in functions.

6. Python 3.5 brings a new operator `@` for matrix multiplication. It is supported by NumPy 1.10 and later:

```
>>> import numpy as np
      M = np.array([[0, 1], [1, 0]])
```

The `*` operator is the element-wise multiplication:

```
>>> M * M
array([[0, 1],
       [1, 0]])
```

Previously, matrix multiplication could be performed with `np.dot()`. The new syntax is clearer:

```
>>> M @ M
array([[1, 0],
       [0, 1]])
```

7. Python 3.3 brings the new `yield from` syntax that allows you, among other things, to compose multiple generators. For example, the two following functions are equivalent:

```
>>> def gen1():
        for i in range(5):
            for j in range(i):
                yield j
>>> def gen2():
        for i in range(5):
            yield from range(i)
>>> list(gen1())
[0, 0, 1, 0, 1, 2, 0, 1, 2, 3]
>>> list(gen2())
[0, 0, 1, 0, 1, 2, 0, 1, 2, 3]
```

8. The `functools` library provides a `@lru_cache` decorator to implement a simple in-memory caching system for Python functions:

```
>>> import time

def f1(x):
    time.sleep(1)
    return x
>>> %timeit -n1 -r1 f1(0)
1 s ± 0 ns per loop (mean ± std. dev. of 1 run,
 1 loop each)
>>> %timeit -n1 -r1 f1(0)
1 s ± 0 ns per loop (mean ± std. dev. of 1 run,
 1 loop each)
```

Here, the two successive identical calls to `f1(0)` take one second. Now, let's define a cached version of this function:

```
>>> from functools import lru_cache
```

```
@lru_cache(maxsize=32) # keep the latest 32 calls
def f2(x):
    time.sleep(1)
    return x
>>> %timeit -n1 -r1 f2(0)
1 s ± 0 ns per loop (mean ± std. dev. of 1 run,
1 loop each)
>>> %timeit -n1 -r1 f2(0)
6.14 µs ± 0 ns per loop (mean ± std. dev. of 1 run,
1 loop each)
```

The first call takes one second, whereas the next one returns immediately. In the second case, the function is not called but the output corresponding to the argument of 0 was cached and returned.

9. The new `pathlib` module offers filesystem facilities that are more convenient to use than the Python 2 `os.path` methods. The main class is `Path`:

```
>>> from pathlib import Path
```

We instantiate a `Path` object representing the current directory:

```
>>> p = Path('..')
```

Let's list all Markdown files in the directory:

```
>>> sorted(p.glob('*md'))
[PosixPath('00_intro.md'),
 PosixPath('01_py3.md'),
 PosixPath('02_workflows.md'),
 PosixPath('03_git.md'),
 PosixPath('04_git_advanced.md'),
 PosixPath('05_tips.md'),
 PosixPath('06_high_quality.md'),
 PosixPath('07_test.md'),
 PosixPath('08_debugging.md')]
```

We can easily get the contents of a text file:

```
>>> _[0].read_text()
'# Introduction\n\n...\\n'
```

Let's obtain the list of subdirectories:

```
>>> [d for d in p.iterdir() if d.is_dir()]
[PosixPath('images'),
 PosixPath('.ipynb_checkpoints'),
 PosixPath('__pycache__'),
```

Finally, we list all files in the `images` subfolder (note the slash / operator on `Path` instances):

```
>>> list((p / 'images').iterdir())
[PosixPath('images/github_new.png'),
 PosixPath('images/folder.png')]
```

10. Python 3.4 provides a new `statistics` module which implements basic statistical routines. This module may be useful when depending on NumPy or SciPy is not desirable. Let's import the module:

```
>>> import random as r
      import statistics as st
```

We create a list of normally-distributed random variables:

```
>>> my_list = [r.normalvariate(0, 1)
              for _ in range(100000)]
```

We compute the mean, median, and standard deviation:

```
>>> print(st.mean(my_list),
          st.median(my_list),
          st.stdev(my_list),
          )
0.00073 -0.00052 1.00050
```

There's more...

Other interesting features of Python 3 include coroutines with the `asyncio` module and asynchronous operations with the new `await` and `async` keywords.

Here are a few references:

- ▶ *What's New In Python 3.6?* at <https://docs.python.org/3/whatsnew/3.6.html>
- ▶ *f-strings*, at https://docs.python.org/3/reference/lexical_analysis.html#f-strings
- ▶ *The yield from syntax*, at <https://docs.python.org/3/whatsnew/3.3.html#pep-380>
- ▶ *functools*, at <https://docs.python.org/3/library/functools.html>
- ▶ *pathlib*, at <https://docs.python.org/3/library/pathlib.html>
- ▶ *statistics*, at <https://docs.python.org/3/library/statistics.html>
- ▶ *10 awesome features of Python that you can't use because you refuse to upgrade to Python 3*, at <http://www.asmeurer.com/python3-presentation/slides.html>

- ▶ *Python 3 for Scientists*, at <http://python-3-for-scientists.readthedocs.io/en/latest/>
- ▶ *Cool New Features in Python 3.6*, at <https://www.youtube.com/watch?v=k1KdMxjDaa0>
- ▶ *Python Cookbook, 3rd Edition*, Brian Jones and David Beazley, O'Reilly Media, at <http://shop.oreilly.com/product/0636920027072.do>
- ▶ Find the best Python books, at <http://pythonbooks.org/>
- ▶ *Buggy Python Code: The 10 Most Common Mistakes That Python Developers Make*, at <https://www.toptal.com/python/top-10-mistakes-that-python-programmers-make>
- ▶ Python 3 statement, to promote the deprecation of Python 2 support by 2020, at <http://www.python3statement.org>

Learning the basics of the distributed version control system Git

Using a **version control system** is an absolute requirement in programming and research. This is the tool that makes it just about impossible to lose one's work. In this recipe, we will cover the basics of Git.

Getting ready

Notable distributed version control systems include **Git**, **Mercurial**, and **Bazaar**, among others. In this chapter, we will use the popular Git system. You can download the Git program and Git GUI clients from <http://git-scm.com>.



Distributed systems tend to be more popular than centralized systems such as SVN or CVS. Distributed systems allow local (offline) changes and offer more flexible collaboration systems.

An online provider allows you to host your code in the cloud. You can use it as a backup of your work and as a platform to share your code with your colleagues. These services include **GitHub** (<https://github.com>), **GitLab** (<https://gitlab.com>), and **Bitbucket** (<https://bitbucket.org>). All of these websites offer free and paid plans with unlimited public and/or private repositories.

GitHub offers desktop applications for Windows and macOS at <https://desktop.github.com/>.

This book's code is stored on GitHub. Most Python libraries are also developed on GitHub.

How to do it...

1. The very first thing to do when starting a new project or computing experiment is create a new folder locally:

```
$ mkdir myproject  
$ cd myproject
```

2. We initialize a Git repository:

```
$ git init  
Initialized empty Git repository in  
~/git/cookbook-2nd/chapter02/myproject/.git/  
$ pwd  
~/git/cookbook-2nd/chapter02/myproject  
$ ls -a  
. .. .git
```

Git created a `.git` subdirectory that contains all the parameters and history of the repository.

3. Let's set our name and email address globally:

```
$ git config --global user.name "My Name"  
$ git config --global user.email "me@home.com"
```

4. We create a new file, and we tell Git to track it:

```
$ echo "Hello world" > file.txt  
$ git add file.txt
```

5. Let's create our first commit:

```
$ git commit -m "Initial commit"  
[master (root-commit) 02971c0] Initial commit  
1 file changed, 1 insertion(+)  
create mode 100644 file.txt
```

6. We can check the list of commits:

```
$ git log  
commit 02971c0e1176cd26ec33900e359b192a27df2821  
Author: My Name <me@home.com>  
Date:   Tue Dec 12 10:50:37 2017 +0100
```

Initial commit

7. Next, we edit the file by appending an exclamation mark:

```
$ echo "Hello world!" > file.txt  
$ cat file.txt  
Hello world!
```

8. We can see the differences between the current state of our repository, and the state in the last commit:

```
$ git diff  
diff --git a/file.txt b/file.txt  
index 802992c..cd08755 100644  
--- a/file.txt  
+++ b/file.txt  
@@ -1 +1 @@  
-Hello world  
+Hello world!
```

The output of `git diff` shows that the contents of `file.txt` were changed from `Hello world` to `Hello world!`. Git compares the states of all tracked files and computes the differences between the files.

9. We can also get a summary of the changes as follows:

```
$ git status  
On branch master  
Changes not staged for commit:  
  (use "git add <file>..." to update what will  
   be committed)  
  
      modified:   file.txt  
  
no changes added to commit (use "git add")  
$ git diff --stat  
file.txt | 2 +-  
1 file changed, 1 insertion(+), 1 deletion(-)
```

The `git status` command gives a summary of all changes since the last commit. The `git diff --stat` command shows, for each modified text file, the number of changed lines.

- Finally, we commit our change with a shortcut that automatically adds all changes in the tracked files (-a option):

```
$ git commit -am "Add exclamation mark to file.txt"
[master 045df6a] Add exclamation mark to file.txt
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git log
commit 045df6a6f8a62b19f45025d15168d6d7382a8429
Author: My Name <me@home.com>
Date:   Tue Dec 12 10:59:39 2017 +0100

Add exclamation mark to file.txt

commit 02971c0e1176cd26ec33900e359b192a27df2821
Author: My Name <me@home.com>
Date:   Tue Dec 12 10:50:37 2017 +0100

Initial commit
```

How it works...

When you start a new project or a new computing experiment, create a new folder on your computer. You will eventually add code, text files, datasets, and other resources in this folder. The distributed version control system keeps track of the changes you make to your files as your project evolves. It is more than a simple backup, as every change you make on any file can be saved along with the corresponding timestamp. You can even revert to a previous state at any time; never be afraid of breaking your code anymore!



Git works best with text files. It can handle binary files but with limitations. It is better to use a separate system such as **Git Large File Storage**, or **Git LFS** (see <https://git-lfs.github.com/>).

Specifically, you can take a snapshot of your project at any time by doing a **commit**. The snapshot includes all staged (or tracked) files. You are in total control of which files and changes will be tracked. With Git, you specify a file as staged for your next commit with `git add`, before committing your changes with `git commit`. The `git commit -a` command allows you to commit all changes in the files that are already being tracked.

When committing, you should provide a clear and short message describing the changes you made. This makes the repository's history considerably more informative than just writing *work in progress*. If the commit message is long, write a short title (less than 50 characters), insert two line breaks, and write a longer description.

How often should you commit?

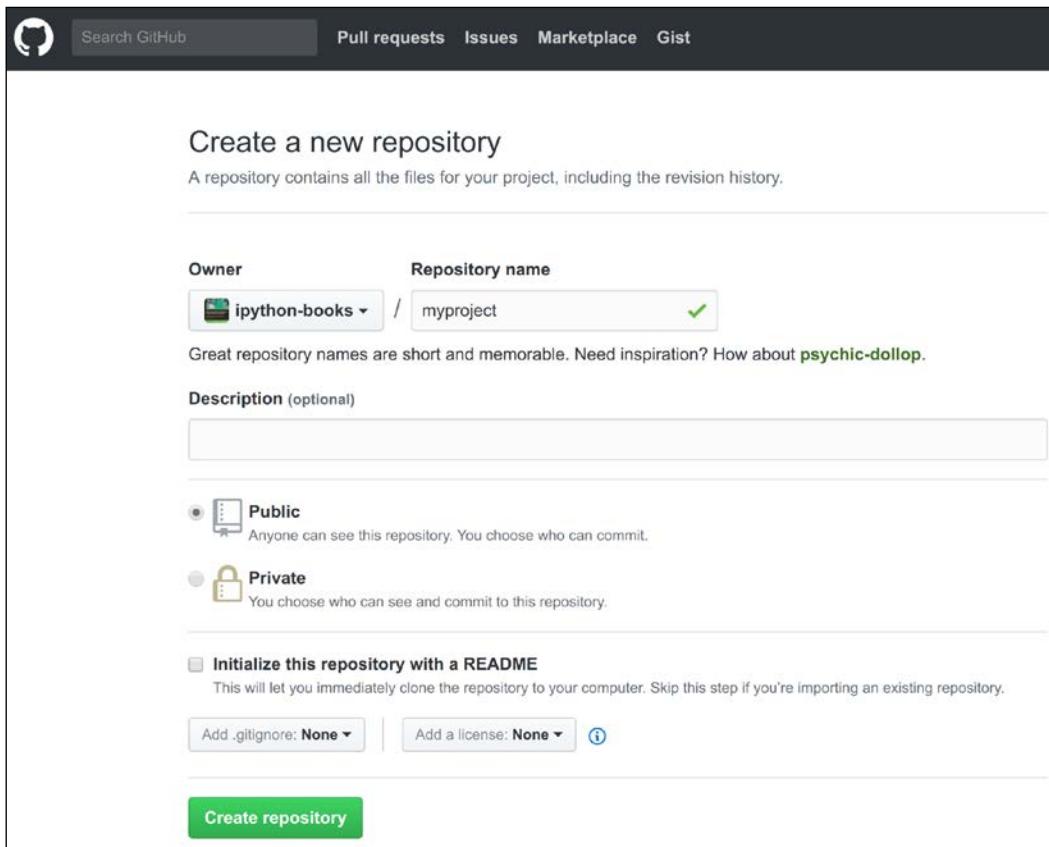


The answer is very often. Git only takes responsibility for your work when you commit changes. What happens between two commits may be lost, so it's better to commit very regularly. Besides, commits are quick and cheap as they are local; that is, they do not involve any remote communication with an external server.

Git is a *distributed* version control system; your local repository does not need to synchronize with an external server. However, you should synchronize if you need to work on several computers, or if you prefer to have a remote backup. Synchronization with a remote repository can be done with `git push` (send your local commits on the remote server), `git fetch` (download remote branches and objects), and `git pull` (synchronize the remote changes on your local repository), after you've set up remotes.

There's more...

We can also create a new repository on an online Git provider such as GitHub:



New project on GitHub

On the main web page of the newly created project, click on the **Clone or download** button to get the repository URL and type in a Terminal:

```
$ git clone https://github.com/mylogin/myproject.git
```

If the local repository already exists, do not tick the **Initialize this repository with a README** box on the GitHub page, and add the remote with `git remote add origin https://github.com/yourlogin/myproject.git`. See <https://help.github.com/articles/adding-a-remote/> for more details.

The simplistic workflow shown in this recipe is linear. In practice, though, workflows with Git are typically nonlinear; this is the concept of branching. We will describe this idea in the next recipe, *A typical workflow with Git branching*.

Here are some references on Git:

- ▶ Hands-on tutorial, available at <https://try.github.io>
- ▶ Git, a simple guide by Roger Dudler, available at <http://rogerdudler.github.io/git-guide/>
- ▶ Git Immersion, a guided tour, at <http://gitimmersion.com>
- ▶ Atlassian Git tutorial, available at <http://www.atlassian.com/git>
- ▶ Online Git course, available at <http://www.codeschool.com/courses/try-git>
- ▶ Git tutorial by Lars Vogel, available at <http://www.vogella.com/tutorials/Git/article.html>
- ▶ GitHub and Git tutorial, available at <http://git-lectures.github.io>
- ▶ *Intro to Git for scientists*, available at http://karthik.github.io/git_intro/
- ▶ GitHub help, available at <https://help.github.com>

See also

- ▶ The *A typical workflow with Git branching* recipe

A typical workflow with Git branching

A distributed version control system such as Git is designed for the complex and nonlinear workflows that are typical in interactive computing and exploratory research. A central concept is **branching**, which we will discuss in this recipe.

Getting ready

You need to work in a local Git repository for this recipe (see the previous recipe, *Learning the basics of the distributed version control system Git*).

How to do it...

1. We go to the `myproject` repository and we create a new branch named `newidea`:

```
$ pwd  
/home/cyrille/git/cookbook-2nd/chapter02  
$ cd myproject
```

```
$ git branch newidea  
$ git branch  
* master  
  newidea
```

As indicated by the star *, we are still on the master branch.

2. We switch to the newly-created newidea branch:

```
$ git checkout newidea  
Switched to branch 'newidea'  
$ git branch  
  Master  
* newidea
```

3. We make changes to the code, for instance, by creating a new file:

```
$ echo "print('new')" > newfile.py  
$ cat newfile.py  
print('new')
```

4. We add this file to the staging area and we commit our changes:

```
$ git add newfile.py  
$ git commit -m "Testing new idea"  
[newidea 8ebee32] Testing new idea  
  1 file changed, 1 insertion(+)  
  create mode 100644 newfile.py  
$ ls  
file.txt  newfile.py
```

5. If we are happy with the changes, we merge the branch to the master branch (the default):

```
$ git checkout master  
Switched to branch 'master'
```

On the master branch, our new file is not there:

```
$ ls  
file.txt
```

If we merge the new branch into the master branch, the file appears:

```
$ git merge newidea  
Updating 045df6a..8ebee32  
Fast-forward
```

```
newfile.py | 1 +  
1 file changed, 1 insertion(+)  
create mode 100644 newfile.py  
$ ls  
file.txt newfile.py
```

6. If we are not happy with the changes, we can just delete the branch, and the new file will be deleted. Here, since we have just merged the branch, we need to undo the last commit:

```
$ git reset --hard HEAD~1  
HEAD is now at 045df6 Add exclamation mark to file.txt
```

We are still on the master branch, but before we merged the newidea branch:

```
$ git branch  
* master  
  newidea
```

We can delete the branch as follows:

```
$ git branch -D newidea  
Deleted branch newidea (was 8ebee32).
```

The Python file is gone:

```
$ ls  
file.txt
```

7. It may happen that while we are halfway through some work, we need to make some other change in another commit or another branch. We could commit our half-done work, but this is not ideal. A better idea is to stash our working copy in a secure location so that we can recover all of our uncommitted changes later. We save our uncommitted changes with the following command:

```
$ echo "new line" >> file.txt  
$ cat file.txt  
Hello world!  
new line  
$ git stash  
Saved working directory and index state WIP on master:  
045df6a Add exclamation mark to file.txt  
HEAD is now at 045df6 Add exclamation mark to file.txt  
$ cat file.txt  
Hello world!
```

We can do anything we want with the repository: checkout a branch, commit changes, pull or push from a remote repository, and so on. When we want to recover our uncommitted changes, we type the following command:

```
$ git stash pop
On branch master
Changes not staged for commit:

modified:   file.txt

no changes added to commit
(use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (c9071a)
$ cat file.txt
Hello world!
new line
```

We can have several stashed states in the repository. More information about stashing can be found with `git stash --help`.

How it works...

Let's imagine that in order to test a new idea, you need to make non-trivial changes to your code in multiple files. You create a new branch, test your idea, and end up with a modified version of your code. If this idea was a dead end, you switch back to the original branch of your code. However, if you are happy with the changes, you **merge** it into the main branch.

The strength of this workflow is that the main branch can evolve independently from the branch with the new idea. This is particularly useful when multiple collaborators are working on the same repository. However, it is also a good habit to have, even when there is a single contributor.

Merging is not always a trivial operation, as it can involve two divergent branches with potential conflicts. Git tries to resolve conflicts automatically, but it is not always successful. In this case, you need to resolve the conflicts manually.

An alternative to merging is **rebasing**, which is useful when the main branch has changed while you were working on your branch. Rebasing your branch on the main branch allows you to move your branching point to a more recent point. This process may require you to resolve conflicts.

Git branches are lightweight objects. Creating and manipulating them is cheap. They are meant to be used frequently. It is important to perfectly grasp all related notions and Git commands (notably `checkout`, `merge`, and `rebase`). The previous recipe contains many references.

There's more...

Many people have thought about effective workflows. For example, a common but complex workflow, called **git-flow**, is described at <http://nvie.com/posts/a-successful-git-branching-model/>. However, it may be preferable to use a simpler workflow in small and mid-size projects, such as the one described at <http://scottchacon.com/2011/08/31/github-flow.html>. The latter workflow elaborates on the simplistic example shown in this recipe.

A related notion to branching is **forking**. There can be multiple copies of the same repository on different servers. Imagine that you want to contribute to IPython's code stored on GitHub. You probably don't have the permission to modify their repository, but you can make a copy into your personal account—this is called forking. In this copy, you can create a branch and propose a new feature or a bug fix. Then, you can propose the IPython developers to merge your branch into their master branch with a **pull request**. They can review your changes, propose suggestions, and eventually merge your work (or not). GitHub is built around this idea and thereby offers a clean way to collaborate on open source projects.

Performing code reviews before merging pull requests leads to higher code quality in a collaborative project. When at least two people review any piece of code, the probability of merging bad or wrong code is reduced.

There is, of course, much more to say about Git. Version control systems are complex and quite powerful in general, and Git is no exception. Mastering Git requires time and experimentation. The previous recipe contains many excellent references.

Here are a few further references about branches and workflows:

- ▶ Git workflows, available at <http://www.atlassian.com/git/workflows>
- ▶ *Learn Git Branching*, at <http://pcottle.github.io/learnGitBranching/>
- ▶ The Git workflow recommended on the NumPy project (and others), described at http://docs.scipy.org/doc/numpy/dev/gitwash/development_workflow.html
- ▶ A post on the IPython mailing list about an efficient Git workflow, by Fernando Perez, available at <http://mail.scipy.org/pipermail/ipython-dev/2010-October/006746.html>

See also

- ▶ The *Learning the basics of the distributed version control system Git* recipe

Efficient interactive computing workflows with IPython

There are multiple ways of using IPython for interactive computing. Some of them are better in terms of flexibility, modularity, reusability, and reproducibility. We will review and discuss them in this recipe.

Any interactive computing workflow is based on the following cycle:

1. Write some code
2. Execute it
3. Interpret the results
4. Repeat

This fundamental loop (also known as **Read-Eval-Print Loop (REPL)**) is particularly useful when doing exploratory research on data or model simulations, or when building a complex algorithm step by step. A more classical workflow (the edit-compile-run-debug loop) would consist of writing a full-blown program, and then performing a complete analysis. This is generally more tedious. It is more common to build an algorithmic solution iteratively, by doing small-scale experiments and tweaking the parameters, and this is precisely what interactive computing is about.

Integrated Development Environments (IDEs), providing comprehensive facilities for software development (such as a source code editor, compiler, and debugger), are widely used for classical workflows. However, when it comes to interactive computing, alternatives to IDEs exist. We will review them here.

How to do it...

Here are a few possible workflows for interactive computing, by increasing order of complexity. Of course, IPython is at the core of all of these methods.

The IPython terminal

IPython is the de facto standard for interactive computing in Python. The IPython Terminal (the `ipython` command) offers a command-line interface specifically designed for REPLs. It is a much more powerful tool than the native Python interpreter (the `python` command). The IPython Terminal is a convenient tool for quick experiments, simple shell interactions, and to find help. Forgot the input arguments of NumPy's `savetxt` function? Just type in `numpy.savetxt?` in IPython (you will first need to use `import numpy`, of course). Some people even use the IPython Terminal as a (sophisticated) calculator!

However, the Terminal quickly becomes limited when it is used alone. The main issue is that the Terminal is not a code editor, and thus entering more than a few lines of code can be inconvenient. Fortunately, there are various ways of solving this problem, as detailed in the following sections.

IPython and text editor

The simplest solution to the not-a-text-editor problem is to use IPython along with a text editor. The `%run` magic command then becomes the central tool in this workflow:

1. Write some code in your favorite text editor and save it in a `myscript.py` Python script file.
2. In IPython, assuming you are in the right directory, type in `%run myscript.py`.
3. The script is executed. The standard output is displayed in real time in the IPython Terminal along with possible errors. Top-level variables defined in the script are accessible in the IPython Terminal at the end of the script's execution.
4. If code changes are required in the script, repeat the process.

With a good text editor, this workflow can be quite efficient. As the script is reloaded when you execute `%run`, your changes will be taken into account automatically. Things become more complicated when your script imports other Python modules that you modify, as these won't be reloaded with `%run`. To overcome this problem, you can use the autoreload IPython extension as described at <http://ipython.readthedocs.io/en/stable/config/extensions/autoreload.html>.

The Jupyter Notebook

The Jupyter Notebook plays a central role in efficient interactive workflows. It is a well-designed mix between a code editor and a Terminal, bringing the best of both worlds within a unified environment.

You can start writing all your code in your notebook's cells. You write, execute, and test your code in the same place, thereby improving your productivity. You can put long comments in Markdown cells and structure your notebook with Markdown headers.

Once portions of your code become mature enough and do not require further changes, you refactor them into reusable Python components (functions, classes, and modules). In practice, you copy and paste the code into Python scripts (files with the .py extension). Jupyter notebooks are currently not easily reusable by third-party code. They are designed for preliminary analyses and exploratory research, not for production-ready code.

A major advantage of notebooks is that they give you documents retracing everything you did with your code, which is particularly useful for reproducible research. Since notebooks are saved in human-readable JSON documents, they don't work that well with version control systems such as Git.

The ipymd module, available at <https://github.com/rossant/ipymd/>, and the more recent podoc module, available at <https://github.com/podoc/podoc>, allow you to use Markdown instead of JSON for notebooks. In podoc, images are saved in external files instead of being embedded in the JSON notebook, which is more convenient when working with a version control system.

JupyterLab, the next generation of the Jupyter Notebook, bridges the gap between the Jupyter Notebook and IDEs. It is covered in the *Introducing JupyterLab* recipe of Chapter 3, *Mastering the Jupyter Notebook*.

Integrated Development Environments

IDEs are particularly well-adapted for classic software development, but they can also be used for interactive computing. A good Python IDE combines a powerful text editor (for example, one that includes features such as syntax highlighting and tab completion), an IPython terminal, and a debugger within a unified environment.

There are multiple open-source and commercial IDEs. **Rodeo** is an IDE for data analysis made by yhat. **Spyder** is another open source IDE with good integration of IPython and Matplotlib.

Eclipse/PyDev is a popular (although slightly heavy) open source cross-platform environment.

PyCharm is one of many commercial environments that support IPython.

Microsoft's IDE for Windows, Visual Studio, has an open source plugin named **Python Tools for Visual Studio (PTVS)**. This tool brings Python support to Visual Studio. PTVS natively supports IPython. You don't necessarily need a paid version of Visual Studio; you can download a free package bundling PTVS with Visual Studio.

There's more...

Here are a few links to various IDEs for Python:

- ▶ Rodeo, at <https://www.yhat.com/products/rodeo>
- ▶ Spyder, at <https://github.com/spyder-ide/spyder>
- ▶ PyDev, at <http://pydev.org>

- ▶ PyCharm, at <http://www.jetbrains.com/pycharm/>
- ▶ PyTools for Microsoft Visual Studio, at <https://microsoft.github.io/PTVS/>

See also

- ▶ The *Learning the basics of the distributed version control system Git* recipe
- ▶ The *Debugging code with IPython* recipe

Ten tips for conducting reproducible interactive computing experiments

In this recipe, we present ten tips that can help you conduct efficient and reproducible interactive computing experiments. These are more guidelines than absolute rules.

First, we will show how you can improve your productivity by minimizing the time spent doing repetitive tasks and maximizing the time spent thinking about your core work.

Second, we will demonstrate how you can achieve more reproducibility in your computing work. Notably, academic research requires experiments to be reproducible so that any result or conclusion can be verified independently by other researchers. It is not uncommon for errors or manipulations in methods to result in erroneous conclusions that can have damaging consequences. For example, in the 2010 research paper in economics *Growth in a Time of Debt*, by Carmen Reinhart and Kenneth Rogoff, computational errors were partly responsible for a flawed study with global ramifications for policy makers (see https://en.wikipedia.org/wiki/Growth_in_a_Time_of_Debt).

How to do it...

1. Organize your directory structure carefully and coherently. The specific structure does not matter. What matters is to be consistent throughout your projects regarding file-naming conventions, folders, subfolders, and so on. Here is a simple example:



File structure

2. Write notes in text files using a lightweight markup language such as Markdown (<http://daringfireball.net/projects/markdown/>), **CommonMark** (<http://commonmark.org/>), or **reStructuredText (reST)**. All meta-information related to your project, files, datasets, code, figures, lab notebooks, and so on, should be written down in text files.
3. Relatedly, document everything non-trivial in your code with comments, docstrings, and so on. You can use a documentation tool such as **Sphinx** (<http://sphinx-doc.org>). However, do not spend too much time documenting unstable and bleeding-edge code while you are working on it; it might change frequently and your documentation may soon be out of date. Write your code in such a way that it's easily understandable without comments (name your variables and functions well, use *Pythonic* patterns, and so on). See also the next recipe, *Writing high-quality Python code*.
4. Use a version control system such as Git for all text-based files, but not binary files (except maybe for very small ones when you really need to). You should use one repository per project. Synchronize the repositories on a remote server, using a free or paid hosting provider (such as GitHub, GitLab, or Bitbucket), or your own server (your host institution might be able to set up one for you). Use a specific system to store and share binary data files, such as <http://figshare.com> or <http://datadryad.org>.

5. Write all your interactive computing code in Jupyter notebooks first and refactor it into standalone Python components only when it is sufficiently mature and stable.
6. Make sure that you record the exact versions of all components in your entire software stack (operating system, Python distribution, modules, and so on). A possibility is to use virtual environments with **virtualenv** or **conda**. Another possibility is to use **Docker** (<https://www.docker.com>).
7. Cache long-to-compute intermediary results using Python's native **pickle** module, **dill** (<https://github.com/uqfoundation/dill>), or Joblib (<http://pythonhosted.org/joblib/>). Joblib notably implements a NumPy-aware **memoize** pattern (not to be confused with memorize), which allows you to cache the results of computationally intensive functions.



How to save persistent data in Python? For purely internal purposes, you can use Joblib, NumPy's `save()` and `savez()` functions for arrays, and pickle for any other Python object (prefer native types such as lists and dictionaries rather than custom classes). For sharing purposes, prefer text files for small datasets (less than 10,000 points); for example, CSV for arrays, and JSON or YAML for highly structured data. For larger datasets, you can use HDF5 (see the *Manipulating large arrays with HDF5* recipe of Chapter 4, Profiling and Optimization).

8. When developing and trying out algorithms on large datasets, run them and compare them on small portions of your data first, before moving to the full sets.
9. When running jobs in a batch, use parallel computing to take advantage of your multicore processing units—for example, with **ipyparallel**, Joblib, **Dask** (<https://dask.pydata.org/en/latest/>), Python's multiprocessing package, or any other parallel computing library.
10. Automate your work as much as possible with Python functions or scripts. Use command-line arguments for user-exposed scripts, but choose Python functions over scripts when possible. On Unix systems, learn Terminal commands to improve your productivity. For repetitive tasks on Windows or GUI-based systems, use automation tools such as **AutoHotKey** (<http://www.autohotkey.com>). Learn keyboard shortcuts in the programs you use a lot, or create your own shortcuts. Automatic steps are reproducible; manual steps are not.

How it works...

The tips given in this recipe ultimately aim to optimize your workflows, in terms of human time, computer time, and quality. Using coherent conventions and structure for your code makes it easier for you to organize your work. Documenting everything saves everyone's time, including (eventually) yours!

Using a distributed version control system with an online hosting service makes it easy for you to work on the same code base from multiple locations, without ever worrying about backups. As you can go back in time in your code, you have very little chance of unintentionally breaking it.

The Jupyter Notebook is an excellent tool for reproducible interactive computing. It lets you keep a detailed record of your work. Also, the ease of using the Jupyter Notebook is that you don't have to worry about reproducibility; just do all of your interactive work in notebooks, put them under version control, and commit regularly. Don't forget to refactor your code into independent reusable components.

Be sure to optimize the time you spend in front of your computer. When working on an algorithm, this cycle frequently happens: you do a slight modification, you launch the code, get the results, make another change, and so on and so forth. If you need to try out a lot of changes, you should ensure that the execution time is fast enough (no more than a few seconds). Using advanced optimization techniques is not necessarily the best option at this stage of experimentation. You should cache your results, try out your algorithms on data subsets, and run your simulations with shorter durations. You can also launch batch jobs in parallel when you want to test different parameter values.

Finally, desperately try to avoid doing repetitive tasks. It is worth spending time automating such tasks when they occur frequently in your day-to-day work. It is more difficult to automate tasks that involve GUIs, but it is feasible thanks to free tools such as AutoHotKey.

There's more...

Here are a few references:

- ▶ *Barbagroup reproducibility syllabus*, at <http://lorenabarba.com/blog/barbagroup-reproducibility-syllabus/>.
- ▶ *An efficient workflow for reproducible science*, a talk by Trevor Bekolay, available at <http://bekolay.org/scipy2013-workflow/>.
- ▶ *Ten Simple Rules for Reproducible Computational Research*, Sandve and others, *PLoS Computational Biology*, 2013, available at <http://dx.doi.org/10.1371/journal.pcbi.1003285>.
- ▶ Software Carpentry, a volunteer organization running workshops for scientists; the workshops cover scientific programming, interactive computing, version control, testing, reproducibility, and task automation. You can find more information at <http://software-carpentry.org>.
- ▶ Reproducible Science, at <https://reproduciblescience.org/>.

See also

- ▶ The [Learning the basics of the Unix shell](#) recipe
- ▶ The [Efficient interactive computing workflows with IPython](#) recipe
- ▶ The [Writing high-quality Python code](#) recipe

Writing high-quality Python code

Writing code is easy. Writing high-quality code is much harder. Quality is to be understood both in terms of actual code (variable names, comments, docstrings, and so on) and architecture (functions, modules, and classes). In general, coming up with a well-designed code architecture is much more challenging than the implementation itself.

In this recipe, we will give a few tips about how to write high-quality code. This is a particularly important topic in academia, as more and more scientists without prior experience in software development need to code.

How to do it...

1. Take the time to learn the Python language seriously. Review the list of all modules in the standard library—you may discover that functions you implemented already exist. Learn to write *Pythonic* code, and do not translate programming idioms from other languages such as Java or C++ to Python.
2. Learn common **design patterns**; these are general reusable solutions to commonly occurring problems in software engineering.
3. Use assertions throughout your code (the `assert` keyword) to prevent future bugs (**defensive programming**).
4. Start writing your code with a bottom-up approach; write independent Python functions that implement focused tasks.
5. Do not hesitate to refactor your code regularly. If your code is becoming too complicated, think about how you can simplify it.
6. Avoid classes when you can. If you can use a function instead of a class, choose the function. A class is only useful when you need to store persistent state between function calls. Make your functions as pure as possible (no side effects).
7. In general, choose Python native types (lists, tuples, dictionaries, and types from Python's `collections` module) over custom types (classes). Native types lead to more efficient, readable, and portable code.
8. Choose keyword arguments over positional arguments in your functions. Argument names are easier to remember than argument ordering. They make your functions self-documenting.

9. Name your variables carefully. Names of functions and methods should start with a verb. A variable name should describe what it is. A function name should describe what it does. The importance of naming things well cannot be overstated.
10. Every function should have a docstring describing its purpose, arguments, and return values, as shown in the following example. You can also look at the conventions chosen in popular libraries such as NumPy. The exact convention does not matter; the point is to be consistent within your code. You can use a markup language such as Markdown or reST:

```
>>> def power(x, n):  
    """Compute the power of a number.  
  
    Arguments:  
    * x: a number  
    * n: the exponent  
  
    Returns:  
    * c: the number x to the power of n  
  
    """  
    return x ** n
```

11. Follow (at least partly) Guido van Rossum's *Style Guide for Python Code*, also known as **Python Enhancement Proposal number 8 (PEP8)**, available at <http://www.python.org/dev/peps/pep-0008/>. It is a long read, but it will help you write well-readable Python code. It covers many little things such as spacing between operators, naming conventions, comments, and docstrings. For instance, you will learn that it is considered a good practice to limit any line of your code to 79 or 99 characters. This way, your code can be correctly displayed in most situations (such as in a command-line interface or on a mobile device) or side by side with another file. Alternatively, you can decide to ignore certain rules. In general, following common guidelines is beneficial on projects involving many developers.
12. You can check your code automatically against most of the style conventions in PEP8 with the `pycodestyle` Python package (<https://github.com/PyCQA/pycodestyle>). You can also automatically make your code PEP8-compatible with the `autopep8` package (<https://github.com/hhatto/autopep8>).
13. Use a tool for static code analysis such as **Flake8** (<http://flake8.pycqa.org/en/latest/>) or **Pylint** (<https://www.pylint.org>). It lets you find potential errors or low-quality code statically—that is, without running your code.
14. Use blank lines to avoid cluttering your code (see PEP8). You can also demarcate sections in a long Python module with salient comments such as this:

```
>>> # Imports  
# -----
```

```
import numpy

# Utility functions
# -----

def fun():
    pass
```

15. A Python module should not contain more than a few hundred lines of code.
Having too many lines of code in a module may be a sign that you need to split it into several modules.
16. Organize important projects (with tens of modules) into subpackages (subdirectories).
17. Take a look at how major Python projects are organized. For example, the code of IPython is well-organized into a hierarchy of subpackages with focused roles. Reading the code itself is also quite instructive.
18. Learn best practices to create and distribute a new Python package. Make sure that you know setuptools, pip, wheels, virtualenv, PyPI, and so on. Also, you are highly encouraged to take a serious look at conda (<http://conda.pydata.org>), a powerful and generic packaging system created by Anaconda. Packaging has long been a rapidly evolving topic in Python, so read only the most recent references. There are a few references in the *There's more...* section.

How it works...

Writing readable code means that other people (or you, in a few months or years) will understand it quicker and will be more willing to use it. It also facilitates bug tracking.

Modular code is also easier to understand and to reuse. Implementing your program's functionality in independent functions that are organized as a hierarchy of packages and modules is an excellent way of achieving high code quality.

It is easier to keep your code loosely coupled when you use functions instead of classes. Spaghetti code is really hard to understand, debug, and reuse.

Iterate between bottom-up and top-down approaches while working on a new project. Starting with a bottom-up approach lets you gain experience with the code before you start thinking about the overall architecture of your program. Still, make sure you know where you're going by thinking about how your components will work together.

There's more...

Much has been written on how to write beautiful code—see the following references. You can find many books on the subject. In the next recipe, we will cover standard techniques to make sure that our code not only looks nice but also works as expected: unit testing, code coverage, and continuous integration.

Here are a few references:

- ▶ *Python Cookbook, 3rd Edition*, David Beazley and Brian K. Jones, O'Reilly Media with many Python advanced recipes, available at <http://shop.oreilly.com/product/0636920027072.do>
- ▶ *The Hitchhiker's Guide to Python!*, available at <http://docs.python-guide.org/en/latest/>
- ▶ Design patterns on Wikipedia, available at https://en.wikipedia.org/wiki/Software_design_pattern
- ▶ Design patterns in Python, described at <https://github.com/faif/python-patterns>
- ▶ Coding standards of Tahoe-LAFS, available at <https://tahoe-lafs.org/trac/tahoe-lafs/wiki/CodingStandards>
- ▶ *How to be a great software developer*, by Peter Nixey, available at <http://peternixey.com/post/83510597580/how-to-be-a-great-software-developer>
- ▶ *Why you should write buggy software with as few features as possible*, a talk by Brian Granger, available at <http://www.youtube.com/watch?v=OrpPDkZef5I>
- ▶ *Python Packaging User Guide*, available at <https://packaging.python.org/>
- ▶ A list of antonyms commonly used in programming, available at <https://github.com/rossant/programming-yin-yang>

See also

- ▶ The *Ten tips for conducting reproducible interactive computing experiments* recipe
- ▶ The *Writing unit tests with pytest* recipe
- ▶ A list of antonyms commonly used in programming, available at <https://github.com/rossant/programming-yin-yang>

Writing unit tests with pytest

Untested code is broken code. Manual testing is essential to ensuring that our software works as expected and does not contain critical bugs. However, manual testing is severely limited because bugs may be introduced at any time in the code.

Nowadays, automated testing is a standard practice in software engineering. In this recipe, we will briefly cover important aspects of automated testing: unit tests, test-driven development, test coverage, and continuous integration. Following these practices is fundamental in order to produce high-quality software.

Getting ready

Python has a native unit testing module that you can readily use (`unittest`). Other third-party unit testing packages exist. In this recipe, we will use **pytest**. It is installed by default in Anaconda, but you can also install it manually with `conda install pytest`.

How to do it...

1. Let's write in a `first.py` file a simple function that returns the first element of a list:

```
>>> %%writefile first.py
def first(l):
    return l[0]
Overwriting first.py
```

2. To test this function, we write another function, the *unit* test, that checks our `first` function using an example and an assertion:

```
>>> %%writefile -a first.py

# This is appended to the file.
def test_first():
    assert first([1, 2, 3]) == 1
Appending to first.py
>>> %cat first.py
def first(l):
    return l[0]

# This is appended to the file.
def test_first():
    assert first([1, 2, 3]) == 1
```

3. To run the unit test, we use the `pytest` executable (the ! means that we're calling an external program from IPython):

```
>>> !pytest first.py
===== test session starts =====
platform linux -- Python 3.6.3, pytest-3.2.1, py-1.4.34
rootdir: ~/git/cookbook-2nd/chapter02_best_practices:
plugins: cov-2.5.1
```

```
collecting 0 items
collecting 1 item
collected 1 item

first.py .

=====
1 passed in 0.00 seconds =====
```

4. Our test passes! Let's add another example with an empty list. We want our function to return `None` in this case:

```
>>> %%writefile first.py
def first(l):
    return l[0]

def test_first():
    assert first([1, 2, 3]) == 1
    assert first([]) is None

Overwriting first.py
>>> !pytest first.py
=====
test session starts =====
platform linux -- Python 3.6.3, pytest-3.2.1, py-1.4.34
rootdir: ~/git/cookbook-2nd/chapter02_best_practices:
plugins: cov-2.5.1

collecting 0 items
collecting 1 item
collected 1 item

first.py F

=====
FAILURES =====
____ test_first ____

def test_first():
    assert first([1, 2, 3]) == 1
>     assert first([]) is None

first.py:6:
-----
l = []

def first(l):
```

```
>         return l[0]
E         IndexError: list index out of range

first.py:2: IndexError
===== 1 failed in 0.02 seconds =====
```

5. This time, our test fails. Let's fix it by modifying the `first()` function:

```
>>> %%writefile first.py
def first(l):
    return l[0] if l else None

def test_first():
    assert first([1, 2, 3]) == 1
    assert first([]) is None

Overwriting first.py
>>> !pytest first.py
===== test session starts =====
platform linux -- Python 3.6.3, pytest-3.2.1, py-1.4.34
rootdir: ~/git/cookbook-2nd/chapter02_best_practices:
plugins: cov-2.5.1

collecting 0 items
collecting 1 item
collected 1 item

first.py .

===== 1 passed in 0.00 seconds =====
```

The test passes again!

How it works...

By definition, a unit test must focus on one specific functionality. All unit tests should be completely independent. Writing a program as a collection of well-tested, mostly decoupled units forces you to write modular code that is more easily maintainable.

In a Python package, a `test_xxx.py` module should accompany every Python module named `xxx.py`. This testing module contains unit tests that test functionality implemented in the `xxx.py` module.

Sometimes, your module's functions require preliminary work to run (for example, setting up the environment, creating data files, or setting up a web server). The unit testing framework can handle this via **fixtures**. The state of the system environment should be exactly the same before and after a testing module runs. If your tests affect the filesystem, they should do so in a temporary directory that is automatically deleted at the end of the tests. Testing frameworks such as pytest provide convenient facilities for this use case.

Tests typically involve many assertions. With pytest, you can simply use the built-in `assert` keyword. Further convenient assertion functions are provided by NumPy (see <http://docs.scipy.org/doc/numpy/reference/routines.testing.html>). They are especially useful when working with arrays. For example, `np.testing.assert_allclose(x, y)` asserts that the `x` and `y` arrays are almost equal, up to a given precision that can be specified.

Writing a full testing suite takes time. It imposes strong (but good) constraints on your code's architecture. It is a real investment, but it is always profitable in the long run. Also, knowing that your project is backed by a full testing suite is a real load off your mind.

First, thinking about unit tests from the beginning forces you to think about a modular architecture. It is really difficult to write unit tests for a monolithic program full of interdependencies.

Second, unit tests make it easier for you to find and fix bugs. If a unit test fails after introducing a change in the program, isolating and reproducing the bugs becomes trivial.

Third, unit tests help you avoid regressions—that is, fixed bugs that silently reappear in a later version. When you discover a new bug, you should write a specific failing unit test for it. To fix it, make this test pass. Now, if the bug reappears later, this unit test will fail and you will immediately be able to address it.

When you write a complex program based on interdependent APIs, having a good test coverage for one module means that you can safely rely on it in other modules, without worrying about its behavior not conforming to its specification.

Unit tests are just one type of automated tests. Other important types of tests include integration tests (making sure that different parts of the program work together) and functional tests (testing typical use cases).

There's more...

Automated testing is a wide topic, and we only scratched the surface in this recipe. We give some further information here.

Test coverage

Using unit tests is good. However, measuring test coverage is even better: it quantifies how much of our code is being covered by your testing suite. The `coverage.py` module (<https://coverage.readthedocs.io/>) does precisely this. It integrates well with pytest.

The **coveralls.io** service brings test-coverage features to a continuous integration server (refer to the *Unit testing and continuous integration* section). It works seamlessly with GitHub.

Workflows with unit testing

Note the particular workflow we have used in this example. After writing our function, we created a first unit test that passed. Then we created a second test, which failed. We investigated the issue and fixed the function. The second test passed. We could continue writing more and more complex unit tests, until we are confident that the function works as expected in most situations.



Run `pytest --pdb` to drop into the Python debugger on failures. This is quite convenient to find out quickly why a unit test fails.



We could even write the tests *before* the function itself. This is **Test-driven development (TDD)**, which consists of writing unit tests before writing the actual code. This workflow forces us to think about what our code does and how one uses it, instead of how it is implemented.

Unit testing and continuous integration

A good habit to get into is running the full testing suite of our project at every commit. In fact, it is even possible to do this completely transparently and automatically through **continuous integration**. We can set up a server that automatically runs our testing suite in the cloud at every commit. If a test fails, we get an automatic email telling us what the problem is so that we can fix it.

There are many continuous integration systems and services: Jenkins/Hudson, Travis CI (<https://travis-ci.org>), Codeship (<http://codeship.com/>), and others. Some of them play well with GitHub. For example, to use Travis CI with a GitHub project, create an account on Travis CI, link your GitHub project to this account, and then add a `.travis.yml` file with various settings in your repository (see the additional details in the references below).

In conclusion, unit testing, code coverage, and continuous integration are standard practices that should be used in all significant projects.

Here are a few references:

- ▶ *Test-driven development*, at https://en.wikipedia.org/wiki/Test-driven_development
- ▶ Documentation of Travis CI in Python, at <http://about.travis-ci.org/docs/user/languages/python/>

Debugging code with IPython

Debugging is an integral part of software development and interactive computing. A widespread debugging technique consists of placing the `print()` functions in various places in the code. Who hasn't done this? It is probably the simplest solution, but it is certainly not the most efficient (it is the poor man's debugger).

IPython is perfectly adapted for debugging, and the integrated debugger is quite easy to use (actually, IPython merely offers a nice interface to the native Python debugger **pdb**). In particular, tab completion works in the IPython debugger. This recipe describes how to debug code with IPython.

How to do it...

There are two not-mutually exclusive ways of debugging code in Python. In the post-mortem mode, the debugger steps into the code as soon as an exception is raised, so that we can investigate what caused it. In the step-by-step mode, we can stop the interpreter at a breakpoint and resume its execution step by step. This process allows us to check carefully the state of our variables as our code is executed.

Both methods can actually be used simultaneously; we can do step-by-step debugging in the post-mortem mode.

The post-mortem mode

When an exception is raised within IPython, execute the `%debug` magic command to launch the debugger and step into the code. Also, the `%pdb on` command tells IPython to launch the debugger automatically as soon as an exception is raised.

Once you are in the debugger, you have access to several special commands, the most important ones being listed here:

- ▶ `p varname` prints the value of a variable
- ▶ `w` shows your current location within the stack
- ▶ `u` goes up in the stack

- ▶ d goes down in the stack
- ▶ l shows the lines of code around your current location
- ▶ a shows the arguments of the current function

The call stack contains the list of all active functions at a given location in the code's execution. You can easily navigate up and down the stack to inspect the values of the function arguments. Although quite simple to use, this mode should let you resolve most of your bugs. For more complex problems, you may need to do step-by-step debugging.

Step-by-step debugging

You have several options to start the step-by-step debugging mode. First, in order to put a breakpoint somewhere in your code, insert the following command:

```
import pdb  
pdb.set_trace()
```

Second, you can run a script from IPython with the following command:

```
%run -d -b extscript.py:20 script
```

This command runs the `script.py` file under the control of the debugger with a breakpoint on line 20 in `extscript.py` (which is imported by `script.py`). Finally, you can do step-by-step debugging as soon as you are in the debugger.

Step-by-step debugging consists of precisely controlling the course of the interpreter. Starting from the beginning of a script or from a breakpoint, you can resume the execution of the interpreter with the following commands:

- ▶ s executes the current line and stops as soon as possible afterwards (step-by-step debugging—that is, the most fine-grained execution pattern)
- ▶ n continues the execution until the next line in the current function is reached
- ▶ r continues the execution until the current function returns
- ▶ c continues the execution until the next breakpoint is reached
- ▶ j 30 brings you to line 30 in the current file

You can add breakpoints dynamically from within the debugger using the b command or with tbreak (temporary breakpoint). You can also clear all or some of the breakpoints, enable or disable them, and so on. You can find the full details of the debugger at <https://docs.python.org/3/library/pdb.html>.

There's more...

To debug your code with IPython, you typically need to execute it first with IPython—for example, with `%run`. However, you may not always have an easy way of doing this. For instance, your program may run with a custom command-line Python script, it may be launched by a complex bash script, or it may be integrated within a GUI. In these cases, you can embed an IPython interpreter at any point in your code (launched by Python), instead of running your whole program with IPython (which may be overkill if you just need to debug a small portion of your code).

To embed IPython within your program, simply insert the following commands somewhere in your code:

```
from IPython import embed  
embed()
```

When your Python program reaches this code, it will pause and launch an interactive IPython terminal at this specific point. You will then be able to inspect all local variables, run any code you want, and possibly debug your code before resuming normal execution.

Most Python IDEs offer graphical debugging features (see the *Efficient interactive computing workflows with IPython* recipe). A GUI can sometimes be more convenient than a command-line debugger. A list of Python debuggers is available at <https://wiki.python.org/moin/PythonDebuggingTools>.

3

Mastering the Jupyter Notebook

In this chapter, we will cover the following topics:

- ▶ Teaching programming in the Notebook with IPython Blocks
- ▶ Converting a Jupyter notebook to other formats with nbconvert
- ▶ Mastering widgets in the Jupyter Notebook
- ▶ Creating custom Jupyter Notebook widgets in Python, HTML, and JavaScript
- ▶ Configuring the Jupyter Notebook
- ▶ Introducing JupyterLab

Introduction

In this chapter, we will explore several advanced features and usage examples of the Jupyter Notebook. As we have only seen basic features in the previous chapters, we will dive deeper into the architecture of the Notebook here.

The Notebook ecosystem

Jupyter notebooks are represented as **JavaScript Object Notation (JSON)** documents. JSON is a language-independent, text-based file format for representing structured documents. As such, notebooks can be processed by any programming language, and they can be converted to other formats such as Markdown, HTML, LaTeX/PDF, and others.

There is an ecosystem of tools around Jupyter Notebook. Notebooks are being used to create slides, teaching materials, blog posts, research papers, and even books. In fact, this very book is entirely written in the Notebook using the Markdown format and a custom-made Python tool.

JupyterLab is the next generation of the Jupyter Notebook. It is still in an early stage of development at the time of writing. We cover it in the last recipe of this chapter.

Architecture of the Jupyter Notebook

Jupyter implements a two-process model, with a **kernel** and a **client**. The client is the interface offering the user the ability to send code to the kernel. The kernel executes the code and returns the result to the client for display. In the **Read-Evaluate-Print Loop (REPL)** terminology, the kernel implements the *Evaluate*, whereas the client implements the *Read* and the *Print* of the process.

The client can be a Qt widget if we run the Qt console, or a browser if we run the Jupyter Notebook. In the Jupyter Notebook, the kernel receives entire cells at once, so it has no notion of a notebook. There is a strong decoupling between the linear document containing the notebook, and the underlying kernel.

All communication procedures between the different processes are implemented on top of the **ZeroMQ (ZMQ)** messaging protocol (<http://zeromq.org>). The Notebook communicates with the underlying kernel using WebSocket, a TCP-based protocol implemented in modern web browsers.

Connecting multiple clients to one kernel

In a notebook, typing `%connect_info` in a cell gives the information we need to connect a new client (such as a Qt console) to the underlying kernel:

```
>>> %connect_info
{
    "shell_port": 58645,
    "iopub_port": 47422,
    "stdin_port": 60550,
    "control_port": 39092,
    "hb_port": 49409,
    "ip": "127.0.0.1",
    "key": "2298f955-7020b0ce534e7a8d81053d43",
    "transport": "tcp",
    "signature_scheme": "hmac-sha256",
    "kernel_name": ""
}
```

```
Paste the above JSON into a file, and connect with:  
    $> jupyter <app> --existing <file>  
or, if you are local, you can connect with just:  
    $> jupyter <app> --existing kernel-4342f625-a8...  
or even just:  
    $> jupyter <app> --existing  
if this is the most recent Jupyter kernel you  
have started.
```

Here, <app> is console, qtconsole, or notebook.

JupyterHub

JupyterHub, available at <https://jupyterhub.readthedocs.io/en/latest/>, is a Python library that can be used to serve notebooks to a set of end-users, for example students of a particular class, or lab members in a research group. It handles user authentication and other low-level details.

Security in notebooks

It is possible for an attacker to put malicious code in a Jupyter notebook. Since notebooks may contain hidden JavaScript code in a cell output, it is theoretically possible for malicious code to execute surreptitiously when the user opens a notebook.

For this reason, Jupyter has a security model where HTML and JavaScript code in a notebook can be either trusted or untrusted. Outputs generated by the user are always trusted. However, outputs that were already there when the user first opened an existing notebook are untrusted.

The security model is based on a cryptographic signature present in every notebook. This signature is generated using a secret key owned by every user.

References

The following are some references about the Notebook architecture:

- ▶ Overview of IPython at <http://ipython.readthedocs.io/en/stable/overview.html>
- ▶ Documentation for the Jupyter Notebook, available at <https://jupyter.readthedocs.io/en/latest/>
- ▶ Security in the Notebook, described at <http://jupyter-notebook.readthedocs.io/en/stable/security.html>

- ▶ The Jupyter messaging protocol, at <http://jupyter-client.readthedocs.io/en/latest/messaging.html>
- ▶ Wrapper kernels at <http://jupyter-client.readthedocs.io/en/latest/wrapperkernels.html>

Here are a few kernels in non-Python languages for the Notebook:

- ▶ IJulia, available at <https://github.com/JuliaLang/IJulia.jl>
- ▶ IRkernel, available at <https://github.com/IRkernel/IRkernel>
- ▶ IHaskell, available at <https://github.com/gibiansky/IHaskell>
- ▶ Dozens of kernels are referenced at <https://github.com/jupyter/jupyter/wiki/Jupyter-kernels>

Teaching programming in the Notebook with IPython Blocks

The Jupyter Notebook is not only a tool for scientific research and data analysis but also a great tool for teaching. In this recipe, we show a simple and fun Python library for teaching programming notions: **IPython Blocks** (available at <http://ipythonblocks.org>). This library allows you or your students to create grids of colorful blocks. You can change the color and size of individual blocks, and you can even animate your grids. There are many basic technical notions you can illustrate with this tool. The visual aspect of this tool makes the learning process more effective and engaging.

In this recipe, we will notably perform the following tasks:

- ▶ Illustrate matrix multiplication with an animation
- ▶ Display an image as a block grid

Getting ready

To install IPython Blocks, type `pip install ipythonblocks` in a Terminal.

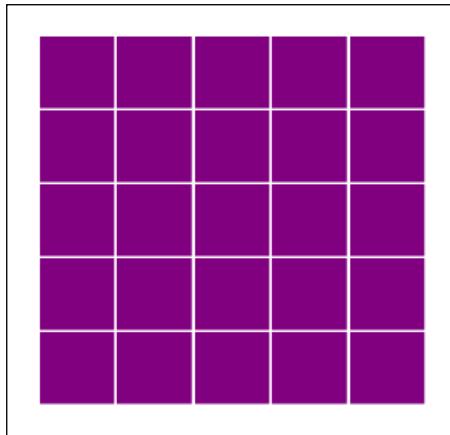
How to do it...

1. First, we import some modules as follows:

```
>>> import time
      from IPython.display import clear_output
      from ipythonblocks import BlockGrid, colors
```

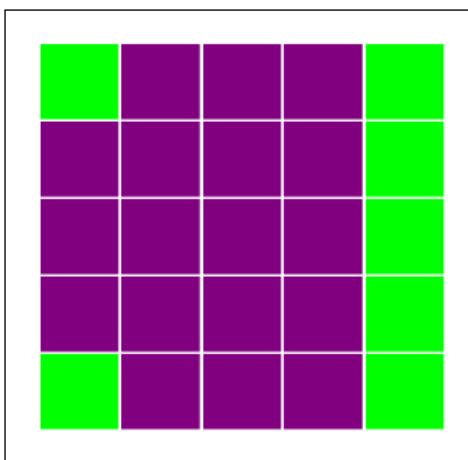
2. Now, we create a block grid with five columns and five rows, and we fill each block with purple:

```
>>> grid = BlockGrid(width=5, height=5,
                     fill=colors['Purple'])
grid.show()
```



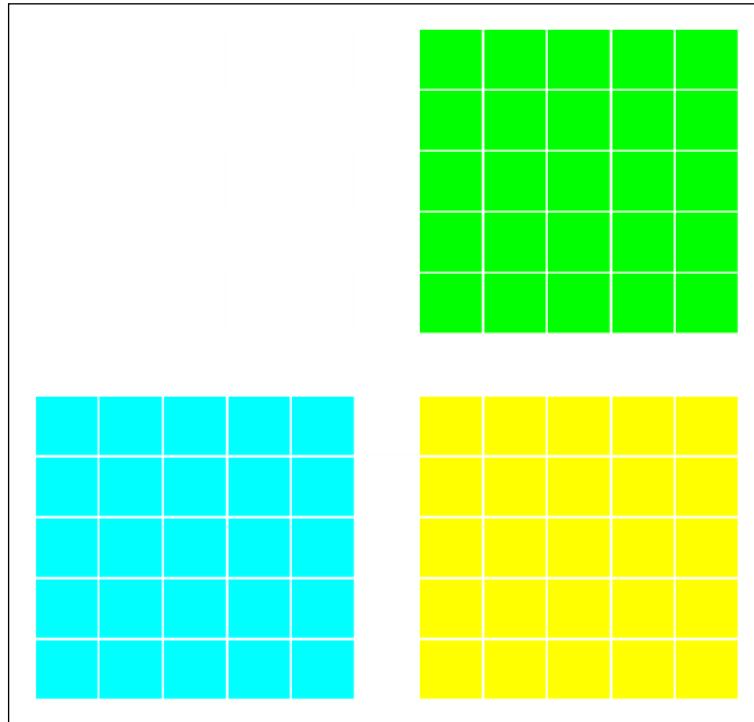
3. We can access individual blocks with 2D indexing. This illustrates the indexing syntax in Python. We can also access an entire row or line with a : (colon). Each block is represented by an RGB color. The library comes with a handy dictionary of colors, assigning RGB tuples to standard color names as follows:

```
>>> grid[0, 0] = colors['Lime']
grid[-1, 0] = colors['Lime']
grid[:, -1] = colors['Lime']
grid.show()
```



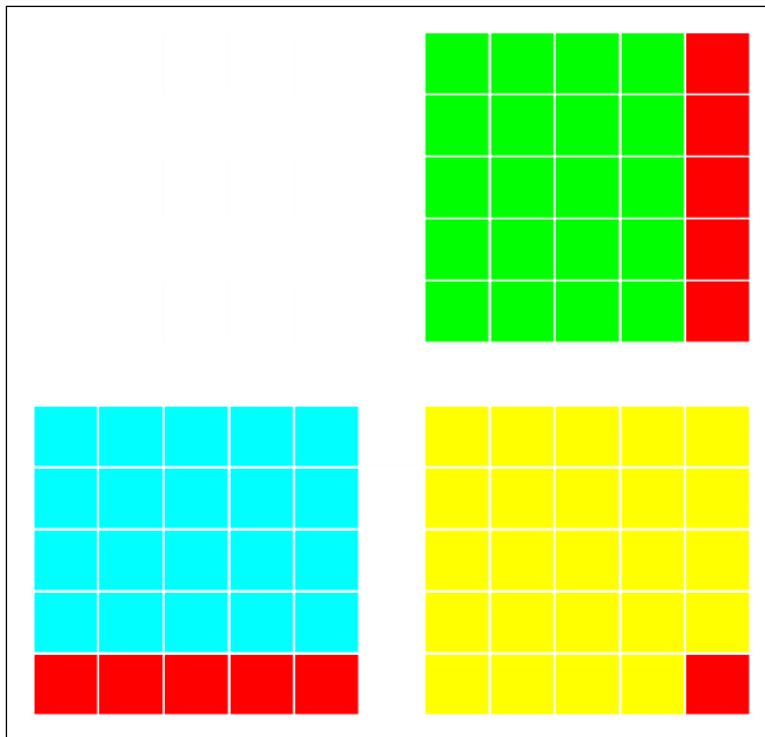
4. Now, we are going to illustrate matrix multiplication. We will represent two (n, n) matrices, A (in cyan) and B (lime), aligned with $C = A B$ (yellow). To do this, we use a small trick: creating a big white grid of size $(2n+1, 2n+1)$. The matrices A, B, and C are just views on parts of the grid.

```
>>> n = 5
      grid = BlockGrid(width=2 * n + 1,
                         height=2 * n + 1,
                         fill=colors['White'])
      A = grid[n + 1:, :n]
      B = grid[:n, n + 1:]
      C = grid[n + 1:, n + 1:]
      A[:, :] = colors['Cyan']
      B[:, :] = colors['Lime']
      C[:, :] = colors['Yellow']
      grid.show()
```



5. Let's turn to matrix multiplication itself. We perform a loop over all rows and columns, and we highlight the corresponding rows and columns in A and B that are multiplied together during the matrix product. We combine IPython's `clear_output()` method with `grid.show()` and `time.sleep()` (pause) to implement the animation as follows:

```
>>> for i in range(n):
    for j in range(n):
        # We reset the matrix colors.
        A[:, :] = colors['Cyan']
        B[:, :] = colors['Lime']
        C[:, :] = colors['Yellow']
        # We highlight the adequate rows
        # and columns in red.
        A[i, :] = colors['Red']
        B[:, j] = colors['Red']
        C[i, j] = colors['Red']
        # We animate the grid in the loop.
        clear_output()
        grid.show()
        time.sleep(.25)
```



6. Finally, we display an image with IPython Blocks. We download and import a PNG image with Matplotlib and we retrieve the data as follows:

```
>>> # We downsample the image by a factor of 4 for
# performance reasons.
img = plt.imread('https://github.com/ipython-books/'
                  'cookbook-2nd-data/blob/master/'
                  'beach.png?raw=true')[::4, ::4, :]
>>> rgb = [img[..., i].ravel() for i in range(3)]
```

7. Now, we create a `BlockGrid` instance with the appropriate number of rows and columns, and we set each block's color to the corresponding pixel's color in the image (multiplying by 255 to convert from a floating-point number in [0, 1] into an 8-bit integer). We use a small block size, and we remove the lines between the blocks as follows:

```
>>> height, width = img.shape[:2]
grid = BlockGrid(width=width, height=height,
                  block_size=2, lines_on=False)
for block, r, g, b in zip(grid, *rgb):
    block.rgb = (r * 255, g * 255, b * 255)
grid.show()
```



Converting a Jupyter notebook to other formats with nbconvert

A Jupyter notebook is saved in a JSON text file. This file contains the entire contents of the notebook: text, code, and outputs. The Matplotlib figures are encoded as base64 strings within the notebooks, resulting in standalone, but sometimes big, notebook files.

 JSON is a human-readable, text-based, open standard format that can represent structured data. Although derived from JavaScript, it is language-independent. Its syntax bears some resemblance to Python dictionaries. JSON can be parsed in many languages including JavaScript and Python (using the `json` module in Python's standard library).

nbconvert (<https://nbconvert.readthedocs.io/en/stable/>) is a tool that can convert notebooks to other formats: raw text, Markdown, HTML, LaTeX/PDF, and even slides with the `reveal.js` library. You will find more information about the different supported formats on the nbconvert documentation.

One typically uses the **nbformat** (<https://nbformat.readthedocs.io/en/latest/>) library to manipulate a notebook. However, in this recipe, we will see how to manipulate the contents of a notebook (which is just a plain-text JSON file) directly with Python, and how to convert it to other formats with nbconvert.

Getting ready

You need to install pandoc, available at <http://pandoc.org>. This tool is used to convert markup files to various formats. On Ubuntu, type `sudo apt-get install pandoc` in a Terminal.

To convert a notebook to PDF, you need a LaTeX distribution, which you can download and install at <http://latex-project.org/ftp.html>.

How to do it...

1. Let's download and open the test notebook. A notebook is just a plain-text file (JSON):

```
>>> import io
      import requests
>>> url = ('https://github.com/ipython-books/'
          'cookbook-2nd-data/blob/master/'
          'test.ipynb?raw=true')
>>> contents = requests.get(url).text
      print(len(contents))
3857
```

2. Here is an excerpt of the `test.ipynb` file:

```
>>> print(contents[:345] + '...' + contents[-33:])
{
  "cells": [
    {
      "cell_type": "markdown",
      "metadata": {},
      "source": [
        "# First chapter"
      ]
    },
    {
      "cell_type": "markdown",
      "metadata": {
        "my_field": [
          "value1",
          "2405"
        ]
      },
      "source": [
        "Let's write some *rich* **text** with
        [links] (http://www.ipython.org) and lists:\n",
        "\n",
        "* item1...rmat": 4,
      ],
      "nbformat_minor": 4
    }
  ]
}
```

3. Now that we have loaded the notebook in a string, let's parse it with the `json` module as follows:

```
>>> import json
nb = json.loads(contents)
```

4. Let's have a look at the keys in the notebook dictionary:

```
>>> print(nb.keys())
print('nbformat %d.%d' % (nb['nbformat'],
                           nb['nbformat_minor']))
dict_keys(['cells', 'metadata',
           'nbformat', 'nbformat_minor'])
nbformat 4.4
```

5. Each cell has a type, optional metadata, some contents (text or code), possibly one or several outputs, and other information. Let's look at a Markdown cell and a code cell:

```
>>> nb['cells'][1]
{'cell_type': 'markdown',
 'metadata': {'my_field': ['value1', '2405']},
 'source': ["Let's write some *rich* **text** with
            [links] (http://www.ipython.org) and lists:\n",
            '\n',
            '* item1\n',
            '* item2\n',
            '    1. subitem\n',
            '    2. subitem\n',
            '* item3']}
>>> nb['cells'][2]
{'cell_type': 'code',
 'execution_count': 1,
 'metadata': {},
 'outputs': [{"data": {"image/png": "iVBOR...QmCC\n",
      "text/plain": ["<matplotlib Figure at ...>"]},
      "metadata": {},
      "output_type": "display_data"}],
 'source': ['import numpy as np\n',
            'import matplotlib.pyplot as plt\n',
            '%matplotlib inline\n',
            'plt.figure(figsize=(2,2));\n',
            "plt.imshow(np.random.rand(10,10),
                       interpolation='none');\n",
            "plt.axis('off');\n",
            'plt.tight_layout();']}
```

6. Once parsed, the notebook is represented as a Python dictionary. Manipulating it is therefore quite convenient in Python. Here, we count the number of Markdown and code cells as follows:

```
>>> cells = nb['cells']
nm = len([cell for cell in cells
          if cell['cell_type'] == 'markdown'])
nc = len([cell for cell in cells
          if cell['cell_type'] == 'code'])
print((f"There are {nm} Markdown cells and "
       f"{nc} code cells."))
There are 2 Markdown cells and 1 code cells.
```

7. Let's have a closer look at the image output of the cell with the Matplotlib figure:

```
>>> cells[2]['outputs'][0]['data']
{'image/png': 'iVBOR...QmCC\n',
 'text/plain': ['<matplotlib.figure.Figure at ...>']}
```

In general, there can be zero, one, or multiple outputs. Additionally, each output can have multiple representations. Here, the Matplotlib figure has a PNG representation (the base64-encoded image) and a text representation (the internal representation of the figure).

8. Now, we convert our text notebook to HTML using nbconvert:

```
>>> # We write the notebook to a file on disk.
      with open('test.ipynb', 'w') as f:
          f.write(contents)
>>> !jupyter nbconvert --to html test.ipynb
[NbConvertApp] Converting notebook test.ipynb to html
[NbConvertApp] Writing 253784 bytes to test.html
```

9. Let's display this document in an <iFrame> (a small window showing an external HTML document within the notebook):

```
>>> from IPython.display import IFrame
      IFrame('test.html', 600, 200)
```

First chapter

Let's write some *rich text* with [links](#) and lists:

- item1
- item2
 - 1. subitem
 - 2. subitem
- item3

```
In [1]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
plt.figure(figsize=(2,2));
plt.imshow(np.random.rand(10,10), interpolation='none');
plt.axis('off');
plt.tight_layout();
```



10. We can also convert the notebook to LaTeX and PDF. In order to specify the title and author of the document, we need to extend the default LaTeX template. First, we create a file called `temp.tplx` that extends the default `article.tplx` template provided by nbconvert. We specify the contents of the author and title blocks as follows:

```
>>> %%writefile temp.tplx
((*- extends 'article.tplx' -*))

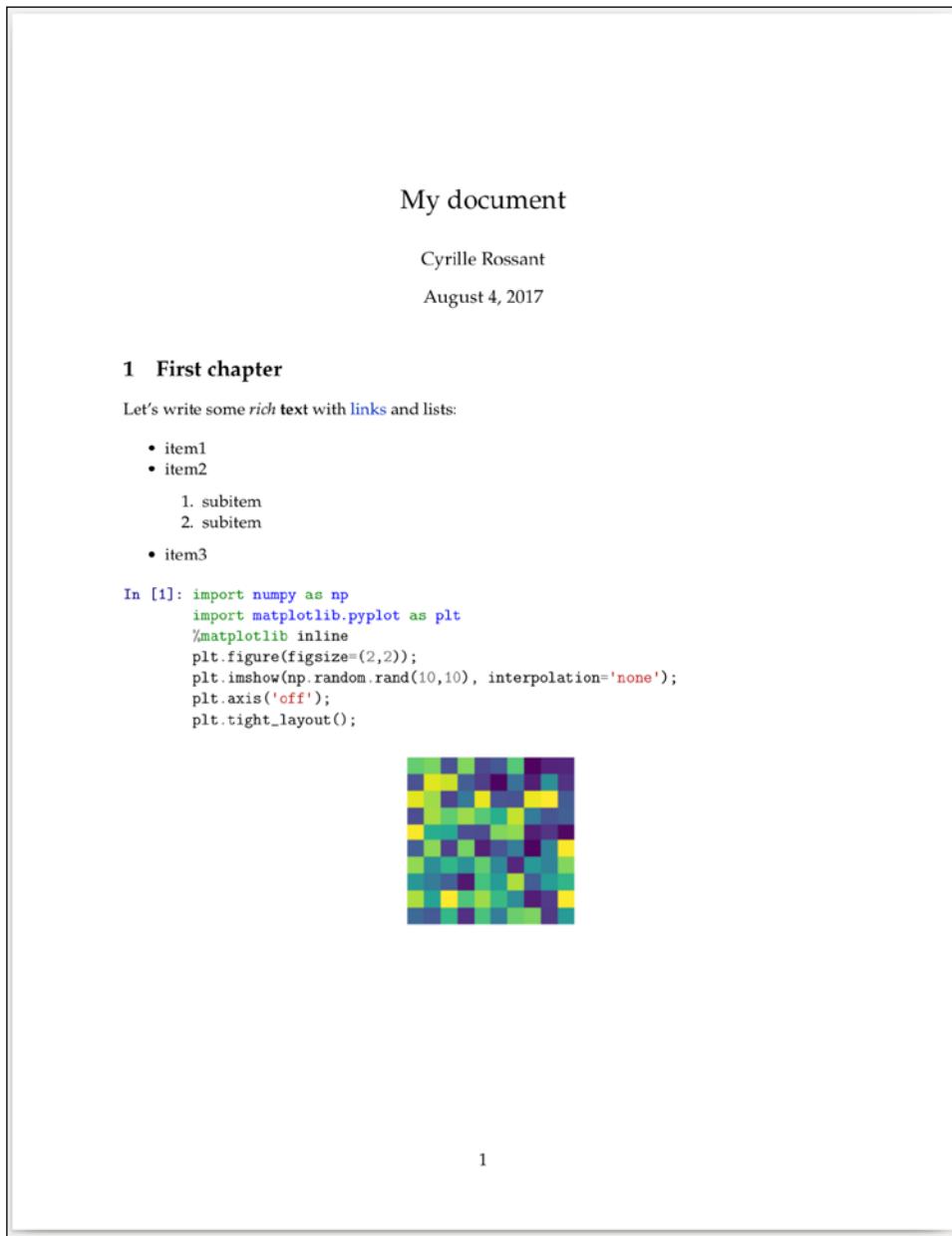
((* block author *))
\author{Cyrille Rossant}
((* endblock author *))

((* block title *))
\title{My document}
((* endblock title *))
Writing temp.tplx
```

11. Then we can run nbconvert by specifying our custom template as follows:

```
>>> %%bash
jupyter nbconvert --to pdf --template temp test.ipynb
[NbConvertApp] Converting notebook test.ipynb to pdf
[NbConvertApp] Support files will be in test_files/
[NbConvertApp] Making directory test_files
[NbConvertApp] Writing 16695 bytes to notebook.tex
[NbConvertApp] Building PDF
[NbConvertApp] Running xelatex 3 times:
    ['xelatex', 'notebook.tex']
[NbConvertApp] Running bibtex 1 time:
    ['bibtex', 'notebook']
[NbConvertApp] PDF successfully created
[NbConvertApp] Writing 16147 bytes to test.pdf
```

We used `nbconvert` to convert the notebook to LaTeX, and `pdflatex` (from our LaTeX distribution) to compile the LaTeX document to PDF. The following screenshot shows the PDF version of the notebook:



How it works...

As we have seen in this recipe, an `.ipynb` file contains a structured representation of the notebook. This JSON file can be easily parsed and manipulated in Python and other languages. However, it is better practice to use the `nbformat` package to manipulate a notebook. The internal JSON format may change, whereas the `nbformat` API is not expected to change.

`nbconvert` is a tool for converting a notebook to another format. The conversion can be customized in several ways. Here, we extended an existing template using `Jinja2`, a templating package (see <http://jinja.pocoo.org/docs/>).

There's more...

There is a free online service, **nbviewer**, that lets us render Jupyter notebooks in HTML dynamically in the cloud. The idea is that we provide nbviewer a URL to with a raw notebook (in JSON), and we get a rendered HTML output. The main page of nbviewer (<http://nbviewer.jupyter.org/>) contains a few examples. This service is maintained by the Jupyter developers and is hosted on Rackspace (<https://www.rackspace.com>).

GitHub automatically renders Jupyter notebooks stored in repositories.

binder, available at <https://mybinder.org>, allows one to turn a GitHub repository into a collection of interactive notebooks in the cloud. The service is free and the code is open source, so that anyone can provide their own binder service.

Here are some more references:

- ▶ Documentation for `nbconvert`, at <https://nbconvert.readthedocs.io/en/stable/>
- ▶ RISE, to create interactive slideshows out of Jupyter notebooks, at <https://daminavila.github.io/RISE/>

Mastering widgets in the Jupyter Notebook

The **ipywidgets** package provides many common user interface controls for exploring code and data interactively. These controls can be assembled and customized to create complex graphical user interfaces. In this recipe, we introduce the various ways we can create user interfaces with `ipywidgets`.

Getting ready

The `ipywidgets` package should be installed by default in Anaconda, but you can also install it manually with `conda install ipywidgets`.

Alternatively, you can install ipywidgets with `pip install ipywidgets`, but then you also need to type the following command in order to enable the extension in the Jupyter Notebook:

```
jupyter nbextension enable --py --sys-prefix widgetsnbextension
```

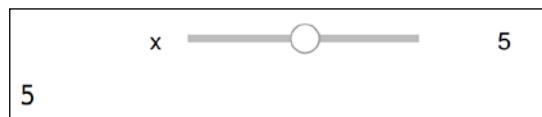
How to do it...

1. Let's import the packages:

```
>>> import ipywidgets as widgets  
      from ipywidgets import HBox, VBox  
      import numpy as np  
      import matplotlib.pyplot as plt  
      from IPython.display import display  
      %matplotlib inline
```

2. The `@interact` decorator shows a widget for controlling the arguments of a function. Here, the function `f()` accepts an integer as an argument. By default, the `@interact` decorator displays a slider to control the value passed to the function:

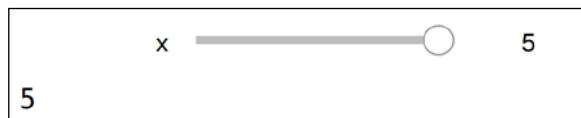
```
>>> @widgets.interact  
      def f(x=5):  
          print(x)
```



The function `f()` is called whenever the slider value changes.

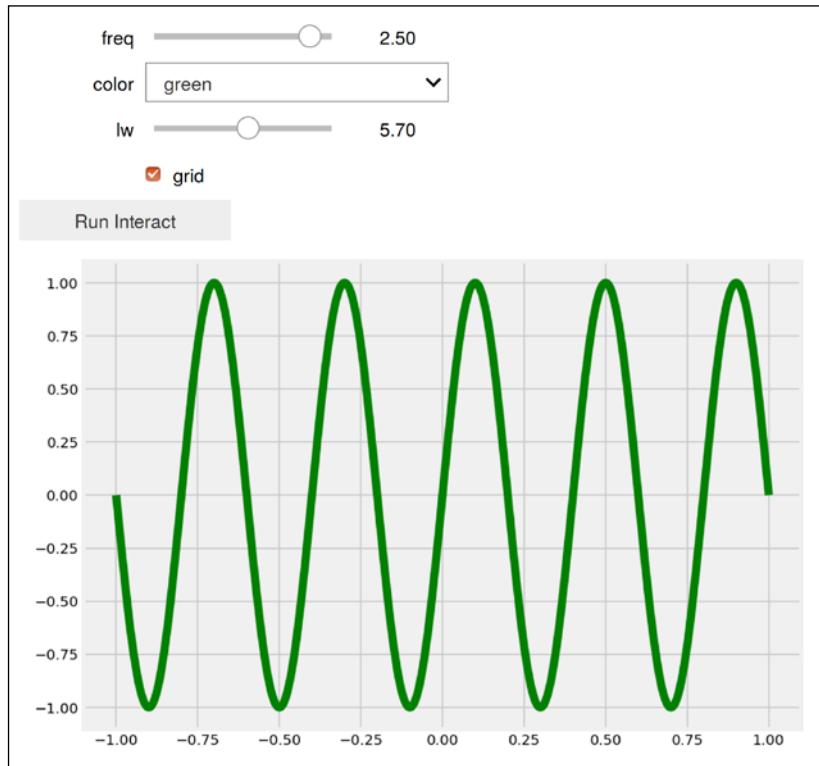
3. We can customize the slider parameters. Here, we specify a minimum and maximum integer range for the slider:

```
>>> @widgets.interact(x=(0, 5))  
      def f(x=5):  
          print(x)
```



4. There is also an `@interact_manual` decorator that provides a button to call the function manually. This is useful with long-lasting computations that should not run every time a widget value changes. Here, we create a simple user interface for controlling four parameters of a function that displays a plot. There are two floating-point sliders, a drop-down menu for choosing a value among a few predefined options, and a checkbox for Boolean values:

```
>>> @widgets.interact_manual(
    color=['blue', 'red', 'green'], lw=(1., 10.))
def plot(freq=1., color='blue', lw=2, grid=True):
    t = np.linspace(-1., +1., 1000)
    fig, ax = plt.subplots(1, 1, figsize=(8, 6))
    ax.plot(t, np.sin(2 * np.pi * freq * t),
            lw=lw, color=color)
    ax.grid(grid)
```



5. In addition to the `@interact` and `@interact_manual` decorators, ipywidgets provides a simple API to create individual widgets. Here, we create a floating-point slider:

```
>>> freq_slider = widgets.FloatSlider(  
        value=2.,  
        min=1.,  
        max=10.0,  
        step=0.1,  
        description='Frequency:',  
        readout_format='.1f',  
)  
freq_slider
```



6. Here is an example of a slider for selecting pairs of numbers, such as intervals and ranges:

```
>>> range_slider = widgets.FloatRangeSlider(  
        value=[-1., +1.],  
        min=-5., max=+5., step=0.1,  
        description='xlim:',  
        readout_format='.1f',  
)  
range_slider
```



7. The toggle button can control a Boolean value:

```
>>> grid_button = widgets.ToggleButton(  
        value=False,  
        description='Grid',  
        icon='check'  
)  
grid_button
```



8. Drop-down menus and toggle buttons are useful when selecting a value among a predefined set of options:

```
>>> color_buttons = widgets.ToggleButtons(  
        options=['blue', 'red', 'green'],  
        description='Color:',  
)  
color_buttons
```



9. The text widget allows the user to write a string:

```
>>> title_textbox = widgets.Text(  
        value='Hello World',  
        description='Title:',  
)  
title_textbox
```



10. We can let the user choose a color using the built-in system color picker:

```
>>> color_picker = widgets.ColorPicker(  
        concise=True,  
        description='Background color:',  
        value='#efefef',  
)  
color_picker
```



11. We can also simply create a button:

```
>>> button = widgets.Button(  
    description='Plot',  
)  
button
```



12. Now, we will see how to combine these widgets into a complex Graphical User Interface, and how to react to user interactions with these controls. We create a function that will display a plot as defined by the created controls. We can access the control value with the `value` property of the widgets:

```
>>> def plot2(b=None):  
    xlim = range_slider.value  
    freq = freq_slider.value  
    grid = grid_button.value  
    color = color_buttons.value  
    title = title_textbox.value  
    bgcolor = color_picker.value  
  
    t = np.linspace(xlim[0], xlim[1], 1000)  
    f, ax = plt.subplots(1, 1, figsize=(8, 6))  
    ax.plot(t, np.sin(2 * np.pi * freq * t),  
            color=color)  
    ax.grid(grid)
```

13. The `on_click` decorator of a `button` widget lets us react to click events. Here, we simply declare that the plotting function should be called when the button is pressed:

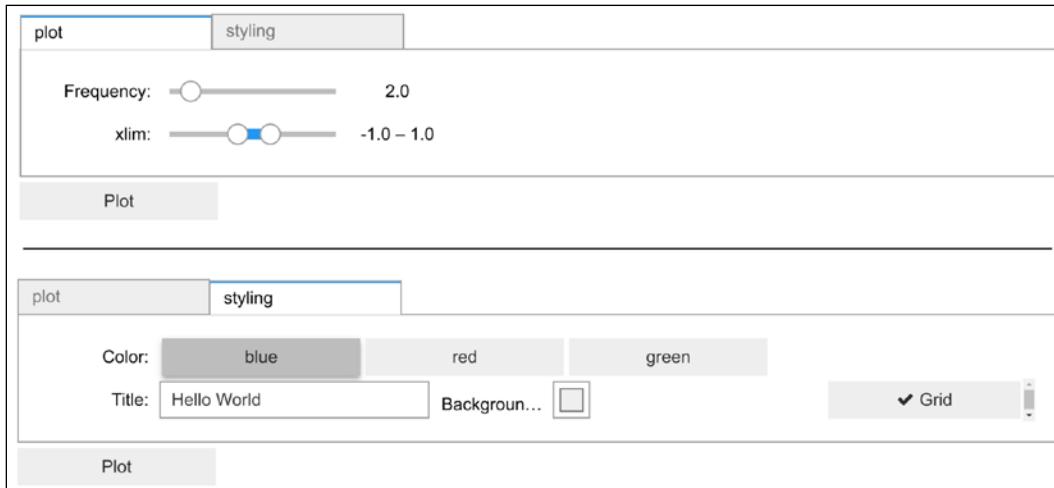
```
>>> @button.on_click  
def plot_on_click(b):  
    plot2()
```

14. To display all of our widgets in a unified graphical interface, we define a layout with two tabs. The first tab shows widgets related to the plot itself, whereas the second tab shows widgets related to the styling of the plot. Each tab contains a vertical stack of widgets defined with the `VBox` class:

```
>>> tab1 = VBox(children=[freq_slider,  
                        range_slider,  
                        ])  
tab2 = VBox(children=[color_buttons,  
                      HBox(children=[title_textbox,  
                                    color_picker,  
                                    grid_button]),  
                      ])
```

15. Finally, we create the Tab instance with our two tabs, we set the titles of the tabs, and we add the plot button below the tabs:

```
>>> tab = widgets.Tab(children=[tab1, tab2])
       tab.set_title(0, 'plot')
       tab.set_title(1, 'styling')
       VBox(children=[tab, button])
```



There's more...

The documentation for ipywidgets demonstrates many other features of the package. Styling the widgets can be customized. New widgets can be created by writing Python and JavaScript code (see the *Creating custom Jupyter Notebook widgets in Python, HTML, and JavaScript* recipe). Widgets can also remain at least partly functional in a static notebook export.

Here are a few references:

- ▶ ipywidgets user guide at https://ipywidgets.readthedocs.io/en/stable/user_guide.html
- ▶ Building a custom widget at <https://ipywidgets.readthedocs.io/en/stable/examples/Widget%20Custom.html>

See also

- ▶ The *Creating custom Jupyter Notebook widgets in Python, HTML, and JavaScript* recipe

Creating custom Jupyter Notebook widgets in Python, HTML, and JavaScript

The ipywidgets package provides many built-in control widgets to interact with code and data in the Jupyter Notebook. In this recipe, we show how to build a custom interactive widget from scratch, using Python on the kernel side, and HTML/JavaScript on the client side (frontend). The widget just displays two buttons to increase and decrease a number. The number can be accessed and updated either from the kernel (Python code) or the client (browser).

How to do it...

1. Let's import the packages:

```
>>> import ipywidgets as widgets  
      from traitlets import Unicode, Int, validate
```

2. We create a CounterWidget class deriving from DOMWidget:

```
>>> class CounterWidget(widgets.DOMWidget):  
      _view_name = Unicode('CounterView').tag(sync=True)  
      _view_module = Unicode('counter').tag(sync=True)  
      value = Int(0).tag(sync=True)
```

This class represents the Python part of the widget. The `_view_name` and `_view_module` attributes refer to the name and module of the JavaScript part. We use the `traitlets` package to specify the type of the variables. The `value` attribute is the counter value, an integer initialized at 0. All of these attributes' values are synchronized between Python and JavaScript, hence the `sync=True` option.

3. We now turn to the JavaScript side of the widget. We can write the code directly in the notebook using the `%%javascript` cell magic. The widget framework relies on several JavaScript libraries: jQuery (represented as the `$` variable), RequireJS (modules and dependencies), and Backbone.js (a model view controller framework):

```
>>> %%javascript  
      // We make sure the 'counter' module is defined  
      // only once.  
      require.undef('counter');  
  
      // We define the 'counter' module depending on the  
      // Jupyter widgets framework.  
      define('counter', ["@jupyter-widgets/base"],  
            function(widgets) {  
  
              // We create the CounterView frontend class,  
              // deriving from DOMWidgetView.
```

```
var CounterView = widgets.DOMWidgetView.extend({  
  
    // This method creates the HTML widget.  
    render: function() {  
        // The value_changed() method should be  
        // called when the model's value changes  
        // on the kernel side.  
        this.value_changed();  
        this.model.on('change:value',  
                    this.value_changed, this);  
  
        var model = this.model;  
        var that = this;  
  
        // We create the plus and minus buttons.  
        this.bm = $('>')  
            .text('-')  
            .click(function() {  
                // When the button is clicked,  
                // the model's value is updated.  
                var x = model.get('value');  
                model.set('value', x - 1);  
                that.touch();  
            });  
  
        this.bp = $('>')  
            .text('+')  
            .click(function() {  
                var x = model.get('value');  
                model.set('value', x + 1);  
                that.touch();  
            });  
  
        // This element displays the current  
        // value of the counter.  
        this.span = $('            .text('0')  
            .css({marginLeft: '10px',  
                  marginRight: '10px'});  
  
        // this.el represents the widget's DOM  
        // element. We add the minus button,  
        // the span element, and the plus button.  
        $(this.el)
```

```
.append(this.bm)
.append(this.span)
.append(this.bp);
},
value_changed: function() {
    // Update the displayed number when the
    // counter's value changes.
    var x = this.model.get('value');
    $($($this.el).children()[1]).text(x);
},
));
return {
    CounterView : CounterView
};
});
});
```

4. Let's display the widget:

```
>>> w = CounterWidget()
w
```

In [4]: `w = CounterWidget()`
`w`
- 0 +

5. Pressing the buttons updates the value immediately.

In [4]: `w = CounterWidget()`
`w`
- 3 +
Custom widget

6. The counter's value is automatically updated on the kernel side:

```
>>> print(w.value)
4
```

7. Conversely, we can update the value from Python, and it is updated in the frontend:

```
>>> w.value = 5
```

```
: w = CounterWidget()  
w  
- 5 +
```

There's more...

Here are a few references:

- ▶ Custom widget tutorial at <https://ipywidgets.readthedocs.io/en/stable/examples/Widget%20Custom.html>
- ▶ RequireJS library at <http://requirejs.org/>
- ▶ Backbone.js library at <http://backbonejs.org/>

See also

- ▶ The *Mastering widgets in the Jupyter Notebook* recipe

Configuring the Jupyter Notebook

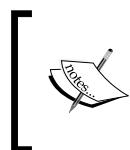
Many aspects of the Jupyter Notebook can be configured. We covered the configuration of the IPython kernel in the *Mastering IPython's configuration system* recipe in *Chapter 1, A Tour of Interactive Computing with Jupyter and IPython*. In this recipe, we show how to configure the Jupyter application and the Jupyter Notebook frontend.

How to do it...

1. Let's check whether the Jupyter Notebook configuration file already exists:

```
>>> %ls ~/.jupyter/jupyter_notebook_config.py  
~/.jupyter/jupyter_notebook_config.py
```

If it does not, type `!jupyter notebook --generate-config -y` in the notebook. If the file already exists, this command will delete its contents and replace it with the default file.



A Jupyter configuration file may exist in Python or in JSON (the same location and filename, but different file extension). JSON files have a higher priority. Unlike Python files, JSON files may be edited programmatically.

2. We can inspect the contents of the file with the following command:

```
>>> %cat ~/.jupyter/jupyter_notebook_config.py
# Configuration file for jupyter-notebook.

#-----
# Application(SingletonConfigurable) configuration
#-----

## This is an application.

## The date format used by logging formatters
#c.Application.log_datefmt = '%Y-%m-%d %H:%M:%S'

[...]

#-----
# JupyterApp(Application) configuration
#-----


## Base class for Jupyter applications

## Answer yes to any prompts.
#c.JupyterApp.answer_yes = False

## Full path of a config file.
#c.JupyterApp.config_file = ''


...
```

For example, to change the default name of a new notebook, we can add the following line to this file:

```
c.ContentsManager.untitled_notebook = 'MyNotebook'
```

3. We now turn to the configuration of the Jupyter Notebook frontend. The configuration files are in the following folder:

```
>>> %ls ~/.jupyter/nbconfig/
notebook.json  tree.json
```

4. Let's inspect the contents of the notebook configuration file (in JSON):

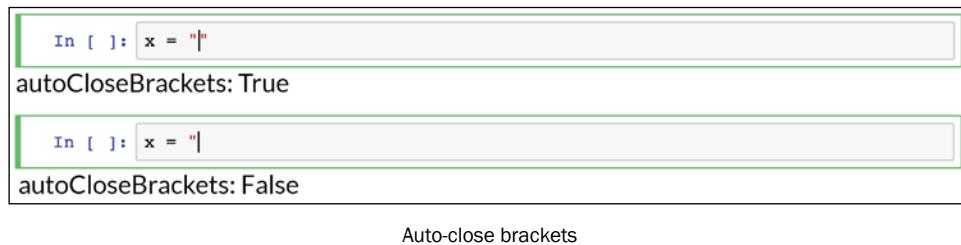
```
>>> %cat ~/.jupyter/nbconfig/notebook.json
{
    "Cell": {
        "cm_config": {
```

```
        "lineNumbers": false
    }
},
"Notebook": {
    "Header": false,
    "Toolbar": false
}
}
```

5. There are several ways to configure the Notebook frontend. We can directly edit this JSON file and reload the notebook. We can also do it in the client using JavaScript. For example, here is how we can disable the auto-closing brackets option in code cells:

```
>>> %%javascript
var cell = Jupyter.notebook.get_selected_cell();
var config = cell.config;
var patch = {
    CodeCell:{
        cm_config: {autoCloseBrackets: false}
    }
}
config.update(patch)
```

If we reload the notebook, this option will be permanently turned off.



6. In fact, this command automatically updates the JSON file:

```
>>> %cat ~/.jupyter/nbconfig/notebook.json
{
    "Cell": {
        "cm_config": {
            "lineNumbers": false
        }
    },
    "Notebook": {
        "Header": false,
```

```
        "Toolbar": false
    },
    "CodeCell": {
        "cm_config": {
            "autoCloseBrackets": false
        }
    }
}
```

7. We can also get and change the frontend options from Python:

```
>>> from notebook.services.config import ConfigManager
      c = ConfigManager()
      c.get('notebook').get('CodeCell')
      {'cm_config': {'autoCloseBrackets': False}}
>>> c.update('notebook', {"CodeCell":
      {"cm_config": {"autoCloseBrackets": True}}})
{'Cell': {'cm_config': {'lineNumbers': False}},
 'CodeCell': {'cm_config': {'autoCloseBrackets': True}},
 'Notebook': {'Header': False, 'Toolbar': False}}
>>> %cat ~/.jupyter/nbconfig/notebook.json
{
    "Cell": {
        "cm_config": {
            "lineNumbers": false
        }
    },
    "Notebook": {
        "Header": false,
        "Toolbar": false
    },
    "CodeCell": {
        "cm_config": {
            "autoCloseBrackets": true
        }
    }
}
```

There's more...

The code cell editor used in the Notebook is handled by the CodeMirror JavaScript library. All options are detailed in the CodeMirror documentation.

Here are a few references:

- ▶ Notebook configuration at <http://jupyter-notebook.readthedocs.io/en/stable/config.html>
- ▶ Notebook frontend configuration at https://jupyter-notebook.readthedocs.io/en/stable/frontend_config.html
- ▶ CodeMirror options at <https://codemirror.net/doc/manual.html#option-indentUnit>

See also

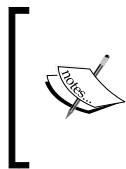
- ▶ The *Mastering IPython's configuration system* recipe in Chapter 1, *A Tour of Interactive Computing with Jupyter and IPython*

Introducing JupyterLab

JupyterLab is the next generation of the Jupyter Notebook. It aims at fixing many Notebook usability issues and it greatly expands its scope. JupyterLab offers a general framework for interactive computing and data science in the browser, using Python, Julia, R, or one of many other languages.

In addition to providing an improved interface to existing notebooks, JupyterLab also brings, within the same interface, a file browser, consoles, terminals, text editors, Markdown editors, CSV editors, JSON editors, interactive maps, widgets, and so on. The architecture is completely extensible and open to developers. In a word, JupyterLab is a web-based, hackable IDE for data science and interactive computing.

JupyterLab uses the exact same Notebook server and file format as the classic Jupyter Notebook, so that it is fully compatible with existing notebooks and kernels. Notebook and JupyterLab can run side to side on the same computer. You can easily switch between the two interfaces.



At the time of writing, JupyterLab is still in an early stage of development. However, it is already fairly usable. The interface may change until the production release. The developer API used to customize JupyterLab is still not stable. There is no user documentation yet.

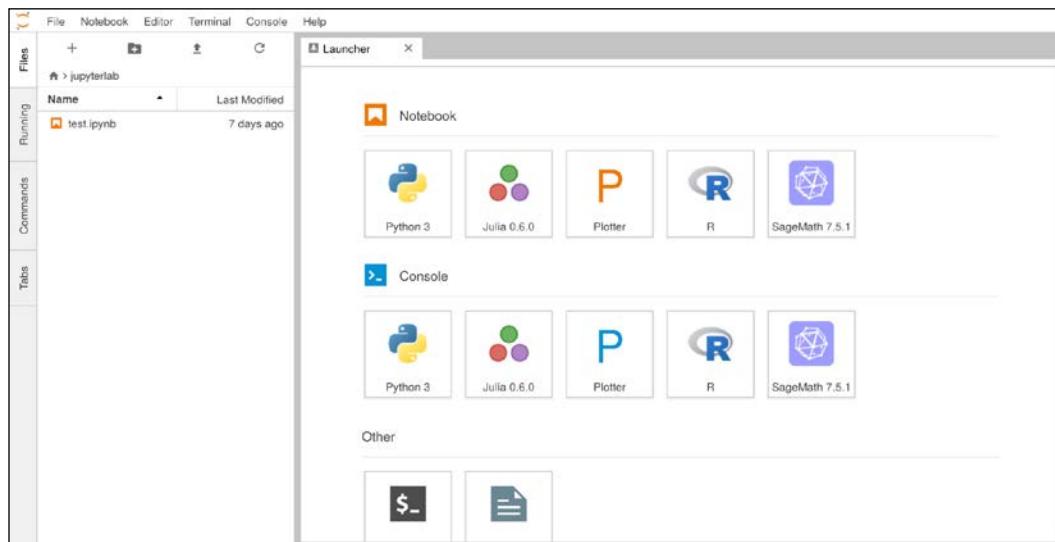
Getting ready

To install JupyterLab, type `conda install -c conda-forge jupyterlab` in a Terminal.

To be able to render GeoJSON files in an interactive map, install the GeoJSON JupyterLab extension with: `jupyter labextension install @jupyterlab/geojson-extension`.

How to do it...

1. We can launch JupyterLab by typing `jupyter lab` in a Terminal. Then we go to `http://localhost:8888/lab` in the web browser.
2. The dashboard shows, on the left, a list of files and subdirectories in the current working directory. On the right, the launcher lets us create notebooks and text files, or open a Jupyter console or a Terminal. Available Jupyter kernels are automatically displayed (here, IPython, but also IR and IJulia).

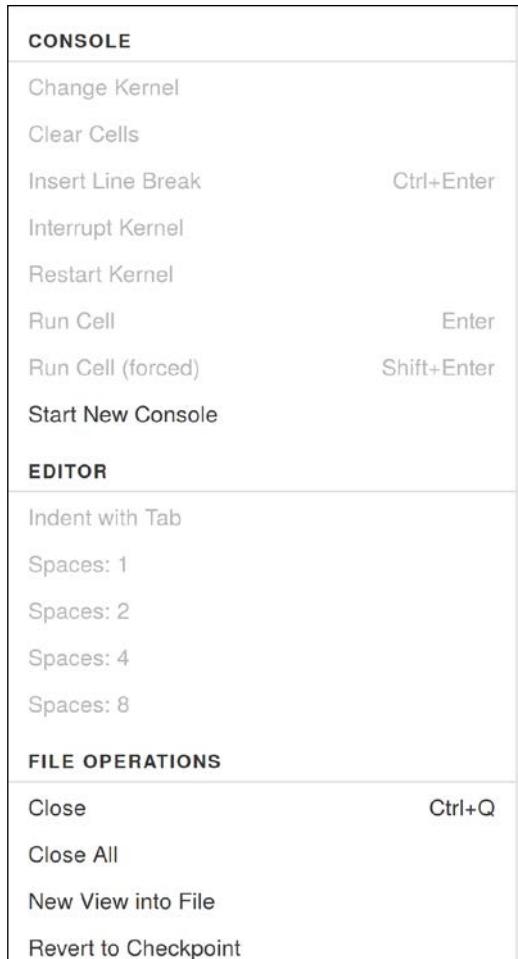


JupyterLab home

3. On the left panel, we can also see a list of open tabs, a list of running sessions, or a list of available commands:

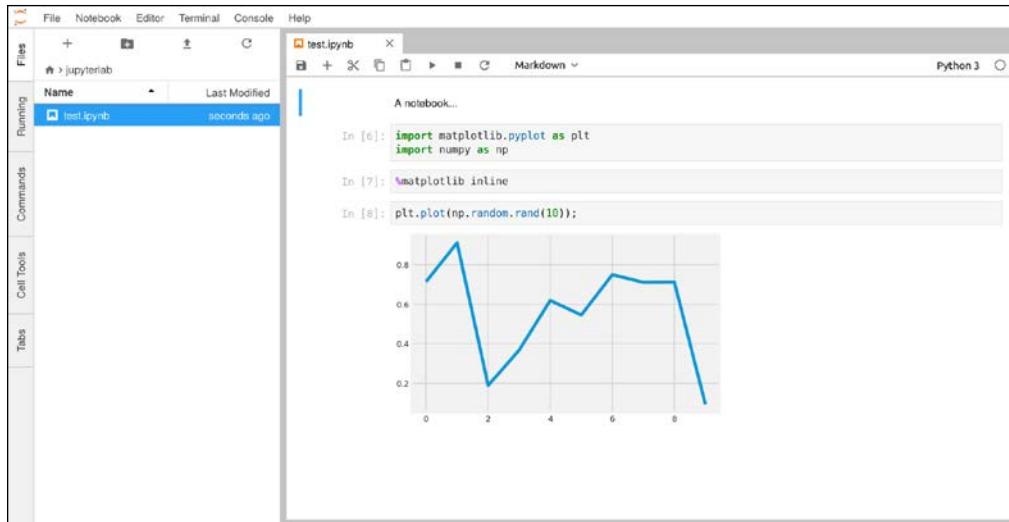
| C | | |
|-------------------|----------------|----------|
| TERMINAL SESSIONS | | |
| \$- | terminals/1 | SHUTDOWN |
| KERNEL SESSIONS | | |
| ▶ | test.ipynb | SHUTDOWN |
| ▶ | Console 1 | SHUTDOWN |
| ▶ | Console 2 | SHUTDOWN |
| ▶ | Untitled.ipynb | SHUTDOWN |

Running sessions



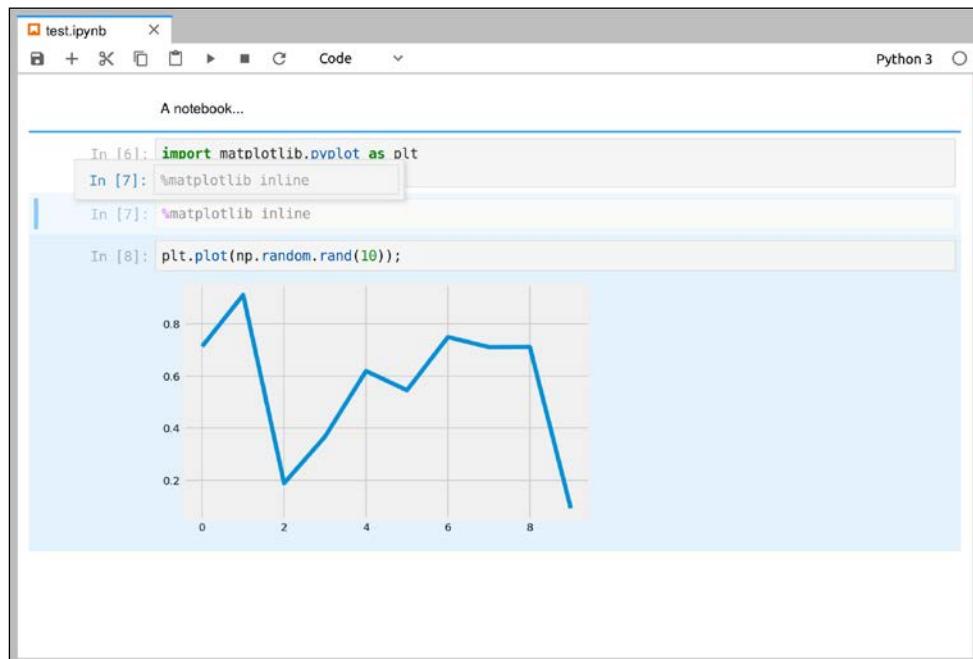
Available commands

4. If we open a Jupyter notebook, we get an interface that closely resembles the classic Notebook interface:



A notebook

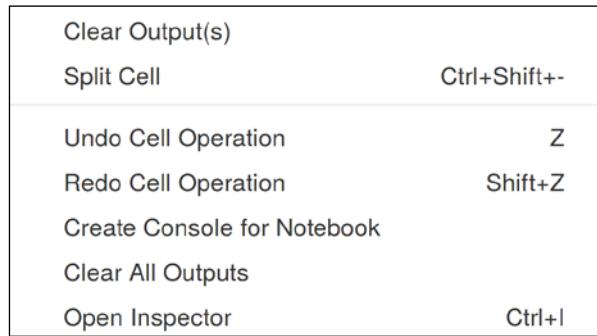
There are a few improvements compared to the classic Notebook. For example, we can drag and drop one or several cells:



Drag and drop in the notebook

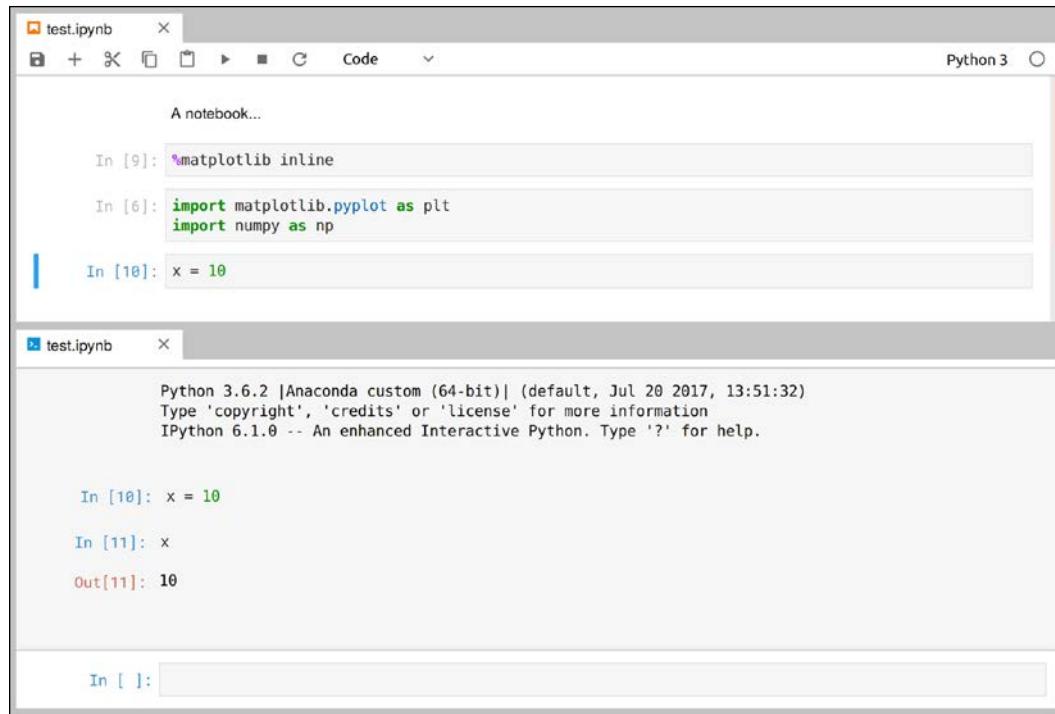
We can also collapse cells.

5. If we right-click in the notebook, a contextual menu appears:



Contextual menu in the notebook

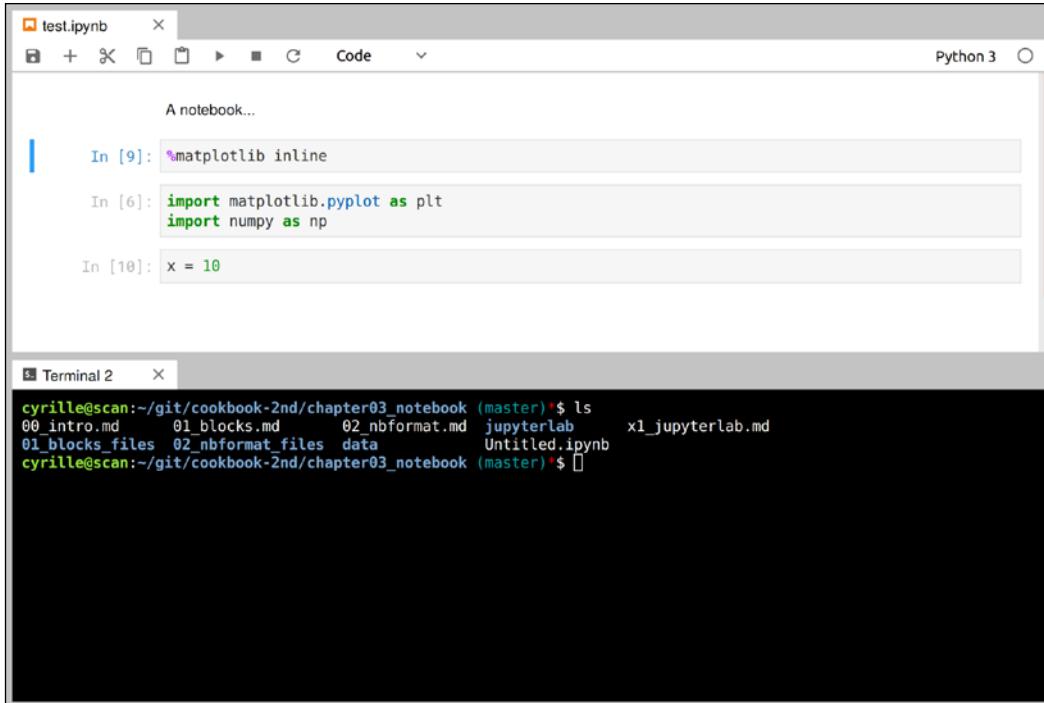
If we click on **Create Console for Notebook**, a new tab appears with a standard IPython console. We can drag and drop the tab anywhere in the screen, for example below the notebook panel:



Notebook and console

The IPython console is connected to the same kernel as the Notebook, so they share the same namespace. We can also open a new IPython console from the launcher, running in a separate kernel.

6. We can also open a system shell directly in the browser, using the `term.js` library:



Notebook and shell

7. JupyterLab includes a text editor. We can create a new text file from the launcher, rename it by giving it the `.md` extension, and edit it:



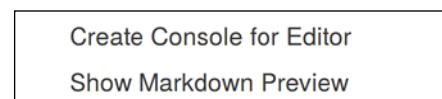
The screenshot shows a JupyterLab interface with a code editor window. The code is as follows:

```
1 # Title
2
3 Here is some code:
4
5 ```python
6 x = 10
7 ```
8
9 Another code block:
10
11 ```python
12 x * 2
13 ```


```

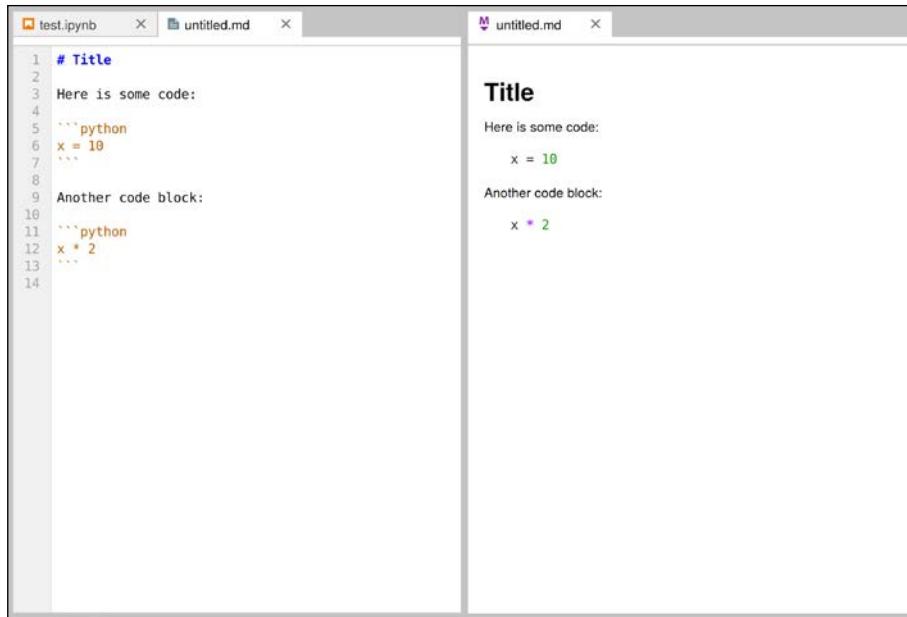
Markdown document

Let's right-click on the Markdown file. A contextual menu appears:



Contextual menu in a Markdown file

We can add a new panel that renders the Markdown file in real time:



The screenshot shows two tabs open in a browser window: 'test.ipynb' and 'untitled.md'. The 'test.ipynb' tab contains Python code cells:

```
1 # Title
2
3 Here is some code:
4
5 ````python
6 x = 10
7 ````

8 Another code block:
9
10 ````python
11 x * 2
12 ````

13
14
```

The 'untitled.md' tab shows the rendered output:

Title

Here is some code:

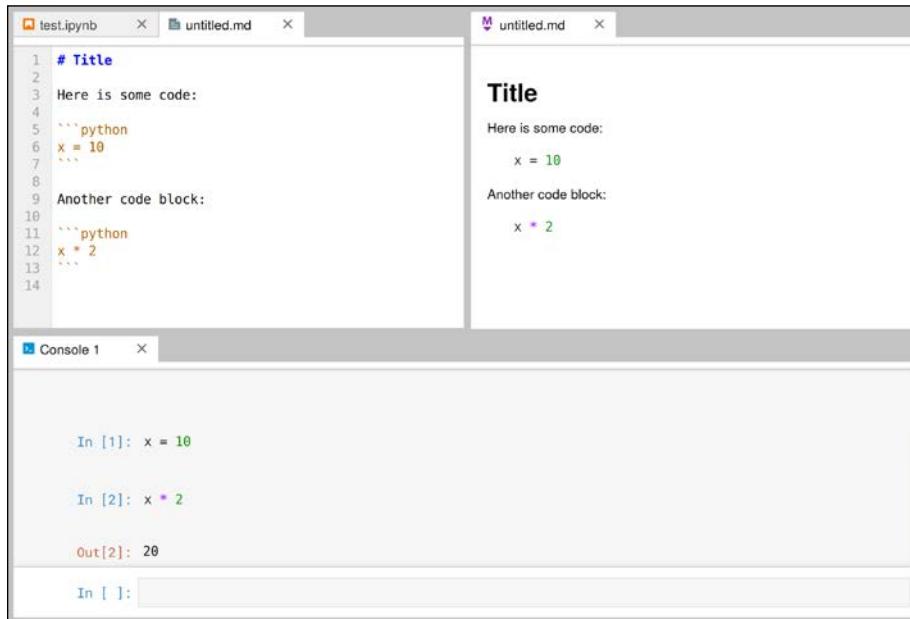
```
x = 10
```

Another code block:

```
x * 2
```

Markdown rendered

We can also attach an IPython console to our Markdown file. By clicking within a code block and pressing *Shift + Enter*, we send the code directly to the console:



The screenshot shows three tabs open: 'test.ipynb', 'untitled.md', and 'Console 1'. The 'test.ipynb' tab contains the same Python code as before. The 'untitled.md' tab shows the rendered output. The 'Console 1' tab displays the IPython session:

```
In [1]: x = 10

In [2]: x * 2

Out[2]: 20

In [ ]:
```

Markdown and console

8. We can also create and open CSV files in JupyterLab:

```
In [17]: import numpy as np
In [12]: M = np.random.rand(10000, 10)
In [20]: np.savetxt('file.csv', M, delimiter=",", fmt='%.3f',
                 header=','.join('col%d' % i for i in range(10)),
                 comments='')
In [21]: !head file.csv
col0,col1,col2,col3,col4,col5,col6,col7,col8,col9
0.912,0.673,0.826,0.587,0.035,0.030,0.884,0.193,0.610,0.645,0.857,0.816,0.620,0.863,0.735,0.141
0.889,0.534,0.232,0.614,0.774,0.428,0.452,0.994,0.402,0.790,0.825,0.531,0.669,0.659,0.248,0.01
2.0,0.943,0.194,0.826,0.013,0.078,0.048,0.436,0.658,0.361,0.744,0.916,0.361,0.726,0.091,0.605,0.8
99.0,0.978,0.549,0.526,0.967,0.340,0.701,0.297,0.382,0.483,0.514,0.922,0.963,0.926,0.024,0.396,0.
128,0.285,0.375,0.208,0.671,0.674,0.999,0.409,0.141,0.292,0.353,0.368,0.187,0.708,0.082,0.826,0
.798,0.114,0.468,0.037,0.022,0.409,0.122,0.979,0.013,0.528,0.891,0.243,0.344,0.940,0.813,0.774,
0.976,0.494,0.170,0.895,0.806,0.764,0.814,0.707,0.653,0.714,0.617,0.573,0.392,0.855,0.818,0.379
,0.678,0.868,0.050,0.743,0.337,0.479,0.731,0.284,0.273,0.871,0.728,0.960,0.841,0.159,0.525,0.58
3.0,0.082,0.894,0.816,0.688,0.076,0.102,0.043,0.715,0.051,0.864,0.922,0.536,0.557,0.007,0.228,0.5
00,0.069,0.826,0.376,0.834,0.452,0.521,0.417,0.347,0.600,0.235,0.455,0.698,0.654,0.455,0.848,0.
641,0.188,0.267,0.586,0.422,0.127,0.393,0.873,0.866,0.113,0.686,0.844,0.314,0.537,0.668,0.239,0
.139,0.241,0.142,0.900,0.681,0.080,0.948,0.022,0.236,0.499,0.880,0.469,0.413,0.552,0.430,0.114,
0.499,0.066,0.875,0.769,0.737,0.941,0.336,0.641,0.097,0.736,0.348,0.791,0.902,0.829,0.476,0.077
,0.947,0.538,0.478,0.845,0.619,0.973,0.762,0.579,0.033,0.054,0.971,0.378,0.435,0.856,0.436,0.11
6,0.777,0.905,0.023,0.878,0.191,0.595,0.239,0.270,0.660,0.676,0.578,0.679,0.437,0.107,0.712,0.1
21,0.982,0.931,0.534,0.173,0.018,0.777,0.461,0.212,0.573,0.391,0.882,0.678,0.706,0.408,0.488,0.
627,0.305,0.434,0.024,0.521,0.964,0.085,0.059,0.998,0.823,0.428,0.386,0.600,0.261,0.359,0.169,0
.052,0.268,0.363,0.542,0.372,0.842,0.371,0.727,0.617,0.030,0.934,0.402,0.811,0.730,0.892,0.920,
0.682,0.800,0.556,0.442,0.475,0.033,0.578,0.272,0.581,0.581,0.335,0.302,0.594,0.143,0.681,0.422
```

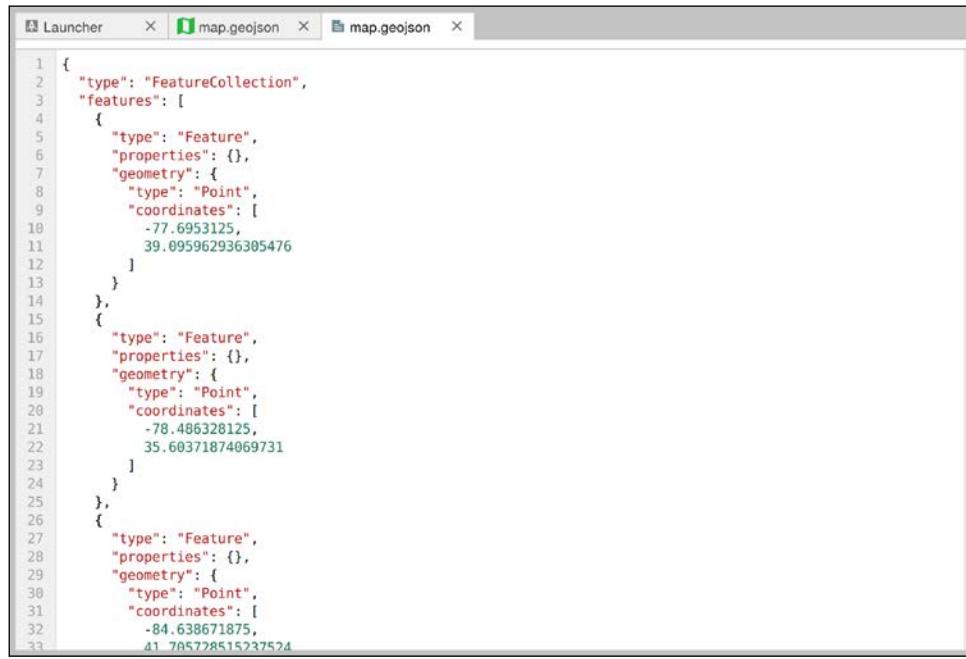
CSV file

The CSV viewer is highly efficient. It can smoothly display huge tables with millions or even billions of values:

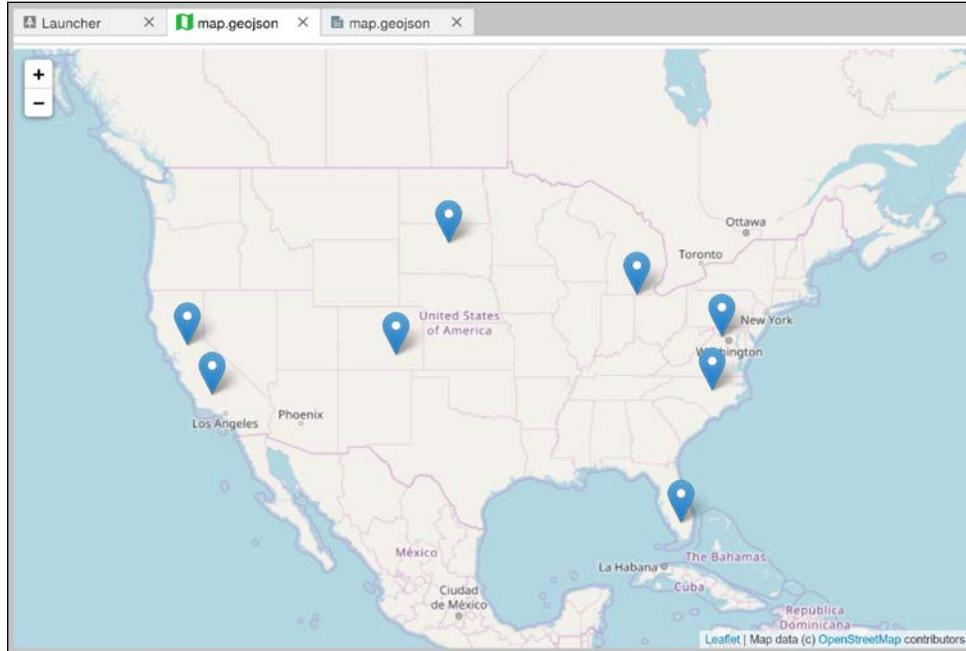
| col1 | col2 | col3 | col4 | col5 | col6 | col7 | col8 | col9 |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 0.350 | 0.002 | 0.402 | 0.344 | 0.345 | 0.365 | 0.255 | 0.005 | 0.335 |
| 0.851 | 0.176 | 0.045 | 0.917 | 0.357 | 0.659 | 0.162 | 0.033 | 0.683 |
| 0.818 | 0.647 | 0.209 | 0.752 | 0.921 | 0.068 | 0.850 | 0.813 | 0.211 |
| 0.959 | 0.914 | 0.211 | 0.874 | 0.379 | 0.558 | 0.778 | 0.714 | 0.930 |
| 0.347 | 0.130 | 0.607 | 0.113 | 0.760 | 0.347 | 0.124 | 0.809 | 0.028 |
| 0.371 | 0.820 | 0.096 | 0.871 | 0.990 | 0.774 | 0.310 | 0.892 | 0.655 |
| 0.694 | 0.340 | 0.334 | 0.759 | 0.454 | 0.385 | 0.301 | 0.765 | 0.699 |
| 0.693 | 0.484 | 0.971 | 0.519 | 0.502 | 0.304 | 0.643 | 0.844 | 0.309 |
| 0.606 | 0.210 | 0.186 | 0.405 | 0.948 | 0.481 | 0.108 | 0.216 | 0.640 |
| 0.519 | 0.782 | 0.488 | 0.485 | 0.956 | 0.905 | 0.679 | 0.044 | 0.672 |
| 0.816 | 0.400 | 0.435 | 0.096 | 0.801 | 0.155 | 0.934 | 0.889 | 0.181 |
| 0.480 | 0.420 | 0.889 | 0.660 | 0.919 | 0.025 | 0.752 | 0.547 | 0.137 |
| 0.919 | 0.720 | 0.413 | 0.883 | 0.394 | 0.547 | 0.329 | 0.290 | 0.241 |
| 0.758 | 0.583 | 0.054 | 0.378 | 0.620 | 0.650 | 0.210 | 0.830 | 0.535 |
| 0.593 | 0.297 | 0.020 | 0.699 | 0.408 | 0.275 | 0.659 | 0.762 | 0.506 |
| 0.697 | 0.853 | 0.979 | 0.198 | 0.984 | 0.542 | 0.224 | 0.602 | 0.108 |
| 0.177 | 0.989 | 0.811 | 0.101 | 0.586 | 0.874 | 0.441 | 0.391 | 0.267 |
| 0.311 | 0.082 | 0.030 | 0.147 | 0.975 | 0.212 | 0.729 | 0.361 | 0.071 |
| 0.064 | 0.274 | 0.750 | 0.576 | 0.336 | 0.316 | 0.044 | 0.106 | 0.577 |
| 0.586 | 0.417 | 0.807 | 0.547 | 0.293 | 0.108 | 0.334 | 0.245 | 0.321 |
| 0.252 | 0.794 | 0.729 | 0.015 | 0.371 | 0.869 | 0.946 | 0.063 | 0.286 |
| 0.279 | 0.008 | 0.511 | 0.838 | 0.663 | 0.805 | 0.799 | 0.737 | 0.989 |
| 0.967 | 0.751 | 0.782 | 0.495 | 0.103 | 0.117 | 0.926 | 0.718 | 0.423 |
| 0.954 | 0.966 | 0.828 | 0.934 | 0.495 | 0.861 | 0.863 | 0.736 | 0.153 |
| 0.809 | 0.733 | 0.066 | 0.518 | 0.515 | 0.148 | 0.377 | 0.368 | 0.643 |

Viewing a CSV file

9. GeoJSON files (files that contain geographic information) can also be edited or viewed with the Leaflet mapping library:



```
1 {  
2     "type": "FeatureCollection",  
3     "features": [  
4         {  
5             "type": "Feature",  
6             "properties": {},  
7             "geometry": {  
8                 "type": "Point",  
9                 "coordinates": [  
10                     -77.6953125,  
11                     39.095962936305476  
12                 ]  
13             }  
14         },  
15         {  
16             "type": "Feature",  
17             "properties": {},  
18             "geometry": {  
19                 "type": "Point",  
20                 "coordinates": [  
21                     -78.486328125,  
22                     35.60371874069731  
23                 ]  
24             }  
25         },  
26         {  
27             "type": "Feature",  
28             "properties": {},  
29             "geometry": {  
30                 "type": "Point",  
31                 "coordinates": [  
32                     -84.638671875,  
33                     41.705728515237524  
34             ]  
35         }  
36     ]  
37 }
```



There's more...

JupyterLab is fully extendable. In fact, its philosophy is that all existing features are implemented as plugins.

It is possible to work collaboratively on a notebook, as with Google Docs. This feature is still in active development at the time of writing.

Here are a few references:

- ▶ JupyterLab GitHub project at <https://github.com/jupyterlab/jupyterlab>
- ▶ Jupyter renderers at <https://github.com/jupyterlab/jupyter-renderers>
- ▶ Talk at PyData 2017, available at <https://channel9.msdn.com/Events/PyData/Seattle2017/BRK11>
- ▶ Talk at PLOTCON 2017, available at <https://www.youtube.com/watch?v=p7Hr54VhOp0>
- ▶ Talk at ESIP Tech, available at <https://www.youtube.com/watch?v=K1AsGeak51A>
- ▶ JupyterLab screencast at <https://www.youtube.com/watch?v=sf8PuLcijuA>
- ▶ Realtime collaboration and cloud storage for JupyterLab through Google Drive, at <https://github.com/jupyterlab/jupyterlab-google-drive>

See also

- ▶ The *Introducing IPython and the Jupyter Notebook* recipe in *Chapter 1, A Tour of Interactive Computing with Jupyter and IPython*

4

Profiling and Optimization

In this chapter, we will cover the following topics:

- ▶ Evaluating the time taken by a command in IPython
- ▶ Profiling your code easily with cProfile and IPython
- ▶ Profiling your code line-by-line with line_profiler
- ▶ Profiling the memory usage of your code with memory_profiler
- ▶ Understanding the internals of NumPy to avoid unnecessary array copying
- ▶ Using stride tricks with NumPy
- ▶ Implementing an efficient rolling average algorithm with stride tricks
- ▶ Processing large NumPy arrays with memory mapping
- ▶ Manipulating large arrays with HDF5

Introduction

Although Python is not generally considered one of the fastest languages (which is somewhat unfair), it is possible to achieve excellent performance with the appropriate methods. This is the objective of this chapter and the next. This chapter describes how to evaluate (**profile**) what makes a program slow, and how this information can be used to **optimize** the code and make it more efficient. The next chapter will deal with more advanced high-performance computing methods that should only be tackled when the methods described here are not sufficient.

The recipes of this chapter are organized into three parts:

- ▶ **Time and memory profiling:** Evaluating the performance of your code
- ▶ **NumPy optimization:** Using NumPy more efficiently, particularly with large arrays
- ▶ **Memory mapping with arrays:** Implementing memory mapping techniques for out-of-core computations on huge arrays

Evaluating the time taken by a command in IPython

The `%timeit` magic and the `%%timeit` cell magic (which applies to an entire code cell) allow us to quickly evaluate the time taken by one or several Python statements. The next recipes in this chapter will show methods for more extensive profiling.

How to do it...

We are going to estimate the time taken to calculate the sum of the inverse squares of all positive integer numbers up to a given `n`.

1. Let's define `n`:

```
>>> n = 100000
```

2. Let's time this computation in pure Python:

```
>>> %timeit sum([1. / i**2 for i in range(1, n)])
21.6 ms ± 343 µs per loop (mean ± std. dev. of 7 runs,
10 loops each)
```

3. Now, let's use the `%%timeit` cell magic to time the same computation written on two lines:

```
>>> %%timeit s = 0.
      for i in range(1, n):
          s += 1. / i**2
22 ms ± 522 µs per loop (mean ± std. dev. of 7 runs,
10 loops each)
```

4. Finally, let's time the NumPy version of this computation:

```
>>> import numpy as np
>>> %timeit np.sum(1. / np.arange(1., n) ** 2)
160 µs ± 959 ns per loop (mean ± std. dev. of 7 runs,
10000 loops each)
```

Here, the NumPy vectorized version is 137x faster than the pure Python version.

How it works...

The `%timeit` command accepts several optional parameters. One such parameter is the number of statement evaluations. By default, this number is chosen automatically so that the `%timeit` command returns within a few seconds in most cases. However, this number can be specified directly with the `-r` and `-n` parameters. Type `%timeit?` in IPython to get more information.

The `%%timeit` cell magic also accepts an optional setup statement in the first line (on the same line as `%%timeit`), which is executed but not timed. All variables created in this statement are available inside the cell.

There's more...

If you are not in an IPython interactive session or in a Jupyter Notebook, you can use `import timeit; timeit.timeit()`. This function benchmarks a Python statement stored in a string. IPython's `%timeit` magic command is a convenient wrapper around `timeit()`, useful in an interactive session. For more information on the `timeit` module, refer to <https://docs.python.org/3/library/timeit.html>.

See also

- ▶ The *Profiling your code easily with cProfile and IPython* recipe
- ▶ The *Profiling your code line-by-line with line_profiler* recipe

Profiling your code easily with cProfile and IPython

The `%timeit` magic command is often helpful, yet a bit limited when we need detailed information about what takes up most of the execution time. This magic command is meant for **benchmarking** (comparing the execution times of different versions of a function) rather than **profiling** (getting a detailed report of the execution time, function by function).

Python includes a profiler named **cProfile** that breaks down the execution time into the contributions of all called functions. IPython provides convenient ways to leverage this tool in an interactive session.

How to do it...

IPython offers the %prun line magic and the %%prun cell magic to easily profile one or multiple lines of code. The %run magic command also accepts a -p flag to run a Python script under the control of the profiler. These commands accept a lot of options as can be seen with %prun? and %run?.

In this example, we will profile a numerical simulation of random walks. We will cover these kinds of simulation in more detail in *Chapter 13, Stochastic Dynamical Systems*.

1. Let's import NumPy:

```
>>> import numpy as np
```

2. Let's create a function generating random +1 and -1 values in an array:

```
>>> def step(*shape):  
    # Create a random n-vector with +1 or -1 values.  
    return 2 * (np.random.random_sample(shape) < .5) - 1
```

3. Now, we write simulation code in a cell starting with %%prun in order to profile the entire simulation. The various options allow us to save the report in a file and to sort the first 10 results by cumulative time. We will explain these options in more detail in the *How it works...* section.

```
>>> %%prun -s cumulative -q -l 10 -T prun0  
    # We profile the cell, sort the report by "cumulative  
    # time", limit it to 10 lines, and save it to a file  
    # named "prun0".  
  
n = 10000  
iterations = 50  
x = np.cumsum(step(iterations, n), axis=0)  
bins = np.arange(-30, 30, 1)  
y = np.vstack([np.histogram(x[i,:], bins)[0]  
               for i in range(iterations)])  
*** Profile printout saved to text file 'prun0'.
```

4. The profiling report has been saved in a text file named prun0. Let's display it (the following output is a stripped down version that fits on this page):

```
>>> print(open('prun0', 'r').read())
```

```

3914 function calls in 0.027 seconds

Ordered by: cumulative time
List reduced from 49 to 10 due to restriction <10>

ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
    1    0.000    0.000    0.027    0.027 {built-in method builtins.exec}
    1    0.000    0.000    0.027    0.027 <string>:7(<module>)
    1    0.000    0.000    0.016    0.016 <string>:11(<listcomp>)
   50    0.002    0.000    0.016    0.000 function_base.py:431(histogram)
   50    0.000    0.000    0.010    0.000 fromnumeric.py:709(sort)
   50    0.009    0.000    0.009    0.000 {method 'sort' of 'numpy.ndarray' objects}
    1    0.002    0.002    0.008    0.008 <ipython-input-8-7b2aa0313928>:1(step)
    1    0.006    0.006    0.006    0.006 {method 'random_sample' of 'mtrand.RandomState' objects}
    1    0.000    0.000    0.002    0.002 fromnumeric.py:2033(cumsum)
    1    0.000    0.000    0.002    0.002 fromnumeric.py:55(_wrapfunc)

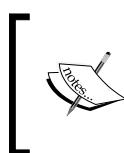
```

Here, we observe the time taken by the different functions involved, directly or indirectly, in our code.

How it works...

Python's profiler creates a detailed report of the execution time of our code, function by function. Here, we can observe the number of calls of the `histogram()`, `cumsum()`, `step()`, `sort()`, and `rand()` functions, and the total time spent by those functions during the code's execution. Internal functions are also profiled. For each function, we get the total number of calls, the total and cumulative times, and their per-call counterparts (division by `ncalls`). The **total time** represents how long the interpreter stays in a given function, *excluding* the time spent in calls to subfunctions. The **cumulative time** is similar but includes the time spent in calls to subfunctions. The filename, function name, and line number are displayed in the last column.

The `%prun` and `%%prun` magic commands accept multiple optional options (type `%prun?` for more details). In the example, `-s` allows us to sort the report by a particular column, `-q` to suppress (quell) the pager output (which is useful when we want to integrate the output in a notebook), `-l` to limit the number of lines displayed or to filter the results by function name (which is useful when we are interested in a particular function), and `-T` to save the report in a text file. In addition, we can choose to save (dump) the binary report in a file with `-D`, or to return it in IPython with `-r`. This database-like object contains all information about the profiling and can be analyzed through Python's `pstats` module.



Every profiler brings its own overhead that can bias the profiling results (**probe effect**). In other words, a profiled program may run significantly slower than a non-profiled program. That's a point to keep in mind.

"Premature optimization is the root of all evil"

As Donald Knuth's well-known quote suggests, optimizing code prematurely is generally considered a bad practice. Code optimization should only be conducted when it's really needed, that is, when the code is really too slow in normal situations. Additionally, we should know exactly where we need to optimize our code; typically, the vast majority of the execution time comprises a relatively small part of the code. The only way to find out is by profiling your code; optimization should never be done without preliminary profiling.



I was once dealing with some fairly complicated code that was slower than expected. I thought I had a pretty good idea of what was causing the problem and how I could resolve it. The solution would involve significant changes in the code. By profiling my code first, I discovered that my diagnosis was wrong: I had written somewhere `max(x)` instead of `np.max(x)` by mistake, where `x` was a huge vector. It was Python's built-in function that was called, instead of NumPy's heavily optimized routine for arrays. If I hadn't profiled my code, I would probably have missed this mistake forever. The program was working perfectly fine, only 150 times slower!

For more general advice on programming optimization, see https://en.wikipedia.org/wiki/Program_optimization.

There's more...

Profiling code in IPython is particularly simple (especially in the Notebook), as we have seen in this recipe. However, it may be undesirable or difficult to execute the code that we need to profile from IPython (GUIs, for example). In this case, we can use `cProfile` directly. It is slightly less straightforward than with IPython.

1. First, we call the following command:

```
$ python -m cProfile -o profresults myscript.py
```

The file `profresults` will contain the dump of the profiling results of `myscript.py`.

2. Then, we execute the following code from Python or IPython to display the profiling results in a human-readable form:

```
import pstats  
p = pstats.Stats('profresults')  
p.strip_dirs().sort_stats("cumulative").print_stats()
```

Explore the documentation for the `cProfile` and `pstats` modules to discover all of the analyses that you can perform on profiling reports.

There are a few GUI tools for exploring and visualizing the output of a profiling session. For example, **SnakeViz** allows you to view profile dumps in a GUI program.

Here are a few references:

- ▶ Documentation of `cProfile` and `pstats`, available at <https://docs.python.org/3/library/profile.html>
- ▶ SnakeViz, available at <https://jiffyclub.github.io/snakeviz/>
- ▶ Heat, a magic command to profile and view Python code as a heat map, at <https://github.com/csurfer/pyheatmagic>
- ▶ Python profiling tools, available at <http://blog.ionelmc.ro/2013/06/08/python-profiling-tools/>
- ▶ `accelerate.profiling` at <https://docs.anaconda.com/accelerate/profiling>

See also

- ▶ The *Profiling your code line-by-line with line_profiler* recipe

Profiling your code line-by-line with line_profiler

Python's native `cProfile` module and the corresponding `%prun` magic break down the execution time of code function by function. Sometimes, we may need an even more fine-grained analysis of code performance with a line-by-line report. Such reports can be easier to read than reports from `cProfile`.

To profile code line-by-line, we need an external Python module named `line_profiler`. In this recipe, we will demonstrate how to use this module within IPython.

Getting ready

To install `line_profiler`, type `conda install line_profiler` in a Terminal.

How do to it...

We will profile the same simulation code as in the previous recipe, line-by-line.

1. First, let's import NumPy and the `line_profiler` IPython extension module that comes with the package:

```
>>> import numpy as np  
>>> %load_ext line_profiler
```

2. This IPython extension module provides an `%lprun` magic command to profile a Python function line-by-line. It works best when the function is defined in a file and not in the interactive namespace or in the Notebook. Therefore, here we write our code in a Python script using the `%%writefile` cell magic:

```
>>> %%writefile simulation.py  
import numpy as np  
  
def step(*shape):  
    # Create a random n-vector with +1 or -1 values.  
    return 2 * (np.random.random_sample(shape) < .5) - 1  
  
def simulate(iterations, n=10000):  
    s = step(iterations, n)  
    x = np.cumsum(s, axis=0)  
    bins = np.arange(-30, 30, 1)  
    y = np.vstack([np.histogram(x[i,:], bins)[0]  
                  for i in range(iterations)])  
    return y
```

3. Now, let's import this script into the interactive namespace so that we can execute and profile our code:

```
>>> from simulation import simulate
```

4. We execute the function under the control of the line profiler. The functions to be profiled need to be explicitly specified in the `%lprun` magic command. We also save the report in a file named `lprof0`:

```
>>> %lprun -T lprof0 -f simulate simulate(50)  
*** Profile printout saved to text file 'lprof0'.
```

5. Let's display the report:

```
>>> print(open('lprof0', 'r').read())
```

```

Timer unit: 1e-06 s
Total time: 0.051297 s
Function: simulate at line 7

Line #    Hits         Time  Per Hit   % Time  Line Contents
=====
 7          1      25393  25393.0     49.5      def simulate(iterations, n=10000):
 8          1       1914   1914.0      3.7          s = step(iterations, n)
 9          1        22     22.0      0.0          x = np.cumsum(s, axis=0)
10          1        4      4.0      0.0          bins = np.arange(-30, 30, 1)
11          1       23962  23962.0     46.7          y = np.vstack([np.histogram(x[i,:], bins)[0]
12                                     for i in range(iterations)])
13          1        2     2.0      0.0
                                         return y

```

How it works...

The `%lprun` command accepts a Python statement as its main argument. The functions to profile need to be explicitly specified with `-f`. Other optional arguments include `-D`, `-T`, and `-r`, and they work in a similar way to their `%prun` magic command counterparts.

The `line_profiler` module displays the time spent on each line of the profiled functions, either in timer units or as a fraction of the total execution time. These details are essential when we are looking for hotspots in our code.

There's more...

Tracing is a related method. Python's `trace` module allows us to trace program execution of Python code. That's particularly useful during in-depth debugging and profiling sessions. We can follow the entire sequence of instructions executed by the Python interpreter. More information on the `trace` module is available at <https://docs.python.org/3/library/trace.html>.

In addition, the Online Python Tutor is an online interactive educational tool that can help us understand what the Python interpreter is doing step-by-step as it executes a program's source code. The Online Python Tutor is available at <http://pythontutor.com/>.

Here are a few references:

- ▶ The `line_profiler` repository at https://github.com/rkern/line_profiler

See also

- ▶ The *Profiling your code easily with cProfile and IPython* recipe
- ▶ The *Profiling the memory usage of your code with memory_profiler* recipe

Profiling the memory usage of your code with `memory_profiler`

The methods described in the previous recipe were about CPU time profiling. That may be the most obvious factor when it comes to code profiling. However, memory is also a critical factor. Writing memory-optimized code is not trivial and can really make your program faster. This is particularly important when dealing with large NumPy arrays, as we will see later in this chapter.

In this recipe, we will look at a simple memory profiler unsurprisingly named `memory_profiler`. Its usage is very similar to `line_profiler`, and it can be conveniently used from IPython.

Getting ready

You can install `memory_profiler` with `conda install memory_profiler`.

How to do it...

1. We load the `memory_profiler` IPython extension:

```
>>> %load_ext memory_profiler
```

2. We define a function that allocates big objects:

```
>>> %%writefile memscript.py
def my_func():
    a = [1] * 1000000
    b = [2] * 9000000
    del b
    return a
```

3. Now, let's run the code under the control of the memory profiler:

```
>>> from memscript import my_func
%mprun -T mprof0 -f my_func my_func()
*** Profile printout saved to text file mprof0.
```

4. Let's show the results:

```
>>> print(open('mprof0', 'r').read())
Line #  Mem usage  Increment  Line Contents
=====
1      93.4 MiB    0.0 MiB  def my_func():
2     100.9 MiB    7.5 MiB      a = [1] * 1000000
3     169.7 MiB   68.8 MiB      b = [2] * 9000000
4     101.1 MiB   -68.6 MiB     del b
5     101.1 MiB    0.0 MiB     return a
```

We can observe line after line the allocation and deallocation of objects.

How it works...

The `memory_profiler` package checks the memory usage of the interpreter at every line. The increment column allows us to spot those places in the code where large amounts of memory are allocated. This is especially important when working with arrays. Unnecessary array creations and copies can considerably slow down a program. We will tackle this issue in the next few recipes.

There's more...

The `memory_profiler` IPython extension also comes with a `%memit` magic command that lets us benchmark the memory used by a single Python statement. Here is a simple example:

```
>>> %%memit import numpy as np
        np.random.randn(1000000)
peak memory: 101.20 MiB, increment: 7.77 MiB
```

The `memory_profiler` package offers other ways to profile the memory usage of a Python program, including plotting the memory usage as a function of time. For more details, refer to the documentation at https://github.com/pythonprofilers/memory_profiler.

See also

- ▶ The *Profiling your code line-by-line with line_profiler* recipe
- ▶ The *Understanding the internals of NumPy to avoid unnecessary array copying* recipe

Understanding the internals of NumPy to avoid unnecessary array copying

We can achieve significant performance speed enhancement with NumPy over native Python code, particularly when our computations follow the **Single Instruction, Multiple Data (SIMD)** paradigm. However, it is also possible to unintentionally write non-optimized code with NumPy.

In the next few recipes, we will see some tricks that can help us write optimized NumPy code. In this recipe, we will see how to avoid unnecessary array copies in order to save memory. In that respect, we will need to dig into the internals of NumPy.

Getting ready

First, we need a way to check whether two arrays share the same underlying data buffer in memory. Let's define a function `aid()` that returns the memory location of the underlying data buffer:

```
>>> import numpy as np
>>> def aid(x):
    # This function returns the memory
    # block address of an array.
    return x.__array_interface__['data'][0]
```

Two arrays with the same data location (as returned by `aid()`) share the same underlying data buffer. However, the opposite is true only if the arrays have the same **offset** (meaning that they have the same first element). Two shared arrays with different offsets will have slightly different memory locations, as shown in the following example:

```
>>> a = np.zeros(3)
      aid(a), aid(a[1:])
(21535472, 21535480)
```

In the next few recipes, we'll make sure to use this method with arrays that have the same offset. Here is a more general and reliable solution for finding out whether two arrays share the same data:

```
>>> def get_data_base(arr):
    """For a given NumPy array, find the base array
    that owns the actual data."""
    base = arr
    while isinstance(base.base, np.ndarray):
        base = base.base
    return base

def arrays_share_data(x, y):
    return get_data_base(x) is get_data_base(y)
>>> print(arrays_share_data(a, a.copy()))
False
>>> print(arrays_share_data(a, a[:1]))
True
```

Thanks to *Michael Droettboom* for pointing this out and proposing this alternative solution.

How to do it...

Computations with NumPy arrays may involve internal copies between blocks of memory. These copies are not always necessary, in which case they should be avoided, as we will see in the following tips.

1. We may sometimes need to make a copy of an array; for instance, if we need to manipulate an array while keeping an original copy in memory:

```
>>> import numpy as np
      a = np.zeros(10)
      ax = aid(a)
      ax
      32250112
>>> b = a.copy()
      aid(b) == ax
      False
```

2. Array computations can involve in-place operations (the first example in the following code: the array is modified) or implicit-copy operations (the second example: a new array is created):

```
>>> a *= 2
      aid(a) == ax
      True
>>> c = a * 2
      aid(c) == ax
      False
Implicit-copy operations are slower, as shown here:
>>> %%timeit a = np.zeros(10000000)
      a *= 2
      4.85 ms ± 24 µs per loop (mean ± std. dev. of 7 runs,
      100 loops each)
>>> %%timeit a = np.zeros(10000000)
      b = a * 2
      7.7 ms ± 105 µs per loop (mean ± std. dev. of 7 runs,
      100 loops each)
```

3. Reshaping an array may or may not involve a copy. The reasons will be explained in the *How it works...* section of this recipe. For instance, reshaping a 2D matrix does not involve a copy, unless it is transposed (or more generally, **non-contiguous**):

```
>>> a = np.zeros((100, 100))
      ax = aid(a)
>>> b = a.reshape((1, -1))
      aid(b) == ax
      True
```

```
>>> c = a.T.reshape((1, -1))
      aid(c) == ax
      False
```

Therefore, the latter instruction is significantly slower than the former:

```
>>> %timeit b = a.reshape((1, -1))
330 ns ± 0.517 ns per loop (mean ± std. dev. of 7 runs
   1000000 loops each)
>>> %timeit a.T.reshape((1, -1))
5 µs ± 5.68 ns per loop (mean ± std. dev. of 7 runs,
   100000 loops each)
```

4. Both the `flatten()` and the `ravel()` methods of an array reshape it into a 1D vector (a flattened array). However, the `flatten()` method always returns a copy, and the `ravel()` method returns a copy only if necessary (thus it's faster, especially with large arrays).

```
>>> d = a.flatten()
      aid(d) == ax
      False
>>> e = a.ravel()
      aid(e) == ax
      True
>>> %timeit a.flatten()
2.3 µs ± 18.1 ns per loop (mean ± std. dev. of 7 runs,
   100000 loops each)
>>> %timeit a.ravel()
199 ns ± 5.02 ns per loop (mean ± std. dev. of 7 runs,
   10000000 loops each)
```

5. **Broadcasting rules** allow us to make computations on arrays with different but compatible shapes. In other words, we don't always need to reshape or tile our arrays to make their shapes match. The following example illustrates two ways of doing an **outer product** between two vectors: the first method involves array tiling, the second one (faster) involves broadcasting:

```
>>> n = 1000
>>> a = np.arange(n)
      ac = a[:, np.newaxis] # column vector
      ar = a[np.newaxis, :] # row vector
>>> %timeit np.tile(ac, (1, n)) * np.tile(ar, (n, 1))
5.7 ms ± 42.6 µs per loop (mean ± std. dev. of 7 runs,
   100 loops each)
>>> %timeit ar * ac
784 µs ± 2.39 µs per loop (mean ± std. dev. of 7 runs,
   1000 loops each)
```

How it works...

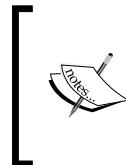
In this section, we will see what happens under-the-hood when using NumPy, and how this knowledge allows us to understand the tricks given in this recipe.

Why are NumPy arrays efficient?

A NumPy array is basically described by metadata (notably the number of dimensions, the shape, and the data type) and the actual data. The data is stored in a homogeneous and contiguous block of memory, at a particular address in system memory (**Random Access Memory (RAM)**). This block of memory is called the **data buffer**. This is the main difference between an array and a pure Python structure, such as a list, where the items are scattered across the system memory. This aspect is the critical feature that makes NumPy arrays so efficient.

Why is this so important? Here are the main reasons:

- ▶ Computations on arrays can be written very efficiently in a low-level language such as C (and a large part of NumPy is actually written in C). Knowing the address of the memory block and the data type, it is just simple arithmetic to loop over all items, for example. There would be a significant overhead if we did that in Python with a list.
- ▶ **Spatial locality** in memory access patterns results in performance gains notably due to the CPU cache. Indeed, the cache loads bytes in chunks from the RAM to the CPU registers. Adjacent items are then loaded very efficiently (**sequential locality**, or **locality of reference**).
- ▶ Finally, the fact that items are stored contiguously in memory allows NumPy to take advantage of **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.



Additionally, NumPy can be linked to highly optimized linear algebra libraries such as **BLAS** and **LAPACK** through **ATLAS** or the **Intel Math Kernel Library (MKL)**. A few specific matrix computations may also be multithreaded, taking advantage of the power of modern multicore processors.

In conclusion, storing data in a contiguous block of memory ensures that the architecture of modern CPUs is used optimally, in terms of memory access patterns, CPU cache, and vectorized instructions.

What is the difference between in-place and implicit-copy operations?

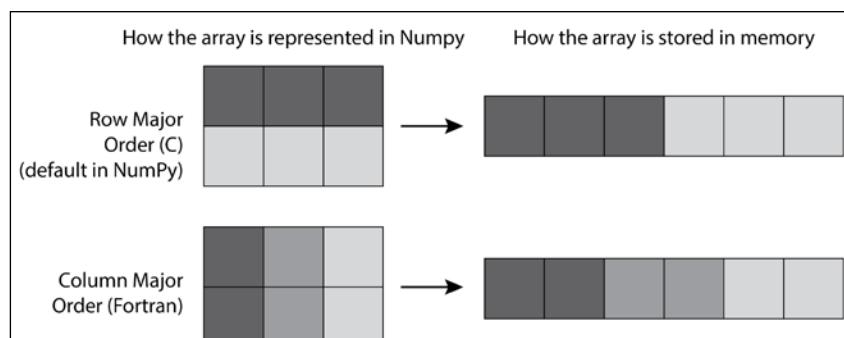
Let's explain the example in step 2. An expression such as `a *= 2` corresponds to an in-place operation, where all values of the array are multiplied by two. By contrast, `a = a*2` means that a new array containing the values of `a*2` is created, and the variable `a` now points to this new array. The old array becomes unreferenced and will be deleted by the garbage collector. No memory allocation happens in the first case, unlike the second case.

More generally, expressions such as `a[i:j]` are **views** to parts of an array; they point to the memory buffer containing the data. Modifying them with in-place operations changes the original array.

Knowing this subtlety of NumPy can help you fix some bugs (where an array is implicitly and unintentionally modified because of an operation on a view), and optimize the speed and memory consumption of your code by reducing the number of unnecessary copies.

Why can't some arrays be reshaped without a copy?

We explain the example in step 3 here, where a transposed 2D matrix cannot be flattened without a copy. A 2D matrix contains items indexed by two numbers (row and column), but it is stored internally as a 1D contiguous block of memory, accessible with a single number. There is more than one way of storing matrix items in a 1D block of memory: we can put the elements of the first row first, then the second row, and so on, or the elements of the first column first, then the second column, and so on. The first method is called **row-major order**, whereas the latter is called **column-major order**. Choosing between the two methods is only a matter of internal convention: NumPy uses row-major order, like C, but unlike FORTRAN.



Internal array layouts: row-major and column-major orders

More generally, NumPy uses the notion of **strides** to convert between a multidimensional index and the memory location of the underlying (1D) sequence of elements. The specific mapping between `array[i1, i2]` and the relevant byte address of the internal data is given by:

```
offset = array.strides[0] * i1 + array.strides[1] * i2
```

When reshaping an array, NumPy avoids copies when possible by modifying the **strides** attribute. For example, when transposing a matrix, the order of strides is reversed, but the underlying data remains identical. However, flattening a transposed array cannot be accomplished simply by modifying strides, so a copy is needed.

Internal array layout can also explain some unexpected performance discrepancies between very similar NumPy operations. As a small exercise, can you explain the following benchmarks?

```
>>> a = np.random.rand(5000, 5000)
>>> %timeit a[0, :].sum()
2.91 µs ± 20 ns per loop (mean ± std. dev. of 7 runs,
    100000 loops each)
>>> %timeit a[:, 0].sum()
33.7 µs ± 22.7 ns per loop (mean ± std. dev. of 7 runs
    10000 loops each)
```

What are NumPy broadcasting rules?

Broadcasting rules describe how arrays with different dimensions and/or shapes can be used for computations. The general rule is that *two dimensions are compatible when they are equal, or when one of them is 1*. NumPy uses this rule to compare the shapes of the two arrays element-wise, starting with the trailing dimensions and working its way forward. The smallest dimension is internally stretched to match the other dimension, but this operation does not involve any memory copy.

There's more...

Here are a few references:

- ▶ Broadcasting rules and examples, available at <http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html>
- ▶ Array interface in NumPy, at <http://docs.scipy.org/doc/numpy/reference/arrays.interface.html>
- ▶ Locality of reference, at https://en.wikipedia.org/wiki/Locality_of_reference
- ▶ Internals of NumPy in the SciPy lectures notes, available at http://scipy-lectures.github.io/advanced/advanced_numpy/
- ▶ 100 NumPy exercises by Nicolas Rougier, available at [http://www.loria.fr/~rougier/teaching\(numpy.100/index.html](http://www.loria.fr/~rougier/teaching(numpy.100/index.html)

See also

- ▶ The *Using stride tricks with NumPy* recipe

Using stride tricks with NumPy

In this recipe, we will dig deeper into the internals of NumPy arrays, by generalizing the notion of row-major and column-major orders to multidimensional arrays. The general notion is that of strides, which describe how the items of a multidimensional array are organized within a one-dimensional data buffer. Strides are mostly an implementation detail, but they can also be used in specific situations to optimize some algorithms.

Getting ready

We suppose that NumPy has been imported and that the `aid()` function has been defined (refer to the *Understanding the internals of NumPy to avoid unnecessary array copying* recipe).

```
>>> import numpy as np
>>> def aid(x):
    # This function returns the memory
    # block address of an array.
    return x.__array_interface__['data'][0]
```

How to do it...

1. Strides are integer numbers describing the byte step in the contiguous block of memory for each dimension.

```
>>> x = np.zeros(10)
      x.strides
(8,)
```

This vector `x` contains double-precision floating point numbers (`float64`, 8 bytes); you need to go 8 bytes *forward* to go from one item to the next.

2. Now, let's look at the strides of a 2D array:

```
>>> y = np.zeros((10, 10))
      y.strides
(80, 8)
```

In the first dimension (vertical), you need to go 80 bytes (10 `float64` items) *forward* to go from one item to the next, because the items are internally stored in row-major order. In the second dimension (horizontal), you need to go 8 bytes *forward* to go from one item to the next.

3. Let's show how we can revisit the broadcasting rules from the previous recipe using strides:

```
>>> n = 1000
      a = np.arange(n)
```

We will create a new array, `b`, pointing to the same memory block as `a`, but with a different shape and different strides. This new array will look like a vertically-tiled version of `a`. We use a special function in NumPy to change the strides of an array:

```
>>> b = np.lib.stride_tricks.as_strided(a, (n, n), (0, 8))
>>> b
array([[ 0,  1,  2, ..., 997, 998, 999],
       [ 0,  1,  2, ..., 997, 998, 999],
       [ 0,  1,  2, ..., 997, 998, 999],
       ...,
       [ 0,  1,  2, ..., 997, 998, 999],
       [ 0,  1,  2, ..., 997, 998, 999],
       [ 0,  1,  2, ..., 997, 998, 999]])
>>> b.size, b.shape, b nbytes
(1000000, (1000, 1000), 8000000)
```

NumPy believes that this array contains one million different elements, whereas the data buffer actually contains the same 1,000 elements as `a`.

4. We can now perform an efficient outer product using the same principle as with broadcasting rules:

```
>>> %timeit b * b.T
766 µs ± 2.59 µs per loop (mean ± std. dev. of 7 runs,
1000 loops each)
>>> %%timeit
    np.tile(a, (n, 1)) * np.tile(a[:, np.newaxis], (1, n))
5.55 ms ± 9.1 µs per loop (mean ± std. dev. of 7 runs,
100 loops each)
```

How it works...

Every array has a number of dimensions, a shape, a data type, and strides. Strides describe how the items of a multidimensional array are organized in the data buffer. There are many different schemes for arranging the items of a multidimensional array in a one-dimensional block. NumPy implements a **strided indexing scheme**, where the position of any element is a **linear combination** of the dimensions, the coefficients being the strides. In other words, strides describe, in any dimension, how many bytes we need to jump over in the data buffer to go from one item to the next.

The position of any element in a multidimensional array is given by a linear combination of its indices, as follows:

| | |
|--|---|
| an array $\text{ndim} = N$ $\text{shape} = (d_0, \dots, d_{N-1})$ $\text{strides} = (s_0, \dots, s_{N-1})$ $\text{itemsize} = \alpha$ | one item $\text{indices} = (i_0, \dots, i_{N-1})$ $\text{position} = \sum_{k=0}^{N-1} s_k i_k$ |
| examples: column-major and row-major orders | |
| $s_k^{\text{column}} = \prod_{j=0}^{k-1} \alpha d_j, \quad s_k^{\text{row}} = \prod_{j=k+1}^{N-1} \alpha d_j$ | |

Strides

Artificially changing the strides allows us to make some array operations more efficient than with standard methods, which may involve array copies. Internally, that's how broadcasting works in NumPy.

The `as_strided()` method takes an array, a shape, and strides as arguments. It creates a new array, but uses the same data buffer as the original array. The only thing that changes is the metadata. This trick lets us manipulate NumPy arrays as usual, except that they may take much less memory than what NumPy thinks. Here, using 0 in the strides implies that any array item can be addressed by many multidimensional indices, resulting in memory savings.

 Be very careful with strided arrays! The `as_strided()` function does not check whether you stay inside the memory block bounds. This means that you need to handle edge effects manually; otherwise, you may end up with garbage values in your arrays. The documentation says: "This function has to be used with extreme care, see notes. (...) It is advisable to avoid `as_strided()` when possible."

We will see a more useful application of stride tricks in the next recipe.

See also

- ▶ The *Implementing an efficient rolling average algorithm with stride tricks* recipe

Implementing an efficient rolling average algorithm with stride tricks

Stride tricks can be useful for local computations on arrays, when the computed value at a given position depends on the neighboring values. Examples include dynamical systems, digital filters, and cellular automata.

In this recipe, we will implement an efficient **rolling average** algorithm (a particular type of convolution-based linear filter) with NumPy stride tricks. A rolling average of a 1D vector contains, at each position, the average of the elements around this position in the original vector. Roughly speaking, this process filters out the noisy components of a signal so as to keep only the slower components.

How to do it...

The idea is to start from a 1D vector, and make a *virtual* 2D array where each line is a shifted version of the previous line. When using stride tricks, this process is very efficient as it does not involve any copy.

1. Let's generate a 1D vector:

```
>>> import numpy as np
      from numpy.lib.stride_tricks import as_strided
>>> def aid(x):
        # This function returns the memory
        # block address of an array.
        return x.__array_interface__['data'][0]
>>> n = 5
      k = 2
      a = np.linspace(1, n, n)
      ax = aid(a)
```

2. Let's change the strides of `a` to add shifted rows:

```
>>> as_strided(a, (k, n), (8, 8))
array([[ 1e+000,  2e+000,  3e+000,  4e+000,  5e+000],
       [ 2e+000,  3e+000,  4e+000,  5e+000,  9e-321]])
```

The last value indicates an out-of-bounds problem: stride tricks can be dangerous as memory access is not checked. Here, we should take edge effects into account by limiting the shape of the array.

3. Now, let's implement the computation of the rolling average. The first version (standard method) involves explicit array copies, whereas the second version uses stride tricks:

```
>>> def shift1(x, k):  
        return np.vstack([x[i:n - k + i + 1]  
                         for i in range(k)])  
>>> def shift2(x, k):  
        return as_strided(x, (k, n - k + 1),  
                           (x.itemsize, x.itemsize))
```

4. These two functions return the same result, except that the array returned by the second function refers to the original data buffer:

```
>>> b = shift1(a, k)  
>>> b  
array([[ 1.,  2.,  3.,  4.],  
       [ 2.,  3.,  4.,  5.]])  
>>> aid(b) == ax  
False  
And now with the second function:  
>>> c = shift2(a, k)  
>>> c  
array([[ 1.,  2.,  3.,  4.],  
       [ 2.,  3.,  4.,  5.]])  
>>> aid(c) == ax  
True
```

5. Let's generate a signal:

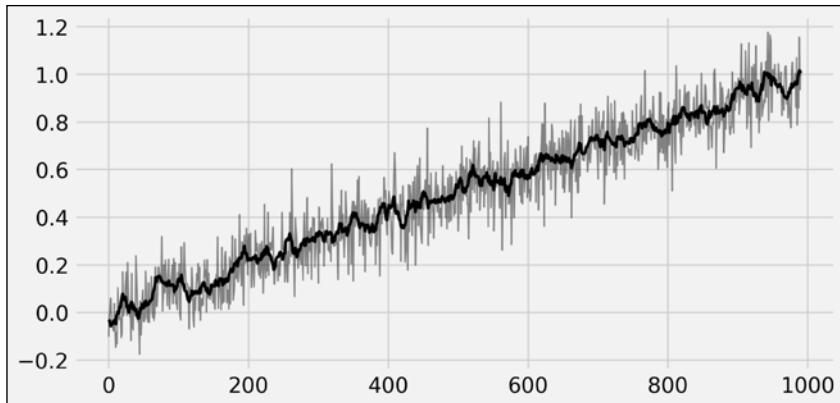
```
>>> n, k = 1000, 10  
      t = np.linspace(0., 1., n)  
      x = t + .1 * np.random.randn(n)
```

6. We compute the signal rolling average by creating the shifted version of the signal, and averaging along the vertical dimension:

```
>>> y = shift2(x, k)  
      x_avg = y.mean(axis=0)
```

7. Let's plot these arrays:

```
>>> import matplotlib.pyplot as plt  
      %matplotlib inline  
>>> fig, ax = plt.subplots(1, 1, figsize=(8, 4))  
      ax.plot(x[:-k + 1], '-k', lw=1, alpha=.5)  
      ax.plot(x_avg, '-k', lw=2)
```



8. Let's evaluate the time taken by the first method:

```
>>> %timeit shift1(x, k)
15.4 µs ± 302 ns per loop (mean ± std. dev. of 7 runs,
    100000 loops each)
>>> %%timeit y = shift1(x, k)
z = y.mean(axis=0)
10.3 µs ± 123 ns per loop (mean ± std. dev. of 7 runs,
    100000 loops each)
```

Here, most of the total time is spent in the array copy (the `shift1()` function).

9. Let's benchmark the second method:

```
>>> %timeit shift2(x, k)
4.77 µs ± 70.3 ns per loop (mean ± std. dev. of 7 runs,
    100000 loops each)
>>> %%timeit y = shift2(x, k)
z = y.mean(axis=0)
9 µs ± 179 ns per loop (mean ± std. dev. of 7 runs,
    100000 loops each)
```

This time, thanks to the stride tricks, most of the time is instead spent in the computation of the average.

See also

- ▶ The *Using stride tricks with NumPy* recipe

Processing large NumPy arrays with memory mapping

Sometimes, we need to deal with NumPy arrays that are too big to fit in the system memory. A common solution is to use **memory mapping** and implement **out-of-core computations**. The array is stored in a file on the hard drive, and we create a memory-mapped object to this file that can be used as a regular NumPy array. Accessing a portion of the array results in the corresponding data being automatically fetched from the hard drive. Therefore, we only consume what we use.

How to do it...

1. Let's create a memory-mapped array in write mode:

```
>>> import numpy as np
>>> nrows, ncols = 1000000, 100
>>> f = np.memmap('memmapped.dat', dtype=np.float32,
                  mode='w+', shape=(nrows, ncols))
```

2. Let's feed the array with random values, one column at a time because our system's memory is limited!

```
>>> for i in range(ncols):
    f[:, i] = np.random.rand(nrows)
```

We save the last column of the array:

```
>>> x = f[:, -1]
```

3. Now, we flush memory changes to disk by deleting the object:

```
>>> del f
```

4. Reading a memory-mapped array from disk involves the same `memmap()` function. The data type and the shape need to be specified again, as this information is not stored in the file:

```
>>> f = np.memmap('memmapped.dat', dtype=np.float32,
                  shape=(nrows, ncols))
>>> np.array_equal(f[:, -1], x)
True
>>> del f
```



This method is not adapted for long-term storage of data and data sharing. The following recipe in this chapter will show a better way based on the HDF5 file format.

How it works...

Memory mapping lets you work with huge arrays almost as if they were regular arrays. Python code that accepts a NumPy array as input will also accept a `memmap` array. However, we need to ensure that the array is used efficiently. That is, the array is never loaded as a whole (otherwise, it would waste system memory and would obviate any advantage of the technique).

Memory mapping is also useful when you have a huge file containing raw data in a homogeneous binary format with a known data type and shape. In this case, an alternative solution is to use NumPy's `fromfile()` function with a file handle created with Python's native `open()` function. Using `f.seek()` lets you position the cursor at any location and load a given number of bytes into a NumPy array.

There's more...

Another way of dealing with huge NumPy matrices is to use **sparse matrices** through SciPy's `sparse` subpackage. It is adapted to when matrices contain mostly zeros, as is often the case with simulations of partial differential equations, graph algorithms, or specific machine learning applications. Representing matrices as dense structures can be a waste of memory, and sparse matrices offer a more efficient compressed representation.

Using sparse matrices in SciPy is not straightforward as multiple implementations exist. Each implementation is best for a particular kind of application. Here are a few references:

- ▶ SciPy lecture notes about sparse matrices, available at http://scipy-lectures.github.io/advanced/scipy_sparse/index.html
- ▶ Reference documentation on sparse matrices, at <http://docs.scipy.org/doc/scipy/reference/sparse.html>
- ▶ Documentation of `memmap`, at <http://docs.scipy.org/doc/numpy/reference/generated/numpy.memmap.html>

See also

- ▶ The *Manipulating large arrays with HDF5* recipe
- ▶ The *Performing out-of-core computations on large arrays with Dask* recipe in Chapter 5, *High-Performance Computing*

Manipulating large arrays with HDF5

NumPy arrays can be persistently saved on disk using built-in functions in NumPy such as `np.savetxt()`, `np.save()`, or `np.savez()`, and loaded in memory using analogous functions. Common file formats for data arrays include raw binary files as in the previous recipe, the NPY file format implemented by NumPy (raw binary files with a header containing the metadata), and **Hierarchical Data Format (HDF5)**.

An HDF5 file contains one or several datasets (arrays or heterogeneous tables) organized into a POSIX-like hierarchy. Datasets may be accessed lazily with memory mapping. In this recipe, we will use `h5py`, a Python package designed to deal with HDF5 files with a NumPy-like programming interface.

Getting ready

You need `h5py` for this recipe and the next one. It should be included with Anaconda, but you can also install it with `conda install h5py`.

How to do it...

1. First, we need to import NumPy and `h5py`:

```
>>> import numpy as np  
      import h5py
```

2. Let's create a new empty HDF5 file in write mode:

```
>>> f = h5py.File('myfile.h5', 'w')
```

3. We create a new top-level group named `experiment1`:

```
>>> f.create_group('/experiment1')  
<HDF5 group "/experiment1" (0 members)>
```

4. Let's also add some metadata to this group:

```
>>> f['/experiment1'].attrs['date'] = '2018-01-01'
```

5. In this group, we create a 1000×1000 array named `array1`:

```
>>> x = np.random.rand(1000, 1000)  
f['/experiment1'].create_dataset('array1', data=x)  
<HDF5 dataset "array1": shape (1000, 1000), type "<f8">
```

6. Finally, we need to close the file to commit the changes to disk:

```
>>> f.close()
```

7. Now, let's open this file in read mode. We could have done this in another Python session since the array has been saved in the HDF5 file.

```
>>> f = h5py.File('myfile.h5', 'r')
```

8. We can retrieve an attribute by giving the group path and the attribute name:

```
>>> f['/experiment1'].attrs['date']
'2018-01-01'
```

9. Let's access our array:

```
>>> y = f['/experiment1/array1']
      type(y)
h5py._hl.dataset.Dataset
```

10. The array can be used as a NumPy array, but an important distinction is that it is stored on disk instead of system memory. Performing a computation on this array automatically loads the requested section of the array into memory, thus it is more efficient to access only the array's views.

```
>>> np.array_equal(x[0, :], y[0, :])
True
```

11. We're done for this recipe, so let's do some clean-up:

```
>>> f.close()
>>> import os
      os.remove('myfile.h5')
```

How it works...

In this recipe, we stored a single array in the file, but HDF5 is especially useful when many arrays need to be saved in a single file. HDF5 is generally used in big projects, when large arrays have to be organized within a hierarchical structure. For example, it is largely used at NASA and other scientific institutions. Researchers can store recorded data across multiple devices, multiple trials, and multiple experiments.

In HDF5, the data is organized within a tree. Nodes are either **groups** (analogous to folders in a file system) or **datasets** (analogous to files). A group can contain subgroups and datasets, whereas datasets only contain data. Both groups and datasets can contain attributes (metadata) that have a basic data type (integer or floating point number, string, and so on).

There's more...

HDF5 files created with `h5py` can be accessed in other languages such as C, FORTRAN, MATLAB, and others.

In HDF5, a dataset may be stored in a **contiguous** block of memory, or in **chunks**. Chunks are atomic objects and HDF5 can only read and write entire chunks. Chunks are internally organized within a tree data structure called a **B-tree**. When we create a new array or table, we can specify the **chunk shape**. It is an internal detail, but it can greatly affect performance when writing and reading parts of the dataset.

The optimal chunk shape depends on how we plan to access the data. There is a trade-off between many small chunks (large overhead due to managing lots of chunks) and a few large chunks (inefficient disk I/O). In general, the chunk size is recommended to be smaller than 1 MB. The chunk cache is also an important parameter that may affect performance.



Another HDF5 library in Python is PyTables. There is work in progress to make the two libraries share more code and reduce the duplication of development efforts.



Here are a few references:

- ▶ NPY file format at <https://docs.scipy.org/doc/numpy-dev/neps/npy-format.html>
- ▶ h5py at <http://www.h5py.org/>
- ▶ HDF5 chunking, at <http://www.hdfgroup.org/HDF5/doc/Advanced/Chunking/>
- ▶ Python and HDF5, a vision, <https://www.hdfgroup.org/2015/09/python-hdf5-a-vision/>
- ▶ PyTables optimization guide, available at <http://pytables.github.io/usersguide/optimization.html>
- ▶ Difference between PyTables and h5py, from the perspective of h5py, at <https://github.com/h5py/h5py/wiki/FAQ#whats-the-difference-between-h5py-and-ptables>
- ▶ A personal piece about the limitations of HDF5, at <http://cyrille.rossant.net/moving-away-hdf5/>

See also

- ▶ Processing huge NumPy arrays with memory mapping
- ▶ Manipulating large heterogeneous tables with HDF5
- ▶ The *Ten tips for conducting reproducible interactive computing experiments* recipe

5

High-Performance Computing

In this chapter, we will cover the following topics:

- ▶ Using Python to write faster code
- ▶ Accelerating pure Python code with Numba and Just-In-Time compilation
- ▶ Accelerating array computations with NumExpr
- ▶ Wrapping a C library in Python with ctypes
- ▶ Accelerating Python code with Cython
- ▶ Optimizing Cython code by writing less Python and more C
- ▶ Releasing the GIL to take advantage of multi-core processors with Cython and OpenMP
- ▶ Writing massively parallel code for NVIDIA graphics cards (GPUs) with CUDA
- ▶ Distributing Python code across multiple cores with IPython
- ▶ Interacting with asynchronous parallel tasks in IPython
- ▶ Performing out-of-core computations on large arrays with Dask
- ▶ Trying the Julia programming language in the Jupyter Notebook

Introduction

The previous chapter presented techniques for code optimization. Sometimes, these methods are not sufficient, and we need to resort to advanced high-performance computing techniques.

In this chapter, we will see three broad, but not mutually exclusive, categories of methods:

- ▶ **Just-In-Time (JIT)** compilation of Python code
- ▶ Resorting to a lower-level language, such as C, from Python
- ▶ Dispatching tasks across multiple computing units using parallel computing

With JIT compilation, Python code is dynamically compiled into a lower-level language.

Compilation occurs at runtime rather than ahead of execution. The translated code runs faster since it is compiled rather than interpreted. JIT compilation is a popular technique as it can lead to fast and high-level languages, whereas these two characteristics used to be mutually exclusive in general.

JIT compilation techniques are implemented in packages such as **Numba** or **NumExpr**, which we will cover in this chapter.

We will also use Julia, a programming language that uses JIT compilation to achieve high performance. This language can be used effectively in the Jupyter Notebook, thanks to the `IJulia` package.

 **PyPy** (<http://pypy.org>), successor to Psyco, is another related project. This alternative implementation of Python (the reference implementation being CPython) integrates a JIT compiler. Thus, it is typically faster than CPython. Since October 2017, PyPy supports NumPy and Pandas (but with Legacy Python rather than Python 3). See <https://morepypy.blogspot.fr/2017/10/pypy-v59-released-now-supports-pandas.html> for more details.

Resorting to a lower-level language such as C is another interesting method. Popular libraries include `ctypes` and `Cython`. Using `ctypes` requires writing C code and having access to a C compiler, or using a compiled C library. By contrast, `Cython` lets us write code in a superset of Python, which is translated to C with various performance results. In this chapter, we will cover `ctypes` and `Cython`, and we will see how to achieve interesting speed enhancement on relatively complex examples.

Finally, we will cover two classes of parallel computing techniques: using multiple CPU cores with IPython and using massively parallel architectures such as **Graphics Processing Units (GPUs)**.

Here are a few references:

- ▶ Interfacing Python with C, a tutorial in the scikit-learn lectures notes available at http://scipy-lectures.github.io/advanced/interfacing_with_c/interfacing_with_c.html
- ▶ *Extending Python with C or C++*, at <https://docs.python.org/3.6/extending/extending.html>
- ▶ xtensor, a NumPy-like library in C++ at <http://quantstack.net/xtensor>

C_{Python} and concurrent programming

The mainstream implementation of the Python language is **C_{Python}**, written in C. C_{Python} integrates a mechanism called **Global Interpreter Lock (GIL)**. This is discussed at <http://wiki.python.org/moin/GlobalInterpreterLock>:

"The GIL facilitates memory management by preventing multiple native threads from executing Python bytecodes at once."

In other words, by disabling concurrent threads within one Python process, the GIL considerably simplifies the memory management system. Memory management is therefore not thread-safe in C_{Python}.

An important implication is that C_{Python} makes it non trivial to leverage multiple CPUs in a single Python process. This is an important issue as modern processors contain more and more cores.

What possible solutions do we have in order to take advantage of multi-core processors?

- ▶ Removing the GIL in C_{Python}. This solution has been tried but has never made it into C_{Python}. It would bring too much complexity in the implementation of C_{Python}, and it would degrade the performance of single-threaded programs.
- ▶ Using multiple processes instead of multiple threads. This is a popular solution; it can be done with the native multiprocessing module or with IPython. We will cover this latter in this chapter.
- ▶ Rewriting specific portions of your code in Cython and replacing all Python variables with C variables. This allows you to remove the GIL temporarily in a loop, thereby enabling use of multi-core processors. We will cover this solution in the *Releasing the GIL to take advantage of multi-core processors with Cython and OpenMP* recipe.
- ▶ Implementing a specific portion of your code in a language that offers better support for multi-core processors and calling it from your Python program.
- ▶ Making your code use the NumPy functions that benefit from multi-core processors, such as `numpy.dot()`. NumPy needs to be compiled with BLAS/LAPACK/ATLAS/MKL.

A must-read reference on the GIL can be found at <http://www.dabeaz.com/GIL/>.

Compiler-related installation instructions

In this section, we will give a few instructions on using compilers with Python. Use cases include using ctypes, using Cython, and building C extensions for Python.

On Linux, you should install GCC. For example, on Ubuntu type `sudo apt-get install build-essential` in a Terminal.

On macOS, install Xcode or the Xcode Command Line Tools. Alternatively, type `gcc` in a Terminal. If it is not installed, macOS should provide you with some options to install it.

On Windows, install a version of Microsoft Visual Studio, Visual C++, or the Visual C++ Build Tools that corresponds to your version of Python. If you use Python 3.6 (which is the latest stable version of Python at the time of writing), the corresponding version of the Microsoft compiler is 2017. All of these programs are free or have a free version that is sufficient for Python.

Here are a few references:

- ▶ Documentation for Installing Cython at <http://cython.readthedocs.io/en/latest/src/quickstart/install.html>
- ▶ Windows compilers for Python, at <https://wiki.python.org/moin/WindowsCompilers>
- ▶ Microsoft Visual Studio downloads at <https://www.visualstudio.com/downloads/>

Using Python to write faster code

The first way to make Python code run faster is to know all features of the language. Python brings many syntax features and modules in the standard library that run much faster than anything you could write by hand. Moreover, although Python may be slow if you write in Python like you would write in C or Java, it is often fast enough when you write Pythonic code.

In this section, we show how badly-written Python code can be significantly improved when using all the features of the language.



Leveraging NumPy for efficient array operations is of course another possibility that we explored in the *Introducing the multidimensional array in NumPy for fast array computations* recipe in Chapter 1, *A Tour of Interactive Computing with Jupyter and IPython*. This recipe focuses on cases where, for one reason or another, depending on and using NumPy is not a possible or desirable option. For example, operations on dictionaries, graphs, or text may be easier to write in Python than in NumPy. In these cases, Python brings many features that can still let you make your code faster.

How to do it...

1. Let's define a list of normally-distributed random variables, using the `random` built-in module instead of NumPy.

```
>>> import random
l = [random.normalvariate(0,1) for i in range(100000)]
```

2. Let's write a function that computes the sum of all numbers in that list. Someone inexperienced with Python may write in Python as if it was C, which would give the following function:

```
>>> def sum1():
    # BAD: not Pythonic and slow
    res = 0
    for i in range(len(l)):
        res = res + l[i]
    return res
>>> sum1()
319.346
>>> %timeit sum1()
6.64 ms ± 69.1 µs per loop (mean ± std. dev. of 7 runs,
100 loops each)
```

Six milliseconds to compute the sum of *only* 100,000 numbers is slow, which may lead some to say rather unfairly that *Python is slow*.

3. Now, let's write a slightly improved version of this code, taking into account the fact that we can enumerate the elements of a list using `for x in l` instead of iterating with an index:

```
>>> def sum2():
    # STILL BAD
    res = 0
    for x in l:
        res = res + x
    return res
```

```
>>> sum2()
319.346
>>> %timeit sum2()
3.3 ms ± 54.7 µs per loop (mean ± std. dev. of 7 runs,
100 loops each)
```

This slight modification gave us a two-fold speed improvement.

4. Finally, we realize that Python brings a built-in function to compute the sum of all elements in a list:

```
>>> def sum3():
    # GOOD
    return sum(l)
>>> sum3()
319.346
>>> %timeit sum3()
391 µs ± 840 ns per loop (mean ± std. dev. of 7 runs,
1000 loops each)
```

This version is 17 times faster than the first version, and we only wrote pure Python code!

5. Let's move to another example involving strings. We'll create a list of strings representing all numbers in our previous list:

```
>>> strings = ['%.3f' % x for x in l]
>>> strings[:3]
['-0.056', '-0.417', '-0.357']
```

6. We define a function concatenating all strings in that list. Again, an inexperienced Python programmer could write code such as the following:

```
>>> def concat1():
    # BAD: not Pythonic
    cat = strings[0]
    for s in strings[1:]:
        cat = cat + ', ' + s
    return cat
>>> concat1()[:24]
'-0.056, -0.417, -0.357, '
>>> %timeit concat1()
1.31 s ± 12.1 ms per loop (mean ± std. dev. of 7 runs,
1 loop each)
```

This function is very slow because a large number of tiny strings are allocated.

7. Next, we realize that Python offers the option to easily concatenate several strings:

```
>>> def concat2():
    # GOOD
    return ', '.join(strings)
>>> concat2()[:24]
'-0.056, -0.417, -0.357, '
>>> %timeit concat2()
797 µs ± 13.7 µs per loop (mean ± std. dev. of 7 runs,
1000 loops each)
```

This function is 1640 times faster!

8. Finally, we want to count the number of occurrences of all numbers between 0 and 99 in a list containing 100,000 integers between 0 and 99:

```
>>> l = [random.randint(0, 100) for _ in range(100000)]
```

9. The naive way would be to iterate over all elements in the list and construct the histogram using a dictionary:

```
>>> def hist1():
    # BAD
    count = {}
    for x in l:
        # We need to initialize every number
        # the first time it appears in the list.
        if x not in count:
            count[x] = 0
        count[x] += 1
    return count
>>> hist1()
{0: 979,
 1: 971,
 2: 990,
 ...
 99: 995,
 100: 1009}
>>> %timeit hist1()
8.7 ms ± 27.6 µs per loop (mean ± std. dev. of 7 runs,
100 loops each)
```

10. Next, we realize that Python offers a `defaultdict` structure that handles the automatic creation of dictionary keys:

```
>>> from collections import defaultdict
>>> def hist2():
    # BETTER
```

```
count = defaultdict(int)
for x in l:
    # The key is created and the value
    # initialized at 0 when needed.
    count[x] += 1
return count
>>> hist2()
defaultdict(int,
{0: 979,
 1: 971,
 ...
 99: 995,
 100: 1009})
>>> %timeit hist2()
6.82 ms ± 217 µs per loop (mean ± std. dev. of 7 runs,
 100 loops each)
```

This version is slightly faster.

11. Finally, we realize that the built-in `collections` module offers a `Counter` class that does exactly what we need:

```
>>> from collections import Counter
>>> def hist3():
        # GOOD
        return Counter(l)
>>> hist3()
Counter({0: 979,
 1: 971,
 ...
 99: 995,
 100: 1009})
>>> %timeit hist3()
3.69 ms ± 105 µs per loop (mean ± std. dev. of 7 runs,
 100 loops each)
```

This version is twice as fast as the first one.

There's more...

When your code is too slow, the first step is to make sure you're not reinventing the wheel and that you're making good use of all the features of the language.

You can have an overview of all the syntax features and built-in modules of Python by reading the documentation and other references:

- ▶ Documentation for Python 3 at <https://docs.python.org/3/library/index.html>
- ▶ *Python Cookbook, 3rd Edition, Brian Jones and David Beazley, O'Reilly Media* at <http://shop.oreilly.com/product/0636920027072.do>

See also

- ▶ The *Using the latest features of Python 3* recipe, in *Chapter 2, Best Practices in Interactive Computing*

Accelerating pure Python code with Numba and Just-In-Time compilation

Numba (<http://numba.pydata.org>) is a package created by Anaconda, Inc (<http://www.anaconda.com>). Numba takes pure Python code and translates it automatically (JIT) into optimized machine code. In practice, this means that we can write a non-vectorized function in pure Python, using `for` loops, and have this function vectorized automatically by using a single decorator. Performance speedups when compared to pure Python code can reach several orders of magnitude and may even outmatch manually-vectorized NumPy code.

In this section, we will show you how to accelerate pure Python code generating a Mandelbrot fractal.

Getting ready

Numba should already be installed in Anaconda, but you can also install it manually with `conda install numba`.

How to do it...

1. Let's import NumPy and define a few variables:

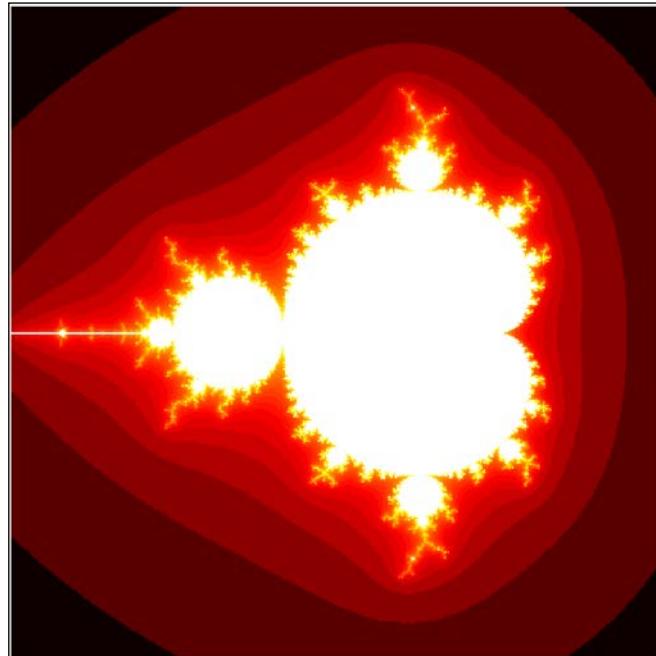
```
>>> import numpy as np
      import matplotlib.pyplot as plt
      %matplotlib inline
>>> size = 400
      iterations = 100
```

2. The following function generates a fractal in pure Python. It accepts an empty array `m` as argument.

```
>>> def mandelbrot_python(size, iterations):
    m = np.zeros((size, size))
    for i in range(size):
        for j in range(size):
            c = (-2 + 3. / size * j +
                  1j * (1.5 - 3. / size * i))
            z = 0
            for n in range(iterations):
                if np.abs(z) <= 10:
                    z = z * z + c
                    m[i, j] = n
                else:
                    break
    return m
```

3. Let's run the simulation and display the fractal:

```
>>> m = mandelbrot_python(size, iterations)
>>> fig, ax = plt.subplots(1, 1, figsize=(10, 10))
ax.imshow(np.log(m), cmap=plt.cm.hot)
ax.set_axis_off()
```



4. Now, we evaluate the time taken by this function:

```
>>> %timeit mandelbrot_python(size, iterations)
5.45 s ± 18.6 ms per loop (mean ± std. dev. of 7 runs,
1 loop each)
```

5. Let's try to accelerate this function using Numba. First, we import the package:

```
>>> from numba import jit
```

6. Next, we add the `@jit` decorator right above the function definition, without changing a single line of code in the body of the function:

```
>>> @jit
def mandelbrot_numba(size, iterations):
    m = np.zeros((size, size))
    for i in range(size):
        for j in range(size):
            c = (-2 + 3. / size * j +
                  1j * (1.5 - 3. / size * i))
            z = 0
            for n in range(iterations):
                if np.abs(z) <= 10:
                    z = z * z + c
                    m[i, j] = n
                else:
                    break
    return m
```

7. This function works just like the pure Python version. How much faster is it?

```
>>> mandelbrot_numba(size, iterations)
>>> %timeit mandelbrot_numba(size, iterations)
34.5 ms ± 59.4 µs per loop (mean ± std. dev. of 7 runs,
10 loops each)
```

The Numba version is about 150 times faster than the pure Python version here!

How it works...

Python bytecode is normally interpreted at runtime by the Python interpreter (most often, CPython). By contrast, a Numba function is parsed and translated directly to machine code ahead of execution, using a powerful compiler architecture named **Low Level Virtual Machine (LLVM)**.

Numba supports a significant but not exhaustive subset of Python semantics. You can find a list of supported Python features at <http://numba.pydata.org/numba-doc/latest/reference/pysupported.html>. When Numba cannot compile Python code to assembly, it will automatically fall back to a much slower mode. You can prevent this behavior with `@jit(nopython=True)`.

Numba generally gives the most impressive speedups on functions that involve tight loops on NumPy arrays (such as in this recipe). This is because there is an overhead running loops in Python, and this overhead becomes non-negligible when there are many iterations of a few cheap operations. In this example, the number of iterations is `size * size * iterations = 16,000,000`.

There's more...

Let's compare the performance of Numba with manually-vectorized code using NumPy, which is the standard way of accelerating pure Python code such as the code given in this recipe. In practice, it means replacing the code inside the two loops over `i` and `j` with array computations. This is relatively easy here as the operations closely follow the **Single Instruction, Multiple Data (SIMD)** paradigm:

```
>>> def initialize(size):
    x, y = np.meshgrid(np.linspace(-2, 1, size),
                       np.linspace(-1.5, 1.5, size))
    c = x + 1j * y
    z = c.copy()
    m = np.zeros((size, size))
    return c, z, m
>>> def mandelbrot_numpy(c, z, m, iterations):
    for n in range(iterations):
        indices = np.abs(z) <= 10
        z[indices] = z[indices] ** 2 + c[indices]
        m[indices] = n
>>> %%timeit -n1 -r10 c, z, m = initialize(size)
           mandelbrot_numpy(c, z, m, iterations)
174 ms ± 2.91 ms per loop (mean ± std. dev. of 10 runs,
  1 loop each)
```

In this example, Numba still beats NumPy.

Numba supports many other features, such as multiprocessing and GPU computing.

Here are a few references:

- ▶ Documentation for Numba available at <http://numba.pydata.org>
- ▶ Supported Python features in Numba, available at <http://numba.pydata.org/numba-doc/latest/reference/pysupported.html>
- ▶ Supported NumPy features in Numba, available at <http://numba.pydata.org/numba-doc/latest/reference/numpysupported.html>

See also

- ▶ The Accelerating array computations with NumExpr recipe

Accelerating array computations with NumExpr

NumExpr is a package that can offer some speedup on complex computations on NumPy arrays. NumExpr evaluates algebraic expressions involving arrays, parses them, compiles them, and finally executes them, possibly on multiple processors.

This principle is somewhat similar to Numba, in that normal Python code is compiled dynamically to machine code. However, NumExpr only tackles algebraic array expressions rather than arbitrary Python code. We will see how that works in this recipe.

Getting ready

NumExpr should already be installed in Anaconda, but you can also install it manually with `conda install numexpr`.

How to do it...

1. Let's import NumPy and NumExpr:

```
>>> import numpy as np  
      import numexpr as ne
```

2. Then we generate three large vectors:

```
>>> x, y, z = np.random.rand(3, 1000000)
```

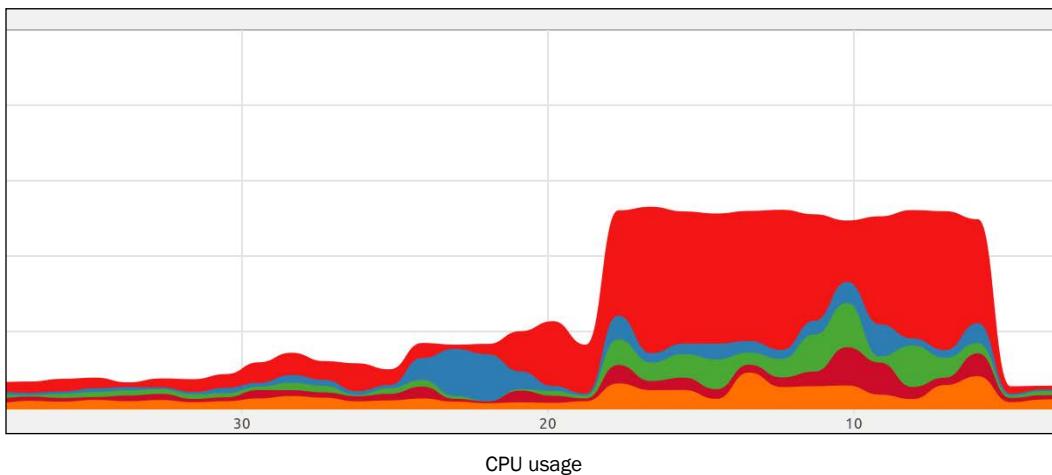
3. Now, we evaluate the time taken by NumPy to calculate a complex algebraic expression involving our vectors:

```
>>> %timeit x + (y**2 + (z*x + 1)*3)  
6.94 ms ± 223 µs per loop (mean ± std. dev. of 7 runs,  
100 loops each)
```

4. Let's perform the same calculation with NumExpr. We need to give the expression as a string:

```
>>> %timeit ne.evaluate('x + (y**2 + (z*x + 1)*3)')
1.47 ms ± 8.07 µs per loop (mean ± std. dev. of 7 runs,
1000 loops each)
```

The following screenshot shows the CPU usage when we ran the code with NumPy and then with NumExpr, which automatically use multiple CPUs:



5. NumExpr can use multiple cores. Here, we have four physical cores and eight virtual threads with Intel's **Hyper-Threading Technology (HTT)**. We can specify how many cores we want NumExpr to use using the `set_num_threads()` function:

```
>>> ne.ncores
8
>>> for i in range(1, 5):
    ne.set_num_threads(i)
    %timeit ne.evaluate('x + (y**2 + (z*x + 1)*3)',)
3.53 ms ± 12.9 µs per loop (mean ± std. dev. of 7 runs,
100 loops each)
2.35 ms ± 276 µs per loop (mean ± std. dev. of 7 runs,
100 loops each)
1.6 ms ± 60 µs per loop (mean ± std. dev. of 7 runs,
1000 loops each)
1.5 ms ± 24.6 µs per loop (mean ± std. dev. of 7 runs,
1000 loops each)
```

How it works...

NumExpr analyzes the array expression, parses it, and compiles it into a lower-level language. NumExpr is aware of CPU-vectorized instructions as well as CPU cache characteristics. As such, NumExpr can optimize vectorized computations dynamically.

Here are a few references:

- ▶ NumExpr on GitHub, at <https://github.com/pydata/numexpr>
- ▶ NumExpr documentation at <https://numexpr.readthedocs.io/en/latest/intro.html>

See also

- ▶ The Accelerating pure Python code with Numba and Just-In-Time compilation recipe

Wrapping a C library in Python with `ctypes`

Wrapping a C library in Python allows us to leverage existing C code or to implement a critical part of the code in a fast language such as C.

It is relatively easy to use externally-compiled libraries with Python. The first possibility is to call a command-line executable with the `os.system()` command, but this method does not extend to compiled libraries.

A more powerful method consists of using a native Python module called `ctypes`. This module allows us to call functions defined in a compiled library (written in C) from Python. The `ctypes` module takes care of data type conversions between C and Python. In addition, the `numpy.ctypeslib` module provides facilities to use NumPy arrays wherever data buffers are used in the external library.

In this example, we will rewrite the code of the Mandelbrot fractal in C, compile it in a shared library, and call it from Python.

Getting ready

The code in this recipe is written for Unix systems and has been tested on Ubuntu. It can be adapted to other systems with minor changes.

A C compiler is required. You will find all compiler-related instructions in this chapter's introduction.

How to do it...

First, we write and compile the Mandelbrot example in C. Then, we access it from Python using ctypes.

1. Let's write the code for the Mandelbrot fractal in C:

```
>>> %%writefile mandelbrot.c
#include "stdio.h"
#include "stdlib.h"

void mandelbrot(int size, int iterations, int *col)
{
    // Variable declarations.
    int i, j, n, index;
    double cx, cy;
    double z0, z1, z0_tmp, z0_2, z1_2;

    // Loop within the grid.
    for (i = 0; i < size; i++)
    {
        cy = -1.5 + (double)i / size * 3;
        for (j = 0; j < size; j++)
        {
            // We initialize the loop of the system.
            cx = -2.0 + (double)j / size * 3;
            index = i * size + j;
            // Let's run the system.
            z0 = 0.0;
            z1 = 0.0;
            for (n = 0; n < iterations; n++)
            {
                z0_2 = z0 * z0;
                z1_2 = z1 * z1;
                if (z0_2 + z1_2 <= 100)
                {
                    // Update the system.
                    z0_tmp = z0_2 - z1_2 + cx;
                    z1 = 2 * z0 * z1 + cy;
                    z0 = z0_tmp;
                    col[index] = n;
                }
            }
        }
    }
}
```

```
        else
        {
            break;
        }
    }
}
```

2. Now, let's compile this C source file with `gcc` into the `mandelbrot.so` dynamic library:

```
>>> ! !gcc -fPIC -shared -Wl,-soname,mandelbrot \
           -o mandelbrot.so \
           mandelbrot.c
```

3. Let's access the library with `ctypes`:

```
>>> import ctypes
>>> lib = ctypes.CDLL('mandelbrot.so')
>>> mandelbrot = lib.mandelbrot
```

4. NumPy and `ctypes` allow us to wrap the C function defined in the library:

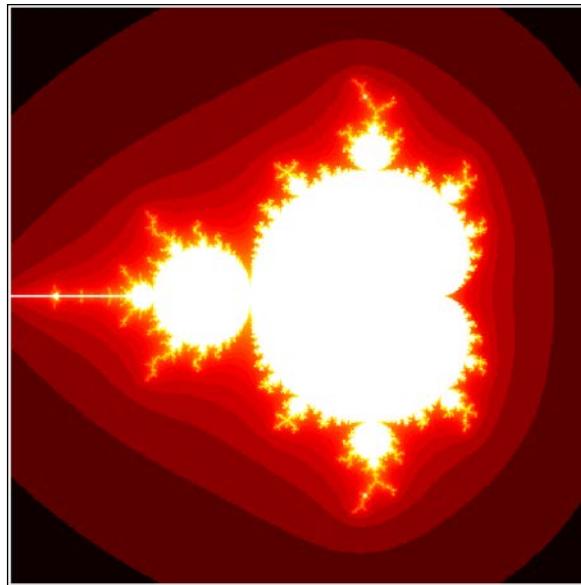
```
>>> from numpy.ctypeslib import ndpointer
>>> # Define the types of the output and arguments of
      # this function.
      mandelbrot.restype = None
      mandelbrot.argtypes = [ctypes.c_int,
                            ctypes.c_int,
                            ndpointer(ctypes.c_int),
                            ]
```

5. To use this function, we first need to initialize an empty array and pass it as an argument to the `mandelbrot()` wrapper function:

```
>>> import numpy as np
      # We initialize an empty array.
      size = 400
      iterations = 100
      col = np.empty((size, size), dtype=np.int32)
      # We execute the C function, which will update
      # the array.
      mandelbrot(size, iterations, col)
```

6. Let's show the result:

```
>>> import numpy as np
      import matplotlib.pyplot as plt
      %matplotlib inline
>>> fig, ax = plt.subplots(1, 1, figsize=(10, 10))
      ax.imshow(np.log(col), cmap=plt.cm.hot)
      ax.set_axis_off()
```



7. How fast is this function?

```
>>> %timeit mandelbrot(size, iterations, col)
28.9 ms ± 73.1 µs per loop (mean ± std. dev. of 7 runs,
 10 loops each)
```

The wrapped C version is slightly faster than the Numba version in the first recipe of this chapter.

How it works...

The `mandelbrot()` function accepts as arguments:

- ▶ The **size** of the `col` buffer (the `col` value is the last iteration where the corresponding point is within a disc around the origin)
- ▶ The number of **iterations**
- ▶ A **pointer** to the buffer of integers

The `mandelbrot()` C function does not return any value; rather, it updates the buffer that was passed by reference to the function (it is a pointer).

To wrap this function in Python, we need to declare the types of the input arguments. The `ctypes` module defines constants for the different data types. In addition, the `numpy.ctypeslib.ndpointer()` function lets us use a NumPy array wherever a pointer is expected in the C function. The data type given as argument to `ndpointer()` needs to correspond to the NumPy data type of the array passed to the function.

Once the function has been correctly wrapped, it can be called as if it was a standard Python function. Here, the initially-empty NumPy array is filled with the Mandelbrot fractal after the call to `mandelbrot()`.

There's more...

An alternative to `ctypes` is `cffi` (<http://cffi.readthedocs.org>), which may be a bit faster and more convenient to use. You can also refer to <http://eli.thegreenplace.net/2013/03/09/python-ffi-with-ctypes-and-cffi/>.

See also

- ▶ The Accelerating pure Python code with Numba and Just-In-Time compilation recipe

Accelerating Python code with Cython

Cython is both a language (a superset of Python) and a Python library. With Cython, we start from a regular Python program and we add annotations about the type of the variables. Then, Cython translates that code to C and compiles the result into a Python extension module. Finally, we can use this compiled module in any Python program.

While dynamic typing comes with a performance cost in Python, statically-typed variables in Cython generally lead to faster code execution.

Performance gains are most significant in CPU-bound programs, notably in tight Python loops. By contrast, I/O bound programs are not expected to benefit much from a Cython implementation.

In this recipe, we will see how to accelerate the Mandelbrot code example with Cython.

Getting ready

A C compiler is required. You will find all compiler-related instructions in the introduction the this chapter.

You will also need Cython, which should be installed by default with Anaconda. If needed, you can also install it with `conda install cython`.

How to do it...

1. Let's define some variables:

```
>>> import numpy as np  
>>> size = 400  
     iterations = 100
```

2. To use Cython in the Jupyter Notebook, we first need to import the Cython Jupyter extension:

```
>>> %load_ext cython
```

3. As a first try, let's just add the `%%cython` magic before the definition of the `mandelbrot()` function. Internally, this cell magic compiles the cell into a standalone Cython module, hence the need for all required imports to occur within the same cell. This cell does not have access to any variable or function defined in the interactive namespace:

```
>>> %%cython -a  
import numpy as np  
  
def mandelbrot_cython(m, size, iterations):  
    for i in range(size):  
        for j in range(size):  
            c = -2 + 3./size*j + 1j*(1.5-3./size*i)  
            z = 0  
            for n in range(iterations):  
                if np.abs(z) <= 10:  
                    z = z*z + c  
                    m[i, j] = n  
                else:  
                    break
```

```
Generated by Cython 0.26
Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

+01: import numpy as np
+02:
+03: def mandelbrot_cython(m, size, iterations):
+04:     for i in range(size):
+05:         for j in range(size):
+06:             c = -2 + 3./size*j + 1j*(1.5-3./size*i)
+07:             z = 0
+08:             for n in range(iterations):
+09:                 if np.abs(z) <= 10:
+10:                     z = z*z + c
+11:                     m[i, j] = n
+12:                 else:
+13:                     break
```

The `-a` option tells Cython to annotate lines of code with a background color indicating how optimized it is. The darker the color, the less optimized the line. The color depends on the relative number of Python API calls at each line. We can click on any line to see the generated C code. Here, this version does not appear to be optimized.

4. How fast is this version?

```
>>> s = (size, size)
>>> %timeit -n1 -r1 m = np.zeros(s, dtype=np.int32)
    mandelbrot_cython(m, size, iterations)
4.52 s ± 0 ns per loop (mean ± std. dev. of 1 run,
 1 loop each)
```

We get virtually no speedup here. We need to specify the type of our Python variables.

5. Let's add type information using typed memory views for NumPy arrays (we explain these in the *How it works...* section). We also use a slightly different way to test whether particles have escaped from the domain (the `if` test):

```
>>> %%cython -a
import numpy as np

def mandelbrot_cython(int[:,::1] m,
                      int size,
                      int iterations):
    cdef int i, j, n
    cdef complex z, c
    for i in range(size):
        for j in range(size):
            c = -2 + 3./size*j + 1j*(1.5-3./size*i)
            z = 0
            for n in range(iterations):
```

```
if z.real**2 + z.imag**2 <= 100:  
    z = z*z + c  
    m[i, j] = n  
else:  
    break
```

```
Generated by Cython 0.26  
Yellow lines hint at Python interaction.  
Click on a line that starts with a "+" to see the C code that Cython generated for it.  
+01: import numpy as np  
02:  
+03: def mandelbrot_cython(int[:, ::1] m,  
04:                           int size,  
05:                           int iterations):  
06:     cdef int i, j, n  
07:     cdef complex z, c  
08:     for i in range(size):  
09:         for j in range(size):  
+10:             c = -2 + 3./size*j + 1j*(1.5-3./size*i)  
+11:             z = 0  
+12:             for n in range(iterations):  
+13:                 if z.real**2 + z.imag**2 <= 100:  
+14:                     z = z*z + c  
+15:                     m[i, j] = n  
16:                 else:  
+17:                     break
```

6. How fast is this new version?

```
>>> %%timeit -n1 -r1 m = np.zeros(s, dtype=np.int32)  
        mandelbrot_cython(m, size, iterations)  
12.7 ms ± 0 ns per loop (mean ± std. dev. of 1 run,  
1 loop each)
```

This version is almost 350 times faster than the first version!

All we have done is specify the type of the local variables and function arguments, and bypass NumPy's `np.abs()` function when computing the absolute value of `z`. These changes have helped Cython generate more optimized C code from Python code.

How it works...

The `cdef` keyword declares a variable as a statically-typed C variable. C variables lead to faster code execution because the overhead from Python's dynamic typing is bypassed. Function arguments can also be declared as statically-typed C variables.

There are two ways of declaring NumPy arrays as C variables with Cython: using **array buffers** or using **typed memory views**. In this recipe, we used typed memory views. We will cover array buffers in the next recipe.

Typed memory views allow efficient access to data buffers with a NumPy-like indexing syntax. For example, we can use `int[:, ::1]` to declare a C-ordered 2D NumPy array with integer values, with `::1` meaning a contiguous layout in this dimension. Typed memory views can be indexed just like NumPy arrays.

However, memory views do not implement element-wise operations like NumPy. Thus, memory views act as convenient data containers within tight `for` loops. For element-wise NumPy-like operations, array buffers should be used instead.

We could achieve a significant performance speedup by replacing the call to `np.abs()` with a faster expression. The reason is that `np.abs()` is a NumPy function with a slight call overhead. It is designed to work with relatively large arrays, not scalar values. This overhead results in a significant performance hit in a tight loop such as here. This bottleneck can be spotted with Cython annotations.

There's more...

Using Cython from Jupyter is very convenient with the `%%cython` cell magic. However, it is sometimes necessary to create a reusable C extension module with Cython. This is actually what the `%%cython` cell magic does under-the-hood. You will find more information at <http://cython.readthedocs.io/en/latest/src/quickstart/build.html>.

Here are a few references:

- ▶ Distributing Cython modules, explained at http://docs.cython.org/src/userguide/source_files_and_compilation.html
- ▶ Compilation with Cython, explained at <http://docs.cython.org/src/reference/compilation.html>
- ▶ Cython and Numpy, at <http://cython.readthedocs.io/en/latest/src/userguide/memoryviews.html>

See also

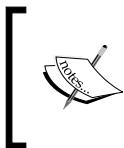
- ▶ The *Optimizing Cython code by writing less Python and more C* recipe
- ▶ The *Releasing the GIL to take advantage of multi-core processors with Cython and OpenMP* recipe

Optimizing Cython code by writing less Python and more C

In this recipe, we will consider a more complicated Cython example. Starting from a slow implementation in pure Python, we will use different Cython features to speed it up progressively.

We will implement a very simple ray tracing engine. **Ray tracing** consists of rendering a scene by simulating the physical properties of light propagation. This rendering method leads to photorealistic scenes, but it is computationally intensive.

Here, we will render a single sphere with diffuse and specular lighting. First we'll give the example's code in pure Python. Then, we will accelerate it incrementally with Cython.



The code is long and contains many functions. We will first give the full code of the pure Python version. Then, we will just describe the changes required to accelerate the code with Cython. The full scripts are available on the book's website.

How to do it...

1. First, let's implement the pure Python version:

```
>>> import numpy as np
      import matplotlib.pyplot as plt
>>> %matplotlib inline
>>> w, h = 400, 400 # Size of the screen in pixels.
```

2. We create a normalization function for vectors:

```
>>> def normalize(x):
      # This function normalizes a vector.
      x /= np.linalg.norm(x)
      return x
```

3. We create a function that computes the intersection of a ray with a sphere:

```
>>> def intersect_sphere(O, D, S, R):
      # Return the distance from O to the intersection
      # of the ray (O, D) with the sphere (S, R), or
      # +inf if there is no intersection.
      # O and S are 3D points, D (direction) is a
      # normalized vector, R is a scalar.
      a = np.dot(D, D)
      OS = O - S
      b = 2 * np.dot(D, OS)
      c = np.dot(OS, OS) - R * R
      disc = b * b - 4 * a * c
      if disc > 0:
          distSqrt = np.sqrt(disc)
          q = (-b - distSqrt) / 2.0 if b < 0 \
              else (-b + distSqrt) / 2.0
          t0 = q / a
          t1 = c / q
          t0, t1 = min(t0, t1), max(t0, t1)
```

```
    if t1 >= 0:  
        return t1 if t0 < 0 else t0  
    return np.inf
```

4. The following function traces a ray:

```
>>> def trace_ray(O, D):  
    # Find first point of intersection with the scene.  
    t = intersect_sphere(O, D, position, radius)  
    # No intersection?  
    if t == np.inf:  
        return  
    # Find the point of intersection on the object.  
    M = O + D * t  
    N = normalize(M - position)  
    toL = normalize(L - M)  
    toO = normalize(O - M)  
    # Ambient light.  
    col = ambient  
    # Lambert shading (diffuse).  
    col += diffuse * max(np.dot(N, toL), 0) * color  
    # Blinn-Phong shading (specular).  
    col += specular_c * color_light * \  
        max(np.dot(N, normalize(toL + toO)), 0) \  
        ** specular_k  
    return col
```

5. Finally, the main loop is implemented in the following function:

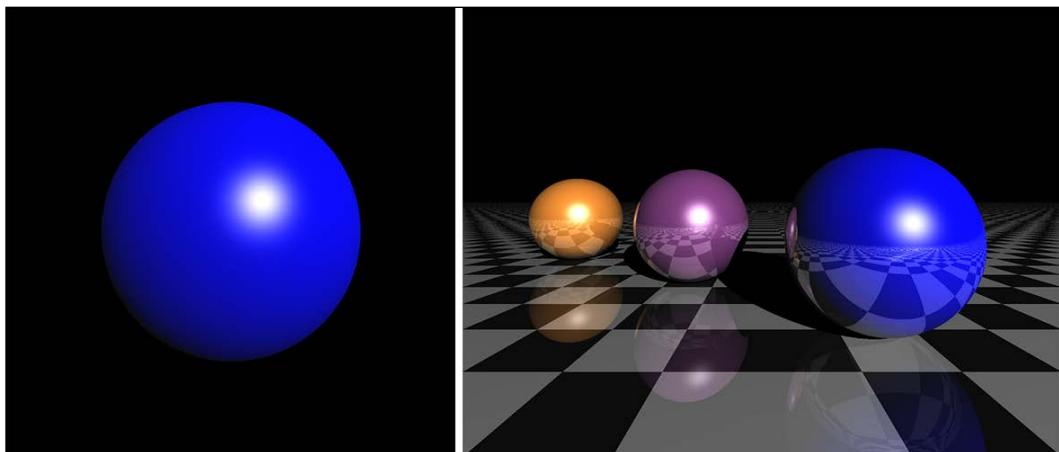
```
>>> def run():  
    img = np.zeros((h, w, 3))  
    # Loop through all pixels.  
    for i, x in enumerate(np.linspace(-1, 1, w)):  
        for j, y in enumerate(np.linspace(-1, 1, h)):  
            # Position of the pixel.  
            Q[0], Q[1] = x, y  
            # Direction of the ray going through  
            # the optical center.  
            D = normalize(Q - O)  
            # Launch the ray and get the color  
            # of the pixel.  
            col = trace_ray(O, D)  
            if col is None:  
                continue  
            img[h - j - 1, i, :] = np.clip(col, 0, 1)  
    return img
```

6. Now, we initialize the scene and define a few parameters:

```
>>> # Sphere properties.  
    position = np.array([0., 0., 1.])  
    radius = 1.  
    color = np.array([0., 0., 1.])  
    diffuse = 1.  
    specular_c = 1.  
    specular_k = 50  
  
    # Light position and color.  
    L = np.array([5., 5., -10.])  
    color_light = np.ones(3)  
    ambient = .05  
  
    # Camera.  
    O = np.array([0., 0., -1.]) # Position.  
    Q = np.array([0., 0., 0.]) # Pointing to.
```

7. Let's render the scene:

```
>>> img = run()  
fig, ax = plt.subplots(1, 1, figsize=(10, 10))  
ax.imshow(img)  
ax.set_axis_off()
```



In this screenshot, the left panel shows the result of this recipe's code. The right panel shows an extended version of the simple ray tracing engine implemented here.

8. How slow is this implementation (the ray1 example on the book's website)?

```
>>> %timeit run()
2.75 s ± 29.9 ms per loop (mean ± std. dev. of 7 runs,
1 loop each)
```

9. If we just use the %%cython magic with the adequate `import numpy as np` and `cimport numpy as np` commands at the top of the cell, we only get an approximate 6% speed improvement (the ray2 example).
10. We could do better by giving information about the type of the variables. Since we use vectorized computations on NumPy arrays, we cannot easily use memory views. Rather, we will use array buffers. First, at the very beginning of the Cython module (or %%cython cell), we declare NumPy data types as follows:

```
import numpy as np
cimport numpy as np
DBL = np.double ctypedef
np.double_t DBL_C
```

Then, we declare a NumPy array with `cdef np.ndarray [DBL_C, ndim=1]` (in this example, a 1D array of double precision floating point numbers). There is a difficulty here because NumPy arrays can only be declared inside functions, not at the top level. Thus, we need to slightly tweak the overall architecture of the code by passing some arrays as function arguments instead of using global variables. However, even by declaring the type of all variables, we gain no speed enhancement at all (the ray3 example).

11. In the current implementation, we incur a performance hit because of the large number of NumPy function calls on tiny arrays (three elements). NumPy is designed to deal with large arrays, and it does not make much sense to use it for arrays that small. In this specific situation, we can try to bypass NumPy by rewriting some functions using the C standard library. We use the `cdef` keyword to declare a C-style function. These functions can yield significant performance speedups. Here is the C function replacing `normalize()`:

```
from libc.math cimport sqrt
cdef normalize(np.ndarray [DBL_C, ndim=1] x):
    cdef double n
    n = sqrt(x[0]*x[0] + x[1]*x[1] + x[2]*x[2])
    x[0] /= n
    x[1] /= n
    x[2] /= n
    return x
```

We obtain a 25% speed improvement (the ray4 example).

12. To get the most interesting speedups, we need to completely bypass NumPy. Where do we use NumPy precisely?

- ❑ Many variables are NumPy arrays (mostly 1D vectors with three elements)
- ❑ Element-wise operations yield implicit NumPy API calls
- ❑ We also use a few NumPy built-in functions such as `np.dot()`

In order to bypass NumPy in our example, we need to reimplement all these features for our specific needs. The first possibility is to use a native Python type for vectors (for example, tuples), and write C-style functions that implement operations on tuples (always assuming they have exactly three elements). For example, the addition between two tuples can be implemented as follows:

```
cdef tuple add(tuple x, tuple y):  
    return (x[0]+y[0], x[1]+y[1], x[2]+y[2])
```

This time, we get an 18x speed enhancement compared to the pure Python version (the ray5 example)! But we can do even better.

13. We are going to define a pure C structure instead of using a Python type for our vectors. In other words, we are not only bypassing NumPy, but we are also bypassing Python by moving to pure C code. To declare a C structure representing a 3D vector in Cython, we can use the following code:

```
cdef struct Vec3:  
    double x, y, z
```

To create a new `Vec3` variable, we can use the following function:

```
cdef Vec3 vec3(double x, double y, double z):  
    cdef Vec3 v  
    v.x = x  
    v.y = y  
    v.z = z  
    return v
```

As an example, here is the function used to add two `Vec3` objects:

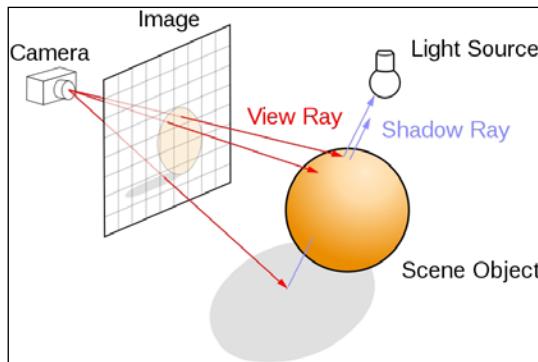
```
cdef Vec3 add(Vec3 u, Vec3 v):  
    return vec3(u.x + v.x, u.y + v.y, u.z + v.z)
```

The code can be updated to make use of these fast C-style functions. Finally, the image can be declared as a 3D memory view. With all these changes, the Cython implementation runs in approximately 8 ms instead of almost a couple of seconds, or 330 times faster (the ray6 example)!

In summary, we have achieved a very interesting speed enhancement by basically rewriting the entire implementation in C with an enhanced Python syntax.

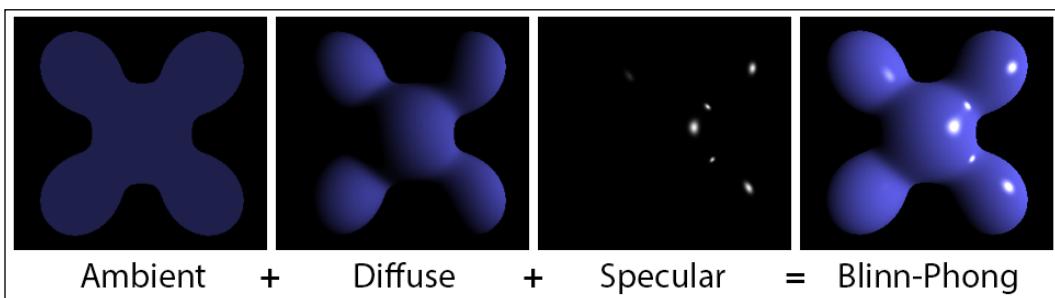
How it works...

Let's explain briefly how ray tracing works. We model a 3D scene with objects such as planes and spheres (here, there is only one sphere). There is also a camera and a plane representing the rendered image:



Principles of ray tracing (Ray trace diagram by Henrik, Wikimedia Commons)

There is a main loop over all pixels of the image. For each pixel, we launch a ray from the camera center to the scene through the current pixel and compute the first intersection point between that ray and an object from the scene. Then, we compute the pixel's color as a function of the object material's color, the position of the lights, the normal of the object at the intersection point, and so on. There are several physics-based lighting equations that describe how the color depends on these parameters. Here, we use the Blinn-Phong shading model with ambient, diffuse, and specular lighting components:



Blinn-Phong shading model (Phong components, Wikimedia Commons)

Of course, a full ray tracing engine is far more complex than what we have implemented in this example. We can model other optic and lighting characteristics such as reflections, refractions, shadows, depth of field, and others. It is also possible to implement ray tracing algorithms on the graphics card for real-time photorealistic rendering. Here are a few references:

- ▶ Blinn-Phong shading model on Wikipedia, available at https://en.wikipedia.org/wiki/Blinn-Phong_shading_model
- ▶ Ray tracing on Wikipedia, available at https://en.wikipedia.org/wiki/Ray_tracing_%28graphics%29

There's more...

Although powerful, Cython requires a good understanding of Python, NumPy, and C. The most interesting performance speedups are achieved when dynamically-typed Python variables are converted to statically-typed C variables, notably within tight loops.

Here are a few references:

- ▶ Cython extension types, available at http://docs.cython.org/src/userguide/extension_types.html
- ▶ Extended version of the ray tracing example, available at <http://gist.github.com/rossant/6046463>

See also

- ▶ The *Accelerating Python code with Cython* recipe
- ▶ The *Releasing the GIL to take advantage of multi-core processors with Cython and OpenMP* recipe

Releasing the GIL to take advantage of multi-core processors with Cython and OpenMP

As we have seen in this chapter's introduction, CPython's GIL prevents pure Python code from taking advantage of multi-core processors. With Cython, we have a way to release the GIL temporarily in a portion of the code in order to enable multi-core computing. This is done with **OpenMP**, a multiprocessing API that is supported by most C compilers.

In this recipe, we will see how to parallelize the previous recipe's code on multiple cores.

Getting ready

To enable OpenMP in Cython, you just need to specify some options to the compiler. There is nothing special to install on your computer besides a good C compiler. See the instructions in this chapter's introduction for more details.

The code in this recipe has been written for GCC on Ubuntu. It can be adapted to other systems with minor changes to the `%%cython` options.

How to do it...

Our simple ray tracing engine implementation is *embarrassingly parallel* (see https://en.wikipedia.org/wiki/Embarrassingly_parallel); there is a main loop over all pixels, within which the exact same function is called repetitively. There is no crosstalk between loop iterations. Therefore, it would be theoretically possible to execute all iterations in parallel.

Here, we will execute one loop (over all columns in the image) in parallel with OpenMP.

You will find the entire code on the book's website (the `ray7` example). We will only show the most important steps here:

1. We use the following magic command:

```
>>> %%cython --compile-args=-fopenmp --link-args=-fopenmp --force
```

2. We import the `prange()` function:

```
>>> from cython.parallel import prange
```

3. We add `nogil` after each function definition in order to remove the GIL. We cannot use any Python variable or function inside a function annotated with `nogil`. For example:

```
cdef Vec3 add(Vec3 x, Vec3 y) nogil:  
    return vec3(x.x + y.x, x.y + y.y, x.z + y.z)
```

4. To run a loop in parallel over the cores with OpenMP, we use `prange()`:

```
with nogil:  
    for i in prange(w):  
        # ...
```

The GIL needs to be released before using any parallel computing feature such as `prange()`.

5. With these changes, we reach a 3x speed enhancement on a quad-core processor compared to the fastest version of the previous recipe.

How it works...

The GIL has been described in the introduction to this chapter. The `nogil` keyword tells Cython that a particular function or code section should be executed without the GIL. When the GIL is released, it is not possible to make any Python API calls, meaning that only C variables and C functions (declared with `cdef`) can be used.

See also

- ▶ The *Accelerating Python code with Cython* recipe
- ▶ The *Optimizing Cython code by writing less Python and more C* recipe
- ▶ The *Distributing Python code across multiple cores with IPython* recipe

Writing massively parallel code for NVIDIA graphics cards (GPUs) with CUDA

Graphics Processing Units (GPUs) are powerful processors specialized for real-time rendering. We find GPUs in virtually any computer, laptop, video game console, tablet, or smartphone. Their massively parallel architecture comprises tens to thousands of cores. The video game industry has been fostering the development of increasingly powerful GPUs over the last two decades.

Since the mid-2000s, GPUs are no longer limited to graphics processing. We can now implement scientific algorithms on a GPU. The only condition is that the algorithm follows the **SIMD paradigm**, where a sequence of instructions is executed in parallel with multiple data. This is called **General Purpose Programming on Graphics Processing Units (GPGPU)**. GPGPU is used in many areas: meteorology, machine learning (most particularly deep learning), computer vision, image processing, finance, physics, bioinformatics, and many more. Writing code for GPUs can be challenging as it requires understanding the internal architecture of the hardware.

CUDA is a proprietary GPGPU framework created in 2007 by NVIDIA Corporation, one of the main GPU manufacturers. Programs written in CUDA only work on NVIDIA graphics cards. There is another competing GPGPU framework called **OpenCL**, an open standard supported by other major companies. OpenCL programs can work on GPUs and CPUs from most manufacturers (notably NVIDIA, AMD, and Intel).

CUDA kernels are typically written in a C dialect that runs on the GPU. However, Numba allows us to CUDA kernels in Python. Numba takes care of compiling the code automatically for the GPU.

In this recipe, we will implement the embarrassingly parallel computation of the Mandelbrot fractal in CUDA using Numba.

Getting ready

You need an NVIDIA GPU installed on your computer. You also need the CUDA toolkit, which you can install with `conda install cudatoolkit`.

How to do it...

1. Let's import the packages:

```
>>> import math
      import numpy as np
      from numba import cuda
      import matplotlib.pyplot as plt
      %matplotlib inline
```

2. Let's check whether Numba correctly identified our GPU:

```
>>> len(cuda.gpus)
1
>>> cuda.gpus[0].name
b'GeForce GTX 980M'
```

3. We write our function in Python. It will be compiled to CUDA code. The object `m` represents a pointer to the array stored on the GPU. The function is called on the GPU in parallel on every pixel of the image. Numba provides a `cuda.grid()` function that gives the index of the pixel in the image:

```
>>> @cuda.jit
      def mandelbrot_numba(m, iterations):
          # Matrix index.
          i, j = cuda.grid(2)
          size = m.shape[0]
          # Skip threads outside the matrix.
          if i >= size or j >= size:
              return
          # Run the simulation.
          c = (-2 + 3. / size * j +
                1j * (1.5 - 3. / size * i))
          z = 0
          for n in range(iterations):
              if abs(z) <= 10:
                  z = z * z + c
                  m[i, j] = n
              else:
                  break
```

4. We initialize the matrix:

```
>>> size = 400
       iterations = 100
>>> m = np.zeros((size, size))
```

5. We initialize the execution grid (see the *How it works...* section):

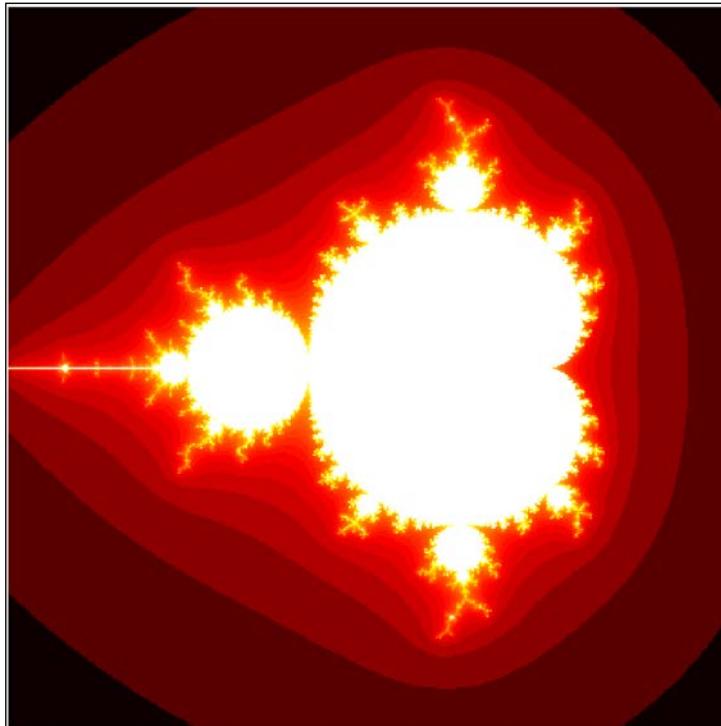
```
>>> # 16x16 threads per block.
       bs = 16
       # Number of blocks in the grid.
       bpg = math.ceil(size / bs)
       # We prepare the GPU function.
       f = mandelbrot_numba [(bpg, bpg), (bs, bs)]
```

6. We execute the GPU function, passing our empty array:

```
>>> f(m, iterations)
```

7. Let's display the result:

```
>>> fig, ax = plt.subplots(1, 1, figsize=(10, 10))
       ax.imshow(np.log(m), cmap=plt.cm.hot)
       ax.set_axis_off()
```



8. Now, let's benchmark this function:

```
>>> %timeit -n10 -r100 f(m, iterations)
2.99 ms ± 173 µs per loop (mean ± std. dev. of 100 runs,
10 loops each)
```

That's about 10 times faster than the CPU version obtained with Numba in the first recipe in this chapter, and 1,800 times faster than the pure Python version! But we can do even better.

9. Numba takes care of transferring arrays automatically between the host machine (CPU) and the device (GPU). These data transfers are slow, sometimes even more than the actual on-device computation. Numba provides facilities to deal with these transfers manually, which can be interesting in some use cases. Let's estimate the time of the data transfers and the computation on the GPU.
10. First, we send the NumPy array to the GPU with the `cuda.to_device()` function:

```
>>> %timeit -n10 -r100 cuda.to_device(m)
481 µs ± 106 µs per loop (mean ± std. dev. of 100 runs,
10 loops each)
```

11. Second, we run the computation on the GPU:

```
>>> %%timeit -n10 -r100 m_gpu = cuda.to_device(m)
f(m_gpu, iterations)
101 µs ± 11.8 µs per loop (mean ± std. dev. of 100 runs,
10 loops each)
```

12. Third, we copy the modified array from the GPU to the CPU.

```
>>> m_gpu = cuda.to_device(m)
>>> %timeit -n10 -r100 m_gpu.copy_to_host()
238 µs ± 67.8 µs per loop (mean ± std. dev. of 100 runs,
10 loops each)
```

If we consider only the GPU computation time excluding the data transfer times, we obtain a version that is 340 times faster than the version compiled on the CPU with Numba, and 54,000 times faster than the pure Python version!

This astronomic speed improvement is explained by the fact that the GPU version is compiled and runs on 1536 CUDA cores on the NVIDIA GTX 980M, whereas the pure Python version is interpreted and runs on 1 CPU.

How it works...

GPU programming is a rich and highly technical topic, encompassing low-level architectural details of GPUs. Of course, we only scratched the surface here with the simplest paradigm possible (an *embarrassingly parallel* problem). We give further references in a later section.

A CUDA GPU has a number of multiprocessors, and each **multiprocessor** has multiple **stream processors** (also called **CUDA cores**). Each multiprocessor executes in parallel with the others. Within a multiprocessor, the stream processors execute the same instruction at the same time, but on multiple data bits (SIMD paradigm).

Concepts central to the CUDA programming model are kernels, threads, blocks, and grids:

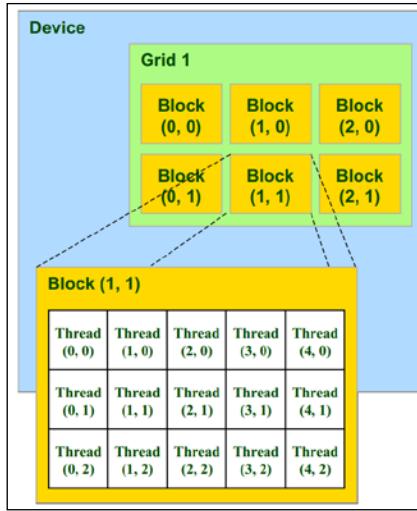
- ▶ A **kernel** is a program written in a C-like language that runs on the GPU
- ▶ A **thread** represents one execution of a kernel on one stream processor
- ▶ A **block** contains multiple threads executing on one multiprocessor
- ▶ A **grid** contains a number of blocks

The number of threads per block is limited by the size of the multiprocessors and depends on the graphics card model (1,024, for example). However, a grid can contain an arbitrary number of blocks.

Within a block, threads are executed within **warps** of typically 32 threads. Better performance is achieved when conditional branching in a kernel is organized into groups of 32 threads.

Threads within a block can synchronize at synchronization barriers using the `CUDA __syncthreads()` function. This feature enables inter-thread communication within one block. However, blocks execute independently so that two threads from different blocks cannot synchronize.

Within a block, threads are organized into a 1D, 2D, or 3D structure, and similarly for blocks within a grid, as shown in the following figure. This structure is convenient as it matches most common multidimensional datasets encountered in real-world problems.



The CUDA programming model (showing threads, blocks, and grids—image by NVIDIA Corporation)

In CUDA, the kernel can retrieve the thread index within the block (`threadIdx`), as well as the block index within the grid (`blockIdx`), to determine which bit of data it should work on. In this recipe, the 2D image of the fractal is partitioned into 16×16 blocks, each block containing 256 pixels, with one thread per pixel. The kernel computes the color of a single pixel.

Numba provides the `cuda.grid(ndim)` function to obtain directly the 1D, 2D, or 3D index of the thread within the grid. Alternatively, one can use the following code snippet to control the exact position of the current thread within the block and the grid (code given in the Numba documentation):

```
# Thread id in a 1D block
tx = cuda.threadIdx.x
# Block id in a 1D grid
ty = cuda.blockIdx.x
# Block width, i.e. number of threads per block
bw = cuda.blockDim.x
# Compute flattened index inside the array
pos = tx + ty * bw
if pos < an_array.size: # Check array boundaries
    # One can access 'an_array[pos]'
```

There are several levels of memory on the GPU, ranging from small, fast, and local memory shared by a few threads within a block to large, slow, and global memory shared by all blocks. We need to tweak the memory access patterns in the code to match the hardware constraints and achieve higher performance. In particular, data access is more efficient when the threads within a warp access consecutive addresses in the global memory; the hardware coalesces all memory accesses into a single access to consecutive **Dynamic Random Access Memory (DRAM)** locations.

There's more...

Here are a few references:

- ▶ Numba CUDA documentation at <http://numba.pydata.org/numba-doc/dev/cuda/index.html>
- ▶ Official CUDA portal at <http://developer.nvidia.com/category/zone/cuda-zone>
- ▶ Education and training for CUDA, at <http://developer.nvidia.com/cuda-education-training>
- ▶ Suggested books about CUDA, at <http://developer.nvidia.com/suggested-reading>

See also

- ▶ The *Accelerating pure Python code with Numba and Just-In-Time compilation* recipe

Distributing Python code across multiple cores with IPython

Despite CPython's GIL, it is possible to execute several tasks in parallel on multi-core computers using multiple processes instead of multiple threads. Python offers a native **multiprocessing** module. IPython's parallel extension, called **ipyparallel**, offers an even simpler interface that brings powerful parallel computing features in an interactive environment. We will describe this tool here.

Getting started

You need to install ipyparallel with `conda install ipyparallel`.

Then, you need to activate the ipyparallel Jupyter extension with `ipcluster nbextension enable --user`.

How to do it...

1. First, we launch four IPython engines in separate processes. We have basically two options to do this:

- ❑ Executing `ipcluster start -n 4` in a system shell
- ❑ Using the web interface provided in Jupyter Notebook's main page by clicking on the IPython **Clusters** tab and launching four engines

2. Then, we create a client that will act as a proxy to the IPython engines. The client automatically detects the running engines:

```
>>> from ipyparallel import Client  
rc = Client()
```

3. Let's check the number of running engines:

```
>>> rc.ids  
[0, 1, 2, 3]
```

4. To run commands in parallel over the engines, we can use the `%px` line magic or the `%%px` cell magic:

```
>>> %%px  
import os  
print(f"Process {os.getpid():d}."  
[stdout:0] Process 10784.  
[stdout:1] Process 10785.  
[stdout:2] Process 10787.  
[stdout:3] Process 10791.
```

5. We can specify which engines to run the commands on using the `--targets` or `-t` option:

```
>>> %%px -t 1,2  
# The os module has already been imported in  
# the previous cell.  
print(f"Process {os.getpid():d}."  
[stdout:1] Process 10785.  
[stdout:2] Process 10787.
```

6. By default, the %px magic executes commands in **blocking mode**; the cell only returns when the commands have completed on all engines. It is possible to run non-blocking commands with the --noblack or -a option. In this case, the cell returns immediately, and the task's status and results can be polled asynchronously from IPython's interactive session:

```
>>> %%px -a
       import time
       time.sleep(5)
<AsyncResult: execute>
```

7. The previous command returned an ASyncResult instance that we can use to poll the task's status:

```
>>> print(_.elapsed, _.ready())
1.522944 False
```

8. The %pxresult blocks until the task finishes:

```
>>> %pxresult
>>> print(_.elapsed, _.ready())
5.044711 True
```

9. ipyparallel provides convenient functions for common use cases, such as a parallel map() function:

```
>>> v = rc[:]
      res = v.map(lambda x: x * x, range(10))
>>> print(res.get())
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

How it works...

There are several steps in distributing code across multiple cores:

1. Launching several IPython **engines** (there is typically one process per core).
2. Creating a client that acts as a proxy to these engines.
3. Using the client to launch tasks on the engines and retrieve the results.

Engines are Python processes that execute code on different computing units. They are very similar to IPython kernels.

There are two main interfaces for accessing the engines:

- ▶ With the **direct interface**, we access engines directly and explicitly with their identifiers
- ▶ With the **load-balanced interface**, we access engines through an interface that automatically and dynamically assigns work to appropriate engines

We can also create custom interfaces for alternative styles of parallelism.

In this recipe, we used the direct interface; we addressed individual engines explicitly by specifying their identifiers in the `%px` magic command.

As we have seen in this recipe, tasks can be launched synchronously or asynchronously. The `%px*` magic commands are particularly convenient in the Notebook, as they let us work seamlessly on multiple engines in parallel.

There's more...

The parallel computing capabilities of ipyparallel offer an easy way to launch independent jobs in parallel over multiple cores. A more advanced use case is when jobs have dependencies.

There are two types of dependency:

- ▶ **Functional dependency:** It determines whether a given task can execute on a given engine, according to the engine's operating system, the presence or absence of specific Python modules, or other conditions. ipyparallel provides a `@require` decorator for functions that need specific Python modules to run on the engines. Another decorator is `@depend`; it lets us define arbitrary conditions implemented in a Python function returning True or False.
- ▶ **Graph dependency:** It determines whether a given task can execute at a given time on a given engine. We may require a task to run only after one or several other tasks have finished. Additionally, we can impose this condition within any individual engine; an engine may need to execute a specific set of tasks before executing our task. For example, here is how to require tasks B and C (with asynchronous results `arB` and `arC`) to finish before task A starts:

```
with view.temp_flags(after=[arB, arC]):  
    arA = view.apply_async(f)
```

ipyparallel provides options to specify whether all or any of the dependencies should be met for the task to run. Additionally, we can specify whether success- and/or failure-dependent tasks should be considered as met or not.

When a task's dependency is unmet, the scheduler reassigns it to one engine, then to another engine, and so on until an appropriate engine is found. If the dependency cannot be met on any engine, an `ImpossibleDependency` error is raised for the task.

Passing data between dependent tasks is not particularly easy with ipyparallel. One initial possibility is to use blocking calls in the interactive session, wait for tasks to finish, retrieve the results, and send them back to the next tasks. Another possibility is to share data between engines via the filesystem, but this solution does not work well on multiple computers. An alternative solution is described at: <http://nbviewer.ipython.org/gist/minrk/11415238>.

References

Here are a few references about ipyparallel:

- ▶ Documentation for ipyparallel available at <https://ipyparallel.readthedocs.io/en/latest/>
- ▶ Dependencies in ipyparallel, explained at <https://ipyparallel.readthedocs.io/en/latest/task.html#dependencies>
- ▶ DAG dependencies, described at https://ipyparallel.readthedocs.io/en/latest/dag_dependencies.html
- ▶ Using MPI with ipyparallel, at <http://ipyparallel.readthedocs.io/en/latest/mpi.html>

Here are some references about alternative parallel computing solutions in Python:

- ▶ Dask, available at <https://dask.pydata.org/en/latest/>
- ▶ Joblib, available at <http://pythonhosted.org/joblib/parallel.html>
- ▶ List of parallel computing packages, available at <http://wiki.python.org/moin/ParallelProcessing>

See also

- ▶ The *Interacting with asynchronous parallel tasks in IPython* recipe
- ▶ The *Performing out-of-core computations on large arrays with Dask* recipe

Interacting with asynchronous parallel tasks in IPython

In this recipe, we will show how to interact with asynchronous tasks running in parallel with ipyparallel.

Getting ready

You need to start the IPython engines (see the previous recipe). The simplest option is to launch them from the IPython **Clusters** tab in the Notebook dashboard. In this recipe, we use four engines.

How to do it...

1. Let's import a few modules:

```
>>> import sys
      import time
      import ipyparallel
      import ipywidgets
      from IPython.display import clear_output, display
```

2. We create a client:

```
>>> rc = ipyparallel.Client()
```

3. Now, we create a load-balanced view on the IPython engines:

```
>>> view = rc.load_balanced_view()
```

4. We define a simple function for our parallel tasks:

```
>>> def f(x):
      import time
      time.sleep(.1)
      return x * x
```

5. We will run this function on 100 integer numbers in parallel:

```
>>> numbers = list(range(100))
```

6. We execute `f` on our list numbers in parallel across all of our engines, using `map_async()`. This function immediately returns an `AsyncResult` object that allows us to interactively retrieve information about the tasks:

```
>>> ar = view.map_async(f, numbers)
```

7. This object has a `metadata` attribute: a list of dictionaries for all engines. We can get the date of submission and completion, the status, the standard output and error, and other information:

```
>>> ar.metadata[0]
{'after': None,
 'completed': None,
 'data': {},
 ...
 'submitted': datetime.datetime(2017, ...)}
```

8. Iterating over the `AsyncResult` instance works normally; the iteration progresses in real-time while the tasks are being completed:

```
>>> for i in ar:
      print(i, end=' ', ')
0, 1, 4, ..., 9801,
```

9. Now, we create a simple progress bar for our asynchronous tasks. The idea is to create a loop polling for the tasks' status at every second. An `IntProgressWidget` widget is updated in real-time and shows the progress of the tasks:

```
>>> def progress_bar(ar):  
    # We create a progress bar.  
    w = ipywidgets.IntProgress()  
    # The maximum value is the number of tasks.  
    w.max = len(ar.msg_ids)  
    # We display the widget in the output area.  
    display(w)  
    # Repeat:  
    while not ar.ready():  
        # Update the widget's value with the  
        # number of tasks that have finished  
        # so far.  
        w.value = ar.progress  
        time.sleep(.1)  
    w.value = w.max  
>>> ar = view.map_async(f, numbers)
```

The progress bar is shown in the following screenshot:

```
>>> progress_bar(ar)
```



How it works...

`AsyncResult` instances are returned by asynchronous parallel functions. They implement several useful attributes and methods, notably:

- ▶ `elapsed`: Elapsed time since submission
- ▶ `progress`: Number of tasks that have completed so far
- ▶ `serial_time`: Sum of the computation time of all of the tasks done in parallel
- ▶ `metadata`: Dictionary with further information about the task
- ▶ `ready()`: Returns whether the call has finished
- ▶ `successful()`: Returns whether the call has completed without raising an exception (an exception is raised if the task has not completed yet)

- ▶ `wait()`: Blocks until the tasks have completed (there is an optional timeout argument)
- ▶ `get()`: Blocks until the tasks have completed and returns the result (there is an optional timeout argument)

There's more...

Here are a few references:

- ▶ Documentation for the `AsyncResult` class available at <http://ipyparallel.readthedocs.io/en/latest/AsyncResult.html>
- ▶ Documentation for the `AsyncResult` of the native multiprocessing module at <https://docs.python.org/3/library/multiprocessing.html#multiprocessing.pool.AsyncResult>
- ▶ Documentation for the task interface available at <http://ipyparallel.readthedocs.io/en/latest/task.html>

See also

- ▶ The *Distributing Python code across multiple cores with IPython* recipe

Performing out-of-core computations on large arrays with Dask

Dask is a parallel computing library that offers not only a general framework for distributing complex computations on many nodes, but also a set of convenient high-level APIs to deal with out-of-core computations on large arrays. Dask provides data structures resembling NumPy arrays (`dask.array`) and Pandas DataFrames (`dask.dataframe`) that efficiently scale to huge datasets. The core idea of Dask is to split a large array into smaller arrays (chunks).

In this recipe, we illustrate the basic principles of `dask.array`.

Getting ready

Dask should already be installed in Anaconda, but you can always install it manually with `conda install dask`. You also need `memory_profiler`, which you can install with `conda install memory_profiler`.

How to do it...

1. Let's import the libraries:

```
>>> import numpy as np
      import dask.array as da
      import memory_profiler
>>> %load_ext memory_profiler
```

2. We initialize a large $10,000 \times 10,000$ array with random values using dask. The array is chunked into 100 smaller arrays with size $1,000 \times 1,000$:

```
>>> Y = da.random.normal(size=(10000, 10000),
                         chunks=(1000, 1000))

>>> Y
dask.array<da.random.normal, shape=(10000, 10000),
           dtype=float64, chunksize=(1000, 1000)>
>>> Y.shape, Y.size, Y.chunks
((10000, 10000),
 1000000000,
((1000, ..., 1000),
 (1000, ..., 1000)))
```

Memory is not allocated for this huge array. Values will be computed on-the-fly at the last moment.

3. Let's say we want to compute the mean of every column:

```
>>> mu = Y.mean(axis=0)
      mu
dask.array<mean_aggregate, shape=(10000,) ,
           dtype=float64, chunksize=(1000,)>
```

This `mu` object is still a dask array and nothing has been computed yet.

4. We need to call the `compute()` method to actually launch the computation. Here, only part of the array is allocated because Dask is smart enough to compute just what is necessary for the computation. Here, the 10 chunks containing the first column of the array are allocated and involved in the computation of `mu[0]`:

```
>>> mu[0].compute()
0.011
```

5. Now, we profile the memory usage and time of the same computation using either NumPy or `dask.array`:

```
>>> def f_numpy():
      X = np.random.normal(size=(10000, 10000))
      x = X.mean(axis=0)[0:100]
>>> %%memit
```

```
f_numpy()
peak memory: 916.32 MiB, increment: 763.00 MiB
>>> %%time
    f_numpy()
CPU times: user 3.86 s, sys: 664 ms, total: 4.52 s
Wall time: 4.52 s
```

NumPy used 763 MB to allocate the entire array, and the entire process (allocation and computation) took more than 4 seconds. NumPy wasted time generating all random values and computing the mean of all columns whereas we only requested the first 100 columns.

6. Next, we use `dask.array` to perform the same computation:

```
>>> def f_dask():
    Y = da.random.normal(size=(10000, 10000),
                          chunks=(1000, 1000))
    y = Y.mean(axis=0)[0:100].compute()
>>> %%memit
    f_dask()
peak memory: 221.42 MiB, increment: 67.64 MiB
>>> %%time
    f_dask()
CPU times: user 492 ms, sys: 12 ms, total: 504 ms
Wall time: 105 ms
```

This time, Dask used only 67 MB and the computation lasted about 100 milliseconds.

7. We can do even better by changing the shape of the chunks. Instead of using 100 square chunks, we use 100 rectangular chunks containing 100 full columns each. The size of the chunks (10,000 elements) remains the same:

```
>>> def f_dask2():
    Y = da.random.normal(size=(10000, 10000),
                          chunks=(10000, 100))
    y = Y.mean(axis=0)[0:100].compute()
>>> %%memit
    f_dask2()
peak memory: 145.60 MiB, increment: 6.93 MiB
>>> %%time
    f_dask2()
CPU times: user 48 ms, sys: 8 ms, total: 56 ms
Wall time: 57.4 ms
```

This is more efficient when computing per-column quantities, because only a single chunk is involved in the computation of the mean of the first 100 columns, compared to 10 chunks in the previous example. The memory usage is therefore 10 times lower here.

8. Finally, we illustrate how we can use multiple cores to perform computations on large arrays. We create a client using `dask.distributed`, a distributed computing library that complements `dask`:

```
>>> from dask.distributed import Client  
>>> client = Client()  
>>> client
```

9. The computation represented by the `Y.sum()` Dask array can be launched locally, or using the `dask.distributed` client:

```
>>> Y.sum().compute()  
4090.221  
>>> future = client.compute(Y.sum())  
>>> future
```

Future: finalize status: finished, type: float64, key: finalize-f148208bfc12510be7a62b1d0c5cba82

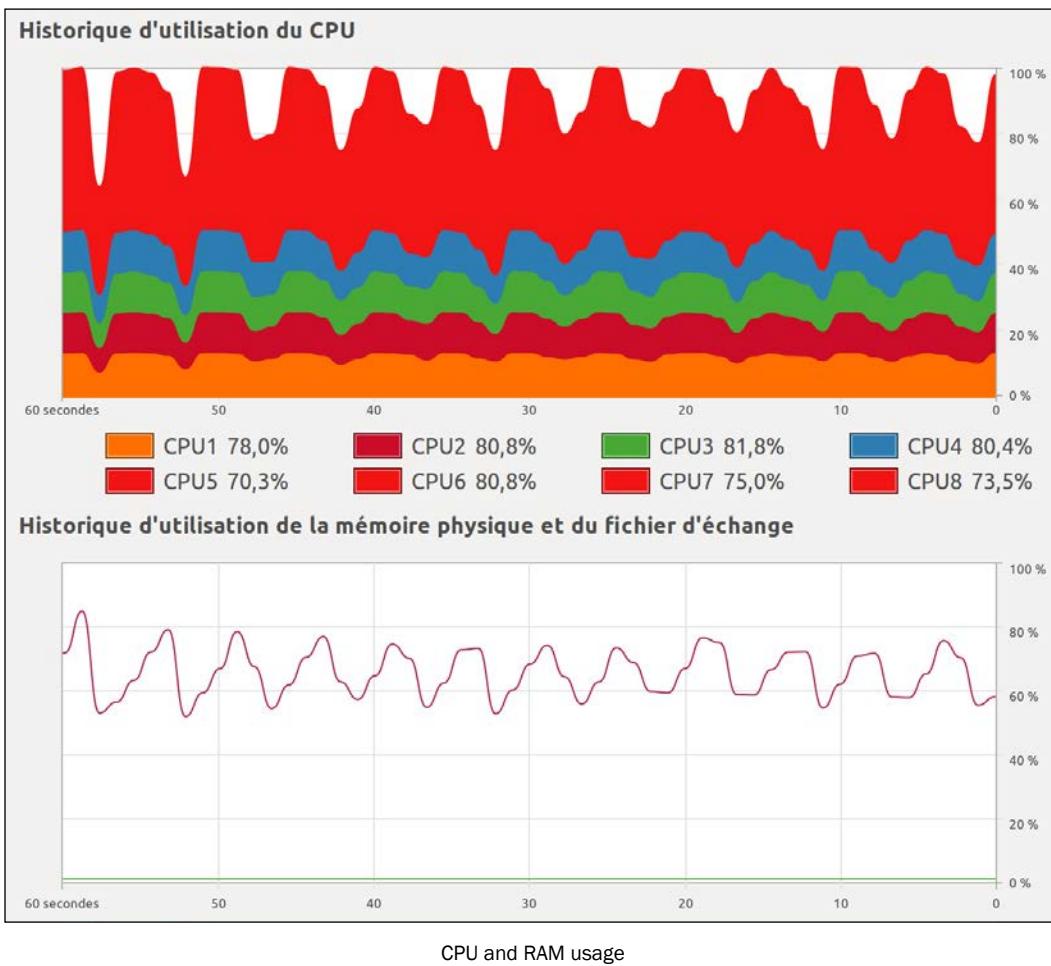
```
>>> future.result()  
4090.221
```

The second method scales to large clusters containing many nodes.

10. We have seen how `dask.array` can help us manage larger datasets in less memory. Now, we show how we can manipulate arrays that would never fit in our computer. For example, let's compute the average of a large terabyte array:

```
>>> huge = da.random.uniform(  
...     size=(1500000, 100000), chunks=(10000, 10000))  
... "Size in memory: %.1f GB" % (huge.nbytes / 1024 ** 3)  
'Size in memory: 1117.6 GB'  
>>> from dask.diagnostics import ProgressBar  
# WARNING: this will take a very long time computing  
# useless values. This is for pedagogical purposes  
# only.  
with ProgressBar():  
    m = huge.mean().compute()  
[##                                ] | 11% Completed | 1min 44.8s
```

The way this task is processed, chunk after chunk, can be seen on this graphic showing CPU and RAM usage as a function of time:



There's more...

The `dask.array` interface shown here is just one of the many possibilities offered by the low-level, graph-based distributed computing framework implemented in Dask. With **task scheduling**, a large computation is split into many smaller computations that may have complex dependencies represented by a dependency graph. A scheduler implements algorithms to execute these computations in parallel by respecting the dependencies.

Here are a few references:

- ▶ Dask documentation at <https://dask.pydata.org/en/latest/index.html>
- ▶ Integrating Dask with IPython at <http://distributed.readthedocs.io/en/latest/ipython.html>
- ▶ Dask examples at <https://dask.pydata.org/en/latest/examples-tutorials.html>
- ▶ *Parallelizing Scientific Python with Dask*, by James Crist, SciPy 2017, a video tutorial at <https://www.youtube.com/watch?v=mbfso3e5DA>
- ▶ Dask tutorial at <https://github.com/dask/dask-tutorial/>

See also

- ▶ The *Distributing Python code across multiple cores with IPython* recipe
- ▶ The *Interacting with asynchronous parallel tasks in IPython* recipe

Trying the Julia programming language in the Jupyter Notebook

Julia (<http://julialang.org>) is a high-level, dynamic language for high-performance numerical computing. The first version was released in 2012 after three years of development at MIT. Julia borrows ideas from Python, R, MATLAB, Ruby, Lisp, C, and other languages. Its major strength is to combine the expressivity and ease of use of high-level, dynamic languages with the speed of C (almost). This is achieved via an LLVM-based JIT compiler that targets machine code for x86-64 architectures.

In this recipe, we will try Julia in the Jupyter Notebook using the **Julia** package available at <https://github.com/JuliaLang/IJulia.jl>. We will also show how to use Python packages (such as NumPy and Matplotlib) from Julia. Specifically, we will compute and display a Julia set.

This recipe is inspired by a Julia tutorial given by David P. Sanders at the SciPy 2014 conference, available at the following:

http://nbviewer.ipython.org/github/dpsanders/scipy_2014_julia/tree/master/

Getting ready

You first need to install Julia. You will find packages for Windows, macOS, and Linux on Julia's website at <http://julialang.org/downloads/>.

Open a Julia Terminal with the `julia` command, and install IJulia by typing `Pkg.add("IJulia")` in the Julia Terminal. Then, quit Julia with `exit()` and launch the Jupyter Notebook as usual with `jupyter notebook`. The IJulia kernel is now available in Jupyter.

How to do it...

1. We can't avoid the customary *Hello World* example. The `println()` function displays a string and adds a line break at the end:

```
println("Hello world!")
Hello world!
```

2. We create a polymorphic function, `f`, that computes the expression `z*z + c`. We will evaluate this function on arrays, so we use element-wise operators with a dot (.) prefix:

```
f(z, c) = z.*z .+ c
f (generic function with 1 method)
```

3. Let's evaluate `f` on scalar complex numbers (the imaginary number `i` is `1.0im`):

```
f(2.0 + 1.0im, 1.0)
4.0 + 4.0im
```

4. Now, we create a (2, 2) matrix. Components are separated by a space and rows are separated by a semicolon (;). The type of this array is automatically inferred from its components. The `Array` type is a built-in data type in Julia, similar, but not identical, to NumPy's `ndarray` type:

```
z = [-1.0 - 1.0im 1.0 - 1.0im;
      -1.0 + 1.0im 1.0 + 1.0im]
2x2 Array{Complex{Float64},2}:
 -1.0-1.0im  1.0-1.0im
 -1.0+1.0im  1.0+1.0im
```

5. We can index arrays with brackets `[]`. A notable difference from Python is that indexing starts from 1 instead of 0. MATLAB has the same convention. Furthermore, the keyword `end` refers to the last item in that dimension:

```
z[1,end]
1.0 - 1.0im
```

6. We can evaluate `f` on the matrix `z` and a scalar `c` (polymorphism):

```
f(z, 0)
2x2 Array{Complex{Float64},2}:
 0.0+2.0im  0.0-2.0im
 0.0-2.0im  0.0+2.0im
```

7. Now, we create a function, `julia`, that computes a Julia set. Optional named arguments are separated from positional arguments by a semicolon (`;`). Julia's syntax for flow control is close to that of Python's, except that colons are dropped, indentation doesn't count, and block `end` keywords are mandatory:

```
function julia(z, c; maxiter=200)
    for n = 1:maxiter
        if abs2(z) > 4.0
            return n-1
        end
        z = f(z, c)
    end
    return maxiter
end
julia (generic function with 1 method)
```

8. We can use Python packages from Julia. First, we have to install the `PyCall` package by using Julia's built-in package manager (`Pkg`). Once the package is installed, we can use it in the interactive session with using `PyCall`:

```
Pkg.add("PyCall")
using PyCall
```

9. We can import Python packages with the `@pyimport` macro (a metaprogramming feature in Julia). This macro is the equivalent of Python's `import` command:

```
@pyimport numpy as np
```

10. The `np` namespace is now available in the Julia interactive session. NumPy arrays are automatically converted to Julia `Array` objects:

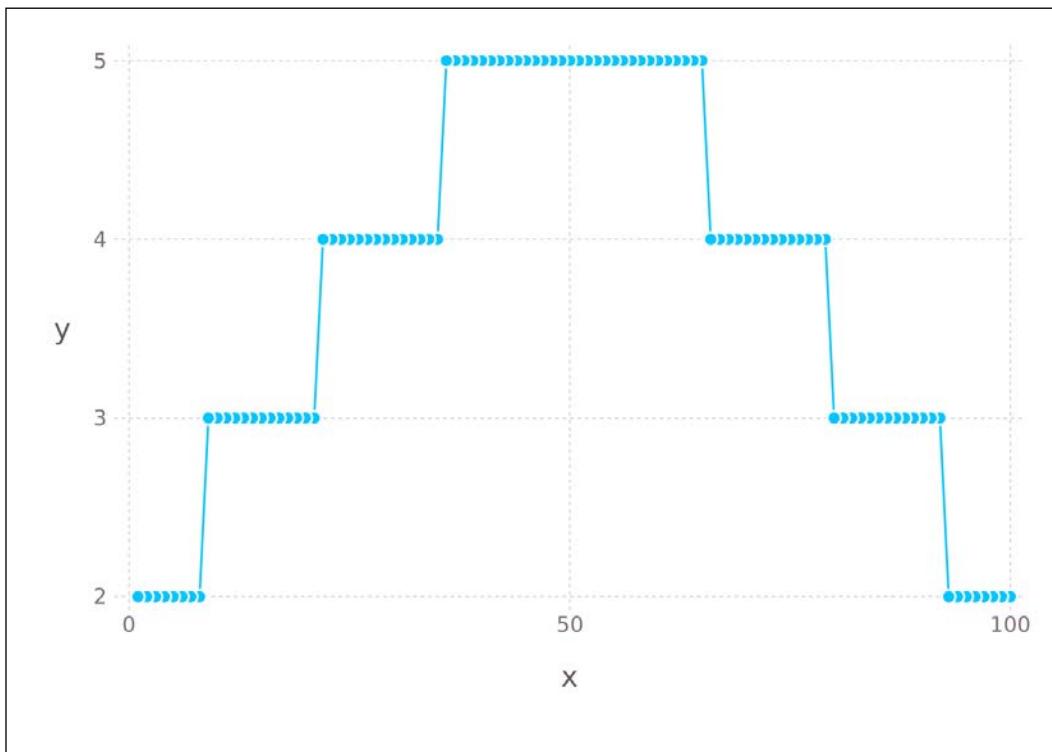
```
z = np.linspace(-1., 1., 100)
100-element Array{Float64,1}:
 -1.0
 -0.979798
 -0.959596
 .
 .
 .
 0.959596
 0.979798
 1.0
```

11. We can use list comprehensions to evaluate the function `julia` on many arguments:

```
m = [julia(z[i], 0.5) for i=1:100]
100-element Array{Int64,1}:
 2
 2
 .
 .
 .
 2
 2
```

12. Let's try the `Gadfly` plotting package. This library offers a high-level plotting interface inspired by Dr. Leland Wilkinson's textbook *The Grammar of Graphics*, Springer. In the Notebook, plots are interactive thanks to the `D3.js` library:

```
Pkg.add("Gadfly")
using Gadfly
plot(x=1:100, y=m, Geom.point, Geom.line)
```

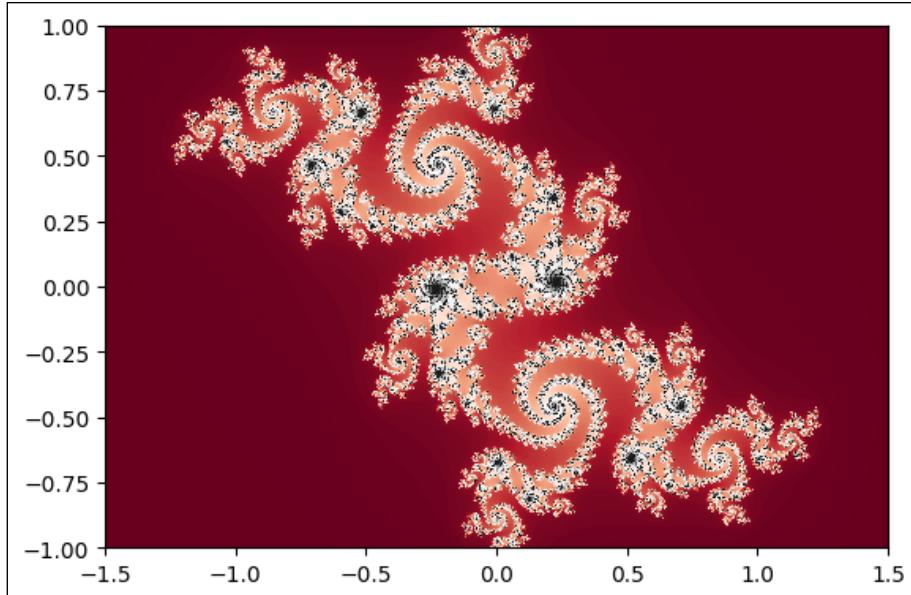


13. Now, we compute a Julia set by using two nested loops. In general, and unlike Python, there is no significant performance penalty in using `for` loops instead of vectorized operations. High-performance code can be written either with vectorized operations or `for` loops:

```
@time m = [julia(complex(r, i), complex(-0.06, 0.67))
           for i = 1:-.001:-1,
           r = -1.5:.001:1.5];
1.99 seconds (12.1 M allocations: 415.8 MiB)
```

14. Finally, we use the `PyPlot` package to draw Matplotlib figures in Julia:

```
Pkg.add("PyPlot")
using PyPlot
imshow(m, cmap="RdGy",
       extent=[-1.5, 1.5, -1, 1]);
```



How it works...

Languages used to be either low-level, difficult to use but fast (such as C); or high-level, easy to use but slow (such as Python). In Python, solutions to this problem include NumPy and Cython, among others.

Julia developers chose to create a new high-level but fast language, bringing the best of both worlds together. This is essentially achieved through JIT compilation techniques implemented with LLVM.

Julia dynamically parses code and generates low-level code in the LLVM **intermediate representation (IR)**. This representation features a language-independent instruction set that is then compiled to machine code. Code written with explicit loops is directly compiled to machine code. This explains why performance-motivated vectorization of code is generally not required with Julia.

There's more...

Strengths of Julia include:

- ▶ A powerful and flexible dynamic type system based on multiple dispatch for parametric polymorphism
- ▶ Facilities for metaprogramming
- ▶ A simple interface for calling C, FORTRAN, or Python code from Julia
- ▶ Built-in support for fine-grained parallel and distributed computing
- ▶ A built-in multidimensional array data type and numerical computing library
- ▶ A built-in package manager based on Git
- ▶ External packages for data analysis such as DataFrames (equivalent of Pandas) and Gadfly (a statistical plotting library)
- ▶ Integration in the Jupyter Notebook

The main strengths of Python as opposed to Julia are its wide community, ecosystem, and the fact that it is a general-purpose language. It is easy to bring numerical computing code written in Python to a Python-based production environment.

Fortunately, one may not have to choose because both Python and Julia can be used in the Jupyter Notebook, and there are ways to make both languages talk to each other via **PyCall** and **pyjulia**.

We have only scratched the surface of the Julia language in this recipe. Topics of interest we couldn't cover in details here include Julia's type system, its metaprogramming features, the support for parallel computing, and the package manager, among others.

Here are some references:

- ▶ The Julia language on Wikipedia available at [https://en.wikipedia.org/wiki/Julia_\(programming_language\)](https://en.wikipedia.org/wiki/Julia_(programming_language))
- ▶ Official documentation for Julia available at <http://docs.julialang.org/en/latest/>
- ▶ PyCall.jl for calling Python from Julia available at <https://github.com/stevengj/PyCall.jl>
- ▶ pyjulia for calling Julia from Python available at <https://github.com/JuliaPy/pyjulia>

- ▶ PyPlot.jl for using Matplotlib in Julia available at <https://github.com/stevengj/PyPlot.jl>
- ▶ Gadfly.jl, a Julia plotting library, available at <http://gadflyjl.org/stable/>
- ▶ DataFrames.jl, an equivalent of Pandas for Julia, available at <https://github.com/JuliaStats/DataFrames.jl>
- ▶ Juno, an IDE for Julia, available at <http://junolab.org/>

6

Data Visualization

In this chapter, we will cover the following topics:

- ▶ Using Matplotlib styles
- ▶ Creating statistical plots easily with seaborn
- ▶ Creating interactive web visualizations with Bokeh and HoloViews
- ▶ Visualizing a NetworkX graph in the Notebook with D3.js
- ▶ Discovering interactive visualization libraries in the Notebook
- ▶ Creating plots with Altair and the Vega-Lite specification

Introduction

While Matplotlib is the main visualization library in Python, it is not the only one. In this chapter, we will introduce some of the many other visualization libraries that cover more domain-specific use cases, or that offer specific interactivity features in the Jupyter Notebook.

Using Matplotlib styles

Recent versions of Matplotlib have significantly improved the default style of its figures. Today, Matplotlib comes with a set of high-quality predefined styles along with a styling system that lets one customize all aspects of these styles.

How to do it...

1. Let's import the libraries:

```
>>> import numpy as np  
      import matplotlib as mpl  
      import matplotlib.pyplot as plt  
      %matplotlib inline
```

2. Let's see a list of all available styles:

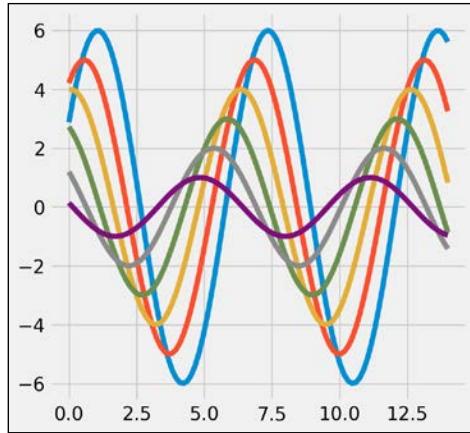
```
>>> sorted(mpl.style.available)  
['bmh',  
 'classic',  
 'dark_background',  
 'fivethirtyeight',  
 'ggplot',  
 'grayscale',  
 'mycustomstyle',  
 'seaborn',  
 ...  
 'seaborn-whitegrid']
```

3. We create a plot:

```
>>> def doplot():  
      fig, ax = plt.subplots(1, 1, figsize=(5, 5))  
      t = np.linspace(-2 * np.pi, 2 * np.pi, 1000)  
      x = np.linspace(0, 14, 100)  
      for i in range(1, 7):  
          ax.plot(x, np.sin(x + i * .5) * (7 - i))  
      return ax
```

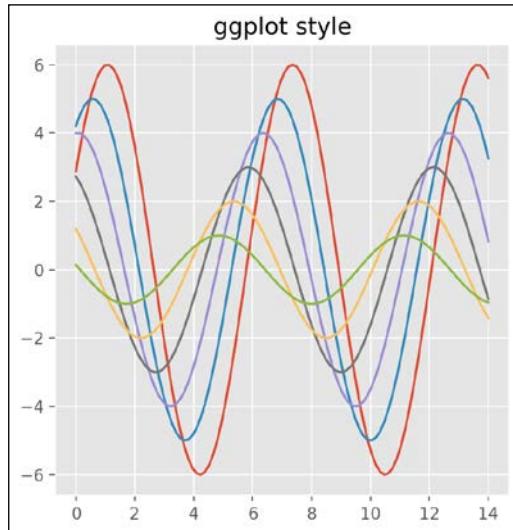
4. We can set a style with `mpl.style.use()`. All subsequent plots will use this style:

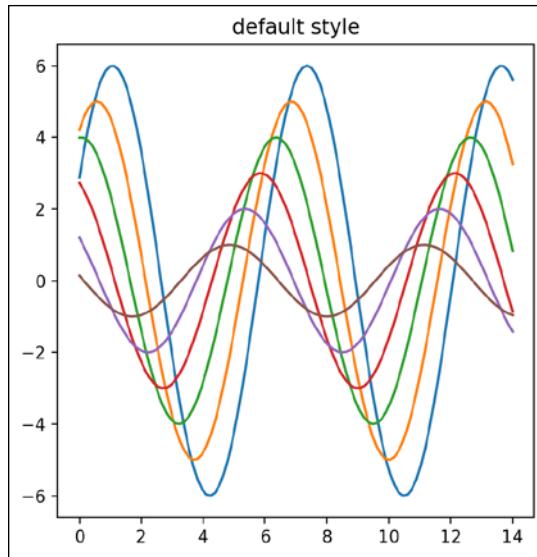
```
>>> mpl.style.use('fivethirtyeight')  
doplot()
```



5. We can temporarily change the style for a given plot using the context manager syntax:

```
>>> # Set the default style.  
    mpl.style.use('default')  
    # Temporarily switch to the ggplot style.  
    with mpl.style.context('ggplot'):  
        ax = doplot()  
        ax.set_title('ggplot style')  
    # Back to the default style.  
    ax = doplot()  
    ax.set_title('default style')
```





6. Now, we will customize the ggplot style by creating a new custom style to be applied in addition to ggplot. First, we specify the path to the custom style file, which should be in `mpl_configdir/stylelib/mycustomstyle.mplstyle`, where `mpl_configdir` is the Matplotlib config directory. Let's get this config directory:

```
>>> cfgdir = matplotlib.get_configdir()
      cfgdir
'/home/cyrille/.config/matplotlib'
```

7. We get the path to the file using the `pathlib` module:

```
>>> from pathlib import Path
      p = Path(cfgdir)
      stylelib = (p / 'stylelib')
      stylelib.mkdir(exist_ok=True)
      path = stylelib / 'mycustomstyle.mplstyle'
```

8. In this file, we specify a few parameters:

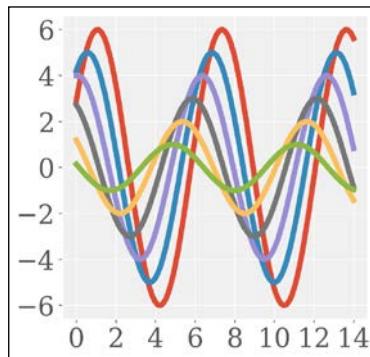
```
>>> path.write_text('''
      axes.facecolor : f0f0f0
      font.family : serif
      lines.linewidth : 5
      xtick.labelsize : 24
      ytick.labelsize : 24
      ''')
```

9. We need to reload the library after we add or change a style:

```
>>> mpl.style.reload_library()
```

10. Here is the result of the new style (we first apply the `ggplot` style, then we customize it by applying the options of our new style):

```
>>> with mpl.style.context(['ggplot', 'mycustomstyle']):  
    doplot()
```



There's more...

Here are a few references:

- ▶ *Customizing matplotlib*, at <http://matplotlib.org/users/customizing.html>
- ▶ *Matplotlib Style Gallery*, at https://tonysyu.github.io/raw_content/matplotlib-style-gallery/gallery.html
- ▶ *Matplotlib: beautiful plots with style*, at <http://www.futurile.net/2016/02/27/matplotlib-beautiful-plots-with-style/>

See also

- ▶ The *Creating statistical plots easily with seaborn* recipe

Creating statistical plots easily with seaborn

seaborn is a library that builds on top of Matplotlib and Pandas to provide easy-to-use statistical plotting routines. In this recipe, we give a few examples, adapted from the official documentation, of the types of statistical plot that can be created with seaborn.

How to do it...

- Let's import NumPy, Matplotlib, and seaborn:

```
>>> import numpy as np
      from scipy import stats
      import matplotlib.pyplot as plt
      import seaborn as sns
      %matplotlib inline
```

- seaborn comes with built-in datasets, which are useful when making demos.

The `tips` dataset contains bills and tips for taxi journeys:

```
>>> tips = sns.load_dataset('tips')
      tips
```

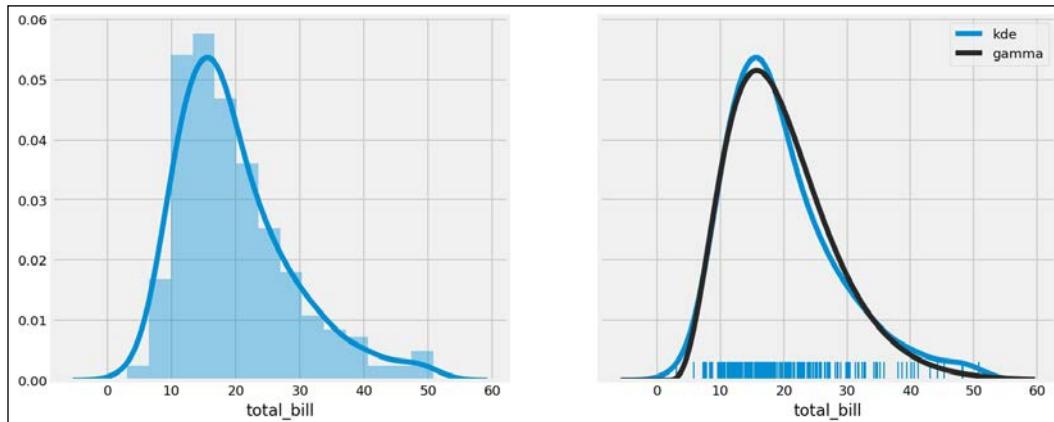
| | total_bill | tip | sex | smoker | day | time | size |
|-----|------------|------|--------|--------|------|--------|------|
| 0 | 16.99 | 1.01 | Female | No | Sun | Dinner | 2 |
| 1 | 10.34 | 1.66 | Male | No | Sun | Dinner | 3 |
| 2 | 21.01 | 3.50 | Male | No | Sun | Dinner | 3 |
| 3 | 23.68 | 3.31 | Male | No | Sun | Dinner | 2 |
| 4 | 24.59 | 3.61 | Female | No | Sun | Dinner | 4 |
| ... | ... | ... | ... | ... | ... | ... | ... |
| 239 | 29.03 | 5.92 | Male | No | Sat | Dinner | 3 |
| 240 | 27.18 | 2.00 | Female | Yes | Sat | Dinner | 2 |
| 241 | 22.67 | 2.00 | Male | Yes | Sat | Dinner | 2 |
| 242 | 17.82 | 1.75 | Male | No | Sat | Dinner | 2 |
| 243 | 18.78 | 3.00 | Female | No | Thur | Dinner | 2 |

244 rows × 7 columns

3. seaborn implements easy-to-use functions to visualize the distribution of datasets.

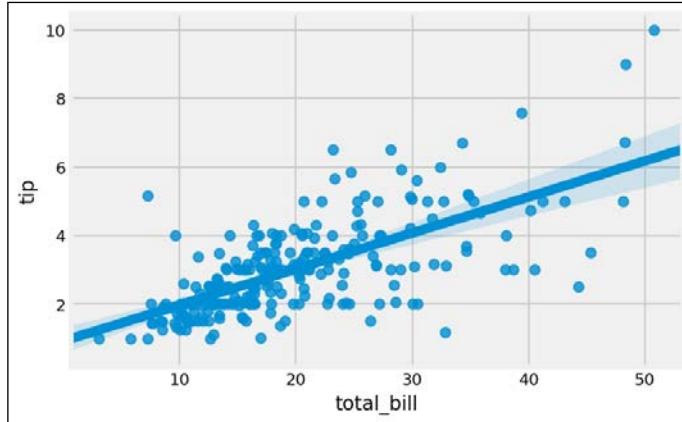
Here, we plot the histogram, **Kernel Density Estimation (KDE)**, and a gamma distribution fit for our dataset:

```
>>> # We create two subplots sharing the same y axis.  
f, (ax1, ax2) = plt.subplots(1, 2,  
                           figsize=(12, 5),  
                           sharey=True)  
  
# Left subplot.  
# Histogram and KDE (active by default).  
sns.distplot(tips.total_bill,  
             ax=ax1,  
             hist=True)  
  
# Right subplot.  
# "Rugplot", KDE, and gamma fit.  
sns.distplot(tips.total_bill,  
             ax=ax2,  
             hist=False,  
             kde=True,  
             rug=True,  
             fit=stats.gamma,  
             fit_kws=dict(label='gamma'),  
             kde_kws=dict(label='kde'))  
ax2.legend()
```



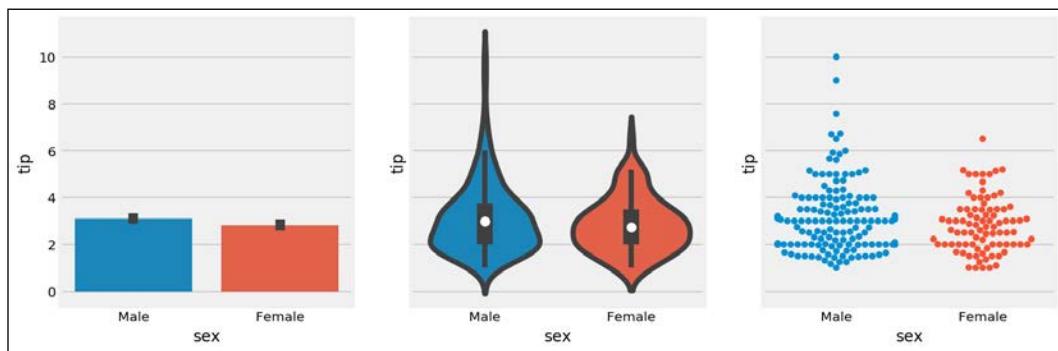
4. We can make a quick linear regression to visualize the correlation between two variables:

```
>>> sns.regplot(x="total_bill", y="tip", data=tips)
```



5. We can also visualize the distribution of categorical data with different types of plot. Here, we display a bar plot, a violin plot, and a swarm plot that show an increasing amount of details:

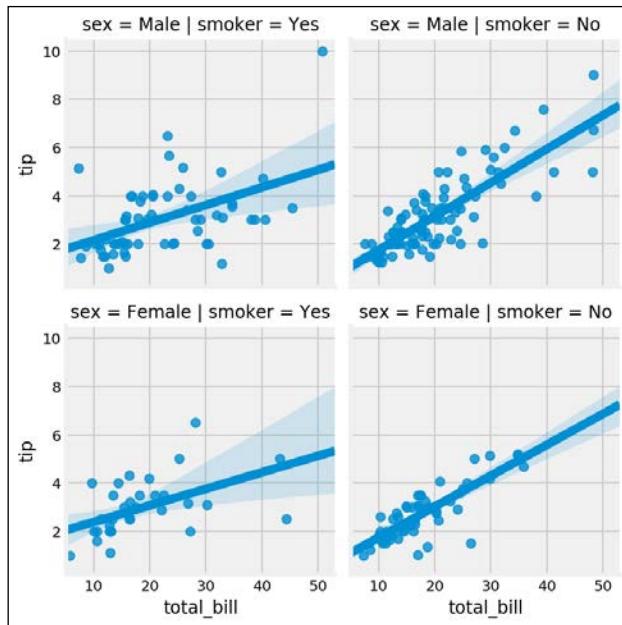
```
>>> f, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(12, 4), sharey=True)
sns.barplot(x='sex', y='tip', data=tips, ax=ax1)
sns.violinplot(x='sex', y='tip', data=tips, ax=ax2)
sns.swarmplot(x='sex', y='tip', data=tips, ax=ax3)
```



The bar plot shows the mean and standard deviation of the tip, for males and females. The violin plot shows an estimation of the distribution in a more informative way than the bar plot, especially with non-Gaussian or multimodal distributions. The swarm plot displays all points, using the non-informative x axis to make them non-overlapping.

6. `FacetGrid` lets us explore a multidimensional dataset with several subplots organized within a grid. Here, we plot the tip as a function of the bill, with a linear regression, for every combination of smoker (Yes/No) and sex (Male/Female):

```
>>> g = sns.FacetGrid(tips, col='smoker', row='sex')
g.map(sns.regplot, 'total_bill', 'tip')
```



There's more...

Besides seaborn, there are other high-level plotting interfaces:

- ▶ **Grammar of Graphics:** *The Grammar of Graphics*, Springer is a book by Dr. Leland Wilkinson that has influenced many high-level plotting interfaces such as R's `ggplot2`, Python's `ggplot` by `ytah`, and others.
- ▶ **Vega**, by Trifacta, is a declarative visualization grammar that can be translated to D3.js (a JavaScript visualization library). **Altair** provides a Python API for the Vega-Lite specification (a higher-level specification that compiles to Vega).

Here are some more references:

- ▶ seaborn tutorial at <https://seaborn.pydata.org/tutorial.html>
- ▶ seaborn gallery at <https://seaborn.pydata.org/examples/index.html>
- ▶ Altair, available at <https://altair-viz.github.io>

- ▶ plotnine, a Grammar of Graphics implementation in Python, at <https://plotnine.readthedocs.io/en/stable/>
- ▶ ggplot for Python available at <http://ggplot.yhathq.com/>
- ▶ ggplot2 for the R programming language, available at <http://ggplot2.org/>
- ▶ *Python Plotting for Exploratory Data Analysis* at <http://pythonplot.com/>

See also

- ▶ The *Using Matplotlib styles* recipe
- ▶ The *Discovering interactive visualization libraries in the Notebook* recipe
- ▶ The *Creating plots with Altair and the Vega-Lite specification* recipe

Creating interactive web visualizations with Bokeh and HoloViews

Bokeh (<http://bokeh.pydata.org/en/latest/>) is a library for creating rich interactive visualizations in a browser. Plots are designed in Python, and they are rendered in the browser.

In this recipe, we will give a few short examples of interactive Bokeh figures in the Jupyter Notebook. We will also introduce **HoloViews**, which provides a high-level API for Bokeh and other plotting libraries.

Getting ready

Bokeh should be installed by default in Anaconda, but you can also install it manually by typing `conda install bokeh` in a Terminal.

To install HoloViews, type `conda install -c ioam holoviews`.

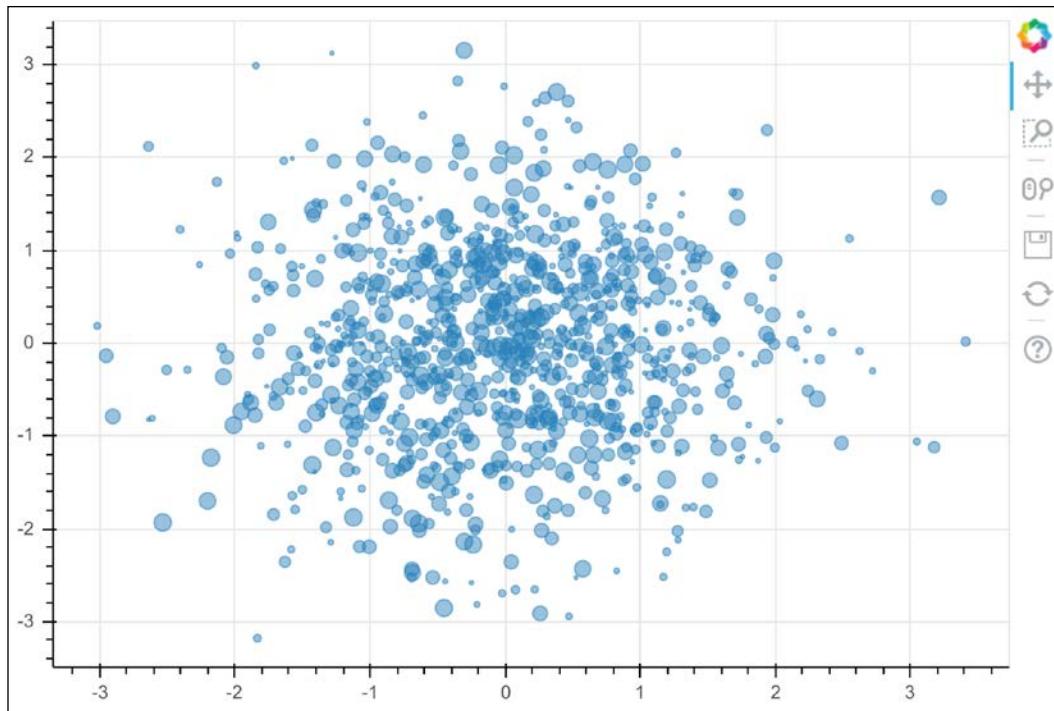
How to do it...

1. Let's import NumPy and Bokeh. We need to call `output_notebook()` to tell Bokeh to render plots in the Jupyter Notebook.

```
>>> import numpy as np
      import pandas as pd
      import bokeh
      import bokeh.plotting as bkh
      bkh.output_notebook()
```

2. Let's create a scatter plot of random data:

```
>>> f = bkh.figure(width=600, height=400)
    f.circle(np.random.randn(1000),
              np.random.randn(1000),
              size=np.random.uniform(2, 10, 1000),
              alpha=.5)
bkh.show(f)
```



An interactive plot is rendered in the notebook. We can pan and zoom by clicking on the toolbar buttons on the right.

3. Let's load a sample dataset, sea_surface_temperature:

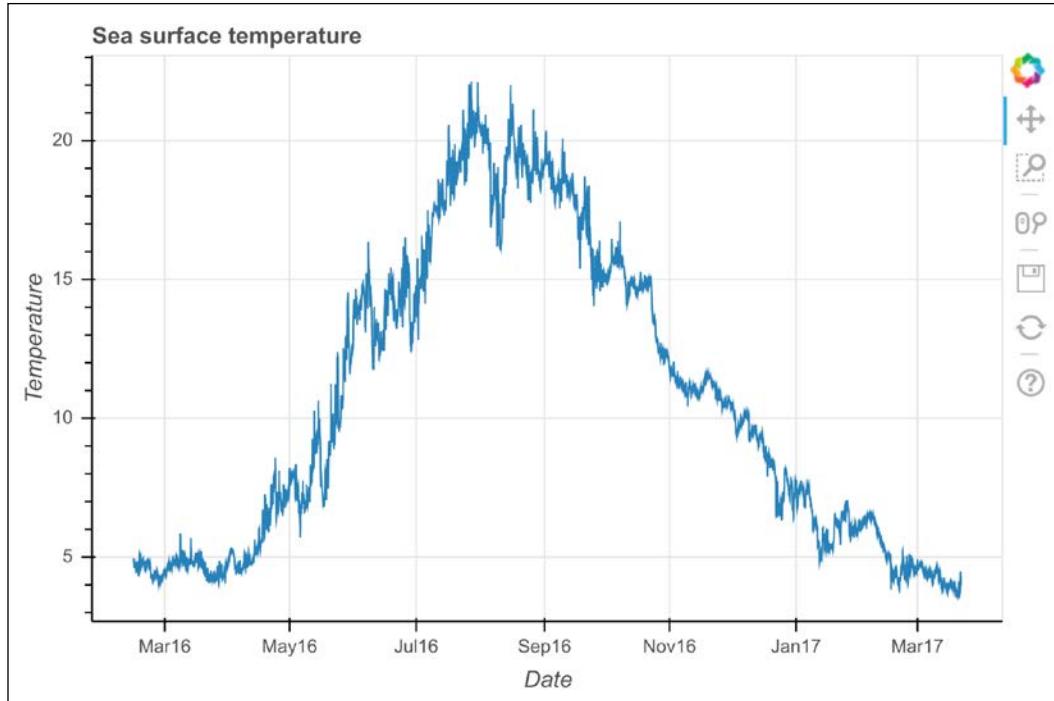
```
>>> from bokeh.sampledata import sea_surface_temperature
      data = sea_surface_temperature.sea_surface_temperature
      data
```

| time | temperature |
|---------------------|-------------|
| 2016-02-15 00:00:00 | 4.929 |
| 2016-02-15 00:30:00 | 4.887 |
| 2016-02-15 01:00:00 | 4.821 |
| 2016-02-15 01:30:00 | 4.837 |
| 2016-02-15 02:00:00 | 4.830 |
| ... | ... |
| 2017-03-21 22:00:00 | 4.000 |
| 2017-03-21 22:30:00 | 3.975 |
| 2017-03-21 23:00:00 | 4.017 |
| 2017-03-21 23:30:00 | 4.121 |
| 2017-03-22 00:00:00 | 4.316 |

19226 rows × 1 columns

4. Now, we plot the evolution of the temperature as a function of time:

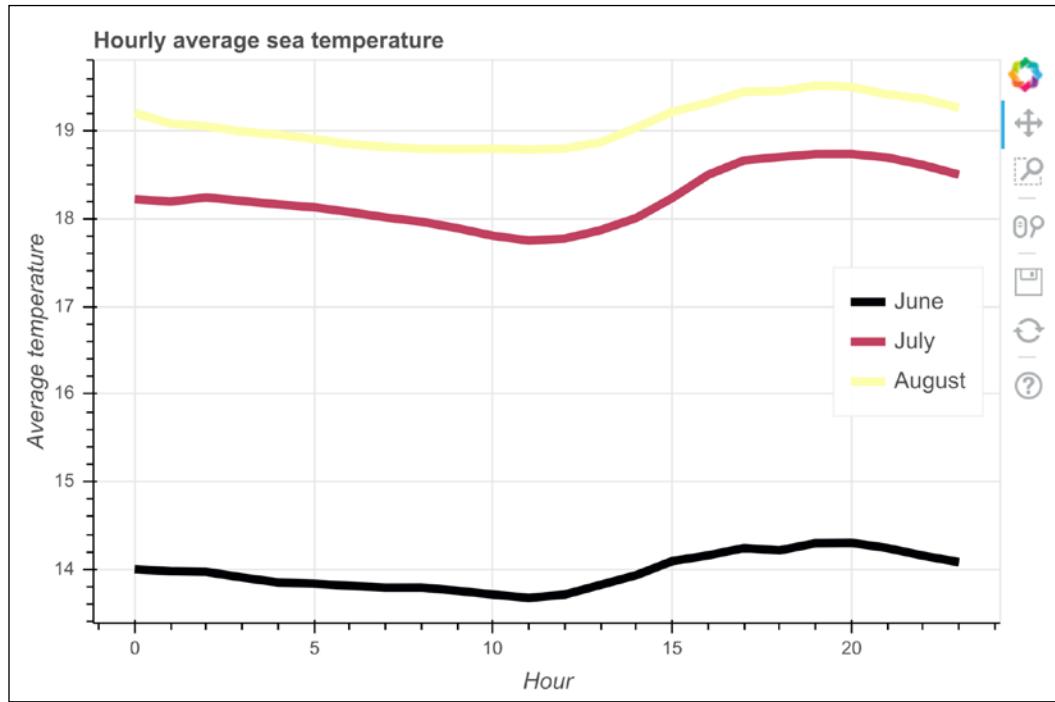
```
>>> f = bkh.figure(x_axis_type="datetime",
                     title="Sea surface temperature",
                     width=600, height=400)
    f.line(data.index, data.temperature)
    f.xaxis.axis_label = "Date"
    f.yaxis.axis_label = "Temperature"
    bkh.show(f)
```



5. We use Pandas to plot the hourly average temperature:

```
>>> months = (6, 7, 8)
      data_list = [data[data.index.month == m]
                  for m in months]
>>> # We group by the hour of the measure:
      data_avg = [d.groupby(d.index.hour).mean()
                  for d in data_list]
>>> f = bkh.figure(width=600, height=400,
                    title="Hourly average sea temperature")
for d, c, m in zip(data_avg,
                     bokeh.palettes.Inferno[3],
                     ('June', 'July', 'August')):
    f.line(d.index, d.temperature,
           line_width=5,
           line_color=c,
           legend=m,
           )
f.xaxis.axis_label = "Hour"
```

```
f.yaxis.axis_label = "Average temperature"  
f.legend.location = 'center_right'  
bkh.show(f)
```



6. Let's move on to HoloViews:

```
>>> import holoviews as hv  
hv.extension('bokeh')
```

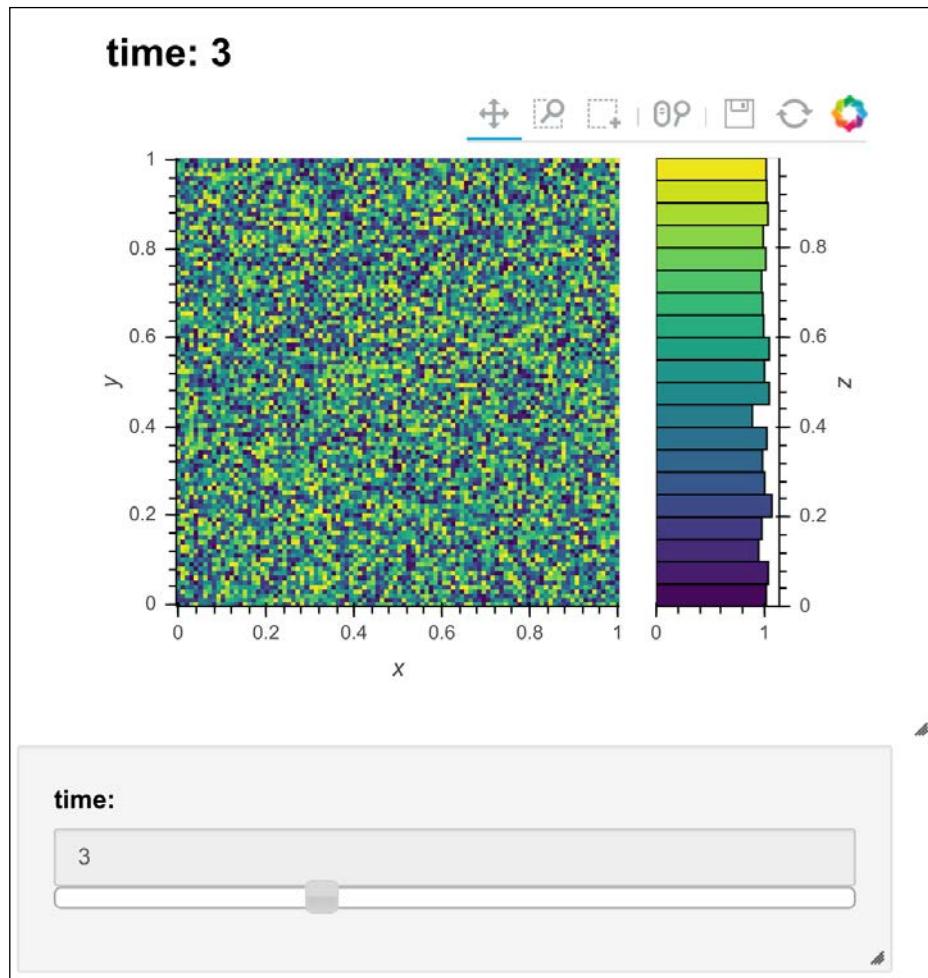
7. We create a 3D array that could represent a time-dependent 2D image:

```
>>> data = np.random.rand(100, 100, 10)  
>>> ds = hv.Dataset((np.arange(10),  
...                   np.linspace(0., 1., 100),  
...                   np.linspace(0., 1., 100),  
...                   data),  
...                   kdims=['time', 'y', 'x'],  
...                   vdims=['z'])  
>>> ds  
:Dataset [time,y,x] (z)
```

The `ds` object is a `Dataset` instance representing our time-dependent data. The **kdims** are the key dimensions (time and space) whereas the **vdims** are the quantities of interest (here, a scalar `z`). In other words, the kdims represent the axes of the 3D array data, whereas the vdims represent the values stored in the array.

8. We can easily display a 2D image with a slider to change the time, and a histogram of `z` as a function of time:

```
>>> %opts Image (cmap='viridis')
ds.to(hv.Image, ['x', 'y']).hist()
```



There's more...

Bokeh figures in the Notebook are interactive even in the absence of a Python server. For example, our figures can be interactive in nbviewer. Bokeh can also generate standalone HTML/JavaScript documents from our plots. More examples can be found in the gallery.

The **xarray** library (see <http://xarray.pydata.org/en/stable/>) provides a way to represent multidimensional arrays with axes. HoloViews can work with xarray objects.

Plotly is a company specializing in interactive visualization. It provides an open-source Python visualization library (see <https://plot.ly/python/>). It also proposes tools for building dashboard-style web-based interfaces (see <https://plot.ly/products/dash/>).

Datashader (<http://datashader.readthedocs.io/en/latest/>) and **vaex** (<http://vaex.astro.rug.nl/>) are two visualization libraries that target very large datasets.

Here are a few references:

- ▶ Bokeh user guide at http://bokeh.pydata.org/en/latest/docs/user_guide.html
- ▶ Bokeh gallery at <http://bokeh.pydata.org/en/latest/docs/gallery.html>
- ▶ Using Bokeh in the Notebook, available at http://bokeh.pydata.org/en/latest/docs/user_guide/notebook.html
- ▶ HoloViews at <http://holoviews.org>
- ▶ HoloViews gallery at <http://holoviews.org/gallery/index.html>
- ▶ HoloViews tutorial at <https://github.com/ioam/jupytercon2017-holoviews-tutorial>

Visualizing a NetworkX graph in the Notebook with D3.js

D3.js (<http://d3js.org>) is a popular interactive visualization framework for the web. Written in JavaScript, it allows us to create data-driven visualizations based on web technologies such as HTML, SVG, and CSS. The official gallery contains many examples (<https://github.com/d3/d3/wiki/gallery>). There are many other JavaScript visualization and charting libraries, but we will focus on D3.js in this recipe.

Being a pure JavaScript library, D3.js has in principle nothing to do with Python. However, the HTML-based Jupyter Notebook can integrate D3.js visualizations seamlessly.

In this recipe, we will create a graph in Python with NetworkX and visualize it in the Jupyter Notebook with D3.js.

Getting ready

You need to know the basics of HTML, JavaScript, and D3.js for this recipe.

How to do it...

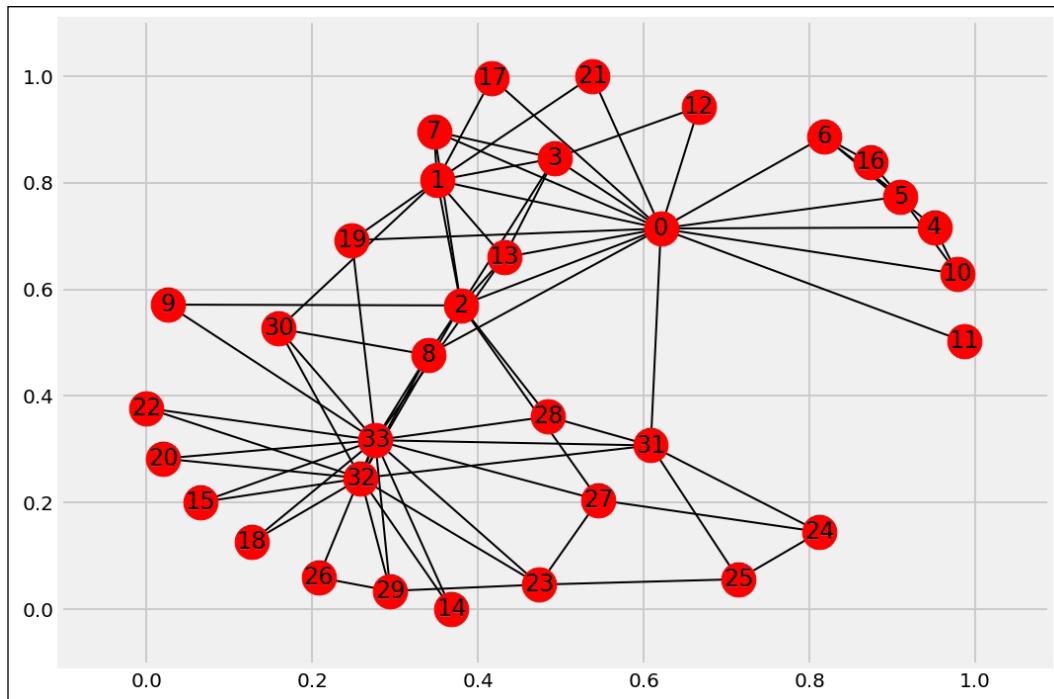
1. Let's import the packages:

```
>>> import json
      import numpy as np
      import networkx as nx
      import matplotlib.pyplot as plt
      %matplotlib inline
```

2. We load a famous social graph published in 1977 called **Zachary's Karate Club**

graph. This graph represents the friendships between members of a karate club. The club's president and the instructor were involved in a dispute, resulting in a schism in this group. Here, we simply display the graph with Matplotlib (using the `networkx.draw()` function):

```
>>> g = nx.karate_club_graph()
      fig, ax = plt.subplots(1, 1, figsize=(8, 6));
      nx.draw_networkx(g, ax=ax)
```



-
3. Now, we're going to display this graph in the Notebook with D3.js. The first step is to bring this graph to JavaScript. Here, we choose to export the graph to JSON. D3.js generally expects each edge to be an object with a source and target. Also, we specify which side each member has taken (the club attribute):

```
>>> nodes = [{"name": str(i), "club": g.node[i]['club']}  
            for i in g.nodes()]  
links = [{"source": u[0], "target": u[1]}  
          for u in g.edges()]  
with open('graph.json', 'w') as f:  
    json.dump({'nodes': nodes, 'links': links},  
              f, indent=4,)
```

4. The next step is to create an HTML object that will contain the visualization. Here, we create a `<div>` element in the notebook. We also specify a few CSS styles for nodes and links (also called edges):

```
>>> %%html  
<div id="d3-example"></div>  
<style>  
.node {stroke: #fff; stroke-width: 1.5px;}  
.link {stroke: #999; stroke-opacity: .6;}  
</style>
```

5. The last step is trickier. We write the JavaScript code to load the graph from the JSON file and display it with D3.js. Knowing the basics of D3.js is required here (see the documentation of D3.js):

```
>>> %%javascript  
// We load the d3.js library from the Web.  
require.config({paths:  
  {d3: "http://d3js.org/d3.v3.min"}});  
require(["d3"], function(d3) {  
  // The code in this block is executed when the  
  // d3.js library has been loaded.  
  
  // First, we specify the size of the canvas  
  // containing the visualization (size of the  
  // <div> element).  
  var width = 300, height = 300;  
  
  // We create a color scale.  
  var color = d3.scale.category10();  
  
  // We create a force-directed dynamic graph layout.  
  var force = d3.layout.force()
```

```
.charge(-120)
.linkDistance(30)
.size([width, height]);

// In the <div> element, we create a <svg> graphic
// that will contain our interactive visualization.
var svg = d3.select("#d3-example").select("svg")
if (svg.empty()) {
  svg = d3.select("#d3-example").append("svg")
    .attr("width", width)
    .attr("height", height);
}

// We load the JSON file.
d3.json("graph.json", function(error, graph) {
  // In this block, the file has been loaded
  // and the 'graph' object contains our graph.

  // We load the nodes and links in the
  // force-directed graph.
  force.nodes(graph.nodes)
    .links(graph.links)
    .start();

  // We create a <line> SVG element for each link
  // in the graph.
  var link = svg.selectAll(".link")
    .data(graph.links)
    .enter().append("line")
    .attr("class", "link");

  // We create a <circle> SVG element for each node
  // in the graph, and we specify a few attributes.
  var node = svg.selectAll(".node")
    .data(graph.nodes)
    .enter().append("circle")
    .attr("class", "node")
    .attr("r", 5) // radius
    .style("fill", function(d) {
      // The node color depends on the club.
      return color(d.club);
    })
})
```

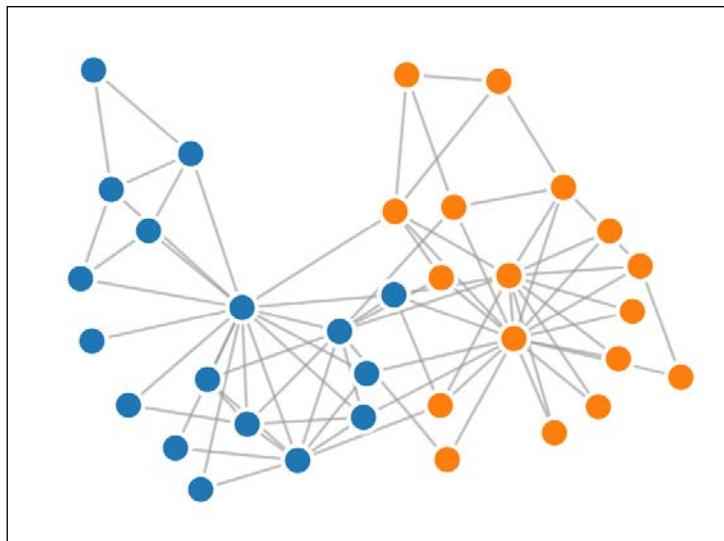
```
.call(force.drag);

// The name of each node is the node number.
node.append("title")
    .text(function(d) { return d.name; });

// We bind the positions of the SVG elements
// to the positions of the dynamic force-directed
// graph, at each time step.
force.on("tick", function() {
    link.attr("x1", function(d){return d.source.x})
        .attr("y1", function(d){return d.source.y})
        .attr("x2", function(d){return d.target.x})
        .attr("y2", function(d){return d.target.y});

    node.attr("cx", function(d){return d.x})
        .attr("cy", function(d){return d.y});
})
})
})
});
```

When we execute this cell, the HTML object created in the previous cell is updated. The graph is animated and interactive; we can click on nodes, see their labels, and move them within the canvas:



An interactive plot in the Notebook with D3.js

There's more...

NetworkX implements routines to import/export graphs from/into files in different formats.

Here are a few references:

- ▶ Reading and writing graphs with NetworkX, at <https://networkx.github.io/documentation/stable/reference/readwrite/index.html>
- ▶ NetworkX and JSON, at https://networkx.github.io/documentation/stable/reference/readwrite/json_graph.html

See also

- ▶ The *Creating interactive web visualizations with Bokeh and HoloViews* recipe

Discovering interactive visualization libraries in the Notebook

Several libraries provide interactive visualization of 2D or 3D data in the Notebook, using the capabilities of Jupyter widgets. We give basic examples using four of these libraries: **ipyleaflet**, **bqplot**, **pythreejs**, and **ipyvolume**.

Getting started

To install the libraries, type `conda install -c conda-forge ipyleaflet bqplot pythreejs ipyvolume` in a Terminal.

How to do it...

1. First, we show a simple example of **ipyleaflet**, which offers a Python interface to use the Leaflet.js interactive mapping library (similar to Google Maps, but based on the open source project OpenStreetMaps):

```
>>> from ipyleaflet import Map, Marker
```

2. We create a map around a given position specified in GPS coordinates:

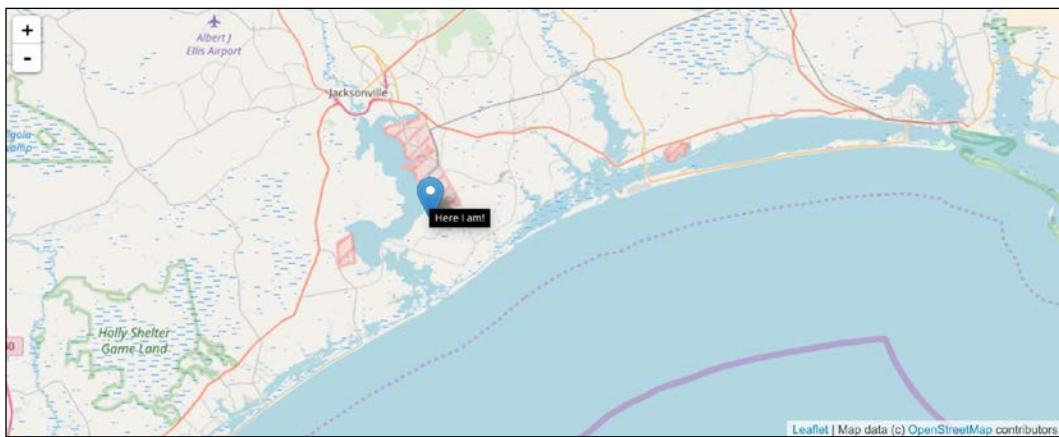
```
>>> pos = [34.62, -77.34]
m = Map(center=pos, zoom=10)
```

3. We also add a marker at that position:

```
>>> marker = Marker(location=pos,
                     rise_on_hover=True,
                     title="Here I am!",
                     )
>>> m += marker
```

4. We display the map in the notebook:

```
>>> m
```

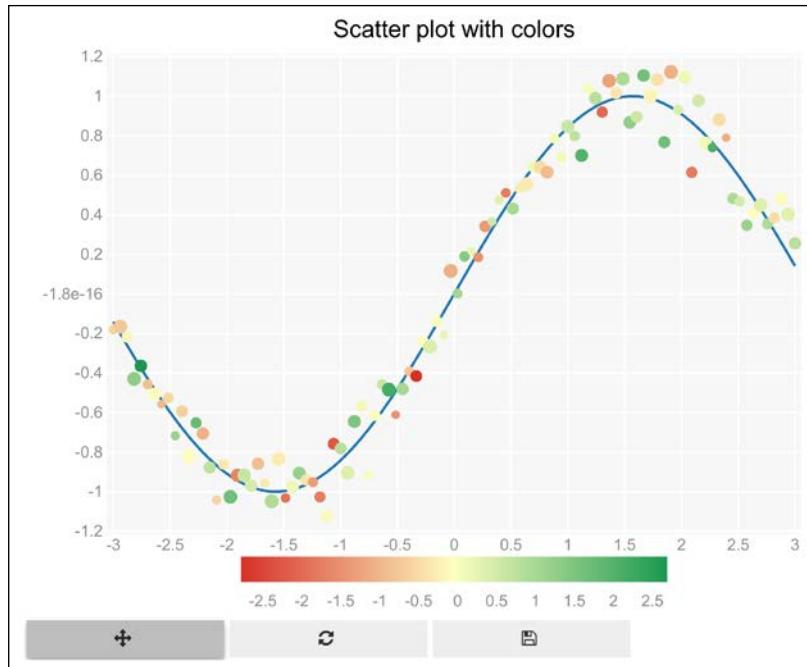


5. Let's move to the `bqplot` interactive plotting library, which implements an API inspired by *Grammar of Graphics*:

```
>>> import numpy as np
      import bqplot.pyplot as plt
```

6. We display an interactive plot using an API that should be familiar to any Matplotlib user:

```
>>> plt.figure(title='Scatter plot with colors')
      t = np.linspace(-3, 3, 100)
      x = np.sin(t)
      y = np.sin(t) + .1 * np.random.randn(100)
      plt.plot(t, x)
      plt.scatter(t, y,
                  size=np.random.uniform(15, 50, 100),
                  color=np.random.randn(100))
      plt.show()
```



7. Next, we show an example of `pythreejs`, a Python bridge to the `three.js` 3D library in JavaScript. This library uses WebGL, an API that leverages the GPU for fast real-time rendering in the browser:

```
>>> from pythreejs import *
```

8. We will display a parametric surface plot. We define the function as a string containing JavaScript code:

```
>>> f = """
        function f(x, y) {
            x = 2 * (x - .5);
            y = 2 * (y - .5);
            r2 = x * x + y * y;
            var z = Math.exp(-2 * r2) * (
                Math.cos(12*x) * Math.sin(12*y));
            return new THREE.Vector3(x, y, z)
        }
        """
```

9. We also create a texture for the surface:

```
>>> texture = np.random.uniform(.5, .9, (20, 20))
material = LambertMaterial(
    map=height_texture(texture))
```

10. We create ambient and directional lights:

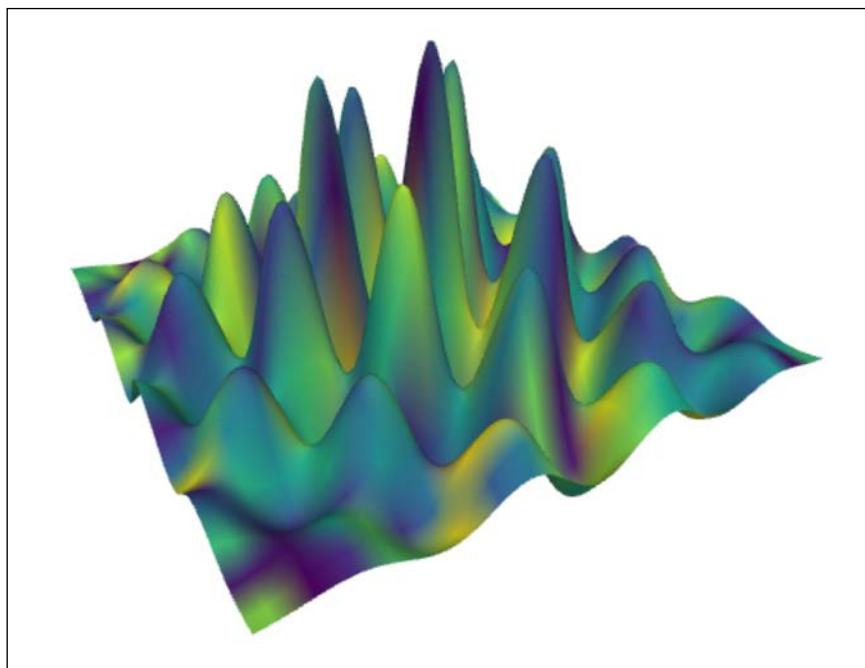
```
>>> alight = AmbientLight(color='#777777')
      dlight = DirectionalLight(color='white',
                                 position=[3, 5, 1],
                                 intensity=0.6)
```

11. We create the surface mesh:

```
>>> surf_g = ParametricGeometry(func=f)
      surf = Mesh(geometry=surf_g,
                  material=material)
```

12. Finally, we initialize the scene and the camera, and we display the plot:

```
>>> scene = Scene(children=[surf, alight])
      c = PerspectiveCamera(position=[2.5, 2.5, 2.5],
                            up=[0, 0, 1],
                            children=[dlight])
      Renderer(camera=c, scene=scene,
                controls=[OrbitControls(controlling=c)])
```

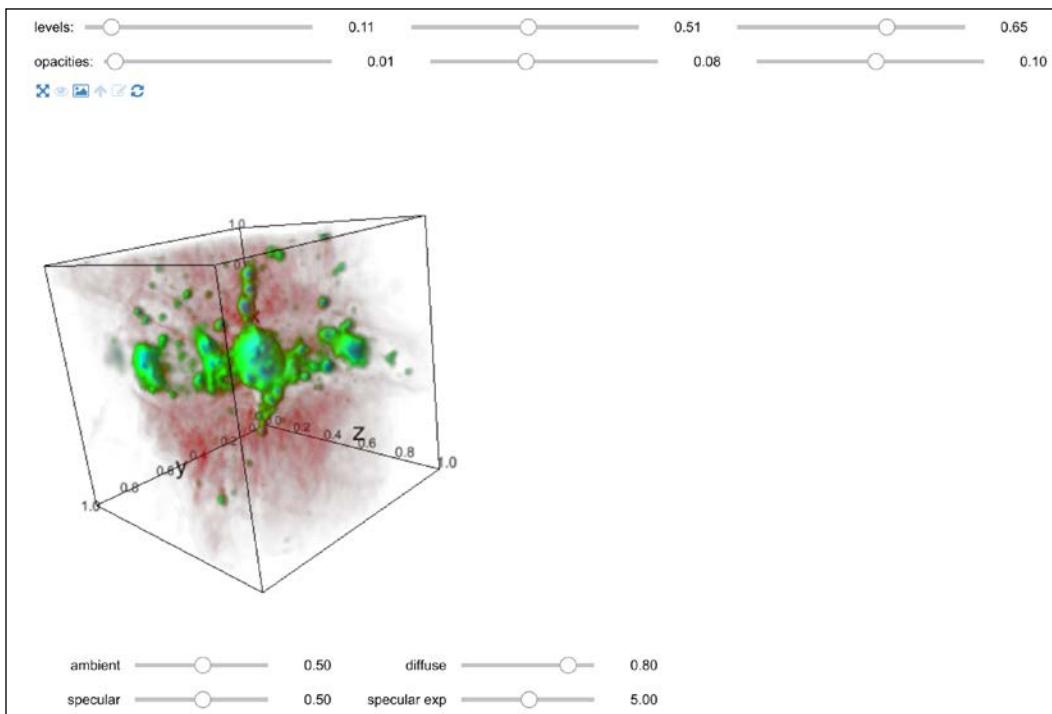


13. The last library is `ipyvolume`, a 3D plotting library in Python that also uses WebGL in the Notebook:

```
>>> import ipyvolume
```

14. This library provides volume rendering features, where a volumetric dataset represented as a 3D array is visualized using ray tracing techniques:

```
>>> ds = ipyvolume.datasets.aquariusA2.fetch()
ipyvolume.quickvolshow(ds.data, lighting=True)
```



There's more

Here are several references:

- ▶ Jupyter widgets at <http://jupyter.org/widgets.html>
- ▶ ipyleaflet at <https://github.com/ellisonbg/ipyleaflet>
- ▶ bqplot at <https://bqplot.readthedocs.io/en/stable/>
- ▶ pythreejs at <https://github.com/jovyan/pythreejs>
- ▶ three.js at <https://threejs.org/>

- ▶ ipyvolume at <https://github.com/maartenbreddels/ipyvolume>
- ▶ Jupyter Google Maps at <http://jupyter-gmaps.readthedocs.io/en/latest/>
- ▶ An interactive 3D molecular viewer for Jupyter, based on NGL, at <http://nglviewer.org/nglview/latest/>

Creating plots with Altair and the Vega-Lite specification

Vega is a declarative format for designing static and interactive visualizations. It provides a JSON-based visualization grammar that focuses on the *what* instead of the *how*. **Vega-Lite** is a higher-level specification that is easier to use than Vega, and that compiles directly to Vega.

Altair is a Python library that provides a simple API to define and display Vega-Lite visualizations. It works in the Jupyter Notebook, JupyterLab, and nteract.



Altair is under active development and some details of the API might change in future versions.



Getting started...

Install Altair with `conda install -c conda-forge altair`.

How to do it...

1. Let's import Altair:

```
>>> import altair as alt
```

2. Altair provides several example datasets:

```
>>> alt.list_datasets()
['airports',
 ...
 'driving',
 'flare',
 'flights-10k',
 'flights-20k',
 'flights-2k',
 'flights-3m',
 'flights-5k',
 'flights-airport',
 'gapminder',
```

```
...  
'wheat',  
'world-110m']
```

3. We load the flights-5k dataset:

```
>>> df = alt.load_dataset('flights-5k')
```

The `load_dataset()` function returns a Pandas DataFrame.

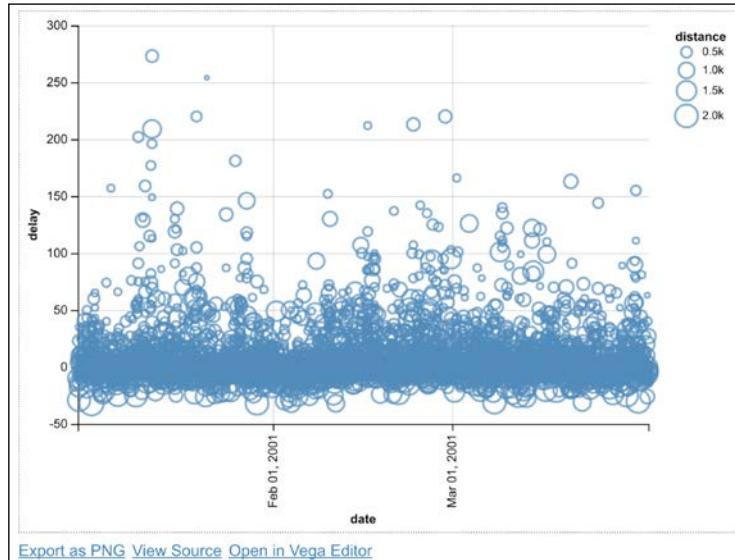
```
>>> df.head(3)
```

| | date | delay | destination | distance | origin |
|---|--------------------|-------|-------------|----------|--------|
| 0 | 2001-01-10 18:2... | 25 | HOU | 192 | SAT |
| 1 | 2001-01-31 16:4... | 17 | OAK | 371 | SNA |
| 2 | 2001-02-16 12:0... | 21 | SAN | 417 | SJC |

This dataset provides the date, origin, destination, flight distance, and delay for many flights.

4. Let's create a scatter plot showing the delay as a function of the date, with the marker size depending on the flight distance:

```
>>> alt.Chart(df).mark_point().encode(  
    x='date',  
    y='delay',  
    size='distance',  
)
```



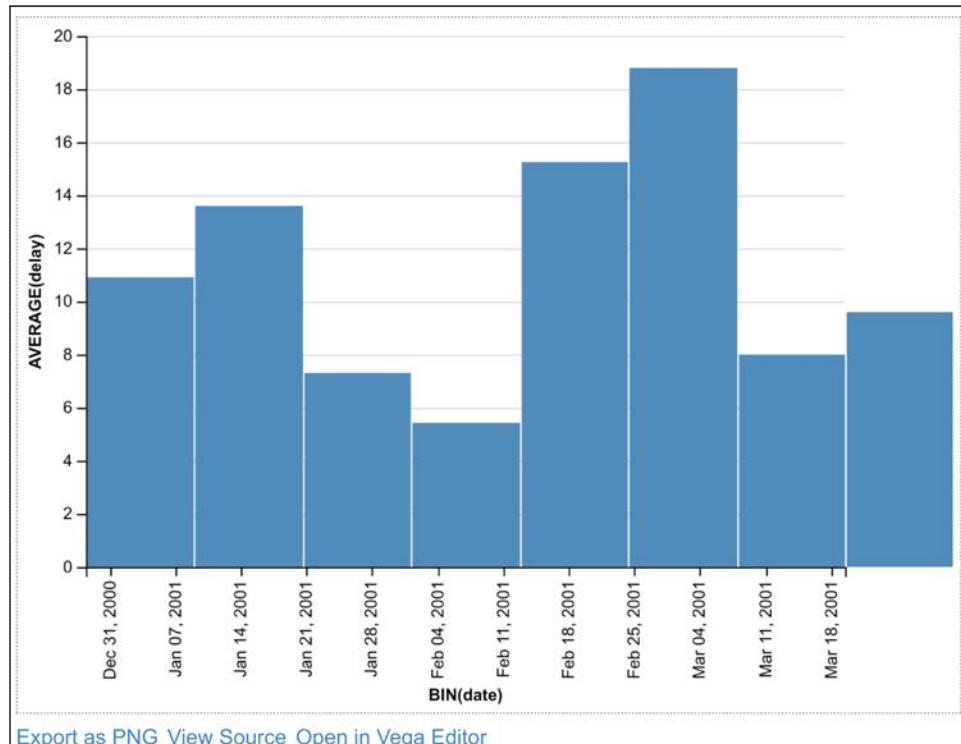
The `mark_point()` method specifies that we're creating a scatter plot. The `encode()` function allows us to link parameters of the plot (the x and y coordinates and the point size) to specific columns in our DataFrame.

5. Next, we create a bar plot with the average delay of all flights departing from Los Angeles, as a function of time:

```
>>> df_la = df[df['origin'] == 'LAX']

x = alt.X('date', bin=True)
y = 'average(delay)'

alt.Chart(df_la).mark_bar().encode(
    x=x,
    y=y,
)
```



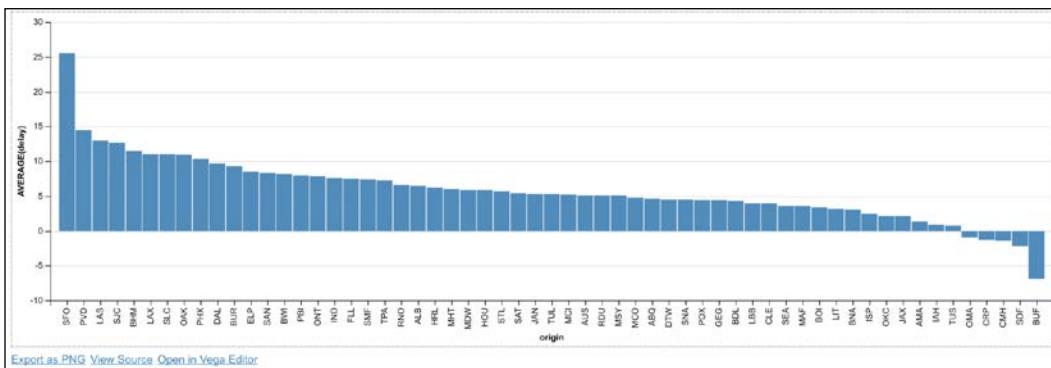
We select all flights departing from the LAX airport using Pandas. For the x coordinate, we use the `alt.X` class to specify that we want a histogram (`bin=True`). For the y coordinate, we specify the average of all delays for every bin.

6. Now, we create a histogram of the average delay of every origin airport. We use the sort option of the x class to specify that we want to order the x axis (origin) as a function of the average delay, in descending order:

```
>>> sort_delay = alt.SortField(
    'delay', op='average', order='descending')

x = alt.X('origin', sort=sort_delay)
y = 'average(delay'

alt.Chart(df).mark_bar().encode(
    x=x,
    y=y,
)
```



How it works...

Altair provides a Python API to generate a Vega-Lite specification in JSON. The `to_json()` method of an Altair chart can be used to inspect the JSON created by Altair. For example, here is the JSON for the last chart example:

```
{
  "$schema": "https://vega.github.io/schema/vega-lite/v1.2.1.json",
  "data": {
    "values": [
      {
        "date": "2001-01-10 18:20:00",
        "delay": 25,
        "destination": "HOU",
        "distance": 192,
        "origin": "SAT"
      },
      ...
    ]
  }
}
```

```
    ...
  ],
},
"encoding": {
  "x": {
    "field": "origin",
    "sort": {
      "field": "delay",
      "op": "average",
      "order": "descending"
    },
    "type": "nominal"
  },
  "y": {
    "aggregate": "average",
    "field": "delay",
    "type": "quantitative"
  }
},
"mark": "bar"
}
```

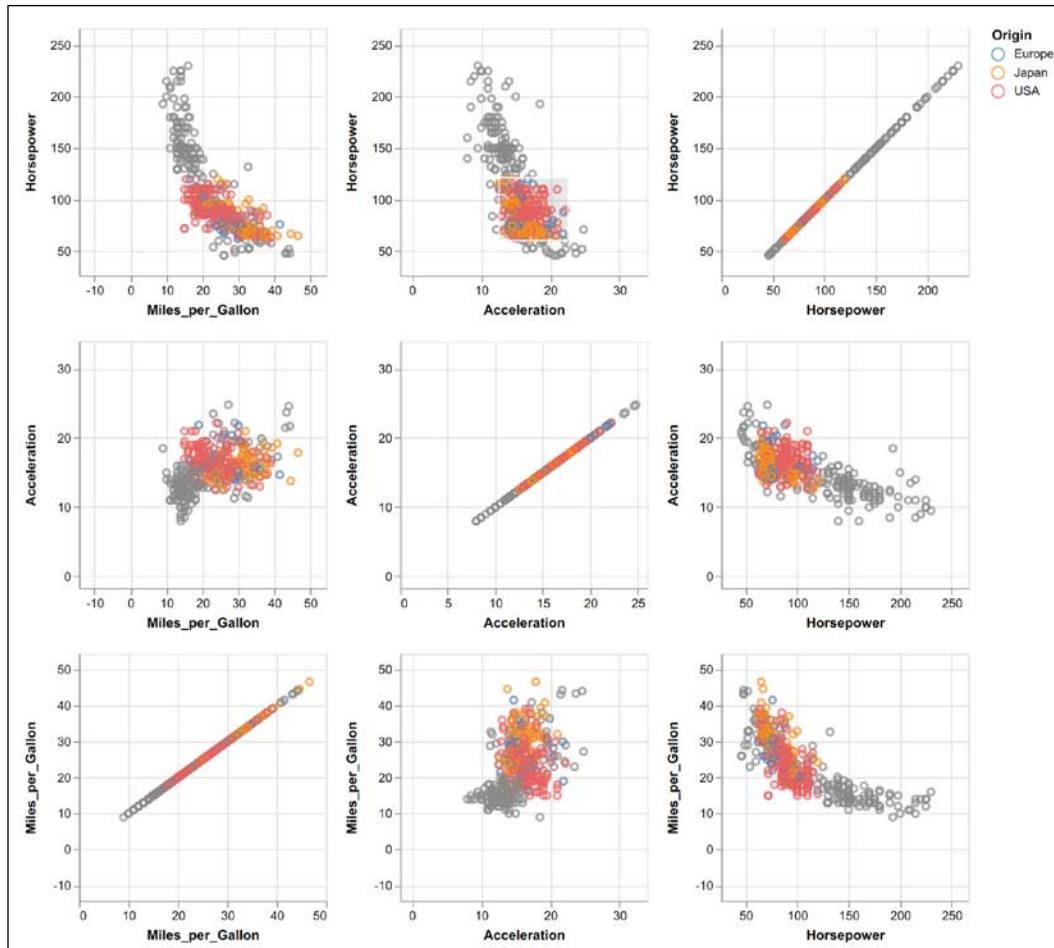
The JSON may contain the data itself, like here, or a URL to a data file. It also defines the *encoding channels* that link the chart parameters to the data.

In the Jupyter Notebook, Altair leverages the Vega-Lite library to create a Canvas or SVG figure with the requested plot.

There's more...

Altair and Vega-Lite support much more complex charts, as can be seen in the galleries for these projects.

Vega-Lite supports interactive plots. The following example from the Vega-Lite gallery illustrates linked brushing between subplots, where a rectangular selection can be drawn with the mouse in any subplot:



Linked brushing

There is also an online editor on the Vega-Lite website that can be used to create plots directly in the browser without installing anything.

Here are a few references:

- ▶ Altair documentation at <https://altair-viz.github.io/>
- ▶ Altair gallery at <https://altair-viz.github.io/gallery/index.html>
- ▶ Vega-Lite documentation at <https://vega.github.io/vega-lite/>
- ▶ Vega-Lite gallery at <https://vega.github.io/vega-lite/examples/>
- ▶ Vega-Lite online editor at <https://vega.github.io/editor/#/custom/vega-lite>

See also

- ▶ The *Creating statistical plots easily with seaborn* recipe
- ▶ The *Discovering interactive visualization libraries in the Notebook* recipe

7

Statistical Data Analysis

In this chapter, we will cover the following topics:

- ▶ Exploring a dataset with pandas and Matplotlib
- ▶ Getting started with statistical hypothesis testing – a simple z-test
- ▶ Getting started with Bayesian methods
- ▶ Estimating the correlation between two variables with a contingency table and a chi-squared test
- ▶ Fitting a probability distribution to data with the maximum likelihood method
- ▶ Estimating a probability distribution nonparametrically with a kernel density estimation
- ▶ Fitting a Bayesian model by sampling from a posterior distribution with a Markov chain Monte Carlo method
- ▶ Analyzing data with the R programming language in the Jupyter Notebook

Introduction

In the previous chapters, we reviewed technical aspects of high-performance interactive computing in Python. We now begin the second part of this book by illustrating a variety of scientific questions that can be tackled with Python.

In this chapter, we introduce statistical methods for data analysis. In addition to covering statistical packages such as pandas, statsmodels, and PyMC3, we will explain the basics of the underlying mathematical principles. Therefore, this chapter will be most profitable if you have basic experience with probability theory and calculus.

The next chapter, *Chapter 8, Machine Learning*, is closely related; the underlying mathematics is very similar, but the goals are slightly different. In this chapter, we show how to gain insight into real-world data and how to make informed decisions in the presence of uncertainty. In the next chapter, the goal is to learn from data—that is, to generalize and to predict outcomes from partial observations.

In this introduction, we will give a broad, high-level overview of the methods we will see in this chapter.

What is statistical data analysis?

The goal of **statistical data analysis** is to understand a complex, real-world phenomenon from partial and uncertain observations. The uncertainty in the data results in uncertainty in the knowledge we get about the phenomenon. A major goal of the theory is to *quantify* this uncertainty.

It is important to make the distinction between the mathematical theory underlying statistical data analysis, and the decisions made after conducting an analysis. The former is perfectly rigorous; perhaps surprisingly, mathematicians were able to build an exact mathematical framework to deal with uncertainty. Nevertheless, there is a subjective part in the way statistical analysis yields actual human decisions. Understanding the risk and the uncertainty behind statistical results is critical in the decision-making process.

In this chapter, we will see the basic notions, principles, and theories behind statistical data analysis, covering in particular how to make decisions with a quantified risk. Of course, we will always show how to implement these methods with Python.

A bit of vocabulary

There are many terms that need introduction before we get started with the recipes. These notions allow us to classify statistical techniques within multiple dimensions.

Exploration, inference, decision, prediction

Exploratory methods allow us to get a preliminary look at a dataset through basic statistical aggregates and interactive visualization. We covered these basic methods in the first chapter of this book and in the prequel book *Learning IPython for Interactive Computing and Data Visualization, Second Edition, Packt Publishing*. The first recipe of this chapter, *Exploring a dataset with pandas and Matplotlib*, shows another example.

Statistical inference consists of getting information about an unknown process through partial and uncertain observations. In particular, **estimation** entails obtaining approximate quantities for the mathematical variables describing this process. Three recipes in this chapter deal with statistical inference:

- ▶ The *Fitting a probability distribution to data with the maximum likelihood method* recipe
- ▶ The *Estimating a probability distribution nonparametrically with a kernel density estimation* recipe
- ▶ The *Fitting a Bayesian model by sampling from a posterior distribution with a Markov chain Monte Carlo* method recipe

Decision theory allows us to make decisions about an unknown process from random observations, with a controlled risk. The following two recipes show how to make statistical decisions:

- ▶ The *Getting started with statistical hypothesis testing – a simple z-test* recipe
- ▶ The *Estimating the correlation between two variables with a contingency table and a chi-squared test* recipe

Prediction consists of learning from data—that is, predicting the outcomes of a random process based on a limited number of observations. This is the topic of the next chapter, *Chapter 8, Machine Learning*.

Univariate and multivariate methods

In most cases, you can consider two dimensions in your data:

- ▶ **Observations** (or **samples**, for machine learning people)
- ▶ **Variables** (or **features**)

Typically, observations are independent realizations of the same random process. Each observation is made of one or several variables. Most of the time, variables are either numbers, or elements belonging to a finite set (that is, taking a finite number of values). The first step in an analysis is to understand what your observations and variables are.

Your problem is **univariate** if you have one variable. It is **bivariate** if you have two variables and **multivariate** if you have at least two variables. Univariate methods are typically simpler. That being said, univariate methods may be used on multivariate data, using one dimension at a time. Although interactions between variables cannot be explored in that case, it is often an interesting first approach.

Frequentist and Bayesian methods

There are at least two different ways of considering uncertainty, resulting in two different classes of methods for inference, decision, and other statistical questions. These are called **frequentist and Bayesian methods**. Some people prefer frequentist methods, while others prefer Bayesian methods.

Frequentists interpret a probability as a statistical average across many independent realizations (law of large numbers). Bayesians interpret it as a degree of belief (no need for many realizations). The Bayesian interpretation is very useful when only a single trial is considered. In addition, Bayesian theory takes into account our **prior knowledge** about a random process. This prior probability distribution is updated into a posterior distribution as we get more and more data.

Both frequentist and Bayesian methods have their advantages and disadvantages. For instance, one could say that frequentist methods might be easier to apply than Bayesian methods, but more difficult to interpret. For classic misuses of frequentist methods, see <http://www.refsmmat.com/statistics/>.

In any case, if you are a beginner in statistical data analysis, you probably want to learn the basics of both approaches before choosing sides. This chapter introduces you to both types of methods.

The following recipes are exclusively Bayesian:

- ▶ The *Getting started with Bayesian methods* recipe
- ▶ The *Fitting a Bayesian model by sampling from a posterior distribution with a Markov chain Monte Carlo method* recipe

Jake VanderPlas has written several blog posts about frequentism and Bayesianism, with examples in Python. The first post of the series is available at <http://jakevdp.github.io/blog/2014/03/11/frequentism-and-bayesianism-a-practical-intro/>.

Parametric and nonparametric inference methods

In many cases, you base your analysis on a **probabilistic model**. This model describes how your data is generated. A probabilistic model has no reality; it is only a mathematical object that guides you in your analysis. A good model can be helpful, whereas a bad model may misguide you.

With a **parametric method**, you assume that your model belongs to a known family of probability distributions. The model has one or multiple numerical parameters that you can estimate.

With a **nonparametric model**, you do not make such an assumption in your model. This gives you more flexibility. However, these methods are typically more complicated to implement and to interpret.

The following recipes are parametric and nonparametric, respectively:

- ▶ The *Fitting a probability distribution to data with the maximum likelihood method* recipe
- ▶ The *Estimating a probability distribution nonparametrically with a kernel density estimation* recipe

This chapter only gives you an idea of the wide range of possibilities that Python offers for statistical data analysis. You can find many books and online courses that cover statistical methods in much greater detail, such as:

- ▶ Statistics resources on Awesome Math, available at <https://github.com/rossant/awesome-math#statistics>
- ▶ Statistics on WikiBooks at <http://en.wikibooks.org/wiki/Statistics>
- ▶ Free statistical textbooks available at <http://stats.stackexchange.com/questions/170/free-statistical-textbooks>

Exploring a dataset with pandas and Matplotlib

In this first recipe, we will show how to conduct a preliminary analysis of a dataset with pandas. This is typically the first step after getting access to the data. pandas lets us load the data very easily, explore the variables, and make basic plots with Matplotlib.

We will take a look at a dataset containing all ATP matches played by the Swiss tennis player Roger Federer until 2012.

How to do it...

1. We import NumPy, pandas, and Matplotlib:

```
>>> from datetime import datetime
      import numpy as np
      import pandas as pd
      import matplotlib.pyplot as plt
      %matplotlib inline
```

2. The dataset is a CSV file—that is, a text file with comma-separated values. pandas lets us load this file with a single function:

```
>>> player = 'Roger Federer'
    df = pd.read_csv('https://github.com/ipython-books/'
                     'cookbook-2nd-data/blob/master/'
                     'federer.csv?raw=true',
                     parse_dates=['start date'],
                     dayfirst=True)
```

We can have a first look at this dataset by just displaying it in the Jupyter Notebook:

```
>>> df.head(3)
```

| year | tournament | start date | type | surface | ... | player2 total se... | player2 total ret... | player2 total ret... | player2 total po... | player2 total po... |
|------|-------------------------|------------|------|--------------|-----|---------------------|----------------------|----------------------|---------------------|---------------------|
| 0 | 1998 Basel, Switzerland | 1998-10-05 | WS | Indoor: Hard | ... | 50.0 | 26.0 | 53.0 | 62.0 | 103.0 |
| 1 | 1998 Toulouse, France | 1998-09-28 | WS | Indoor: Hard | ... | 65.0 | 8.0 | 41.0 | 41.0 | 106.0 |
| 2 | 1998 Toulouse, France | 1998-09-28 | WS | Indoor: Hard | ... | 75.0 | 23.0 | 73.0 | 69.0 | 148.0 |

3 rows × 70 columns

3. There are many columns. Each row corresponds to a match played by Roger Federer. Let's add a Boolean variable indicating whether he has won the match or not.

The `tail()` method displays the last rows of the column:

```
>>> df['win'] = df['winner'] == player
    df['win'].tail()
1174    False
1175     True
1176     True
1177     True
1178    False
Name: win, dtype: bool
```

4. `df['win']` is a Series object. It is very similar to a NumPy array, except that each value has an index (here, the match index). This object has a few standard statistical functions. For example, let's look at the proportion of matches won:

```
>>> won = 100 * df['win'].mean()
    print(f"{player} has won {won:.0f}% of his matches.")
Roger Federer has won 82% of his matches.
```

5. Now, we are going to look at the evolution of some variables across time. The `df['start date']` field contains the start date of the tournament:

```
>>> date = df['start date']
```

6. We are now looking at the proportion of double faults in each match (taking into account that there are logically more double faults in longer matches!). This number is an indicator of the player's state of mind, his level of self-confidence, his willingness to take risks while serving, and other parameters.

```
>>> df['dblfaults'] = (df['player1 double faults'] /  
                      df['player1 total points total'])
```

7. We can use the `head()` and `tail()` methods to take a look at the beginning and the end of the column, and `describe()` to get summary statistics. In particular, let's note that some rows have `NaN` values (that is, the number of double faults is not available for all matches).

```
>>> df['dblfaults'].tail()  
1174    0.018116  
1175    0.000000  
1176    0.000000  
1177    0.011561  
1178      NaN  
Name: dblfaults, dtype: float64  
>>> df['dblfaults'].describe()  
count    1027.000000  
mean      0.012129  
std       0.010797  
min       0.000000  
25%       0.004444  
50%       0.010000  
75%       0.018108  
max       0.060606  
Name: dblfaults, dtype: float64
```

8. A very powerful feature in pandas is `groupby()`. This function allows us to group together rows that have the same value in a particular column. Then, we can aggregate this group by value to compute statistics in each group. For instance, here is how we can get the proportion of wins as a function of the tournament's surface:

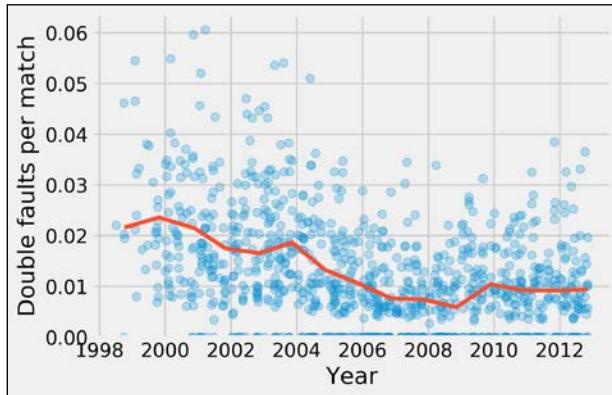
```
>>> df.groupby('surface')['win'].mean()  
Surface  
Indoor: Carpet    0.736842  
Indoor: Clay      0.833333  
Indoor: Hard      0.836283  
Outdoor: Clay     0.779116  
Outdoor: Grass    0.871429  
Outdoor: Hard     0.842324  
Name: win, dtype: float64
```

9. Now, we are going to display the proportion of double faults as a function of the tournament date, as well as the yearly average. To do this, we also use `groupby()`:

```
>>> gb = df.groupby('year')
```

10. `gb` is a `GroupBy` instance. It is similar to a `DataFrame` object, but there are multiple rows per group (all matches played in each year). We can aggregate these rows using the `mean()` operation. We use the `matplotlib plot_date()` function because the x axis contains dates:

```
>>> fig, ax = plt.subplots(1, 1)
    ax.plot_date(date.astype(datetime), df['dblfaults'],
                  alpha=.25, lw=0)
    ax.plot_date(gb['start date'].max().astype(datetime),
                  gb['dblfaults'].mean(), '--', lw=3)
    ax.set_xlabel('Year')
    ax.set_ylabel('Double faults per match')
    ax.set_ylim(0)
```



There's more...

pandas is an excellent tool for data wrangling and exploratory analysis. pandas accepts all sorts of formats (text-based, and binary files) and it lets us manipulate tables in many ways. In particular, the `groupby()` function is particularly powerful.

What we covered here is only the first step in a data-analysis process. We need more advanced statistical methods to obtain reliable information about the underlying phenomena, make decisions and predictions, and so on. This is the topic of the following recipes.

In addition, more complex datasets demand more sophisticated analysis methods. For example, digital recordings, images, sounds, and videos require specific signal processing treatments before we can apply statistical techniques. These questions will be covered in subsequent chapters.

Here are a few references about pandas:

- ▶ pandas website at <https://pandas.pydata.org/>
- ▶ pandas tutorial at <http://pandas.pydata.org/pandas-docs/stable/10min.html>
- ▶ *Python for Data Analysis, 2nd Edition*, Wes McKinney, O'Reilly Media, at <http://shop.oreilly.com/product/0636920050896.do>

Getting started with statistical hypothesis testing — a simple z-test

Statistical hypothesis testing allows us to make decisions in the presence of incomplete data. By definition, these decisions are uncertain. Statisticians have developed rigorous methods to evaluate this risk. Nevertheless, some subjectivity is always involved in the decision-making process. The theory is just a tool that helps us make decisions in an uncertain world.

Here, we introduce the most basic ideas behind statistical hypothesis testing. We will follow a particularly simple example: coin tossing. More precisely, we will show how to perform a z-test, and we will briefly explain the mathematical ideas underlying it. This kind of method (also called the frequentist method), although widely used in science, is not without flaws and interpretation difficulties. We will show another approach based on Bayesian theory later in this chapter. It is very helpful to understand both approaches.

Getting ready

You need to have a basic knowledge of probability theory for this recipe (random variables, distributions, expectancy, variance, central limit theorem, and so on).

How to do it...

Many frequentist methods for hypothesis testing roughly involve the following steps:

1. Writing down the hypotheses, notably the **null hypothesis**, which is the opposite of the hypothesis we want to prove (with a certain degree of confidence).
2. Computing a **test statistic**, a mathematical formula depending on the test type, the model, the hypotheses, and the data.
3. Using the computed value to reject the hypothesis with a given level of uncertainty, or fail to conclude (and, consequently, accept the hypothesis until future studies reject it).

For example, to test the efficacy of a new drug, doctors may consider, as a null hypothesis, that the drug has no statistically significant effect on a group of patients compared to a control group of patients who do not take the drug. If studies reject the null hypothesis, it is an argument in favor of the efficacy of the drug (but it is not a definite proof).

Here, we flip a coin n times and we observe h heads. We want to know whether the coin is fair (null hypothesis). This example is particularly simple yet quite useful for pedagogical purposes. Besides, it is the basis of many more complex methods.

We denote the Bernoulli distribution by $B(q)$ with the unknown parameter q . You can refer to https://en.wikipedia.org/wiki/Bernoulli_distribution for more information.

A Bernoulli variable is:

- ▶ 0 (tail) with probability $1 - q$
- ▶ 1 (head) with probability q

Here are the steps required to conduct a simple statistical z-test:

1. Let's suppose that after $n = 100$ flips, we get $h = 61$ heads. We choose a significance level of 0.05: is the coin fair or not? Our null hypothesis is: the coin is fair ($q = 1/2$). We set these variables:

```
>>> import numpy as np
      import scipy.stats as st
      import scipy.special as sp
>>> n = 100 # number of coin flips
      h = 61 # number of heads
      q = .5 # null-hypothesis of fair coin
```

2. Let's compute the **z-score**, which is defined by the following formula ($xbar$ is the estimated average of the distribution). We will explain this formula in the next section, *How it works....*

```
>>> xbar = float(h) / n
      z = (xbar - q) * np.sqrt(n / (q * (1 - q)))
      # We don't want to display more than 4 decimals.
      z
      2.2000
```

3. Now, from the z-score, we can compute the p-value as follows:

```
>>> pval = 2 * (1 - st.norm.cdf(z))
      pval
      0.0278
```

4. This p-value is less than 0.05, so we reject the null hypothesis and conclude that *the coin is probably not fair.*

How it works...

The coin tossing experiment is modeled as a sequence of n independent random variables $x_i \in \{0, 1\}$ following the Bernoulli distribution $B(q)$. Each x_i represents one coin flip. After our experiment, we get actual values (samples) for these variables. A different notation is sometimes used to distinguish between the random variables (probabilistic objects) and the actual values (samples).

The following formula gives the **sample mean** (proportion of heads here):

$$\bar{x} = \frac{1}{n} \sum_i x_i$$

Knowing the expectancy $\mu = q$ and variance $\sigma^2 = q(1 - q)$ of the distribution $B(q)$, we compute:

$$E[\bar{x}] = \mu = q$$
$$\text{var}(\bar{x}) = \frac{\sigma^2}{n} = \frac{q(1 - q)}{n}$$

The z-test is the normalized version of \bar{x} (we remove its mean, and divide by the standard deviation, thus we get a variable with mean 0 and standard deviation 1):

$$z = \frac{\bar{x} - E[\bar{x}]}{\text{std}(\bar{x})} = (\bar{x} - q) \sqrt{\frac{n}{q(1 - q)}}$$

Under the null hypothesis, what is the probability of obtaining a z-test higher (in absolute value) than some quantity z_0 ? This probability is called the (two-sided) **p-value**. According to the central limit theorem, the z-test approximately follows a standard Gaussian distribution $N(0, 1)$ for large n , so we get:

$$p = P[|z| > z_0] = 2P[z > z_0] \simeq 2(1 - \Phi(z_0))$$

The following diagram illustrates the z-score and the p-value:

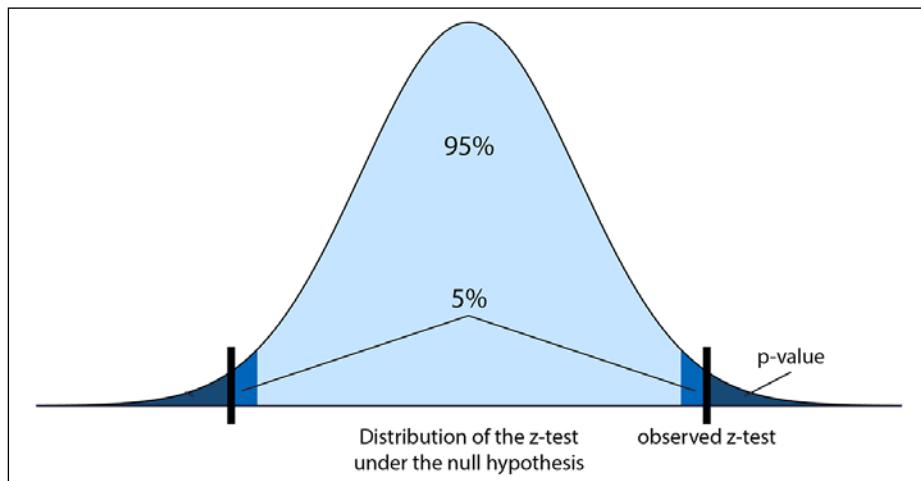


Illustration of the z-score and the p-value.

In this formula, Φ is the cumulative distribution function of a standard normal distribution. In SciPy, we can get it with `scipy.stats.norm.cdf`. So, given the z-test computed from the data, we compute the p-value: the probability of observing a z-test more extreme than the observed test, under the null hypothesis.

If the p-value is less than five percent (a frequently-chosen significance level, for arbitrary and historical reasons), we conclude that either:

- ▶ The null hypothesis is false, thus we conclude that the coin is unfair.
- ▶ The null hypothesis is true, and it's just bad luck if we obtained these values.
We cannot make a conclusion.

We cannot disambiguate between these two options in this framework, but typically the first option is chosen. We hit the limits of frequentist statistics, although there are ways to mitigate this problem (for example, by conducting several independent studies and looking at all of their conclusions).

There's more...

There are many statistical tests that follow this pattern. Reviewing all those tests is largely beyond the scope of this book, but you can take a look at the reference at https://en.wikipedia.org/wiki/Statistical_hypothesis_testing.

As a p-value is not easy to interpret, it can lead to wrong conclusions, even in peer-reviewed scientific publications. For an in-depth treatment of the subject, see <http://www.refsammat.com/statistics/>.

See also

- ▶ The Getting started with Bayesian methods recipe

Getting started with Bayesian methods

In the last recipe, we used a frequentist method to test a hypothesis on incomplete data. Here, we will see an alternative approach based on Bayesian theory. The main idea is to consider that *unknown parameters are random variables*, just like the variables describing the experiment. Prior knowledge about the parameters is integrated into the model. This knowledge is updated as more and more data is observed.

Frequentists and Bayesians interpret probabilities differently. Frequentists interpret a probability as a limit of frequencies when the number of samples tends to infinity. Bayesians interpret it as a belief; this belief is updated as more and more data is observed.

Here, we revisit the previous coin flipping example with a Bayesian approach. This example is sufficiently simple to permit an analytical treatment. In general, as we will see later in this chapter, analytical results cannot be obtained and numerical methods become essential.

Getting ready

This is a math-heavy recipe. Knowledge of basic probability theory (random variables, distributions, Bayes formula) and calculus (derivatives, integrals) is recommended. We use the same notations as in the previous recipe.

How to do it...

Let q be the probability of obtaining a head. Whereas q was just a fixed number in the previous recipe, we consider here that it is a **random variable**. Initially, this variable follows a distribution called the **prior probability distribution**. It represents our knowledge about q before we start flipping the coin. We will update this distribution after each trial (posterior distribution).

1. First, we assume that q is a uniform random variable on the interval $[0, 1]$. That's our prior distribution: for all q , $P(q) = 1$.
2. Then, we flip our coin n times. We note x_i the outcome of the i th flip (0 for tail, 1 for head).

3. What is the probability distribution of q knowing the observations x_i ? **Bayes' theorem** allows us to compute the *posterior distribution* analytically (see the next section for the mathematical details):

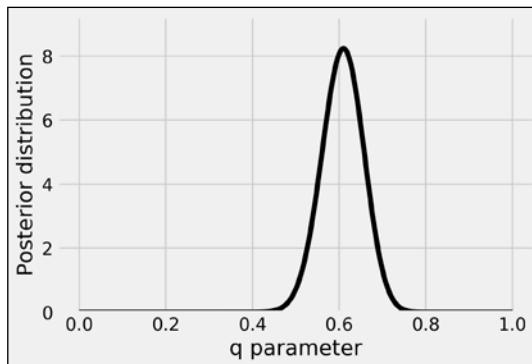
$$P(q | \{x_i\}) = \frac{P(\{x_i\} | q)P(q)}{\int_0^1 P(\{x_i\} | q)P(q)dq} = (n+1) \binom{n}{h} q^h (1-q)^{n-h}$$

4. We define the posterior distribution according to the preceding mathematical formula. We remark that this expression is $(n+1)$ times the **Probability Mass Function (PMF)** of the binomial distribution, which is directly available in `scipy.stats`. (For more information on binomial distribution, refer to https://en.wikipedia.org/wiki/Binomial_distribution.)

```
>>> import numpy as np
     import scipy.stats as st
     import matplotlib.pyplot as plt
     %matplotlib inline
>>> def posterior(n, h, q):
     return (n + 1) * st.binom(n, q).pmf(h)
```

5. Let's plot this distribution for an observation of $h = 61$ heads and $n = 100$ total flips:

```
>>> n = 100
     h = 61
     q = np.linspace(0., 1., 1000)
     d = posterior(n, h, q)
>>> fig, ax = plt.subplots(1, 1)
     ax.plot(q, d, '-k')
     ax.set_xlabel('q parameter')
     ax.set_ylabel('Posterior distribution')
     ax.set_xlim(0, d.max() + 1)
```



This curve represents our belief about the parameter q after we have observed 61 heads.

How it works...

In this section, we explain Bayes' theorem, and we give the mathematical details underlying this example.

Bayes' theorem

There is a very general idea in data science that consists of explaining data with a mathematical model. This is formalized with a one-way process, $model \rightarrow data$.

Once this process is formalized, the task of the data scientist is to exploit the data to recover information about the model. In other words, we want to *invert* the original process and get $data \rightarrow model$.

In a probabilistic setting, the direct process is represented as a **conditional probability distribution** $P(data | model)$. This is the probability of observing the data when the model is entirely specified.

Similarly, the inverse process is $P(model | data)$. It gives us information about the model (what we're looking for), knowing the observations (what we have).

Bayes' theorem is at the core of a general framework for inverting a probabilistic process of $model \rightarrow data$. It can be stated as follows:

$$P(model | data) = \frac{P(data | model) P(model)}{P(data)}$$

This equation gives us information about our model, knowing the observed data. Bayes' equation is widely used in signal processing, statistics, machine learning, inverse problems, and in many other scientific applications.

In Bayes' equation, $P(model)$ reflects our prior knowledge about the model. Also, $P(data)$ is the distribution of the data. It is generally expressed as an integral of $P(data | model) P(model)$.

In conclusion, Bayes' equation gives us a general roadmap for data inference:

1. Specify a mathematical model for the direct process $model \rightarrow data$ (the $P(data | model)$ term).
2. Specify a prior probability distribution for the model ($P(model)$ term).
3. Perform analytical or numerical calculations to solve this equation.

Computation of the posterior distribution

In this recipe's example, we found the posterior distribution with the following equation (deriving directly from Bayes' theorem):

$$P(q | \{x_i\}) = \frac{P(\{x_i\} | q) P(q)}{\int_0^1 P(\{x_i\} | q) P(q) dq}$$

Knowing that the x_i are independent, we get (h being the number of heads):

$$P(\{x_i\} | q) = \prod_{i=1}^n P(x_i | q) = q^h (1-q)^{n-h}$$

In addition, we can compute analytically the following integral (using an integration by parts and an induction):

$$\int_0^1 P(\{x_i\} | q) P(q) dq = \int_0^1 q^h (1-q)^{n-h} dq = \frac{1}{(n+1)\binom{n}{h}}$$

Finally, we get:

$$P(q | \{x_i\}) = \frac{P(\{x_i\} | q) P(q)}{\int_0^1 P(\{x_i\} | q) P(q) dq} = (n+1)\binom{n}{h} q^h (1-q)^{n-h}$$

Maximum a posteriori estimation

We can get a point estimate from the posterior distribution. For example, the **Maximum a posteriori (MAP)** estimation consists of considering the maximum of the posterior distribution as an estimate for q . We can find this maximum analytically or numerically. For more information on MAP, refer to https://en.wikipedia.org/wiki/Maximum_a_posteriori_estimation.

Here, we can get this estimate analytically by deriving the posterior distribution with respect to q . We get (assuming $1 \leq h \leq n-1$):

$$\frac{dP(q | \{x_i\})}{dq} = (n+1) \frac{n!}{(n-h)! h!} (hq^{h-1}(1-q)^{n-h} - (n-h)q^h(1-q)^{n-h-1})$$

This expression is equal to zero when $q = h/n$. This is the MAP estimate of the parameter q . This value happens to be the proportion of heads obtained in the experiment.

There's more...

In this recipe, we showed a few basic notions in Bayesian theory. We illustrated them with a simple example. The fact that we were able to derive the posterior distribution analytically is not very common in real-world applications. This example is nevertheless informative because it explains the core mathematical ideas behind the complex numerical methods we will see later.

Credible interval

The posterior distribution indicates the plausible values for q given the observations. We could use it to derive a credible interval, likely to contain the actual value. **Credible intervals** are the Bayesian analog to confidence intervals in frequentist statistics. For more information on credible intervals, refer to https://en.wikipedia.org/wiki/Credible_interval.

Conjugate distributions

In this recipe, the prior and posterior distributions are **conjugate**, meaning that they belong to the same family (the beta distribution). For this reason, we were able to compute the posterior distribution analytically. You will find more details about conjugate distributions at https://en.wikipedia.org/wiki/Conjugate_prior.

Non-informative (objective) prior distributions

We chose a uniform distribution as prior distribution for the unknown parameter q . It is a simple choice and it leads to tractable computations. It reflects the intuitive fact that we do not favor any particular value a priori. However, there are rigorous ways of choosing completely uninformative priors (see https://en.wikipedia.org/wiki/Prior_probability#Uninformative_priors). An example is the Jeffreys prior, based on the idea that the prior distribution should not depend on the parameterization of the parameters. For more information on Jeffreys prior, refer to https://en.wikipedia.org/wiki/Jeffreys_prior. In our example, the Jeffreys prior is:

$$P(q) = \frac{1}{\sqrt{q(1-q)}}$$

See also

- ▶ The *Fitting a Bayesian model by sampling from a posterior distribution with a Markov chain Monte Carlo method* recipe

Estimating the correlation between two variables with a contingency table and a chi-squared test

Whereas univariate methods deal with single-variable observations, multivariate methods consider observations with several features. Multivariate datasets allow the study of *relations* between variables, more particularly their correlation, or lack thereof (that is, independence).

In this recipe, we will take a look at the same tennis dataset as in the first recipe of this chapter. Following a frequentist approach, we will estimate the correlation between the number of aces and the proportion of points won by a tennis player.

How to do it...

- Let's import NumPy, pandas, SciPy.stats, and Matplotlib:

```
>>> import numpy as np
     import pandas as pd
     import scipy.stats as st
     import matplotlib.pyplot as plt
     %matplotlib inline
```

- We download and load the dataset:

```
>>> player = 'Roger Federer'
     df = pd.read_csv('https://github.com/ipython-books/'
                      'cookbook-2nd-data/blob/master/'
                      'federer.csv?raw=true',
                      parse_dates=['start date'],
                      dayfirst=True)
```

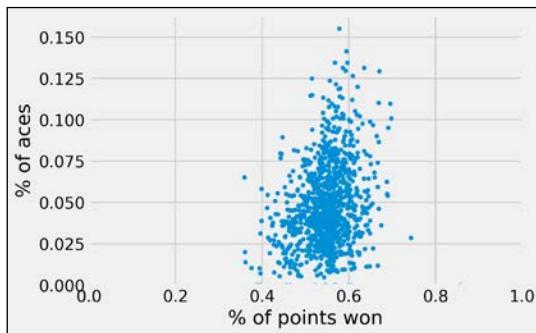
- Each row corresponds to a match, and the 70 columns contain many player characteristics during that match:

```
>>> print(f"Number of columns: {len(df.columns)}")
     df[df.columns[:4]].tail()
Number of columns: 70
```

| | year | tournament | start date | type |
|------|------|--------------------|------------|------|
| 1174 | 2012 | Australian Open... | 2012-01-16 | GS |
| 1175 | 2012 | Doha, Qatar | 2012-01-02 | 250 |
| 1176 | 2012 | Doha, Qatar | 2012-01-02 | 250 |
| 1177 | 2012 | Doha, Qatar | 2012-01-02 | 250 |
| 1178 | 2012 | Doha, Qatar | 2012-01-02 | 250 |

4. Here, we only look at the proportion of points won, and the (relative) number of aces:

```
>>> npoints = df['player1 total points total']
    points = df['player1 total points won'] / npoints
    aces = df['player1 aces'] / npoints
>>> fig, ax = plt.subplots(1, 1)
    ax.plot(points, aces, '.')
    ax.set_xlabel('% of points won')
    ax.set_ylabel('% of aces')
    ax.set_xlim(0., 1.)
    ax.set_ylim(0.)
```



If the two variables were independent, we would not see any trend in the cloud of points. On this plot, it is a bit hard to tell. Let's use pandas to compute a coefficient correlation.

5. For simplicity, we create a new DataFrame object with only these fields. We also remove the rows where one field is missing (using `dropna()`):

```
>>> df_bis = pd.DataFrame({'points': points,
                           'aces': aces}).dropna()
df_bis.tail()
```

| | aces | points |
|------|----------|----------|
| 1173 | 0.024390 | 0.585366 |
| 1174 | 0.039855 | 0.471014 |
| 1175 | 0.046512 | 0.639535 |
| 1176 | 0.020202 | 0.606061 |
| 1177 | 0.069364 | 0.531792 |

6. Let's compute the Pearson's correlation coefficient between the relative number of aces in the match, and the number of points won:

```
>>> df_bis.corr()
```

| manyaces | False | True |
|----------|-------|------|
| result | | |
| False | 300 | 214 |
| True | 214 | 299 |

A correlation of ~ 0.26 seems to indicate a positive correlation between our two variables. In other words, the more aces in a match, the more points the player wins (which is not very surprising!).

7. Now, to determine if there is a statistically significant correlation between the variables, we use a **chi-squared test** of the independence of variables in a **contingency table**.
8. First, we binarize our variables. Here, the value corresponding to the number of aces is True if the player is serving more aces than usual in a match, and False otherwise:

```
>>> df_bis['result'] = (df_bis['points'] >
                         df_bis['points'].median())
    df_bis['manyaces'] = (df_bis['aces'] >
                           df_bis['aces'].median())
```

9. Then, we create a contingency table, with the frequencies of all four possibilities (True and True, True and False, and so on):

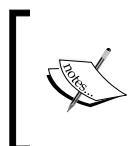
```
>>> pd.crosstab(df_bis['result'], df_bis['manyaces'])
```

| manyaces | False | True |
|----------|-------|------|
| result | | |
| False | 300 | 214 |
| True | 214 | 299 |

10. Finally, we compute the chi-squared test statistic and the associated p-value. The null hypothesis is the independence between the variables. SciPy implements this test in `scipy.stats.chi2_contingency()`, which returns several objects. We're interested in the second result, which is the p-value:

```
>>> st.chi2_contingency(_)
(2.780e+01, 1.338e-07, 1,
 array([[ 257.250,   256.749],
       [ 256.749,   256.250]]))
```

The p-value is much lower than 0.05, so we reject the null hypothesis and conclude that there is a statistically significant correlation between the proportion of aces and the proportion of points won in a match in this dataset.



Correlation does not imply causation. Here, it is likely that external factors influence both variables. See https://en.wikipedia.org/wiki/Correlation_does_not_imply_causation for more details.

How it works...

We give here a few details about the statistical concepts used in this recipe.

Pearson's correlation coefficient

Pearson's correlation coefficient measures the linear correlation between two random variables, X and Y . It is a normalized version of the covariance:

$$\rho = \frac{\text{cov}(X, Y)}{\sqrt{\text{var}(X)\text{var}(Y)}} = \frac{E\left((X - E(X))(Y - E(Y))\right)}{\sqrt{\text{var}(X)\text{var}(Y)}}$$

It can be estimated by substituting, in this formula, the expectancy with the sample mean, and the variance with the sample variance. More details about its inference can be found at https://en.wikipedia.org/wiki/Pearson_product-moment_correlation_coefficient.

Contingency table and chi-squared test

The contingency table contains the frequencies O_{ij} of all combinations of outcomes, when there are multiple random variables that can take a finite number of values. Under the null hypothesis of independence, we can compute the expected frequencies E_{ij} , based on the marginal sums (sums in each row). The chi-squared statistic, by definition, is:

$$\chi^2 = \sum_{i,j} \frac{(O_{ij} - E_{ij})^2}{E_{ij}}$$

When there are sufficient observations, this variable approximately follows a chi-squared distribution (the distribution of the sum of normal variables squared). Once we get the p-value, as explained in the *Getting started with statistical hypothesis testing – a simple z-test recipe*, we can reject or accept the null hypothesis of independence. Then, we can conclude (or not) that there exists a significant correlation between the variables.

There's more...

There are many other sorts of chi-squared tests—that is, tests where the test statistic follows a chi-squared distribution. These tests are widely used for testing the goodness-of-fit of a distribution, or testing the independence of variables. More information can be found in the following pages:

- ▶ Chi-square test in SciPy documentation available at http://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.chi2_contingency.html
- ▶ Contingency table introduced at https://en.wikipedia.org/wiki/Contingency_table
- ▶ Chi-squared test introduced at https://en.wikipedia.org/wiki/Pearson%27s_chi-squared_test

See also

- ▶ The *Getting started with statistical hypothesis testing – a simple z-test recipe*

Fitting a probability distribution to data with the maximum likelihood method

A good way to explain a dataset is to apply a probabilistic model to it. Finding an adequate model can be a job on its own. Once a model is chosen, it is necessary to compare it to the data. This is what statistical estimation is about. In this recipe, we apply the maximum likelihood method on a dataset of survival times after heart transplant (1967-1974 study).

Getting ready

As usual in this chapter, a background in probability theory and real analysis is recommended. In addition, you need the `statsmodels` package to retrieve the test dataset. It should be included in Anaconda, but you can always install it with the `conda install statsmodels` command.

How to do it...

1. statsmodels is a Python package for conducting statistical data analyses. It also contains real-world datasets that we can use when experimenting with new methods. Here, we load the heart dataset:

```
>>> import numpy as np
      import scipy.stats as st
      import statsmodels.datasets
      import matplotlib.pyplot as plt
      %matplotlib inline
>>> data = statsmodels.datasets.heart.load_pandas().data
```

2. Let's take a look at this DataFrame.

```
>>> data.tail()
```

| | survival | censors | age |
|----|----------|---------|------|
| 64 | 14.0 | 1.0 | 40.3 |
| 65 | 167.0 | 0.0 | 26.7 |
| 66 | 110.0 | 0.0 | 23.7 |
| 67 | 13.0 | 0.0 | 28.9 |
| 68 | 1.0 | 0.0 | 35.2 |

This dataset contains censored and uncensored data: a censor of 0 means that the patient was alive at the end of the study, and thus we don't know the exact survival time. We only know that the patient survived at least the indicated number of days. For simplicity here, we only keep uncensored data (we thereby introduce a bias toward patients that did not survive very long after their transplant):

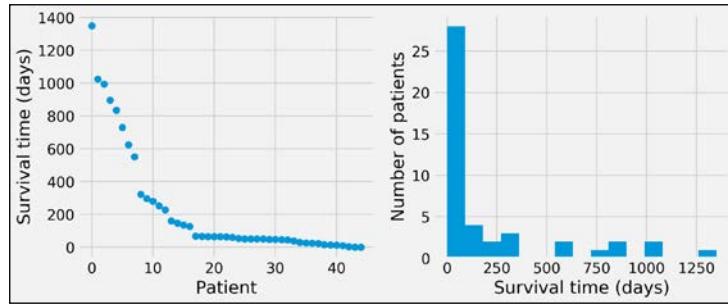
```
>>> data = data[data.censors == 1]
      survival = data.survival
```

3. Let's take a look at the data graphically, by plotting the raw survival data and the histogram:

```
>>> fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))

      ax1.plot(sorted(survival) [::-1], 'o')
      ax1.set_xlabel('Patient')
      ax1.set_ylabel('Survival time (days)')
```

```
ax2.hist(survival, bins=15)
ax2.set_xlabel('Survival time (days)')
ax2.set_ylabel('Number of patients')
```



4. We observe that the histogram is decreasing very rapidly. Fortunately, the survival rates today are much higher (~70 percent after 5 years). Let's try to fit an exponential distribution (more information on the exponential distribution is available at https://en.wikipedia.org/wiki/Exponential_distribution) to the data. According to this model, S (number of days of survival) is an exponential random variable with the parameter λ , and the observations s_i are sampled from this distribution. Let the sample mean be:

$$\bar{s} = \frac{1}{n} \sum s_i$$

The likelihood function of an exponential distribution is as follows, by definition (see proof in the next section):

$$\mathcal{L}(\lambda, \{s_i\}) = P(\{s_i\} | \lambda) = \lambda^n \exp(-\lambda n \bar{s})$$

The maximum likelihood estimate for the rate parameter is, by definition, the value λ that maximizes the likelihood function. In other words, it is the parameter that maximizes the probability of observing the data, assuming that the observations are sampled from an exponential distribution.

Here, it can be shown that the likelihood function has a maximum value when $\lambda = 1/\bar{s}$, which is the maximum likelihood estimate for the rate parameter. Let's compute this parameter numerically:

```
>>> smean = survival.mean()
      rate = 1. / smean
```

5. To compare the fitted exponential distribution to the data, we first need to generate linearly spaced values for the x axis (days):

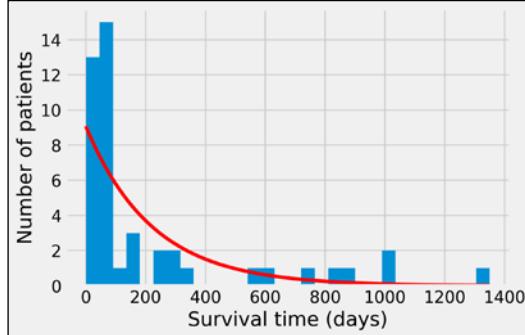
```
>>> smax = survival.max()
      days = np.linspace(0., smax, 1000)
      # bin size: interval between two
      # consecutive values in `days`
      dt = smax / 999.
```

We can obtain the probability density function of the exponential distribution with SciPy. The parameter is the scale, the inverse of the estimated rate.

```
>>> dist_exp = st.expon.pdf(days, scale=1. / rate)
```

6. Now, let's plot the histogram and the obtained distribution. We need to rescale the theoretical distribution to the histogram (depending on the bin size and the total number of data points):

```
>>> nbins = 30
      fig, ax = plt.subplots(1, 1, figsize=(6, 4))
      ax.hist(survival, nbins)
      ax.plot(days, dist_exp * len(survival) * smax / nbins,
              '-r', lw=3)
      ax.set_xlabel("Survival time (days)")
      ax.set_ylabel("Number of patients")
```



The fit is far from perfect. We were able to find an analytical formula for the maximum likelihood estimate here. In more complex situations, that is not always possible. Thus we may need to resort to numerical methods. SciPy actually integrates numerical maximum likelihood routines for a large number of distributions. Here, we use this other method to estimate the parameter of the exponential distribution.

```
>>> dist = st.expon
      args = dist.fit(survival)
      args
(1.000, 222.289)
```

7. We can use these parameters to perform a **Kolmogorov-Smirnov test**, which assesses the goodness of fit of the distribution with respect to the data. This test is based on a distance between the **empirical distribution function** of the data and the **Cumulative Distribution Function (CDF)** of the reference distribution.

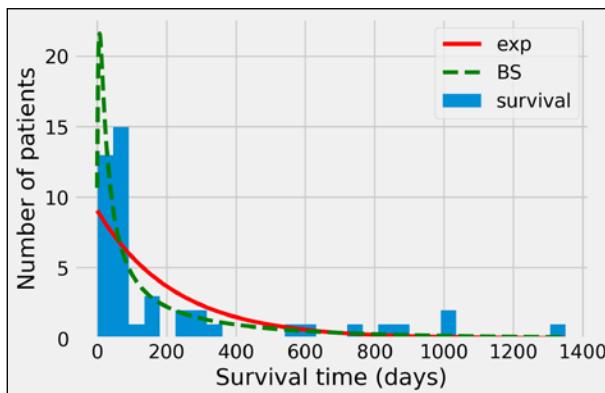
```
>>> st.kstest(survival, dist.cdf, args)
KstestResult(statistic=0.362, pvalue=8.647e-06)
```

Here, the p-value is very low: the null hypothesis (stating that the observed data stems from an exponential distribution with a maximum likelihood rate parameter) can be rejected with high confidence. Let's try another distribution, the **Birnbaum-Sanders distribution**, which is typically used to model failure times. (More information on the Birnbaum-Sanders distribution is available at https://en.wikipedia.org/wiki/Birnbaum-Saunders_distribution.)

```
>>> dist = st.fatigueLife
      args = dist.fit(survival)
      st.kstest(survival, dist.cdf, args)
KstestResult(statistic=0.188, pvalue=0.073)
```

This time, the p-value is about 0.073, so that we would not reject the null hypothesis with a five percent confidence level. When plotting the resulting distribution, we observe a better fit than with the exponential distribution:

```
>>> dist_fl = dist.pdf(days, *args)
nbins = 30
fig, ax = plt.subplots(1, 1, figsize=(6, 4))
ax.hist(survival, nbins)
ax.plot(days, dist_exp * len(survival) * smax / nbins,
        '-r', lw=3, label='exp')
ax.plot(days, dist_fl * len(survival) * smax / nbins,
        '--g', lw=3, label='BS')
ax.set_xlabel("Survival time (days)")
ax.set_ylabel("Number of patients")
ax.legend()
```



How it works...

Here, we give the calculations leading to the maximum likelihood estimation of the rate parameter for an exponential distribution:

$$\begin{aligned}\mathcal{L}(\lambda, \{s_i\}) &= P(\{s_i\} \mid \lambda) \\ &= \prod_{i=1}^n P(s_i \mid \lambda) && \text{(by independence of the } s_i\text{)} \\ &= \prod_{i=1}^n \lambda \exp(-\lambda s_i) \\ &= \lambda^n \exp\left(-\lambda \sum_{i=1}^n s_i\right) \\ &= \lambda^n \exp(-\lambda n \bar{s})\end{aligned}$$

Here, \bar{s} is the sample mean. In more complex situations, we would require numerical optimization methods in which the principle is to maximize the likelihood function using a standard numerical optimization algorithm (see *Chapter 9, Numerical Optimization*).

To find the maximum of this function, let's compute its derivative function with respect to λ :

$$\frac{d\mathcal{L}(\lambda, \{s_i\})}{d\lambda} = \lambda^{n-1} \exp(-\lambda n \bar{s}) (n - n \lambda \bar{s})$$

The root of this derivative is therefore $\lambda = 1/\bar{s}$.

There's more...

Here are a few references:

- ▶ Maximum likelihood on Wikipedia, available at https://en.wikipedia.org/wiki/Maximum_likelihood
- ▶ Kolmogorov-Smirnov test on Wikipedia, available at https://en.wikipedia.org/wiki/Kolmogorov-Smirnov_test
- ▶ Goodness of fit at https://en.wikipedia.org/wiki/Goodness_of_fit

The maximum likelihood method is parametric: the model belongs to a prespecified parametric family of distributions. In the next recipe, we will see a nonparametric kernel-based method.

See also

- ▶ The *Estimating a probability distribution nonparametrically with a kernel density estimation* recipe

Estimating a probability distribution nonparametrically with a kernel density estimation

In the previous recipe, we applied a **parametric estimation method**. We had a statistical model (the exponential distribution) describing our data, and we estimated a single parameter (the rate of the distribution). **Nonparametric estimation** deals with statistical models that do not belong to a known family of distributions. The parameter space is then *infinite-dimensional* instead of *finite-dimensional* (that is, we estimate *functions* rather than *numbers*).

Here, we use a **Kernel Density Estimation (KDE)** to estimate the density of probability of a spatial distribution. We look at the geographical locations of tropical cyclones from 1848 to 2013, based on data provided by the NOAA, the US' National Oceanic and Atmospheric Administration.

Getting ready

You need Cartopy, available at <http://scitools.org.uk/cartopy/>. You can install it with `conda install -c conda-forge cartopy`.

How to do it...

1. Let's import the usual packages. The kernel density estimation with a Gaussian kernel is implemented in `scipy.stats`:

```
>>> import numpy as np
     import pandas as pd
     import scipy.stats as st
     import matplotlib.pyplot as plt
     from matplotlib.colors import ListedColormap
     import cartopy.crs as ccrs
     %matplotlib inline
```

2. Let's open the data with pandas:

```
>>> # www.ncdc.noaa.gov/ibtracs/index.php?name=wmo-data
    df = pd.read_csv('https://github.com/ipython-books/'
                      'cookbook-2nd-data/blob/master/'
                      'Allstorms.ibtracs_wmo.v03r05.csv?'
                      'raw=true')
```

3. The dataset contains information about most storms since 1848. A single storm may appear multiple times across several consecutive days.

```
>>> df[df.columns[[0, 1, 3, 8, 9]]].head()
```

| | Serial_Num | Season | Basin | Latitude | Longitude |
|---|---------------|--------|-------|----------|-----------|
| 0 | 1848011S09080 | 1848 | SI | -8.6 | 79.8 |
| 1 | 1848011S09080 | 1848 | SI | -9.0 | 78.9 |
| 2 | 1848011S09080 | 1848 | SI | -10.4 | 73.2 |
| 3 | 1848011S09080 | 1848 | SI | -12.8 | 69.9 |
| 4 | 1848011S09080 | 1848 | SI | -13.9 | 68.9 |

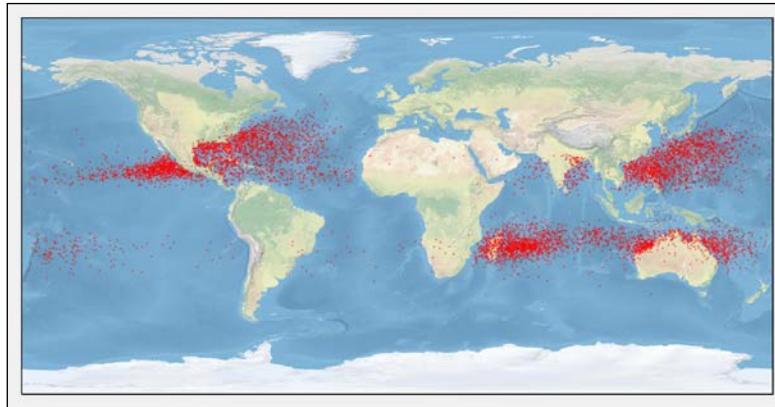
4. We use pandas' groupby() function to obtain the average location of every storm:

```
>>> dfs = df.groupby('Serial_Num')
    pos = dfs[['Latitude', 'Longitude']].mean()
    x = pos.Longitude.values
    y = pos.Latitude.values
    pos.head()
```

| Serial_Num | Latitude | Longitude |
|---------------|------------|-----------|
| 1848011S09080 | -15.918182 | 71.854545 |
| 1848011S15057 | -24.116667 | 52.016667 |
| 1848061S12075 | -20.528571 | 65.342857 |
| 1851080S15063 | -17.325000 | 55.400000 |
| 1851080S21060 | -23.633333 | 60.200000 |

5. We display the storms on a map with Cartopy. This toolkit allows us to easily project the geographical coordinates on the map.

```
>>> # We use a simple equirectangular projection,  
# also called Plate Carree.  
crs = ccrs.PlateCarree()  
# We create the map plot.  
ax = plt.axes(projection=crs)  
# We display the world map picture.  
ax.stock_img()  
# We display the storm locations.  
ax.scatter(x, y, color='r', s=.5, alpha=.25)
```



6. Before performing the kernel density estimation, we transform the storms' positions from the **geodetic coordinate system** (longitude and latitude) into the map's coordinate system, called **plate carrière**.

```
>>> geo = ccrs.Geodetic()  
h = geo.transform_points(crs, x, y)[:, :2].T  
h.shape  
(2, 6940)
```

7. Now, we perform the kernel density estimation on our (2, N) array.

```
>>> kde = st.gaussian_kde(h)
```

8. The `gaussian_kde()` routine returned a Python function. To see the results on a map, we need to evaluate this function on a 2D grid spanning the entire map. We create this grid with `meshgrid()`, and we pass the `x` and `y` values to the `kde()` function:

```
>>> k = 100  
# Coordinates of the four corners of the map.  
x0, x1, y0, y1 = ax.get_extent()
```

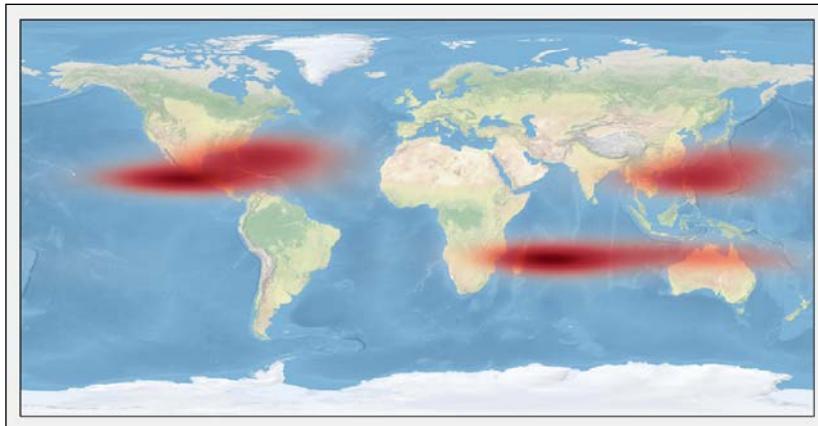
```
# We create the grid.  
tx, ty = np.meshgrid(np.linspace(x0, x1, 2 * k),  
                     np.linspace(y0, y1, k))  
# We reshape the grid for the kde() function.  
mesh = np.vstack((tx.ravel(), ty.ravel()))  
# We evaluate the kde() function on the grid.  
v = kde(mesh).reshape((k, 2 * k))
```

9. Before displaying the KDE heatmap on the map, we need to use a special colormap with a transparent channel. This will allow us to superimpose the heatmap on the stock image:

```
>>> # https://stackoverflow.com/a/37334212/1595060  
cmap = plt.get_cmap('Reds')  
my_cmap = cmap(np.arange(cmap.N))  
my_cmap[:, -1] = np.linspace(0, 1, cmap.N)  
my_cmap = ListedColormap(my_cmap)
```

10. Finally, we display the estimated density with `imshow()`:

```
>>> ax = plt.axes(projection=crs)  
ax.stock_img()  
ax.imshow(v, origin='lower',  
          extent=[x0, x1, y0, y1],  
          interpolation='bilinear',  
          cmap=my_cmap)
```



How it works...

The kernel density estimator of a set of n points $\{x_i\}$ is given as:

$$\hat{f}_h(x) = \frac{1}{nh} \sum_{i=1}^n K\left(\frac{x - x_i}{h}\right)$$

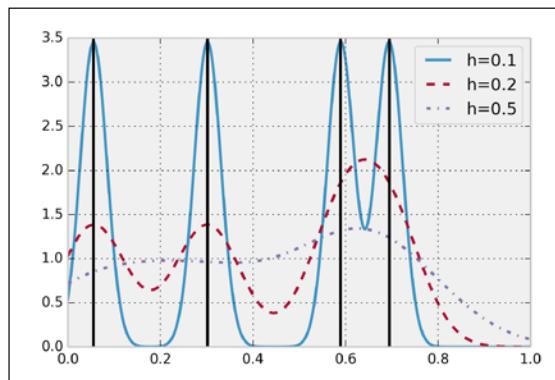
Here, $h > 0$ is a scaling parameter (the **bandwidth**) and $K(u)$ is the **kernel**, a symmetric function that integrates to 1. This estimator is to be compared with a classical histogram, where the kernel would be a top-hat function (a rectangle function taking its values in $\{0, 1\}$), but the blocks would be located on a regular grid instead of the data points. For more information on kernel density estimator, refer to https://en.wikipedia.org/wiki/Kernel_density_estimation.

Multiple kernels can be chosen. Here, we chose a **Gaussian kernel**, so that the KDE is the superposition of Gaussian functions centered on all the data points. It is an estimation of the density.

The choice of the bandwidth is not trivial; there is a tradeoff between a too low value (small bias, high variance: overfitting) and a too high value (high bias, small variance: underfitting). We will return to this important concept of **bias-variance tradeoff** in the next chapter. For more information on the bias-variance tradeoff, refer to https://en.wikipedia.org/wiki/Bias-variance_dilemma.

There are several methods to automatically choose a sensible bandwidth. SciPy uses a **rule of thumb** called **Scott's Rule**: $h = n^{**(-1. / (d + 4))}$. You will find more information at http://scipy.github.io/devdocs/generated/scipy.stats.gaussian_kde.html.

The following figure illustrates the KDE. The sample dataset contains four points in $[0, 1]$ (black lines). The estimated density is a smooth curve, represented here with different bandwidth values.



Kernel density estimation



There are other implementations of KDE in statsmodels and scikit-learn. You can find more information here: <http://jakevdp.github.io/blog/2013/12/01/kernel-density-estimation/>

See also

- ▶ The *Fitting a probability distribution to data with the maximum likelihood method* recipe

Fitting a Bayesian model by sampling from a posterior distribution with a Markov chain Monte Carlo method

In this recipe, we illustrate a very common and useful method for characterizing a posterior distribution in a Bayesian model. Imagine that you have some data and you want to obtain information about the underlying random phenomenon. In a frequentist approach, you could try to fit a probability distribution within a given family of distributions, using a parametric method such as the maximum likelihood method. The optimization procedure would yield parameters that maximize the probability of observing the data if given the null hypothesis.

In a Bayesian approach, you consider the parameters themselves as random variables. Their prior distributions reflect your initial knowledge about these parameters. After the observations, your knowledge is updated, and this is reflected in the posterior distributions of the parameters.

A typical goal for Bayesian inference is to characterize the posterior distributions. Bayes' theorem gives an analytical way to do this, but it is often impractical in real-world problems due to the complexity of the models and the number of dimensions. A **Markov chain Monte Carlo (MCMC)** method, such as the **Metropolis-Hastings algorithm**, gives a numerical method to approximate a posterior distribution.

Here, we introduce the **PyMC3** package, which gives an effective and natural interface for fitting a probabilistic model to data in a Bayesian framework. We will look at the annual frequency of storms in the northern Atlantic Ocean since the 1850s using data from NOAA, the US' National Oceanic and Atmospheric Administration. This example is largely inspired by the tutorial available in the official PyMC3 documentation at http://docs.pymc.io/notebooks/getting_started.html#Case-study-2--Coal-mining-disasters.

Getting ready

You need PyMC3, available at <http://docs.pymc.io>. You can install it with `conda install -c conda-forge pymc3`.

How to do it...

1. Let's import the standard packages and PyMC3:

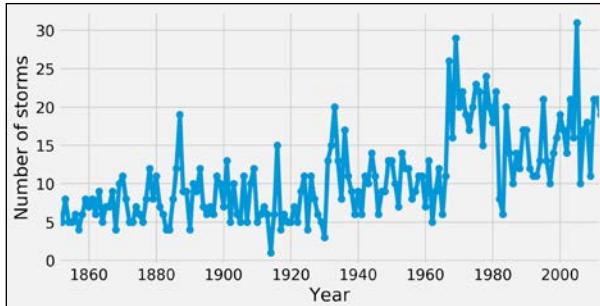
```
>>> import numpy as np
      import pandas as pd
      import pymc3 as pm
      import matplotlib.pyplot as plt
      %matplotlib inline
```

2. Let's import the data with pandas:

```
>>> # www.ncdc.noaa.gov/ibtracs/index.php?name=wmo-data
      df = pd.read_csv('https://github.com/ipython-books/'
                      'cookbook-2nd-data/blob/master/'
                      'Allstorms.ibtracs_wmo.v03r05.csv?'
                      'raw=true',
                      delim_whitespace=False)
```

3. With pandas, it only takes a single line of code to get the annual number of storms in the North Atlantic Ocean. We first select the storms in that basin (NA), then we group the rows by year (Season), and finally we take the number of unique storms (Serial_Num), as each storm can span several days (the `nunique()` method):

```
>>> cnt = df[df['Basin'] == 'NA'].groupby(
      'Season')[['Serial_Num']].nunique()
      # The years from 1851 to 2012.
      years = cnt.index
      y0, y1 = years[0], years[-1]
      arr = cnt.values
      >>> # Plot the annual number of storms.
      fig, ax = plt.subplots(1, 1, figsize=(8, 4))
      ax.plot(years, arr, '-o')
      ax.set_xlim(y0, y1)
      ax.set_xlabel("Year")
      ax.set_ylabel("Number of storms")
```



4. Now we define our probabilistic model. We assume that storms arise following a time-dependent **Poisson process** with a deterministic rate. We assume that this rate is a piecewise-constant function that takes a first value `early_mean` before a switch point `switchpoint`, and a second value `late_mean` after that point. These three unknown parameters are treated as random variables (we will describe them more in the *How it works...* section). In the model, the annual number of storms per year follows a Poisson distribution (this is a property of Poisson processes).



A Poisson process (https://en.wikipedia.org/wiki/Poisson_process) is a particular point process—that is, a stochastic process describing the random occurrence of instantaneous events. The Poisson process is fully random: the events occur independently at a given rate. See also *Chapter 13, Stochastic Dynamical Systems*.

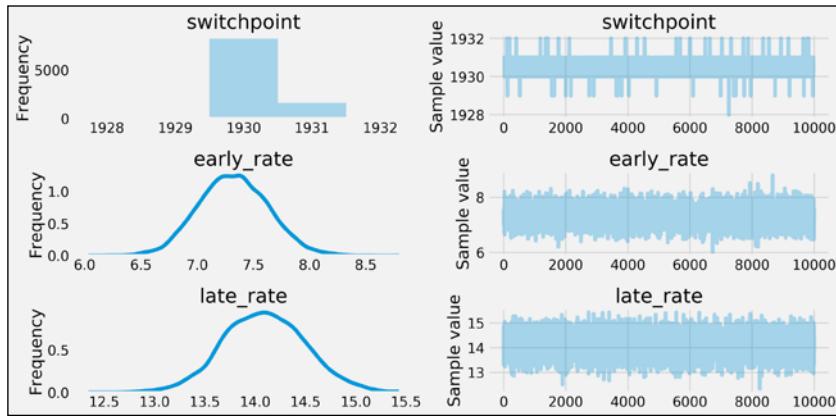
```
>>> # We define our model.
      with pm.Model() as model:
          # We define our three variables.
          switchpoint = pm.DiscreteUniform(
              'switchpoint', lower=y0, upper=y1)
          early_rate = pm.Exponential('early_rate', 1)
          late_rate = pm.Exponential('late_rate', 1)
          # The rate of the Poisson process is a piecewise
          # constant function.
          rate = pm.math.switch(switchpoint >= years,
                                early_rate, late_rate)
          # The annual number of storms per year follows
          # a Poisson distribution.
          storms = pm.Poisson('storms', rate, observed=arr)
```

5. Now, we sample from the posterior distribution given the observed data. The `sample(10000)` method launches the fitting iterative procedure with 10000 iterations, which may take a few seconds:

```
>>> with model:
    trace = pm.sample(10000)
Assigned Metropolis to switchpoint
Assigned NUTS to early_rate_log_
Assigned NUTS to late_rate_log_
100%|██████████| 10500/10500 [00:05<00:00, 1757.23it/s]
```

6. Once the sampling has finished, we can plot the distribution and paths of the Markov chains:

```
>>> pm.traceplot(trace)
```



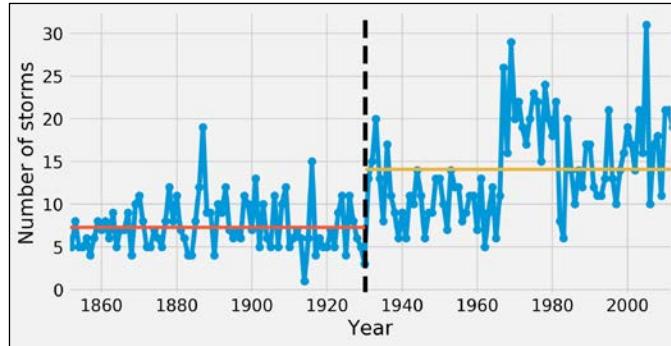
Each row represents a variable. The left plot is a histogram of the corresponding Markov chain, which gives the posterior distribution of the variable. The right plot is an arbitrarily-chosen path of a Markov chain, showing the evolution of the variable during the fitting procedure.

7. Taking the sample mean of these distributions, we get posterior estimates for the three unknown parameters, including the year where the frequency of storms suddenly increased:

```
>>> s = trace['switchpoint'].mean()
em = trace['early_rate'].mean()
lm = trace['late_rate'].mean()
s, em, lm
(1930.171, 7.316, 14.085)
```

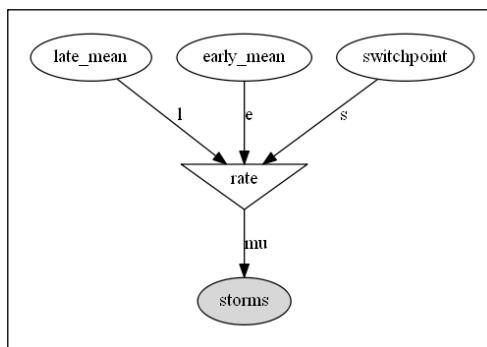
8. Finally, we can plot the estimated rate on top of the observations:

```
>>> fig, ax = plt.subplots(1, 1, figsize=(8, 4))
ax.plot(years, arr, '-o')
ax.axvline(s, color='k', ls='--')
ax.plot([y0, s], [em, em], '--', lw=3)
ax.plot([s, y1], [lm, lm], '--', lw=3)
ax.set_xlim(y0, y1)
ax.set_xlabel("Year")
ax.set_ylabel("Number of storms")
```



How it works...

The general idea is to define a Bayesian probabilistic model and to fit it to the data. This model may be the starting point of an estimation or decision task. The model is essentially described by stochastic or deterministic variables linked together within a **Directed Acyclic Graph (DAG)**. A is linked to B if B is entirely or partially determined by A. The following figure shows the graph of the model used in this recipe:



Dependency graph of the variables

Stochastic variables follow distributions that can be parameterized by fixed numbers or other variables in the model. Parameters may be random variables themselves, reflecting knowledge prior to the observations. This is the core of Bayesian modeling.

The goal of the analysis is to include the observations into the model in order to update our knowledge as more and more data is available. Although Bayes' theorem gives us an exact way to compute those posterior distributions, it is rarely practical in real-world problems. This is notably due to the complexity of the models. Alternatively, numerical methods have been developed in order to tackle this problem.

The MCMC method used here allows us to sample from a complex distribution by simulating a Markov chain that has the desired distribution as its equilibrium distribution. The **Metropolis-Hastings algorithm** is a particular application of this method to our current example.

There's more...

Here are a few references:

- ▶ A free e-book on the subject, by Cameron Davidson-Pilon, entirely written in the Jupyter Notebook, available at <http://camdavidsonpilon.github.io/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers/>
- ▶ The MCMC method introduced at https://en.wikipedia.org/wiki/Markov_chain_Monte_Carlo
- ▶ The Metropolis-Hastings algorithm introduced at https://en.wikipedia.org/wiki/Metropolis-Hastings_algorithm

See also

- ▶ The *Getting started with Bayesian methods* recipe

Analyzing data with the R programming language in the Jupyter Notebook

R (<http://www.r-project.org>) is an open-source domain-specific programming language for statistics. Its syntax is well-adapted to statistical modeling and data analysis. By contrast, Python's syntax is typically more convenient for general-purpose programming. Luckily, Jupyter allows us to have the best of both worlds. For example, we can insert R code snippets anywhere in a normal Jupyter notebook. We can continue using Python and pandas for data loading and wrangling, and switch to R to design and fit statistical models. Using R instead of Python for these tasks is more than a matter of programming syntax; R comes with an impressive statistical toolbox.

In this recipe, we will show how to interface R with Python in the Jupyter Notebook, and we will illustrate the most basic capabilities of R with a simple data analysis example.



There is another way of using R in the Jupyter Notebook, which is to install **IRkernel**, the R kernel for Jupyter. Using this method, all of the code of an IRkernel notebook is written in R, not in Python. You will find more information at <https://irkernel.github.io/installation/>.

Getting ready

You need the `statsmodels` package for this recipe. It should be installed by default with Anaconda, but you can always install it with `conda install statsmodels`.

You also need R and `rpy2` (<https://rpy2.readthedocs.io/>). There are three steps to use R with Python:

1. Download R from <https://cran.r-project.org/> and install it.
2. Install `rpy2` with `conda install rpy2`.
3. Run the `%load_ext rpy2.ipython` command in a Jupyter notebook.



`rpy2` does not appear to work well on Windows. We recommend using Linux or macOS.

How to do it...

Here, we will use the following workflow: first, we load data from Python. Then, we use R to design and fit a model, and to make some plots in the Jupyter Notebook. We could also use R only for the entire recipe, or Python only. The goal of this recipe is precisely to show how to use both languages in the same Jupyter notebook.

1. Let's load the `longley` dataset with the `statsmodels` package. This dataset contains a few economic indicators in the US from 1947 to 1962. We also load the IPython R extension:

```
>>> import statsmodels.datasets as sd  
>>> data = sd.longley.load_pandas()  
>>> %load_ext rpy2.ipython
```

2. We define `x` and `y` as the **exogeneous** (independent) and **endogenous** (dependent) variables, respectively. The endogenous variable quantifies the total employment in the country.

```
>>> data.endog_name, data.exog_name
('TOTEMP', ['GNPDEFL', 'GNP', 'UNEMP',
             'ARMED', 'POP', 'YEAR'])

>>> y, x = data.endog, data.exog
```

3. For convenience, we add the endogenous variable to the `x` DataFrame:

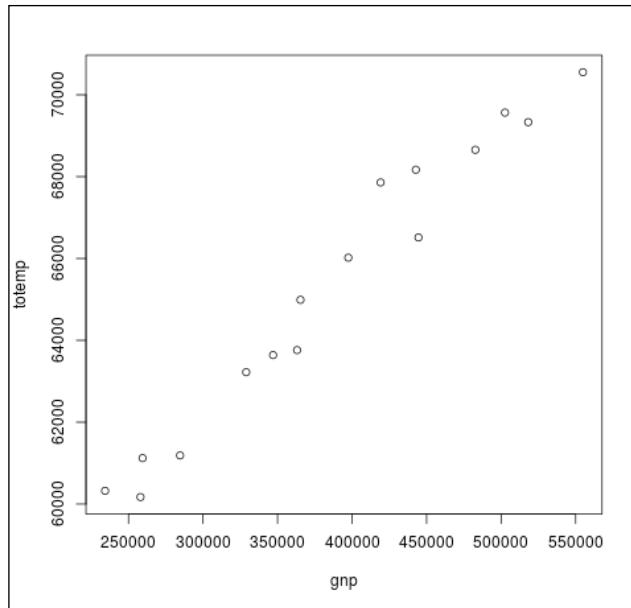
```
>>> x['TOTEMP'] = y
>>> x
```

| | GNPDEFL | GNP | UNEMP | ARMED | POP | YEAR | TOTEMP |
|-----|---------|----------|--------|--------|----------|--------|---------|
| 0 | 83.0 | 234289.0 | 2356.0 | 1590.0 | 107608.0 | 1947.0 | 60323.0 |
| 1 | 88.5 | 259426.0 | 2325.0 | 1456.0 | 108632.0 | 1948.0 | 61122.0 |
| 2 | 88.2 | 258054.0 | 3682.0 | 1616.0 | 109773.0 | 1949.0 | 60171.0 |
| 3 | 89.5 | 284599.0 | 3351.0 | 1650.0 | 110929.0 | 1950.0 | 61187.0 |
| 4 | 96.2 | 328975.0 | 2099.0 | 3099.0 | 112075.0 | 1951.0 | 63221.0 |
| ... | ... | ... | ... | ... | ... | ... | ... |
| 11 | 110.8 | 444546.0 | 4681.0 | 2637.0 | 121950.0 | 1958.0 | 66513.0 |
| 12 | 112.6 | 482704.0 | 3813.0 | 2552.0 | 123366.0 | 1959.0 | 68655.0 |
| 13 | 114.2 | 502601.0 | 3931.0 | 2514.0 | 125368.0 | 1960.0 | 69564.0 |
| 14 | 115.7 | 518173.0 | 4806.0 | 2572.0 | 127852.0 | 1961.0 | 69331.0 |
| 15 | 116.9 | 554894.0 | 4007.0 | 2827.0 | 130081.0 | 1962.0 | 70551.0 |

16 rows × 7 columns

4. We will make a simple plot in R. First, we need to pass Python variables to R. We use the `%R -i var1,var2` magic. Then, we call R's `plot()` command:

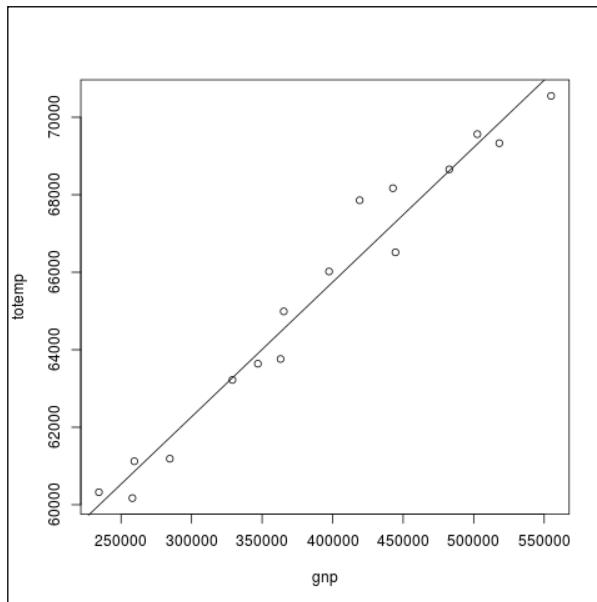
```
>>> gnp = x['GNP']
      totemp = x['TOTEMP']
>>> %R -i totemp,gnp plot(gnp, totemp)
```



5. Now that the data has been passed to R, we can fit a linear model to the data. In R, the `lm()` function lets us perform a linear regression. Here, we want to express `totemp` (total employment) as a function of the country's GNP. We use the `%%R` cell magic to write several lines of R code in a cell:

```
>>> %%R
    # Least-squares regression
    fit <- lm(totemp ~ gnp)
    # Display the coefficients of the fit.
    print(fit$coefficients)
    # Plot the data points.
```

```
plot(gnp, totemp)
# And plot the linear regression.
abline(fit)
```



How it works...

The `-i` and `-o` options of the `%%R` magic allow us to pass variables back and forth between IPython and R. The variable names need to be separated by commas. You can find more information about the `%R` magic in the documentation available at <https://rpy2.readthedocs.io/>.

In R, the tilde (~) expresses the dependence of a dependent variable upon one or several independent variables. The `lm()` function allows us to fit a simple linear regression model to the data. Here, `totemp` is expressed as a function of `gnp`:

$$\text{totemp} = a \times \text{gnp} + b$$

Here, b (intercept) and a are the coefficients of the linear regression model. These two values are returned by `fit$coefficients` in R, where `fit` is the fitted model.

Our data points do not satisfy this relation exactly, but the coefficients are chosen so as to minimize the error between this linear prediction and the actual values. This is typically done by minimizing the following least squares error:

$$r(a, b) = \sum_{i=1}^n (\text{totemp}_i - (a \times \text{gnp}_i + b))^2$$

The data points are $(\text{gnp}_i, \text{totemp}_i)$ here. The coefficients a and b that are returned by `lm()` make this sum minimal: they fit the data best.

There's more...

Regression is an important statistical concept that we will see in greater detail in the next chapter. Here are a few references:

- ▶ Regression analysis on Wikipedia, available at https://en.wikipedia.org/wiki/Regression_analysis
- ▶ Least squares method on Wikipedia, available at https://en.wikipedia.org/wiki/Linear_least_squares_%28mathematics%29

Here are a few references about R:

- ▶ Introduction to R available at <http://cran.r-project.org/doc/manuals/R-intro.html>
- ▶ R tutorial available at <http://www.cyclismo.org/tutorial/R/>
- ▶ **CRAN, or Comprehensive R Archive Network**, containing many packages for R, available at <http://cran.r-project.org>

See also

- ▶ The *Exploring a dataset with pandas and Matplotlib* recipe

8

Machine Learning

In this chapter, we will cover the following topics:

- ▶ Getting started with scikit-learn
- ▶ Predicting who will survive on the Titanic with logistic regression
- ▶ Learning to recognize handwritten digits with a K-nearest neighbors classifier
- ▶ Learning from text – Naive Bayes for Natural Language Processing
- ▶ Using support vector machines for classification tasks
- ▶ Using a random forest to select important features for regression
- ▶ Reducing the dimensionality of a dataset with a principal component analysis
- ▶ Detecting hidden structures in a dataset with clustering

Introduction

In the previous chapter, we were interested in getting insight into data, understanding complex phenomena through partial observations, and making informed decisions in the presence of uncertainty. Here, we are still interested in analyzing and processing data using statistical tools. However, the goal is not necessarily to *understand* the data, but to learn from it.

Learning from data is close to what we do as humans. From our experience, we intuitively learn general facts and relations about the world, even if we don't fully understand their complexity. The increasing computational power of computers makes them able to learn from data too. That's the heart of **machine learning**, a branch of artificial intelligence at the intersection of computer science, statistics, and applied mathematics.

This chapter is a hands-on introduction to some of the most basic methods in machine learning. These methods are routinely used by data scientists. We will use these methods with **scikit-learn**, a popular and user-friendly Python package for machine learning.

A bit of vocabulary

In this introduction, we will explain the fundamental definitions and concepts of machine learning.

Learning from data

In machine learning, most data can be represented as a table of numerical values. Every row is called an **observation**, a **sample**, or a **data point**. Every column is called a **feature** or a **variable**.

Let's call N the number of rows (or the number of points) and D the number of columns (or number of features). The number D is also called the **dimensionality** of the data. The reason is that we can view this table as a set E of vectors in a space with D dimensions (or vector space). Here, a vector x contains D numbers (x_1, \dots, x_D), also called **components**. This mathematical point of view is very useful and we will use it throughout this chapter.

We make the distinction between supervised learning and unsupervised learning:

- ▶ **Supervised learning** is when we have a label y associated with a data point x . The goal is to learn the mapping from x to y from our data. The data gives us this mapping for a finite set of points, but what we want is to generalize this mapping to the full set E , or at least to a larger set of points.
- ▶ **Unsupervised learning** is when we don't have any labels. What we want to do is discover some form of hidden structure in the data.

Supervised learning

Mathematically, supervised learning consists of finding a function f that maps the set of points E to a set of labels F , knowing a finite set of associations (x, y) , which is given by our data. This is what generalization is about: after observing the pairs (x_i, y_i) , given a new x , we are able to find the corresponding y by applying the function f to x .

It is a common practice to split the set of data points into two subsets: the **training set** and the **test set**. We learn the function f on the training set and test it on the test set. This is essential when assessing the predictive power of a model. By training and testing a model on the same set, our model might not be able to generalize well. This is the fundamental concept of **overfitting**, which we will detail later in this chapter.

We generally make the distinction between classification and regression, two particular instances of supervised learning.

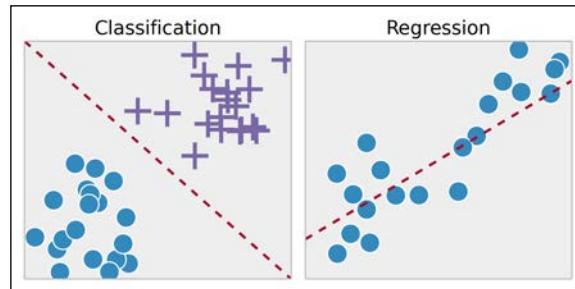
Classification is when our labels y can only take a finite set of values (categories). Examples include:

- ▶ **Handwritten digit recognition:** x is an image with a handwritten digit; y is a digit between 0 and 9
- ▶ **Spam filtering:** x is an email and y is 1 or 0, depending on whether that email is spam or not

Regression is when our labels y can take any real (continuous) value. Examples include:

- ▶ Predicting stock market data
- ▶ Predicting sales
- ▶ Detecting the age of a person from a picture

A classification task yields a division of our space E in different regions (also called **partitions**), each region being associated to one particular value of the label y . A regression task yields a mathematical model that associates a real number to any point x in the space E . This difference is illustrated in the following figure:



Difference between classification and regression

Classification and regression can be combined. For example, in the **probit model**, although the dependent variable is binary (classification), the *probability* that this variable belongs to one category can also be modeled (regression). We will see an example in the recipe about logistic regression. For more information on the probit model, refer to https://en.wikipedia.org/wiki/Probit_model.

Unsupervised learning

Broadly speaking, unsupervised learning helps us discover systemic structures in our data. This is harder to grasp than supervised learning, in that there is generally no precise question and answer.

Here are a few important tasks related to unsupervised learning:

- ▶ **Clustering:** Grouping similar points together within clusters
- ▶ **Density estimation:** Estimating a probability density function that can explain the distribution of the data points
- ▶ **Dimension reduction:** Getting a simple representation of high-dimensional data points by projecting them onto a lower-dimensional space (notably for data visualization)
- ▶ **Manifold learning:** Finding a low-dimensional manifold containing the data points (also known as nonlinear dimension reduction)

Feature selection and feature extraction

In a supervised learning context, when our data contains many features, it is sometimes necessary to choose a subset of them. The features we want to keep are those that are most relevant to our question. This is the problem of **feature selection**.

Additionally, we might want to extract new features by applying complex transformations on our original dataset. This is **feature extraction**. For example, in computer vision, training a classifier directly on the pixels is not the most efficient method in general. We might want to extract the relevant points of interest or make appropriate mathematical transformations. These steps depend on our dataset and on the questions we want to answer.

For example, it is often necessary to preprocess the data before learning models. **Feature scaling** (or **data normalization**) is a common preprocessing step where features are linearly rescaled to fit in the range $[-1, 1]$ or $[0, 1]$.

Feature extraction and feature selection involve a balanced combination of domain expertise, intuition, and mathematical methods. These early steps are crucial, and they might be even more important than the learning steps themselves. The reason is that the few dimensions that are relevant to our problem are generally hidden in the high dimensionality of our dataset. We need to uncover the low-dimensional structure of interest to improve the efficiency of the learning models.

We will see a few feature selection and feature extraction methods in this chapter. Methods that are specific to signals, images, or sounds will be covered in *Chapter 10, Signal Processing*, and *Chapter 11, Image and Audio Processing*.

Deep learning has profoundly revolutionized machine learning in the last few years. A major characteristic of this range of methods is that feature selection and extraction are often included in the model itself. The most relevant features are automatically selected by the algorithm. This method works particularly well on images, sounds, and videos. Typically, however, deep learning requires a huge amount of training data and computational power. Covering deep learning methods in Python is beyond the scope of this book, but we give a few references at the end of this introduction.

Here are a few further references:

- ▶ Feature selection in scikit-learn, documented at http://scikit-learn.org/stable/modules/feature_selection.html
- ▶ Feature selection on Wikipedia at https://en.wikipedia.org/wiki/Feature_selection

Overfitting, underfitting, and the bias-variance tradeoff

A central notion in machine learning is the trade-off between **overfitting** and **underfitting**.

A model may be able to represent our data accurately. However, if it is too accurate, it might not generalize well to unobserved data. For example, in facial recognition, a too-accurate model would be unable to identify someone who styled their hair differently that day. The reason is that our model might learn irrelevant features in the training data. On the contrary, an insufficiently trained model would not generalize well either. For example, it would be unable to correctly recognize twins. For more information on overfitting, refer to <https://en.wikipedia.org/wiki/Overfitting>.

A popular solution to reduce overfitting consists of adding structure to the model—for example, with **regularization**. This method favors simpler models during training (Occam's razor). You will find more information at https://en.wikipedia.org/wiki/Regularization_%28mathematics%29.

The **bias-variance dilemma** is closely related to the issue of overfitting and underfitting. The **bias** of a model quantifies how precise it is across training sets. The **variance** quantifies how sensitive the model is to small changes in the training set. A **robust** model is not overly sensitive to small changes. The dilemma involves minimizing both bias and variance; we want a precise and robust model. Simpler models tend to be less accurate but more robust. Complex models tend to be more accurate but less robust. For more information on the bias-variance dilemma, refer to https://en.wikipedia.org/wiki/Bias-variance_dilemma.

The importance of this trade-off cannot be overstated. This question pervades the entire discipline of machine learning. We will see concrete examples in this chapter.

Model selection

As we will see in this chapter, there are many supervised and unsupervised algorithms. For example, well-known classifiers that we will cover in this chapter include logistic regression, nearest-neighbors, Naive Bayes, and support vector machines. There are many other algorithms that we can't cover here.

No model performs uniformly better than the others. One model may perform well on one dataset and badly on another. This is the question of **model selection**.

We will see systematic methods to assess the quality of a model on a particular dataset (notably cross-validation). In practice, machine learning is not an exact science in that it frequently involves trial and error. We need to try different models and empirically choose the one that performs best.

That being said, understanding the details of the learning models allows us to gain intuition about which model is best adapted to our current problem.

Here are a few references on this question:

- ▶ Model selection on Wikipedia, available at https://en.wikipedia.org/wiki/Model_selection
- ▶ Model evaluation in scikit-learn's documentation, available at http://scikit-learn.org/stable/modules/model_evaluation.html
- ▶ Blog post on how to choose a classifier, available at <http://blog.echen.me/2011/04/27/choosing-a-machine-learning-classifier/>

Machine learning references

Here are a few excellent, math-heavy textbooks on machine learning:

- ▶ *Pattern Recognition and Machine Learning*, Christopher M. Bishop, (2006), Springer
- ▶ *Machine Learning – A Probabilistic Perspective*, Kevin P. Murphy, (2012), MIT Press
- ▶ *The Elements of Statistical Learning*, Trevor Hastie, Robert Tibshirani, Jerome Friedman, (2009), Springer

Here are a few books more oriented toward programmers without a strong mathematical background:

- ▶ *Machine Learning for Hackers*, Drew Conway, John Myles White, (2012), O'Reilly Media
- ▶ *Machine Learning in Action*, Peter Harrington, (2012), Manning Publications Co.
- ▶ *Python Machine Learning*, Sebastian Raschka (2015), Packt Publishing

Further references can be found here:

- ▶ Awesome Machine Learning resources, at <https://github.com/josephmisiti/awesome-machine-learning>
- ▶ Statistical Learning lectures on Awesome Math, at <https://github.com/rossant/awesome-math/#statistical-learning>

Important classes of machine learning methods that we couldn't cover in this chapter include **neural networks** and **deep learning**. Deep learning is the subject of very active research in machine learning. Many state-of-the-art results are currently achieved by using deep learning methods.

Here are few references on deep learning:

- ▶ Awesome Deep Learning resources, at <https://github.com/ChristosChristofidis/awesome-deep-learning>
- ▶ Coursera Deep Learning Specialization course, at <https://www.coursera.org/specializations/deep-learning>
- ▶ Udacity Deep Learning course, at <https://www.udacity.com/course/deep-learning--ud730>
- ▶ Keras Tutorial: Deep Learning in Python, at <https://www.datacamp.com/community/tutorials/deep-learning-python>
- ▶ Deep Learning with Python, a book by François Chollet, Manning Publications, at <https://www.manning.com/books/deep-learning-with-python>

Finally, here are a few lists of public datasets that can be used for data science projects:

- ▶ List of datasets for machine learning research, at https://en.wikipedia.org/wiki/List_of_datasets_for_machine_learning_research
- ▶ Awesome Public Datasets, at <https://github.com/caesar0301/awesome-public-datasets>
- ▶ Datasets for Data Science and Machine Learning, at <https://elitedatascience.com/datasets>
- ▶ Kaggle Datasets, at <https://www.kaggle.com/datasets>

Getting started with scikit-learn

In this recipe, we introduce the basics of the machine learning **scikit-learn** package (<http://scikit-learn.org>). This package is the main tool we will use throughout this chapter. Its clean API makes it easy to define, train, and test models.

We will show here a basic example of linear regression in the context of curve fitting. This toy example will allow us to illustrate key concepts such as linear models, overfitting, underfitting, regularization, and cross-validation.

Getting ready

You can find all instructions to install scikit-learn in the main documentation. For more information, refer to <http://scikit-learn.org/stable/install.html>. Anaconda comes with scikit-learn by default, but, if needed, you can install it manually by typing `conda install scikit-learn` in a Terminal.

How to do it...

We will generate a one-dimensional dataset with a simple model (including some noise), and we will try to fit a function to this data. With this function, we can predict values on new data points. This is a curve fitting regression problem.

1. First, let's make all the necessary imports:

```
>>> import numpy as np  
      import scipy.stats as st  
      import sklearn.linear_model as lm  
      import matplotlib.pyplot as plt  
      %matplotlib inline
```

2. We now define a deterministic nonlinear function underlying our generative model:

```
>>> def f(x):  
      return np.exp(3 * x)
```

3. We generate the values along the curve on $[0, 2]$:

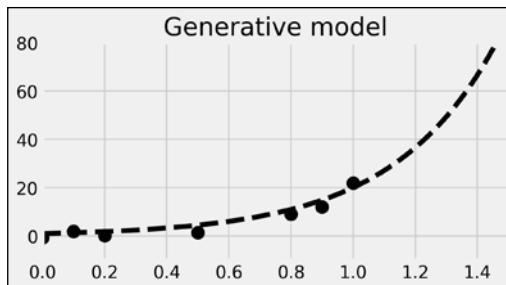
```
>>> x_tr = np.linspace(0., 2, 200)  
y_tr = f(x_tr)
```

4. Now, let's generate data points within $[0, 1]$. We use the function f and we add some Gaussian noise:

```
>>> x = np.array([0, .1, .2, .5, .8, .9, 1])  
y = f(x) + 2 * np.random.randn(len(x))
```

5. Let's plot our data points on $[0, 1]$:

```
>>> fig, ax = plt.subplots(1, 1, figsize=(6, 3))  
ax.plot(x_tr, y_tr, '--k')  
ax.plot(x, y, 'ok', ms=10)  
ax.set_xlim(0, 1.5)  
ax.set_ylim(-10, 80)  
ax.set_title('Generative model')
```



In the image, the dotted curve represents the generative model.

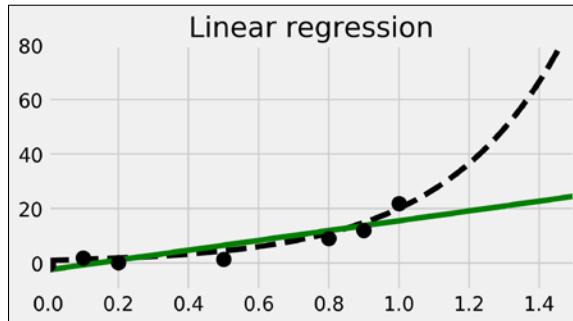
6. Now, we use scikit-learn to fit a linear model to the data. There are three steps. First, we create the model (an instance of the `LinearRegression` class). Then, we fit the model to our data. Finally, we predict values from our trained model.

```
>>> # We create the model.
lr = lm.LinearRegression()
# We train the model on our training dataset.
lr.fit(x[:, np.newaxis], y)
# Now, we predict points with our trained model.
y_lr = lr.predict(x_tr[:, np.newaxis])
```

We need to convert `x` and `x_tr` to column vectors, as it is a general convention in scikit-learn that observations are rows, while features are columns. Here, we have seven observations with one feature.

7. We now plot the result of the trained linear model. We obtain a regression line in green here:

```
>>> fig, ax = plt.subplots(1, 1, figsize=(6, 3))
ax.plot(x_tr, y_tr, '--k')
ax.plot(x_tr, y_lr, 'g')
ax.plot(x, y, 'ok', ms=10)
ax.set_xlim(0, 1.5)
ax.set_ylim(-10, 80)
ax.set_title("Linear regression")
```



8. The linear fit is not well-adapted here, as the data points are generated according to a nonlinear model (an exponential curve). Therefore, we are now going to fit a nonlinear model. More precisely, we will fit a polynomial function to our data points. We can still use linear regression for this, by precomputing the exponents of our data points. This is done by generating a Vandermonde matrix, using the `np.vander()` function. We will explain this trick in *How it works....* In the following code, we perform and plot the fit:

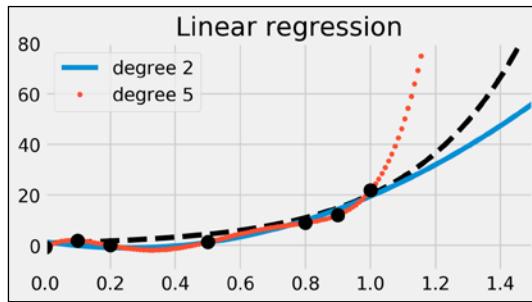
```
>>> lrp = lm.LinearRegression()
fig, ax = plt.subplots(1, 1, figsize=(6, 3))
```

```

    ax.plot(x_tr, y_tr, '--k')

    for deg, s in zip([2, 5], ['-', '.']):
        lrp.fit(np.vander(x, deg + 1), y)
        y_lrp = lrp.predict(np.vander(x_tr, deg + 1))
        ax.plot(x_tr, y_lrp, s,
                label=f'degree {deg}')
    ax.legend(loc=2)
    ax.set_xlim(0, 1.5)
    ax.set_ylim(-10, 80)
    # Print the model's coefficients.
    print(f'Coefficients, degree {deg}:\n\t',
          '\t'.join(f'{c:.2f}' for c in lrp.coef_))
    ax.plot(x, y, 'ok', ms=10)
    ax.set_title("Linear regression")
Coefficients, degree 2: 36.95 -18.92 0.00
Coefficients, degree 5: 903.98 -2245.99 1972.43 -686.45 78.64 0.00

```



We have fitted two polynomial models of degree 2 and 5. The degree 2 polynomial appears to fit the data points less precisely than the degree 5 polynomial. However, it seems more robust; the degree 5 polynomial seems really bad at predicting values outside the data points (look for example at the $x \geq 1$ portion). This is what we call **overfitting**; by using a too-complex model, we obtain a better fit on the trained dataset, but a less robust model outside this set.

9. We will now use a different learning model called **ridge regression**. It works like linear regression except that it prevents the polynomial's coefficients from becoming too big. This is what happened in the previous example. By adding a **regularization term** in the **loss function**, ridge regression imposes some structure on the underlying model. We will see more details in the next section.

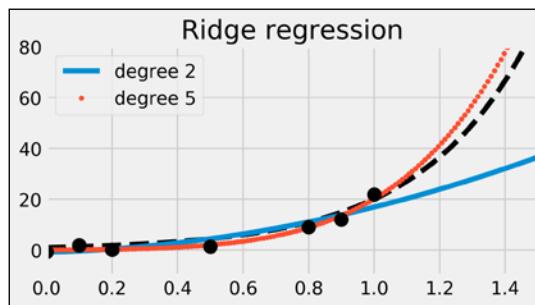
The ridge regression model has a meta-parameter, which represents the weight of the regularization term. We could try different values with trial and error using the Ridge class. However, scikit-learn provides another model called RidgeCV, which includes a parameter search with cross-validation. In practice, this means that we don't have to tweak this parameter by hand—scikit-learn does it for us. As the models of scikit-learn always follow the fit-predict API, all we have to do is replace lm.LinearRegression() with lm.RidgeCV() in the previous code. We will give more details in the next section.

```
>>> ridge = lm.RidgeCV()

    fig, ax = plt.subplots(1, 1, figsize=(6, 3))
    ax.plot(x_tr, y_tr, '--k')

    for deg, s in zip([2, 5], ['-', '.']):
        ridge.fit(np.vander(x, deg + 1), y)
        y_ridge = ridge.predict(np.vander(x_tr, deg + 1))
        ax.plot(x_tr, y_ridge, s,
                label='degree ' + str(deg))
    ax.legend(loc=2)
    ax.set_xlim(0, 1.5)
    ax.set_ylim(-10, 80)
    # Print the model's coefficients.
    print(f'Coefficients, degree {deg}:',
          '\n'.join(f'{c:.2f}' for c in ridge.coef_))

    ax.plot(x, y, 'ok', ms=10)
    ax.set_title("Ridge regression")
Coefficients, degree 2: 14.43 3.27 0.00
Coefficients, degree 5: 7.07 5.88 4.37 2.37 0.40 0.00
```



This time, the degree 5 polynomial seems more precise than the simpler degree 2 polynomial (which now causes **underfitting**). Ridge regression mitigates the overfitting issue here. Observe how the degree 5 polynomial's coefficients are much smaller than in the previous example.

How it works...

In this section, we explain all the aspects covered in this recipe.

scikit-learn API

scikit-learn implements a clean and coherent API for supervised and unsupervised learning. Our data points should be stored in an (N, D) matrix X , where N is the number of observations and D is the number of features. In other words, each row is an observation. The first step in a machine learning task is to define what the matrix X is exactly.

In a supervised learning setup, we also have a target, an N -long vector y with a scalar value for each observation. This value is either continuous or discrete, depending on whether we have a regression or classification problem, respectively.

In scikit-learn, models are implemented in classes that have the `fit()` and `predict()` methods. The `fit()` method accepts the data matrix X as input, and y as well for supervised learning models. This method trains the model on the given data.

The `predict()` method also takes data points as input (as an (M, D) matrix). It returns the labels or transformed points as predicted by the trained model.

Ordinary Least Squares regression

Ordinary Least Squares (OLS) regression is one of the simplest regression methods. It consists of approaching the output values y_i with a linear combination of X_{ij} :

$$\forall i \in \{1, \dots, N\}, \quad \hat{y}_i = \sum_{j=1}^D w_j X_{ij}, \quad \text{or, in matrix form: } \hat{\mathbf{y}} = \mathbf{X}\mathbf{w}.$$

Here, $w = (w_1, \dots, w_D)$ is the (unknown) **parameter vector**. Also, \hat{y} represents the model's output. We want this vector to match the data points y as closely as possible. Of course, the exact equality $\hat{y} = y$ cannot hold in general (there is always some noise and uncertainty—models are idealizations of reality). Therefore, we want to *minimize* the difference between these two vectors. The OLS regression method consists of minimizing the following **loss function**:

$$\min_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 = \min_{\mathbf{w}} \left(\sum_{i=1}^N (y_i - \hat{y}_i)^2 \right)$$

This sum of the components squared is called the **L² norm**. It is convenient because it leads to differentiable loss functions so that gradients can be computed and common optimization procedures can be performed.

Polynomial interpolation with linear regression

OLS regression fits a linear model to the data. The model is linear both in the data points X_i and in the parameters w_j . In our example, we obtain a poor fit because the data points were generated according to a nonlinear generative model (an exponential function).

However, we can still use the linear regression method with a model that is linear in w_j but nonlinear in x_i . To do this, we need to increase the number of dimensions in our dataset by using a basis of polynomial functions. In other words, we consider the following data points:

$$\mathbf{x}_i, \mathbf{x}_i^2, \dots, \mathbf{x}_i^D$$

Here, D is the maximum degree. The input matrix X is therefore the Vandermonde matrix associated to the original data points x_i . For more information on the **Vandermonde matrix**, refer to https://en.wikipedia.org/wiki/Vandermonde_matrix.

Training a linear model on these new data points is equivalent to training a polynomial model on the original data points.

Ridge regression

Polynomial interpolation with linear regression can lead to overfitting if the degree of the polynomials is too large. By capturing the random fluctuations (noise) instead of the general trend of the data, the model loses some of its predictive power. This corresponds to a divergence of the polynomial's coefficients w_j .

A solution to this problem is to prevent these coefficients from growing unboundedly. With **ridge regression** (also known as **Tikhonov regularization**), this is done by adding a regularization term to the loss function. For more details on Tikhonov regularization, refer to https://en.wikipedia.org/wiki/Tikhonov_regularization.

$$\min_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \alpha \|\mathbf{w}\|_2^2$$

By minimizing this loss function, we not only minimize the error between the model and the data (first term, related to the bias), but also the size of the model's coefficients (second term, related to the variance). The bias-variance trade-off is quantified by the hyperparameter α , which specifies the relative weight between the two terms in the loss function.

Here, ridge regression led to a polynomial with smaller coefficients, and thus a better fit.

Cross-validation and grid search

A drawback of the ridge regression model compared to the ordinary least squares model is the presence of an extra hyperparameter α . The quality of the prediction depends on the choice of this parameter. One possibility would be to fine-tune this parameter manually, but this procedure can be tedious and can also lead to overfitting problems.

To solve this problem, we can use a **grid search**: we loop over many possible values for α , and we evaluate the performance of the model for each possible value. Then, we choose the parameter that yields the best performance.

How can we assess the performance of a model with a given α value? A common solution is to use **cross-validation**. This procedure consists of splitting the dataset into a training set and a test set. We fit the model on the train set, and we test its predictive performance on the test set. By testing the model on a different dataset than the one used for training, we reduce overfitting.

There are many ways to split the initial dataset into two parts like this. One possibility is to remove one sample to form the train set and to put this one sample into the test set. This is called **Leave-one-out cross-validation (LOOCV)**. With N samples, we obtain N sets of train and test sets. The cross-validated performance is the average performance on all these set decompositions.

As we will see later, scikit-learn implements several easy-to-use functions to do cross-validation and grid search. In this recipe, we used a special estimator called `RidgeCV` that implements a cross-validation and grid search procedure that is specific to the ridge regression model. Using this class ensures that the best hyperparameter α is found automatically for us.

There's more...

Here are a few references about least squares:

- ▶ Ordinary least squares on Wikipedia, available at https://en.wikipedia.org/wiki/Ordinary_least_squares
- ▶ Linear least squares on Wikipedia, available at https://en.wikipedia.org/wiki/Linear_least_squares_%28mathematics%29

Here are a few references about cross-validation and grid search:

- ▶ Cross-validation in scikit-learn's documentation, available at http://scikit-learn.org/stable/modules/cross_validation.html

- ▶ Grid search in scikit-learn's documentation, available at http://scikit-learn.org/stable/modules/grid_search.html
- ▶ Cross-validation on Wikipedia, available at https://en.wikipedia.org/wiki/Cross-validation_%28statistics%29

Here are a few references about scikit-learn:

- ▶ scikit-learn basic tutorial available at <http://scikit-learn.org/stable/tutorial/basic/tutorial.html>
- ▶ scikit-learn tutorial given at the SciPy 2017 conference, available at <https://www.youtube.com/watch?v=2kT6QOVsgSg>

Predicting who will survive on the Titanic with logistic regression

In this recipe, we will introduce **logistic regression**, a basic classifier. We will apply these techniques on a **Kaggle** dataset where the goal is to predict survival on the Titanic based on real data (see <http://www.kaggle.com/c/titanic>).



Kaggle (<http://www.kaggle.com/competitions>) hosts machine learning competitions where anyone can download a dataset, train a model, and test the predictions on the website.

How to do it...

1. We import the standard packages:

```
>>> import numpy as np
      import pandas as pd
      import sklearn
      import sklearn.linear_model as lm
      import sklearn.model_selection as ms
      import matplotlib.pyplot as plt
      %matplotlib inline
```

-
2. We load the training and test datasets with pandas:

```
>>> train = pd.read_csv('https://github.com/ipython-books'
                      '/cookbook-2nd-data/blob/master/'
                      'titanic_train.csv?raw=true')
    test = pd.read_csv('https://github.com/ipython-books/'
                      'cookbook-2nd-data/blob/master/'
                      'titanic_test.csv?raw=true')
>>> train[train.columns[[2, 4, 5, 1]]].head()
```

| | Pclass | Sex | Age | Survived |
|---|--------|--------|------|----------|
| 0 | 3 | male | 22.0 | 0 |
| 1 | 1 | female | 38.0 | 1 |
| 2 | 3 | female | 26.0 | 1 |
| 3 | 1 | female | 35.0 | 1 |
| 4 | 3 | male | 35.0 | 0 |

3. Let's keep only a few fields for this example, and also convert the `Sex` field to a binary variable so that it can be handled correctly by NumPy and scikit-learn. Finally, we remove the rows that contain NaN values:

```
>>> data = train[['Age', 'Pclass', 'Survived']]
    # Add a 'Female' column.
    data = data.assign(Female=train['Sex'] == 'female')
    # Reorder the columns.
    data = data[['Female', 'Age', 'Pclass', 'Survived']]
    data = data.dropna()
    data.head()
```

| | Female | Age | Pclass | Survived |
|---|--------|------|--------|----------|
| 0 | False | 22.0 | 3 | 0 |
| 1 | True | 38.0 | 1 | 1 |
| 2 | True | 26.0 | 3 | 1 |
| 3 | True | 35.0 | 1 | 1 |
| 4 | False | 35.0 | 3 | 0 |

4. Now, we convert this DataFrame object to a NumPy array so that we can pass it to scikit-learn:

```
>>> data_np = data.astype(np.int32).values
    X = data_np[:, :-1]
    y = data_np[:, -1]
```

5. Let's have a look at the survival of male and female passengers as a function of their age:

```
>>> # We define a few boolean vectors.
    # The first column is 'Female'.
    female = X[:, 0] == 1

    # The last column is 'Survived'.
    survived = y == 1

    # This vector contains the age of the passengers.
    age = X[:, 1]

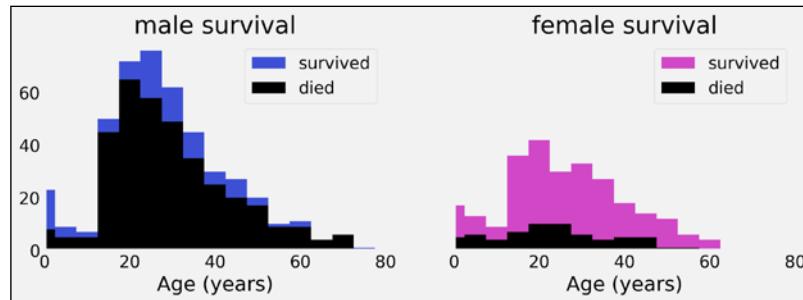
    # We compute a few histograms.
    bins_ = np.arange(0, 81, 5)
    S = {'male': np.histogram(age[survived & ~female],
                               bins=bins_) [0],
          'female': np.histogram(age[survived & female],
                                 bins=bins_) [0]}
    D = {'male': np.histogram(age[~survived & ~female],
                               bins=bins_) [0],
          'female': np.histogram(age[~survived & female],
                                 bins=bins_) [0]}

>>> # We now plot the data.
    bins = bins_[:-1]
    fig, axes = plt.subplots(1, 2, figsize=(10, 3),
                            sharey=True)
    for ax, sex, color in zip(axes, ('male', 'female'),
                             ('#3345d0', '#cc3dc0')):
        ax.bar(bins, S[sex], bottom=D[sex], color=color,
               width=5, label='survived')
```

```

        ax.bar(bins, D[sex], color='k',
                width=5, label='died')
    ax.set_xlim(0, 80)
    ax.set_xlabel("Age (years)")
    ax.set_title(sex + " survival")
    ax.grid(None)
    ax.legend()

```



- Let's try to train a LogisticRegression classifier in order to predict the survival of people based on their gender, age, and class. We first need to create a train and a test dataset:

```

>>> # We split X and y into train and test datasets.
      (X_train, X_test, y_train, y_test) = \
          ms.train_test_split(X, y, test_size=.05)
>>> # We instantiate the classifier.
      logreg = lm.LogisticRegression()

```

- We train the model and we get the predicted values on the test set:

```

>>> logreg.fit(X_train, y_train)
      y_predicted = logreg.predict(X_test)

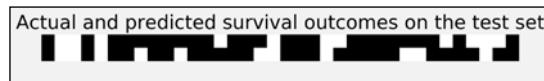
```

The following figure shows the actual and predicted results:

```

>>> fig, ax = plt.subplots(1, 1, figsize=(8, 3))
      ax.imshow(np.vstack((y_test, y_predicted)),
                 interpolation='none', cmap='bone')
      ax.set_axis_off()
      ax.set_title("Actual and predicted survival outcomes "
                  "on the test set")

```



8. To get an estimation of the model's performance, we compute the cross-validation score with the `cross_val_score()` function. This function uses a three-fold stratified cross-validation procedure by default, but this can be changed with the `cv` keyword argument:

```
>>> ms.cross_val_score(logreg, X, y)
array([ 0.78661088,  0.78991597,  0.78059072])
```

This function returns, for each pair of train and test set, a prediction score (we give more details in *How it works...*).

9. The `LogisticRegression` class accepts a `C` hyperparameter as an argument. This parameter quantifies the regularization strength. To find a good value, we can perform a grid search with the generic `GridSearchCV` class. It takes an estimator as input and a dictionary of parameter values. We can also specify the number of cores to use on a multicore processor with the `n_jobs` argument. This new estimator uses cross-validation to select the best parameter:

```
>>> grid = ms.GridSearchCV(
    logreg, {'C': np.logspace(-5, 5, 200)}, n_jobs=4)
grid.fit(X_train, y_train)
grid.best_params_
{'C': 0.042}
```

10. Here is the performance of the best estimator:

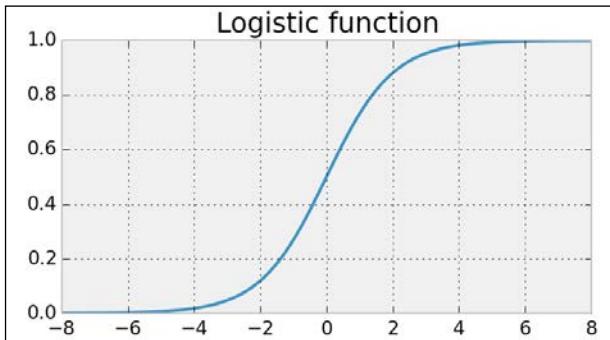
```
>>> ms.cross_val_score(grid.best_estimator_, X, y)
array([ 0.77405858,  0.80672269,  0.78902954])
```

How it works...

Logistic regression is *not* a regression model, it is a classification model. However, it is closely related to linear regression. This model predicts the probability that a binary variable is 1, by applying a **sigmoid function** (more precisely, a logistic function) to a linear combination of the variables. The equation of the sigmoid is:

$$\forall i \in \{1, \dots, N\}, \quad \hat{y}_i = f(\mathbf{x}_i \mathbf{w}) \quad \text{where} \quad f(x) = \frac{1}{1 + \exp(-x)}.$$

The following figure shows a logistic function:



If a binary variable has to be obtained, we can round the value to the closest integer.

The parameter w is obtained with an optimization procedure during the learning step.

There's more...

Here are a few references:

- ▶ Logistic regression in scikit-learn's documentation, available at http://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
- ▶ Logistic regression on Wikipedia, available at https://en.wikipedia.org/wiki/Logistic_regression

See also

- ▶ The *Getting started with scikit-learn* recipe
- ▶ The *Learning to recognize handwritten digits with a K-nearest neighbors classifier* recipe
- ▶ The *Using support vector machines for classification tasks* recipe

Learning to recognize handwritten digits with a K-nearest neighbors classifier

In this recipe, we will see how to recognize handwritten digits with a **K-nearest neighbors (K-NN)** classifier. This classifier is a simple but powerful model, well-adapted to complex, highly nonlinear datasets such as images. We will explain how it works later in this recipe.

How to do it...

1. We import the modules:

```
>>> import numpy as np
     import sklearn
     import sklearn.datasets as ds
     import sklearn.model_selection as ms
     import sklearn.neighbors as nb
     import matplotlib.pyplot as plt
     %matplotlib inline
```

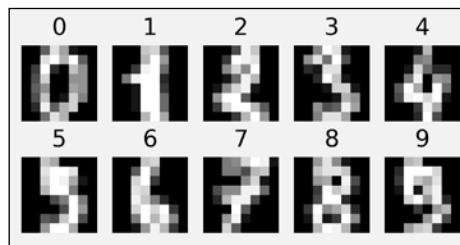
2. Let's load the digits dataset, part of the datasets module of scikit-learn. This dataset contains handwritten digits that have been manually labeled:

```
>>> digits = ds.load_digits()
     X = digits.data
     y = digits.target
     print((X.min(), X.max()))
     print(X.shape)
(0.0, 16.0)
(1797, 64)
```

In the matrix `X`, each row contains $8 \times 8 = 64$ pixels (in grayscale, values between 0 and 16). The row-major ordering is used.

3. Let's display some of the images along with their labels:

```
>>> nrows, ncols = 2, 5
      fig, axes = plt.subplots(nrows, ncols,
                               figsize=(6, 3))
      for i in range(nrows):
          for j in range(ncols):
              # Image index
              k = j + i * ncols
              ax = axes[i, j]
              ax.matshow(digits.images[k, ...],
                         cmap=plt.cm.gray)
              ax.set_axis_off()
              ax.set_title(digits.target[k])
```



4. Now, let's fit a K-NN classifier on the data:

```
>>> (X_train, X_test, y_train, y_test) = \
      ms.train_test_split(X, y, test_size=.25)
>>> knc = nb.KNeighborsClassifier()
>>> knc.fit(X_train, y_train)
```

5. Let's evaluate the score of the trained classifier on the test dataset:

```
>>> knc.score(X_test, y_test)
0.987
```

6. Now, let's see if our classifier can recognize a handwritten digit:

```
>>> # Let's draw a 1.  
one = np.zeros((8, 8))  
one[1:-1, 4] = 16 # The image values are in [0, 16].  
one[2, 3] = 16  
>>> fig, ax = plt.subplots(1, 1, figsize=(2, 2))  
ax.imshow(one, interpolation='none',  
         cmap=plt.cm.gray)  
ax.grid(False)  
ax.set_axis_off()  
ax.set_title("One")
```



```
>>> # We need to pass a (1, D) array.  
knc.predict(one.reshape((1, -1)))  
array([1])
```

Good job!

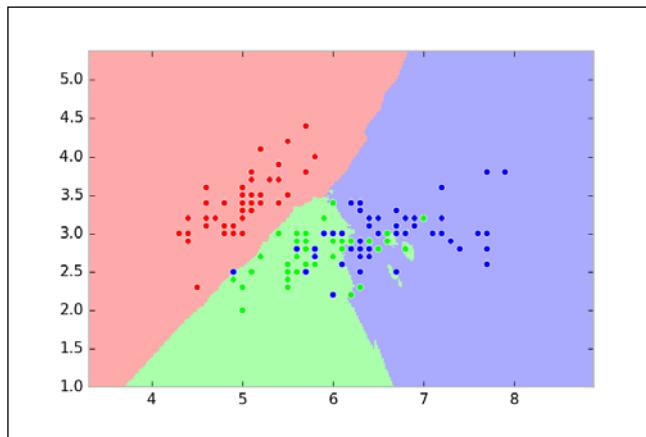
How it works...

This example illustrates how to deal with images in scikit-learn. An image is a 2D (N, M) matrix, which has NM features. This matrix needs to be flattened when composing the data matrix; each row is a full image.

The idea of K-NN is as follows: given a new point in the feature space, find the K closest points from the training set and assign the label of the majority of those points.

The distance is generally the Euclidean distance, but other distances can be used too.

The following plot, obtained from the scikit-learn documentation at <http://scikit-learn.org/stable/modules/neighbors.html>, shows the space partition obtained with a 15-nearest-neighbors classifier on a toy dataset (with three labels):



K-nearest neighbors example

The number K is a hyperparameter of the model. If it is too small, the model will not generalize well (high variance). In particular, it will be highly sensitive to outliers. By contrast, the precision of the model will worsen if K is too large. At the extreme, if K is equal to the total number of points, the model will always predict the exact same value disregarding the input (high bias). There are heuristics to choose this hyperparameter.

It should be noted that no model is learned by a K-NN algorithm; the classifier just stores all data points and compares any new target points with them. This is an example of **instance-based learning**. It is in contrast to other classifiers such as the logistic regression model, which explicitly learns a simple mathematical model on the training data.

The K-NN method works well on complex classification problems that have irregular decision boundaries. However, it might be computationally intensive with large training datasets because a large number of distances have to be computed for testing. Dedicated tree-based data structures such as **K-D trees** or **ball trees** can be used to accelerate the search of nearest neighbors.

The K-NN method can be used for classification, like here, and also for regression problems. The model assigns the average of the target value of the nearest neighbors. In both cases, different weighting strategies can be used.

There's more...

Here are a few references:

- ▶ The K-NN algorithm in scikit-learn's documentation, available at <http://scikit-learn.org/stable/modules/neighbors.html>
- ▶ The K-NN algorithm on Wikipedia, available at https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm
- ▶ Blog post about how to choose the K hyperparameter, available at <http://datasciencegal.wordpress.com/2013/12/27/finding-the-k-in-k-means-clustering/>
- ▶ Instance-based learning on Wikipedia, available at https://en.wikipedia.org/wiki/Instance-based_learning

See also

- ▶ The *Predicting who will survive on the Titanic with logistic regression* recipe
- ▶ The *Using support vector machines for classification tasks* recipe

Learning from text – Naive Bayes for Natural Language Processing

In this recipe, we show how to handle text data with scikit-learn. Working with text requires careful preprocessing and feature extraction. It is also quite common to deal with highly sparse matrices.

We will learn to recognize whether a comment posted during a public discussion is considered insulting to one of the participants. We will use a labeled dataset from Imperium, released during a Kaggle competition (see <http://www.kaggle.com/c/detecting-insults-in-social-commentary>).

How to do it...

1. Let's import our libraries:

```
>>> import numpy as np
      import pandas as pd
      import sklearn
      import sklearn.model_selection as ms
```

```
import sklearn.feature_extraction.text as text
import sklearn.naive_bayes as nb
import matplotlib.pyplot as plt
%matplotlib inline
```

2. Let's open the CSV file with pandas:

```
>>> df = pd.read_csv('https://github.com/ipython-books/'
                     'cookbook-2nd-data/blob/master/'
                     'troll.csv?raw=true')
```

3. Each row is a comment. We will consider two columns: whether the comment is insulting (1) or not (0) and the Unicode-encoded contents of the comment:

```
>>> df[['Insult', 'Comment']].tail()
```

| | Insult | Comment |
|------|---------------|----------------------|
| 3942 | 1 | "you are both mo... |
| 3943 | 0 | "Many toolbars in... |
| 3944 | 0 | @LambeauOrW... |
| 3945 | 0 | "How about Felix... |
| 3946 | 0 | "You're all upset... |

4. Now, we are going to define the feature matrix x and the labels y :

```
>>> y = df['Insult']
```

Obtaining the feature matrix from the text is not trivial. scikit-learn can only work with numerical matrices. So how do we convert text into a matrix of numbers? A classical solution is to first extract a **vocabulary**, a list of words used throughout the corpus. Then, we count, for each sample, the frequency of each word. We end up with a **sparse matrix**, a huge matrix containing mostly zeros. Here, we do this in two lines. We will give more details in *How it works....*



The general rule here is that whenever one of our features is categorical (that is, the presence of a word, a color belonging to a fixed set of n colors, and so on), we should vectorize it by considering one binary feature per item in the class. For example, instead of a feature `color` being red, green, or blue, we should consider three binary features `color_red`, `color_green`, and `color_blue`. We give further references in the *There's more...* section.

```
>>> tf = text.TfidfVectorizer()
      X = tf.fit_transform(df['Comment'])
      print(X.shape)
(3947, 16469)
```

5. There are 3947 comments and 16469 different words. Let's estimate the sparsity of this feature matrix:

```
>>> p = 100 * X.nnz / float(X.shape[0] * X.shape[1])
      print(f"Each sample has ~{p:.2f}% non-zero features.")
Each sample has ~0.15% non-zero features.
```

6. Now, we are going to train a classifier as usual. We first split the data into a train and test set:

```
>>> (X_train, X_test, y_train, y_test) = \
      ms.train_test_split(X, y, test_size=.2)
```

7. We use a **Bernoulli Naive Bayes classifier** with a grid search on the α parameter:

```
>>> bnb = ms.GridSearchCV(
      nb.BernoulliNB(),
      param_grid={'alpha': np.logspace(-2., 2., 50)})
      bnb.fit(X_train, y_train)
```

8. Let's check the performance of this classifier on the test dataset:

```
>>> bnb.score(X_test, y_test)
0.761
```

9. Let's take a look at the words corresponding to the largest coefficients (the words we find frequently in insulting comments):

```
>>> # We first get the words corresponding to each feature
      names = np.asarray(tf.get_feature_names())
      # Next, we display the 50 words with the largest
      # coefficients.
      print(','.join(names[np.argsort(
          bnb.best_estimator_.coef_[0, :])[:-1][:50]]))
you,are,your,to,the,and,of,that,is,in,it,like,have,on,not,for,just
,re,with,be,an,so,this,xa0,all,idiot,what,get,up,go,****,don,stupi
d,no,as,do,can,***,or,but,if,know,who,about,dumb,****,me,*****,*be
cause,back
```

10. Finally, let's test our estimator on a few test sentences:

```
>>> print(bnb.predict(tf.transform([
      "I totally agree with you.",
      "You are so stupid."
    ])))
[0 1]
```

How it works...

scikit-learn implements several utility functions to obtain a sparse feature matrix from text data. A vectorizer such as `CountVectorizer()` extracts a vocabulary from a corpus (`fit()`) and constructs a sparse representation of the corpus based on this vocabulary (`transform()`). Each sample is represented by the vocabulary's word frequencies. The trained instance also contains attributes and methods to map feature indices to the corresponding words (`get_feature_names()`) and conversely (`vocabulary_`).

N-grams can also be extracted: those are pairs or tuples of words occurring successively (`ngram_range` keyword).

The frequency of the words can be weighted in different ways. Here, we have used **tf-idf**, or **term frequency-inverse document frequency**. This quantity reflects how important a word is to a corpus. Frequent words in comments have a high weight except if they appear in most comments (which means that they are common terms, for example, "the" and "and" would be filtered out using this technique).

Naive Bayes algorithms are Bayesian methods based on the naive assumption of independence between the features. This strong assumption drastically simplifies the computations and leads to very fast yet decent classifiers.

There's more...

Here are a few references:

- ▶ Text feature extraction in scikit-learn's documentation, available at http://scikit-learn.org/stable/modules/feature_extraction.html#text-feature-extraction
- ▶ Term frequency-inverse document-frequency on Wikipedia, available at <https://en.wikipedia.org/wiki/Tf-IDF>
- ▶ Vectorizer in scikit-learn's documentation, available at http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.DictVectorizer.html
- ▶ Naive Bayes classifier on Wikipedia, at https://en.wikipedia.org/wiki/Naive_Bayes_classifier
- ▶ Naive Bayes in scikit-learn's documentation, available at http://scikit-learn.org/stable/modules/naive_bayes.html
- ▶ Document classification example in scikit-learn's documentation, at http://scikit-learn.org/stable/datasets/twenty_newsgroups.html

Here are other natural language processing libraries in Python:

- ▶ NLTK available at <http://www.nltk.org>
- ▶ spaCy available at <https://spacy.io/>
- ▶ textacy available at <http://textacy.readthedocs.io/en/stable/>

See also

- ▶ The *Predicting who will survive on the Titanic with logistic regression* recipe
- ▶ The *Learning to recognize handwritten digits with a K-nearest neighbors classifier* recipe
- ▶ The *Using support vector machines for classification tasks* recipe

Using support vector machines for classification tasks

In this recipe, we introduce **support vector machines**, or **SVMs**. These models can be used for classification and regression. Here, we illustrate how to use linear and nonlinear SVMs on a simple classification task. This recipe is inspired by an example in the scikit-learn documentation (see http://scikit-learn.org/stable/auto_examples/svm/plot_svm_nonlinear.html).

How to do it...

1. Let's import the packages:

```
>>> import numpy as np
      import pandas as pd
      import sklearn
      import sklearn.datasets as ds
      import sklearn.model_selection as ms
      import sklearn.svm as svm
      import matplotlib.pyplot as plt
      %matplotlib inline
```

2. We generate 2D points and assign a binary label according to a linear operation on the coordinates:

```
>>> X = np.random.randn(200, 2)
      y = X[:, 0] + X[:, 1] > 1
```

-
3. We now fit a linear **Support Vector Classifier (SVC)**. This classifier tries to separate the two groups of points with a linear boundary (a line here, but more generally a hyperplane):

```
>>> # We train the classifier.  
est = svm.LinearSVC()  
est.fit(X, y)
```

4. We define a function that displays the boundaries and decision function of a trained classifier:

```
>>> # We generate a grid in the square [-3, 3]^2.  
xx, yy = np.meshgrid(np.linspace(-3, 3, 500),  
                     np.linspace(-3, 3, 500))
```

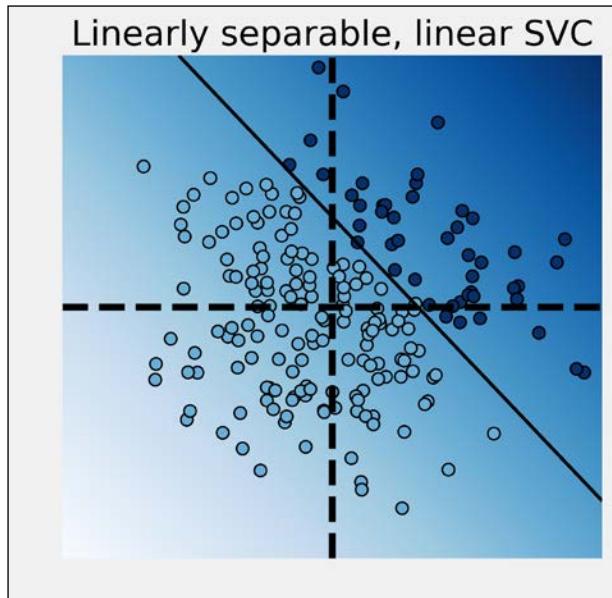
```
# This function takes a SVM estimator as input.
```

```
def plot_decision_function(est, title):  
    # We evaluate the decision function on the grid.  
    Z = est.decision_function(np.c_[xx.ravel(),  
                                    yy.ravel()])  
    Z = Z.reshape(xx.shape)  
    cmap = plt.cm.Blues  
  
    # We display the decision function on the grid.  
    fig, ax = plt.subplots(1, 1, figsize=(5, 5))  
    ax.imshow(Z,  
              extent=(xx.min(), xx.max(),  
                      yy.min(), yy.max()),  
              aspect='auto',  
              origin='lower',  
              cmap=cmap)  
  
    # We display the boundaries.  
    ax.contour(xx, yy, Z, levels=[0],  
               linewidths=2,  
               colors='k')  
  
    # We display the points with their true labels.  
    ax.scatter(X[:, 0], X[:, 1],  
               s=50, c=.5 + .5 * y,  
               edgecolors='k',  
               lw=1, cmap=cmap,  
               vmin=0, vmax=1)
```

```
ax.axhline(0, color='k', ls='--')
ax.axvline(0, color='k', ls='--')
ax.axis([-3, 3, -3, 3])
ax.set_axis_off()
ax.set_title(title)
```

- Let's take a look at the classification results with the linear SVC:

```
>>> ax = plot_decision_function(
    est, "Linearly separable, linear SVC")
```



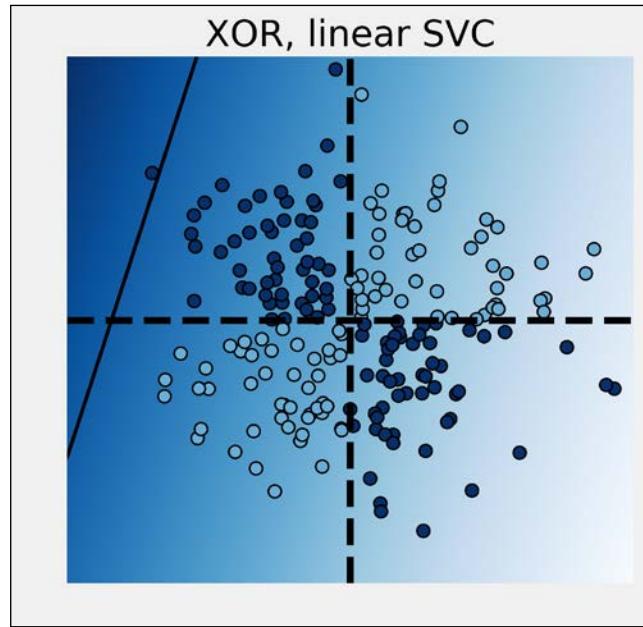
The linear SVC tried to separate the points with a line and it did a pretty good job here.

- We now modify the labels with an XOR function. A point's label is 1 if the coordinates have different signs. This classification is not linearly separable. Therefore, a linear SVC fails completely:

```
>>> y = np.logical_xor(X[:, 0] > 0, X[:, 1] > 0)

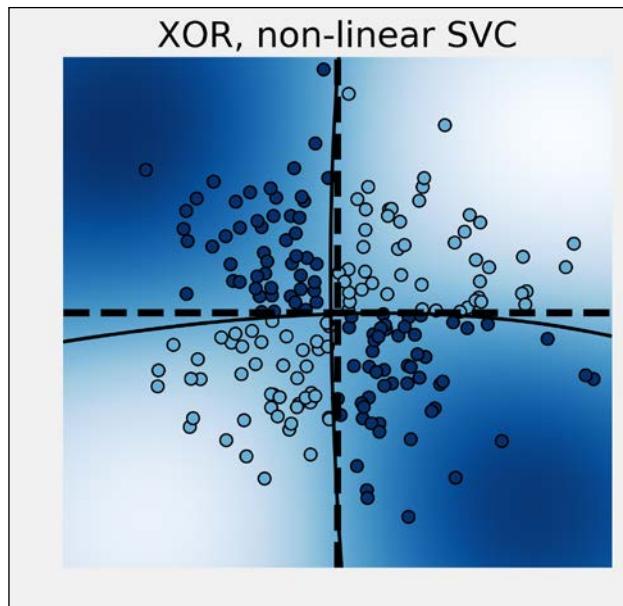
# We train the classifier.
est = ms.GridSearchCV(svm.LinearSVC(),
                      {'C': np.logspace(-3., 3., 10)})
est.fit(X, y)
print("Score: {:.1f}".format(
    ms.cross_val_score(est, X, y).mean()))
```

```
# We plot the decision function.  
ax = plot_decision_function(  
    est, "XOR, linear SVC")  
Score: 0.5
```



7. Fortunately, it is possible to use nonlinear SVCs by using **nonlinear kernels**. Kernels specify a nonlinear transformation of the points into a higher dimensional space. Transformed points in this space are assumed to be more linearly separable. By default, the SVC classifier in scikit-learn uses the **Radial Basis Function (RBF)** kernel:

```
>>> y = np.logical_xor(X[:, 0] > 0, X[:, 1] > 0)  
  
est = ms.GridSearchCV(  
    svm.SVC(), {'C': np.logspace(-3., 3., 10),  
                'gamma': np.logspace(-3., 3., 10)})  
est.fit(X, y)  
print("Score: {:.3f}".format(  
    ms.cross_val_score(est, X, y).mean()))  
  
plot_decision_function(  
    est.best_estimator_, "XOR, non-linear SVC")  
Score: 0.955
```



This time, the nonlinear SVC successfully managed to classify these nonlinearly separable points.

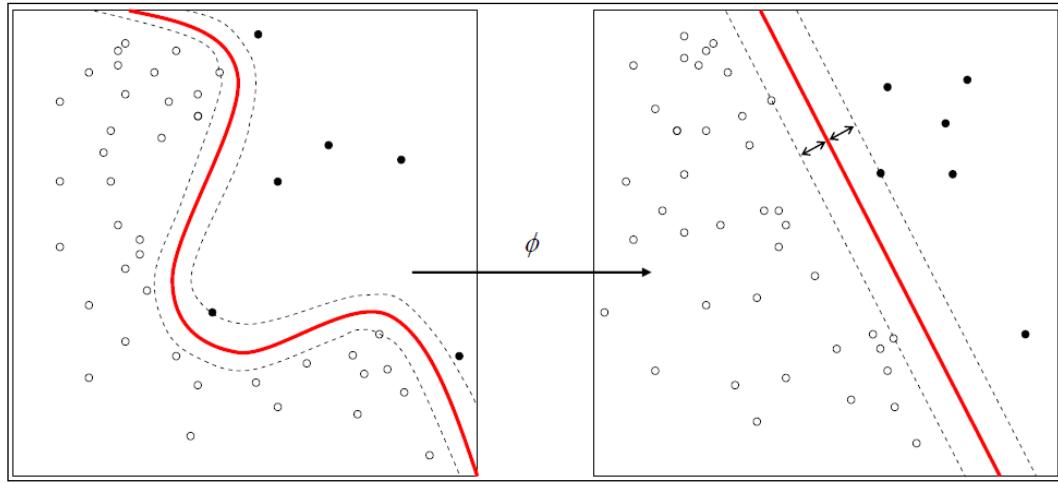
How it works...

A two-class linear SVC tries to find a hyperplane (defined as a linear equation) that best separates the two sets of points (grouped according to their labels). There is also the constraint that this separating hyperplane needs to be as far as possible from the points. This method works best when such a hyperplane exists. Otherwise, this method can fail completely, as we saw in the XOR example. XOR is known as being a nonlinearly separable operation.

The SVM classes in scikit-learn have a C hyperparameter. This hyperparameter trades off misclassification of training examples against simplicity of the decision surface. A low C value makes the decision surface smooth, while a high C value aims at classifying all training examples correctly. This is another example where a hyperparameter quantifies the bias-variance trade-off. This hyperparameter can be chosen with cross-validation and grid search.

The linear SVC can also be extended to multiclass problems. The multiclass SVC is directly implemented in scikit-learn.

The nonlinear SVC works by considering a nonlinear transformation $\phi(x)$ from the original space into a higher dimensional space. This nonlinear transformation can increase the linear separability of the classes. In practice, all dot products are replaced by the $k(x, x') = \phi(x) \cdot \phi(x')$ kernel.



Non-linear SVC

There are several widely-used nonlinear kernels. By default, SVC uses Gaussian radial basis functions:

$$k(\mathbf{x}, \mathbf{x}') = \exp(-\gamma \|\mathbf{x} - \mathbf{x}'\|^2)$$

Here, γ is a hyperparameter of the model that can be chosen with grid search and cross-validation.

The ϕ function does not need to be computed explicitly. This is the kernel trick; it suffices to know the kernel $k(x, x')$. The existence of a function ϕ corresponding to a given kernel $k(x, x')$ is guaranteed by a mathematical theorem in functional analysis (Mercer's theorem).

There's more...

Here are a few references about support vector machines:

- ▶ Exclusive OR on Wikipedia, available at https://en.wikipedia.org/wiki/Exclusive_OR
- ▶ Support vector machines on Wikipedia, available at https://en.wikipedia.org/wiki/Support_vector_machine
- ▶ SVMs in scikit-learn's documentation, available at <http://scikit-learn.org/stable/modules/svm.html>

- ▶ Kernel trick on Wikipedia, available at https://en.wikipedia.org/wiki/Kernel_method
- ▶ Notes about the kernel trick, available at http://www.eric-kim.net/eric-kim-net/posts/1/kernel_trick.html

See also

- ▶ The *Predicting who will survive on the Titanic with logistic regression* recipe
- ▶ The *Learning to recognize handwritten digits with a K-nearest neighbors classifier* recipe

Using a random forest to select important features for regression

Decision trees are frequently used to represent workflows or algorithms. They also form a method for nonparametric supervised learning. A tree mapping observations to target values is learned on a training set and gives the outcomes of new observations.

Random forests are ensembles of decision trees. Multiple decision trees are trained and aggregated to form a model that is more performant than any of the individual trees. This general idea is the purpose of **ensemble learning**.

There are many types of ensemble methods. Random forests are an instance of **bootstrap aggregating**, also called **bagging**, where models are trained on randomly drawn subsets of the training set.

Random forests yield information about the importance of each feature for the classification or regression task. In this recipe, we will find the most influential features of Boston house prices using a classic dataset that contains a range of diverse indicators about the houses' neighborhood.

How to do it...

1. We import the packages:

```
>>> import numpy as np
      import sklearn as sk
      import sklearn.datasets as skd
      import sklearn.ensemble as ske
      import matplotlib.pyplot as plt
      %matplotlib inline
```

2. We load the Boston dataset:

```
>>> data = skd.load_boston()
```

The details of this dataset can be found in `data['DESCR']`. Here is the description of some features:

- ❑ CRIM: Per capita crime rate by town
- ❑ NOX: Nitric oxide concentration (parts per 10 million)
- ❑ RM: Average number of rooms per dwelling
- ❑ AGE: Proportion of owner-occupied units built prior to 1940
- ❑ DIS: Weighted distances to five Boston employment centres
- ❑ PTRATIO: Pupil-teacher ratio by town
- ❑ LSTAT: Percentage of lower status of the population
- ❑ MEDV: Median value of owner-occupied homes in \$1000s

The target value is MEDV.

3. We create a `RandomForestRegressor` model:

```
>>> reg = ske.RandomForestRegressor()
```

4. We get the samples and the target values from this dataset:

```
>>> X = data['data']
      y = data['target']
```

5. Let's fit the model:

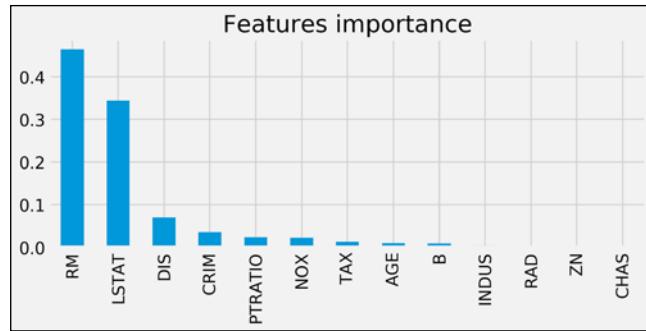
```
>>> reg.fit(X, y)
```

6. The importance of our features can be found in `reg.feature_importances_`. We sort them by decreasing order of importance:

```
>>> fet_ind = np.argsort(reg.feature_importances_)[-1:-1]
      fet_imp = reg.feature_importances_[fet_ind]
```

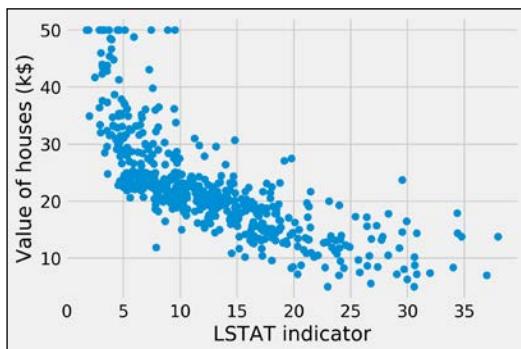
7. Finally, we plot a histogram of the features' importance by creating a pandas Series:

```
>>> fig, ax = plt.subplots(1, 1, figsize=(8, 3))
      labels = data['feature_names'][fet_ind]
      pd.Series(fet_imp, index=labels).plot('bar', ax=ax)
      ax.set_title('Features importance')
```



8. We find that RM (number of rooms per dwelling) and LSTAT (proportion of lower status of the population) are the most important features determining the price of a house. As an illustration, here is a scatter plot of the price as a function of LSTAT:

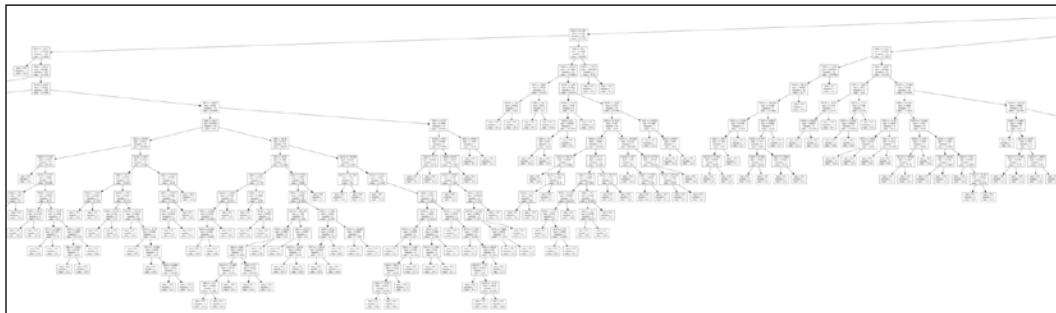
```
>>> fig, ax = plt.subplots(1, 1)
    ax.scatter(X[:, -1], y)
    ax.set_xlabel('LSTAT indicator')
    ax.set_ylabel('Value of houses (k$) ')
```



9. Optionally, we can display a graphic representation of the trees, using the Graphviz package (available at <http://www.graphviz.org>):

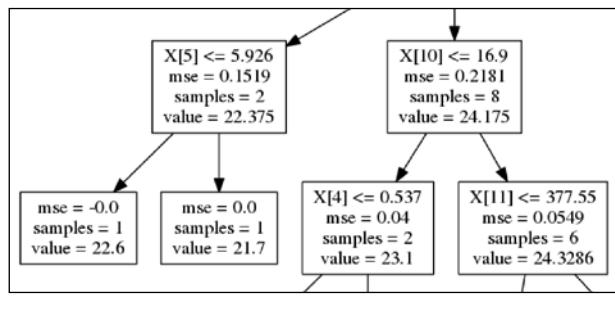
```
>>> from sklearn import tree
    tree.export_graphviz(reg.estimators_[0],
                          'tree.dot')
```

This command exports the first estimator of the random forest into a `.dot` file. We can convert this file into an image with the `dot` command-line executable (available in the `graphviz` package). The following image shows a small part of the image, which is otherwise too large to display:



Tree

The following image shows a close-up of the tree:



Zoom-out tree

The intermediary nodes contain decisions of the form `feature <= value`. Every input point starts from the root and ends up in a leaf node, depending on which conditions are satisfied. The leaf node's value gives the estimated target value for the input point. When using a random forest, an average of the values across trees is computed.

How it works...

Several algorithms can be used to train a decision tree. scikit-learn uses the **CART**, or **Classification and Regression Trees** algorithm. This algorithm constructs binary trees using the feature and threshold that yield the largest information gain at each node. Terminal nodes give the outcomes of input values.

Decision trees are simple to understand. They can also be visualized with **pydot**, a Python package for drawing graphs and trees. This is useful when we want to understand what a tree has learned exactly (**white box model**); the conditions that apply on the observations at each node can be expressed easily with Boolean logic.

However, decision trees may suffer from overfitting, notably when they are too deep, and they might be unstable. Additionally, global convergence toward an optimal model is not guaranteed, particularly when greedy algorithms are used for training. These problems can be mitigated by using ensembles of decision trees, notably random forests.

In a random forest, multiple decision trees are trained on bootstrap samples of the training dataset (randomly sampled with replacement). Predictions are made with the averages of individual trees' predictions (bootstrap aggregating or bagging). Additionally, random subsets of the features are chosen at each node (**random subspace method**). These methods lead to an overall better model than the individual trees.

There's more...

Here are a few references:

- ▶ Ensemble learning in scikit-learn's documentation, available at <http://scikit-learn.org/stable/modules/ensemble.html>
- ▶ API reference of RandomForestRegressor available at <http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html>
- ▶ Random forests on Wikipedia, available at https://en.wikipedia.org/wiki/Random_forest
- ▶ Decision tree learning on Wikipedia, available at https://en.wikipedia.org/wiki/Decision_tree_learning
- ▶ Bootstrap aggregating on Wikipedia, available at https://en.wikipedia.org/wiki/Bootstrap_aggregating
- ▶ Random subspace method on Wikipedia, available at https://en.wikipedia.org/wiki/Random_subspace_method
- ▶ Ensemble learning on Wikipedia, available at https://en.wikipedia.org/wiki/Ensemble_learning

See also

- ▶ The [Using support vector machines for classification tasks](#) recipe

Reducing the dimensionality of a dataset with a principal component analysis

In the previous recipes, we presented *supervised learning* methods; our data points came with discrete or continuous labels, and the algorithms were able to learn the mapping from the points to the labels.

Starting with this recipe, we will present **unsupervised learning** methods. These methods might be helpful prior to running a supervised learning algorithm. They can give a first insight into the data.

Let's assume that our data consists of points x_i without any labels. The goal is to discover some form of hidden structure in this set of points. Frequently, data points have intrinsic low dimensionality: a small number of features suffice to accurately describe the data. However, these features might be hidden among many other features not relevant to the problem. Dimension reduction can help us find these structures. This knowledge can considerably improve the performance of subsequent supervised learning algorithms.

Another useful application of unsupervised learning is **data visualization**; high-dimensional datasets are hard to visualize in 2D or 3D. Projecting the data points on a subspace or submanifold yields more interesting visualizations.

In this recipe, we will illustrate a basic unsupervised linear method, **Principal Component Analysis (PCA)**. This algorithm lets us project data points linearly on a low-dimensional subspace. Along the **principal components**, which are vectors forming a basis of this low-dimensional subspace, the variance of the data points is maximum.

We will use the classic *Iris flower* dataset as an example. This dataset contains the width and length of the petal and sepal of 150 iris flowers. These flowers belong to one of three categories: Iris-setosa, Iris-virginica, and Iris-versicolor. We have access to the category in this dataset (labeled data). However, because we are interested in illustrating an unsupervised learning method, we will only use the data matrix without the labels.

How to do it...

1. We import NumPy, Matplotlib, and scikit-learn:

```
>>> import numpy as np
      import sklearn
      import sklearn.decomposition as dec
```

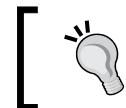
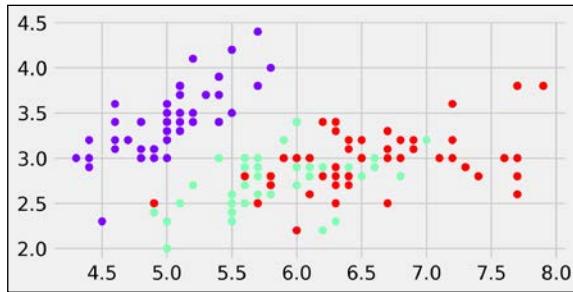
```
import sklearn.datasets as ds
import matplotlib.pyplot as plt
%matplotlib inline
```

2. The Iris flower dataset is available in the datasets module of scikit-learn:

```
>>> iris = ds.load_iris()
      X = iris.data
      y = iris.target
      print(X.shape)
(150, 4)
```

3. Each row contains four parameters related to the morphology of the flower. Let's display the first two dimensions. The color reflects the iris variety of the flower (the label, between 0 and 2):

```
>>> fig, ax = plt.subplots(1, 1, figsize=(6, 3))
      ax.scatter(X[:, 0], X[:, 1], c=y,
                  s=30, cmap=plt.cm.rainbow)
```



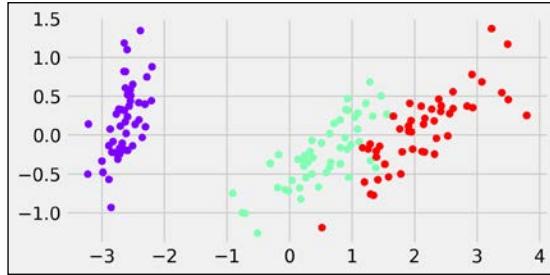
If you're reading the printed version of this book, you might not be able to distinguish the colors. You will find the colored images on the book's website.

4. We now apply PCA on the dataset to get the transformed matrix. This operation can be done in a single line with scikit-learn: we instantiate a PCA model and call the `fit_transform()` method. This function computes the principal components and projects the data on them:

```
>>> X_bis = dec.PCA().fit_transform(X)
```

5. We now display the same dataset, but in a new coordinate system (or equivalently, a linearly transformed version of the initial dataset):

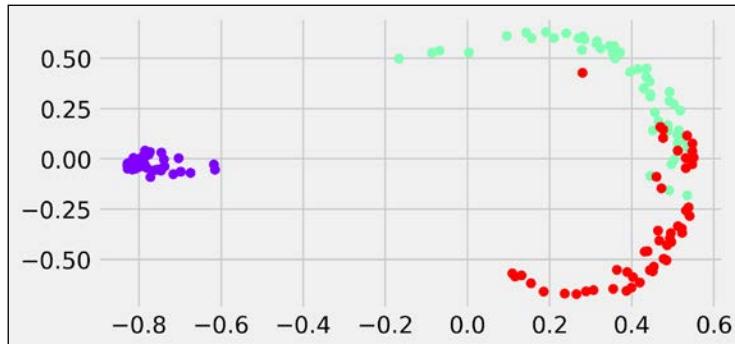
```
>>> fig, ax = plt.subplots(1, 1, figsize=(6, 3))
    ax.scatter(X_bis[:, 0], X_bis[:, 1], c=y,
               s=30, cmap=plt.cm.rainbow)
```



Points belonging to the same classes are now grouped together, even though the PCA estimator did *not* use the labels. The PCA was able to find a projection maximizing the variance, which corresponds here to a projection where the classes are well separated.

6. The `sklearn.decomposition` module contains several variants of the classic PCA estimator: ProbabilisticPCA, SparsePCA, RandomizedPCA, KernelPCA, and others. As an example, let's take a look at KernelPCA, a nonlinear version of PCA:

```
>>> X_ter = dec.KernelPCA(kernel='rbf').fit_transform(X)
    fig, ax = plt.subplots(1, 1, figsize=(6, 3))
    ax.scatter(X_ter[:, 0], X_ter[:, 1], c=y, s=30,
               cmap=plt.cm.rainbow)
```



How it works...

Let's look at the mathematical ideas behind PCA. This method is based on a matrix decomposition called **Singular Value Decomposition (SVD)**:

$$X = U\Sigma V^T$$

Here, X is the (N, D) data matrix, U and V are orthogonal matrices, and Σ is an (N, D) diagonal matrix.

PCA transforms X into X' defined by:

$$X' = X V = U \Sigma$$

The diagonal elements of Σ are the singular values of X . By convention, they are generally sorted in descending order. The columns of U are orthonormal vectors called the **left singular vectors** of X . Therefore, the columns of X' are the **left singular vectors** multiplied by the singular values.

In the end, PCA converts the initial set of observations, which are made of possibly correlated variables, into vectors of linearly uncorrelated variables called **principal components**.

The first new feature (or first component) is a transformation of all original features such that the dispersion (variance) of the data points is the highest in that direction. In the subsequent principal components, the variance is decreasing. In other words, PCA gives us an alternative representation of our data where the new features are sorted according to how much they account for the variability of the points.

There's more...

Here are a few further references:

- ▶ Iris flower dataset on Wikipedia, available at https://en.wikipedia.org/wiki/Iris_flower_data_set
- ▶ PCA on Wikipedia, available at https://en.wikipedia.org/wiki/Principal_component_analysis
- ▶ SVD decomposition on Wikipedia, available at https://en.wikipedia.org/wiki/Singular_value_decomposition
- ▶ Iris dataset example available at http://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html
- ▶ Decompositions in scikit-learn's documentation, available at <http://scikit-learn.org/stable/modules/decomposition.html>

- ▶ Unsupervised learning tutorial with scikit-learn, available at http://scikit-learn.org/dev/tutorial/statistical_inference/unsupervised_learning.html

See also

- ▶ The *Detecting hidden structures in a dataset with clustering* recipe

Detecting hidden structures in a dataset with clustering

A large part of unsupervised learning is devoted to the **clustering** problem. The goal is to group similar points together in a totally unsupervised way. Clustering is a hard problem, as the very definition of **clusters** (or **groups**) is not necessarily well posed. In most datasets, stating that two points should belong to the same cluster may be context-dependent or even subjective.

There are many clustering algorithms. We will see a few of them in this recipe, applied to a toy example.

How to do it...

1. Let's import the libraries:

```
>>> from itertools import permutations
      import numpy as np
      import sklearn
      import sklearn.decomposition as dec
      import sklearn.cluster as clu
      import sklearn.datasets as ds
      import sklearn.model_selection as ms
      import matplotlib.pyplot as plt
      %matplotlib inline
```

2. Let's generate a random dataset with three clusters:

```
>>> X, y = ds.make_blobs(n_samples=200,
                        n_features=2,
                        centers=3,
                        cluster_std=1.5,
                        )
```

3. We need a couple of functions to relabel and display the results of the clustering algorithms:

```
>>> def relabel(cl):
    """Relabel a clustering with three clusters
    to match the original classes."""
    if np.max(cl) != 2:
        return cl
    perms = np.array(list(permutations((0, 1, 2))))
    i = np.argmin([np.sum(np.abs(perm[cl] - y))
                  for perm in perms])
    p = perms[i]
    return p[cl]

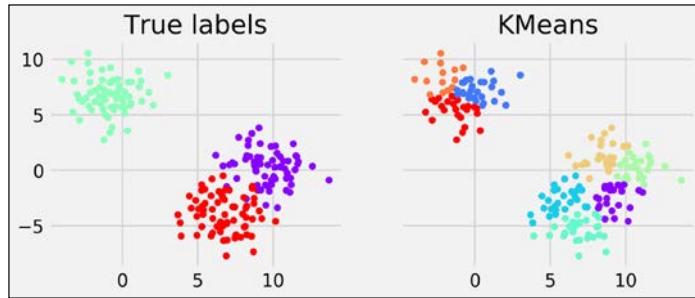
>>> def display_clustering(labels, title):
    """Plot the data points with the cluster
    colors."""

    # We relabel the classes when there are 3 clusters
    labels = relabel(labels)
    fig, axes = plt.subplots(1, 2, figsize=(8, 3),
                           sharey=True)

    # Display the points with the true labels on the
    # left, and with the clustering labels on the
    # right.
    for ax, c, title in zip(
            axes,
            [y, labels],
            ["True labels", title]):
        ax.scatter(X[:, 0], X[:, 1], c=c, s=30,
                   linewidths=0, cmap=plt.cm.rainbow)
        ax.set_title(title)
```

4. Now, we cluster the dataset with the **K-means** algorithm, a classic and simple clustering algorithm:

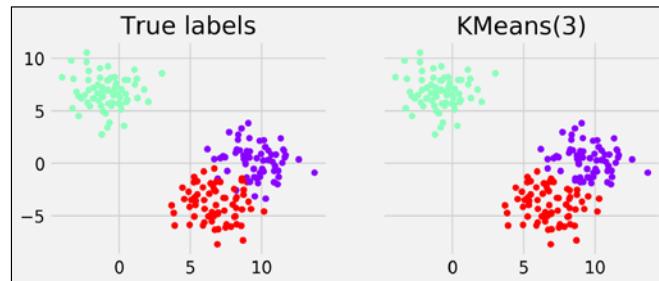
```
>>> km = clu.KMeans()
       km.fit(X)
       display_clustering(km.labels_, "KMeans")
```



 If you're reading the printed version of this book, you might not be able to distinguish the colors. You will find the colored images on the book's website.

5. This algorithm needs to know the number of clusters at initialization time. In general, however, we do not necessarily know the number of clusters in the dataset. Here, let's try with `n_clusters=3` (that's cheating, because we happen to know that there are three clusters):

```
>>> km = clu.KMeans(n_clusters=3)
       km.fit(X)
       display_clustering(km.labels_, "KMeans(3)")
```



6. Let's try a few other clustering algorithms implemented in scikit-learn. The simplicity of the API makes it really easy to try different methods; it is just a matter of changing the name of the class:

```
>>> fig, axes = plt.subplots(2, 3,
                           figsize=(10, 7),
```

```

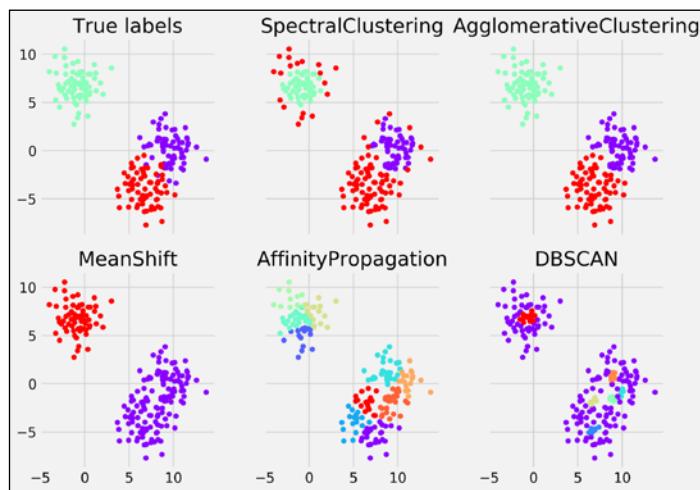
        sharex=True,
        sharey=True)

    axes[0, 0].scatter(X[:, 0], X[:, 1],
                       c=y, s=30,
                       linewidths=0,
                       cmap=plt.cm.rainbow)
    axes[0, 0].set_title("True labels")

    for ax, est in zip(axes.flat[1:], [
        clu.SpectralClustering(3),
        clu.AgglomerativeClustering(3),
        clu.MeanShift(),
        clu.AffinityPropagation(),
        clu.DBSCAN(),
    ]):
        est.fit(X)
        c = relabel(est.labels_)
        ax.scatter(X[:, 0], X[:, 1], c=c, s=30,
                   linewidths=0, cmap=plt.cm.rainbow)
        ax.set_title(est.__class__.__name__)

    # Fix the spacing between subplots.
    fig.tight_layout()

```



The first two algorithms required the number of clusters as input. The next one did not, but it was able to find the right number. The last two failed at finding the correct number of clusters (this is *overclustering*—too many clusters have been found).

How it works...

The K-means clustering algorithm consists of partitioning the data points x_j into K clusters S_i so as to minimize the within-cluster sum of squares:

$$\operatorname{argmin}_{\mathbf{S}} \sum_{i=1}^k \sum_{x_j \in S_i} \|x_j - \mu_i\|_2^2$$

Here, μ_i is the center of the cluster i (average of all points in S_i).

Although it is very hard to solve this problem exactly, approximation algorithms exist. A popular one is **Lloyd's algorithm**. It consists of starting from an initial set of K means μ_i and alternating between two steps:

- ▶ In the *assignment step*, the points are assigned to the cluster associated to the closest mean
- ▶ In the *update step*, the means are recomputed from the last assignments

The algorithm converges to a solution that is not guaranteed to be optimal.

The **expectation-maximization algorithm** can be seen as a probabilistic version of the K-means algorithm. It is implemented in the mixture module of scikit-learn.

The other clustering algorithms used in this recipe are explained in the scikit-learn documentation. There is no clustering algorithm that works uniformly better than all the others, and every algorithm has its strengths and weaknesses. You will find more details in the references in the next section.

There's more...

Here are a few references:

- ▶ The K-means clustering algorithm on Wikipedia, available at https://en.wikipedia.org/wiki/K-means_clustering
- ▶ The expectation-maximization algorithm on Wikipedia, available at https://en.wikipedia.org/wiki/Expectation–maximization_algorithm
- ▶ Clustering in scikit-learn's documentation, available at <http://scikit-learn.org/stable/modules/clustering.html>
- ▶ **t-Distributed Stochastic Neighbor Embedding (t-SNE)** clustering method, at <https://lvdmaaten.github.io/tsne/>

- ▶ scikit-learn t-SNE implementation, at <http://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html>
- ▶ **Uniform Manifold Approximation and Projection (UMAP)**, a t-SNE alternative, at <https://github.com/lmcinnes/umap>

See also

- ▶ The *Reducing the dimensionality of a dataset with a principal component analysis* recipe

9

Numerical Optimization

In this chapter, we will cover the following topics:

- ▶ Finding the root of a mathematical function
- ▶ Minimizing a mathematical function
- ▶ Fitting a function to data with nonlinear least squares
- ▶ Finding the equilibrium state of a physical system by minimizing its potential energy

Introduction

Mathematical optimization is a wide area of applied mathematics. It consists of finding the best solution to a given problem. Many real-world problems can be expressed in an optimization framework. What is the shortest path on the road from point A to point B? What is the best strategy to solve a puzzle? What is the most energy-efficient shape of a car (automotive aerodynamics)? Mathematical optimization is relevant in many domains including engineering, economics, finance, operations research, image processing, data analysis, and others.

Mathematically, an optimization problem consists of finding the maximum or minimum value of a function. We sometimes use the terms **continuous optimization** or **discrete optimization**, according to whether the function variable is real-valued or discrete.

In this chapter, we will focus on numerical methods for solving continuous optimization problems. Many optimization algorithms are implemented in the `scipy.optimize` module. We will come across other instances of optimization problems in several other chapters of this book. For example, we will see discrete optimization problems in *Chapter 14, Graphs, Geometry, and Geographic Information Systems*.

In this introduction, we give a few important definitions and key concepts related to mathematical optimization.

The objective function

We will study methods to find a root or an **extremum** of a real-valued function f called the **objective function**. An extremum is either a maximum or a minimum of a function. It can accept one or several variables, it can be continuous or not, and so on. The more assumptions we have about the function, the easier it can be optimized.



A maximum of f is a minimum of $-f$, so any minimization algorithm can be used to maximize a function by considering the opposite of that function. Therefore, from now on, when we talk about *minimization*, we will really mean *minimization or maximization*.

Convex functions are easier to optimize than non-convex functions, as they satisfy certain useful properties. For example, any local minimum is necessarily a global minimum. The field of **convex optimization** deals with algorithms that are specifically adapted to the optimization of convex functions on convex domains. Convex optimization is an advanced topic, and we can't cover much of it here.

Differentiable functions have gradients, and these gradients can be particularly useful in optimization algorithms. Similarly, **continuous functions** are typically easier to optimize than non-continuous functions.

Also, functions with a single variable are easier to optimize than functions with multiple variables.

The choice of the most adequate optimization algorithm depends on the properties satisfied by the objective function.

Local and global minima

A **minimum** of a function f is a point x_0 such that $f(x) \geq f(x_0)$, for a particular set of points x in E . When this inequality is satisfied on the whole set E , we refer to x_0 as a **global minimum**. When it is only satisfied locally (around the point x_0), we say that x_0 is a **local minimum**. A **maximum** is defined similarly.

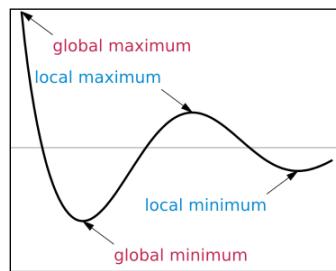
If f is differentiable, an extremum x_0 satisfies:

$$f'(x_0) = 0$$

Therefore, finding the extrema of an objective function is closely related to finding the roots of the derivative. However, a point x_0 satisfying this property is not necessarily an extremum.

It is more difficult to find global minima than to find local minima. In general, when an algorithm finds a local minimum, there is no guarantee that it is also a global minimum. Frequently, an algorithm seeking a global minimum stays stuck in a local minimum. This problem needs to be accounted for, specifically in global minimization algorithms. However, things are simpler with convex functions since these do not have strictly local minima. Moreover, there are many cases where finding a local minimum is good enough (for example, when looking for a good solution to a problem rather than the absolute best solution).

Finally, let's note that a global minimum or maximum does not necessarily exist (the function can go to infinity). In that case, it may be necessary to constrain the space search; this is the subject of **constrained optimization**.



Local and global extrema (from https://en.wikipedia.org/wiki/Maxima_and_minima#/media/File:Extrema_example_original.svg)

Constrained and unconstrained optimization

- ▶ **Unconstrained optimization:** Finding the minimum of a function f on the full set E where f is defined
- ▶ **Constrained optimization:** Finding the minimum of a function f on a subset E' of E ; this set is generally described by equalities and inequalities:

$$\mathbf{x} \in E' \iff \forall i, j, \quad g_i(\mathbf{x}) = c_i, \quad h_j(\mathbf{x}) \leq d_j$$

Here, the g_i and h_j are arbitrary functions defining the constraints.

For example, optimizing the aerodynamic shape of a car requires constraints on parameters such as the volume and mass of the car, the cost of the production process, and others.

Deterministic and stochastic algorithms

Some global optimization algorithms are **deterministic**, others are **stochastic**. Stochastic methods are useful when dealing with the highly irregular and noisy functions that are typical of real-world data. Deterministic algorithms may be stuck in local minima, particularly if there are many non-global local minima. By spending some time exploring the space E , stochastic algorithms may have a chance of finding a global minimum.

References

- ▶ The SciPy lecture notes are an excellent reference on mathematical optimization with SciPy, and they are available at http://scipy-lectures.github.io/advanced/mathematical_optimization/index.html
- ▶ Reference manual of `scipy.optimize` available at <http://docs.scipy.org/doc/scipy/reference/optimize.html>
- ▶ Numerical Analysis on Awesome Math, at <https://github.com/rossant/awesome-math/#numerical-analysis>
- ▶ Overview of mathematical optimization on Wikipedia, available at https://en.wikipedia.org/wiki/Mathematical_optimization
- ▶ Extrema, minima, and maxima on Wikipedia, available at https://en.wikipedia.org/wiki/Maxima_and_minima
- ▶ Convex optimization on Wikipedia, available at https://en.wikipedia.org/wiki/Convex_optimization

Finding the root of a mathematical function

In this short recipe, we will see how to use SciPy to find the root of a simple mathematical function of a single real variable.

How to do it...

1. Let's import NumPy, SciPy, `scipy.optimize`, and `matplotlib`:

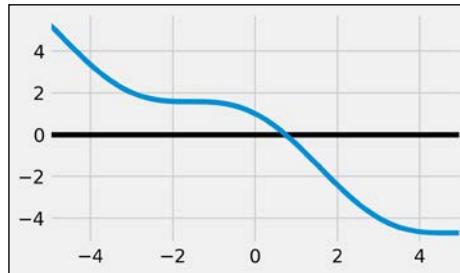
```
>>> import numpy as np
     import scipy as sp
     import scipy.optimize as opt
     import matplotlib.pyplot as plt
     %matplotlib inline
```

2. We define the mathematical function $f(x) = \cos(x) - x$ in Python. We will try to find a root of this function numerically. Here, a root corresponds to a fixed point of the cosine function:

```
>>> def f(x):
     return np.cos(x) - x
```

3. Let's plot this function on the interval $[-5, 5]$ (using 1000 samples):

```
>>> x = np.linspace(-5, 5, 1000)
     y = f(x)
     fig, ax = plt.subplots(1, 1, figsize=(5, 3))
     ax.axhline(0, color='k')
     ax.plot(x, y)
     ax.set_xlim(-5, 5)
```

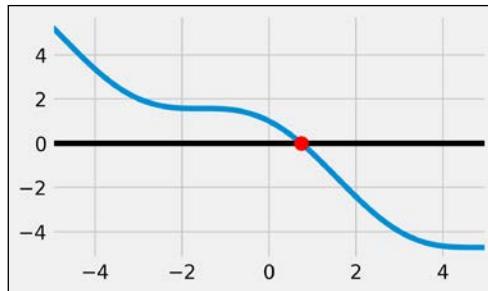


4. We see that this function has a unique root on this interval (this is because the function's sign changes on this interval). The `scipy.optimize` module contains a few root-finding functions that are adapted here. For example, the `bisect()` function implements the **bisection method** (also called the **dichotomy method**). It takes as input the function and the interval to find the root in:

```
>>> opt.bisect(f, -5, 5)
0.739
```

Let's visualize the root on the plot:

```
>>> fig, ax = plt.subplots(1, 1, figsize=(5, 3))
    ax.axhline(0, color='k')
    ax.plot(x, y)
    # The zorder argument is used to put
    # the dot on top of the other elements.
    ax.scatter([_], [0], c='r', s=100,
               zorder=10)
    ax.set_xlim(-5, 5)
```



5. A faster and more powerful method is `brentq()` (**Brent's method**). This algorithm also requires f to be continuous and $f(a)$ and $f(b)$ to have different signs:

```
>>> opt.brentq(f, -5, 5)
0.739
```

The `brentq()` method is faster than `bisect()`. If the conditions are satisfied, it is a good idea to try Brent's method first:

```
>>> %timeit opt.bisect(f, -5, 5)
%timeit opt.brentq(f, -5, 5)
34.5 µs ± 855 ns per loop (mean ± std. dev. of 7 runs,
10000 loops each)
7.71 µs ± 170 ns per loop (mean ± std. dev. of 7 runs,
100000 loops each)
```

How it works...

The bisection method consists of iteratively cutting an interval in half and selecting a subinterval that necessarily contains a root. This method is based on the fact that, if f is a continuous function of a single real variable, $f(a) > 0$, and $f(b) < 0$, then f has a root in (a, b) (**intermediate value theorem**).

Brent's method is a popular hybrid algorithm combining root bracketing, interval bisection, and inverse quadratic interpolation. It is a default method that works in many cases.

Let's also mention **Newton's method**. The idea is to approximate $f(x)$ by its tangent (found with $f'(x)$) and find the intersection with the $y = 0$ line. If f is regular enough, the intersection point will be closer to the actual root of f . By iterating this operation, the algorithm may converge to the sought solution.

There's more...

Here are a few references:

- ▶ Documentation of `scipy.optimize`, available at <http://docs.scipy.org/doc/scipy/reference/optimize.html#root-finding>
- ▶ A course on root finding with SciPy, available at <http://quant-econ.net/scipy.html#roots-and-fixed-points>
- ▶ The bisection method on Wikipedia, available at https://en.wikipedia.org/wiki/Bisection_method
- ▶ The intermediate value theorem on Wikipedia, available at https://en.wikipedia.org/wiki/Intermediate_value_theorem
- ▶ Brent's method on Wikipedia, available at https://en.wikipedia.org/wiki/Brent%27s_method
- ▶ Newton's method on Wikipedia, available at https://en.wikipedia.org/wiki/Newton%27s_method

See also

- ▶ The *Minimizing a mathematical function* recipe

Minimizing a mathematical function

Mathematical optimization deals mainly with the problem of finding a minimum or a maximum of a mathematical function. Frequently, a real-world numerical problem can be expressed as a function minimization problem. Such examples can be found in statistical inference, machine learning, graph theory, and other areas.

Although there are many function minimization algorithms, a generic and universal method does not exist. Therefore, it is important to understand the differences between existing classes of algorithms, their specificities, and their respective use cases. We should also have a good understanding of our problem and our objective function; is it continuous, differentiable, convex, multidimensional, regular, or noisy? Is our problem constrained or unconstrained? Are we seeking local or global minima?

In this recipe, we will demonstrate a few usage examples of the function minimization algorithms implemented in SciPy.

How to do it...

1. We import the libraries:

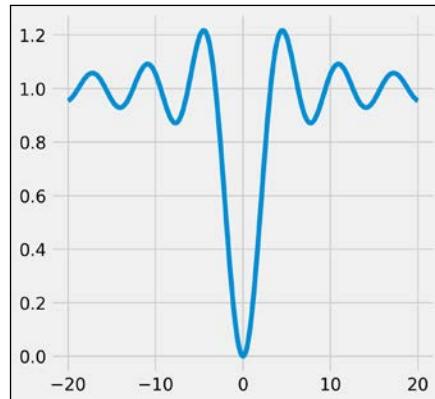
```
>>> import numpy as np  
      import scipy as sp  
      import scipy.optimize as opt  
      import matplotlib.pyplot as plt  
      %matplotlib inline
```

2. First, let's define a simple mathematical function (the opposite of the **cardinal sine**). This function has many local minima but a single global minimum (https://en.wikipedia.org/wiki/Sinc_function):

```
>>> def f(x):  
      return 1 - np.sin(x) / x
```

3. Let's plot this function on the interval $[-20, 20]$ (with 1000 samples):

```
>>> x = np.linspace(-20., 20., 1000)  
y = f(x)  
>>> fig, ax = plt.subplots(1, 1, figsize=(5, 5))  
ax.plot(x, y)
```

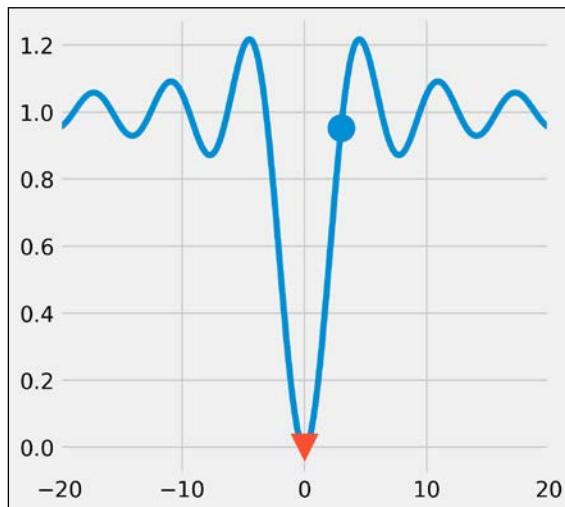


4. The `scipy.optimize` module comes with many function minimization routines. The `minimize()` function offers a unified interface to many algorithms. The **Broyden–Fletcher–Goldfarb–Shanno (BFGS)** algorithm (the default algorithm in `minimize()`) gives good results in general. The `minimize()` function requires an initial point as argument. For scalar univariate functions, we can also use `minimize_scalar()`:

```
>>> x0 = 3
      xmin = opt.minimize(f, x0).x
```

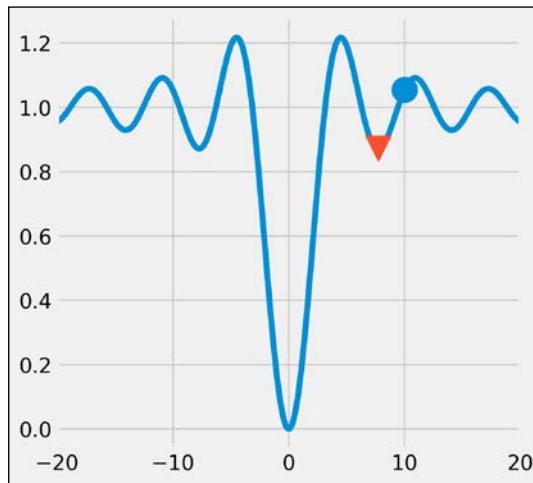
Starting from $x_0 = 3$, the algorithm was able to find the actual global minimum, as shown in the following figure:

```
>>> fig, ax = plt.subplots(1, 1, figsize=(5, 5))
      ax.plot(x, y)
      ax.scatter(x0, f(x0), marker='o', s=300)
      ax.scatter(xmin, f(xmin), marker='v', s=300,
                  zorder=20)
      ax.set_xlim(-20, 20)
```



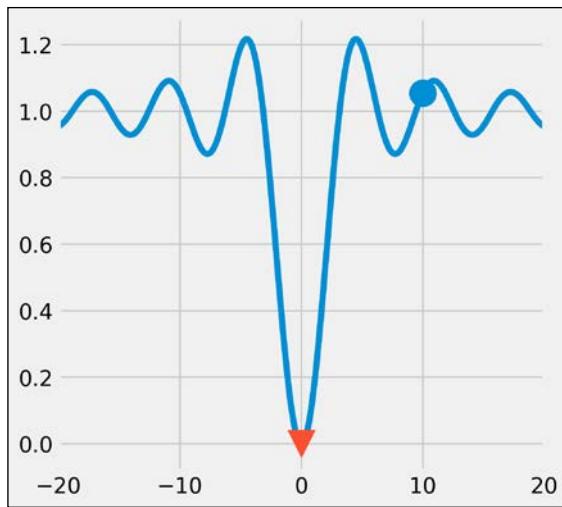
5. Now, if we start from an initial point that is further away from the actual global minimum, the algorithm converges towards a *local* minimum only:

```
>>> x0 = 10
      xmin = opt.minimize(f, x0).x
>>> fig, ax = plt.subplots(1, 1, figsize=(5, 5))
      ax.plot(x, y)
      ax.scatter(x0, f(x0), marker='o', s=300)
      ax.scatter(xmin, f(xmin), marker='v', s=300,
                  zorder=20)
      ax.set_xlim(-20, 20)
```



6. Like most function minimization algorithms, the BFGS algorithm is efficient at finding *local* minima, but not necessarily *global* minima, especially on complicated or noisy objective functions. A general strategy to overcome this problem is to combine such algorithms with an exploratory grid search on the initial points. Another option is to use a different class of algorithms based on heuristics and stochastic methods. An example is the **basin-hopping algorithm**:

```
>>> # We use 1000 iterations.
      xmin = opt.basin hopping(f, x0, 1000).x
>>> fig, ax = plt.subplots(1, 1, figsize=(5, 5))
      ax.plot(x, y)
      ax.scatter(x0, f(x0), marker='o', s=300)
      ax.scatter(xmin, f(xmin), marker='v', s=300,
                  zorder=20)
      ax.set_xlim(-20, 20)
```



This time, the algorithm was able to find the global minimum.

7. Now, let's define a new function, in two dimensions this time, called the **Lévi function**:

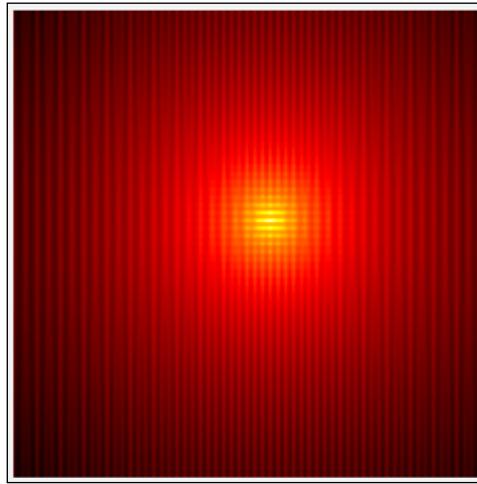
$$f(x, y) = \sin^2(3\pi x) + (x - 1)^2 (1 + \sin^2(3\pi y)) + (y - 1)^2 (1 + \sin^2(2\pi y))$$

This function is very irregular and may be difficult to minimize in general. The expected global minimum is $(1, 1)$. The Lévi function is one of the many **test functions for optimization** that researchers have developed to study and benchmark optimization algorithms (https://en.wikipedia.org/wiki/Test_functions_for_optimization):

```
>>> def g(X):
    # X is a 2*N matrix, each column contains
    # x and y coordinates.
    x, y = X
    return (np.sin(3 * np.pi * x)**2 +
            (x - 1)**2 * (1 + np.sin(3 * np.pi * y)**2) +
            (y - 1)**2 * (1 + np.sin(2 * np.pi * y)**2))
```

8. Let's display this function with `imshow()`, on the square $[-10, 10]^2$:

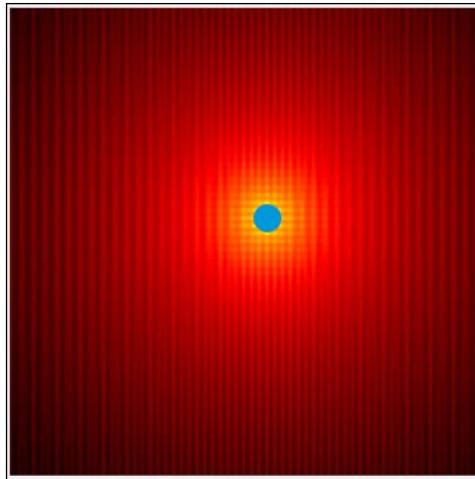
```
>>> n = 500
      k = 10
      X, Y = np.mgrid[-k:k:n * 1j,
                        -k:k:n * 1j]
>>> Z = g(np.vstack((X.ravel(), Y.ravel()))).reshape(n, n)
>>> fig, ax = plt.subplots(1, 1, figsize=(3, 3))
      # We use a logarithmic scale for the color here.
      ax.imshow(np.log(Z), cmap=plt.cm.hot_r,
                extent=(-k, k, -k, k), origin=0)
      ax.set_axis_off()
```



9. The `minimize()` function also works in multiple dimensions:

```
>>> # We use the Powell method.
      x0, y0 = opt.minimize(g, (8, 3),
                            method='Powell').x
      x0, y0
      (1.000, 1.000)

>>> fig, ax = plt.subplots(1, 1, figsize=(3, 3))
      ax.imshow(np.log(Z), cmap=plt.cm.hot_r,
                extent=(-k, k, -k, k), origin=0)
      ax.scatter(x0, y0, s=100)
      ax.set_axis_off()
```



How it works...

Many function minimization algorithms are based on the fundamental idea of **gradient descent**. If a function f is differentiable, then at every point, the opposite of its gradient points to the direction of the greatest decrease rate of the function. By following this direction, we can expect to find a local minimum.

This operation is generally done iteratively, by following the direction of the gradient with a small step. The way this step is computed depends on the optimization method.

Newton's method can also be used in this context of function minimization. The idea is to find a root of f' with Newton's method, thereby making use of the second derivative f'' . In other words, we approximate f with a quadratic function instead of a linear function. In multiple dimensions, this is done by computing the **Hessian** (second derivatives) of f . By performing this operation iteratively, we can expect the algorithm to converge towards a local minimum.

When the computation of the Hessian is too costly, we can compute an approximation of the Hessian. Such methods are called **Quasi-Newton methods**. The BFGS algorithm belongs to this class of algorithms.

These algorithms make use of the objective function's gradient. If we can compute an analytical expression of the gradient, we should provide it to the minimization routine. Otherwise, the algorithm will compute an approximation of the gradient that may not be reliable.

The **basin-hopping algorithm** is a stochastic algorithm that seeks a global minimum by combining random perturbation of the positions and local minimization.

There are many stochastic global optimization methods based on **metaheuristics**. They are generally less well-theoretically grounded than the deterministic optimization algorithms previously described, and convergence is not always guaranteed. However, they may be useful in situations where the objective function is very irregular and noisy, with many local minima. The **Covariance Matrix Adaptation Evolution Strategy (CMA-ES)** algorithm is a metaheuristic that performs well in many situations. It is currently not implemented in SciPy, but there's a Python implementation in one of the references given later.

SciPy's `minimize()` function accepts a `method` keyword argument to specify the minimization algorithm to use. This function returns an object containing the results of the optimization. The `x` attribute is the point reaching the minimum.

There's more...

Here are a few further references:

- ▶ The `scipy.optimize` reference documentation, available at <http://docs.scipy.org/doc/scipy/reference/optimize.html>
- ▶ Documentation of the basin-hopping algorithm, available at <http://scipy.github.io/devdocs/generated/scipy.optimize.basinhopping.html>
- ▶ A lecture on mathematical optimization with SciPy, available at http://scipy-lectures.github.io/advanced/mathematical_optimization/
- ▶ Definition of the gradient on Wikipedia, available at <https://en.wikipedia.org/wiki/Gradient>
- ▶ Newton's method on Wikipedia, available at https://en.wikipedia.org/wiki/Newton%27s_method_in_optimization
- ▶ Quasi-Newton methods on Wikipedia, available at https://en.wikipedia.org/wiki/Quasi-Newton_method
- ▶ Metaheuristics for function minimization on Wikipedia, available at <https://en.wikipedia.org/wiki/Metaheuristic>
- ▶ The CMA-ES algorithm described at <https://en.wikipedia.org/wiki/CMA-ES>
- ▶ A Python implementation of CMA-ES, available at http://www.lri.fr/~hansen/cmaes_inmatlab.html#python

See also

- ▶ The *Finding the root of a mathematical function* recipe

Fitting a function to data with nonlinear least squares

In this recipe, we will show an application of numerical optimization to **nonlinear least squares curve fitting**. The goal is to fit a function, depending on several parameters, to data points. In contrast to the linear least squares method, this function does not have to be linear in those parameters.

We will illustrate this method on artificial data.

How to do it...

1. Let's import the usual libraries:

```
>>> import numpy as np  
      import scipy.optimize as opt  
      import matplotlib.pyplot as plt  
      %matplotlib inline
```

2. We define a logistic function with four parameters:

$$f_{a,b,c,d}(x) = \frac{a}{1 + \exp(-c(x - d))} + b$$

```
>>> def f(x, a, b, c, d):  
      return a / (1. + np.exp(-c * (x - d))) + b
```

3. Let's define four random parameters:

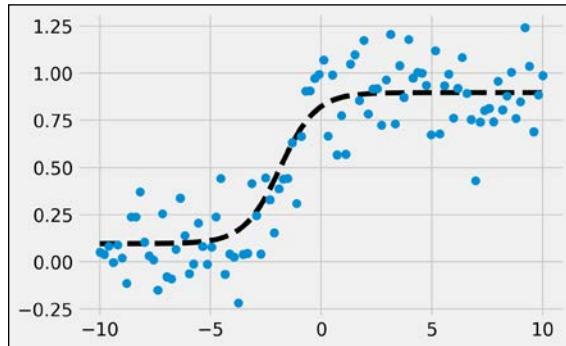
```
>>> a, c = np.random.exponential(size=2)  
      b, d = np.random.randn(2)
```

4. Now, we generate random data points by using the sigmoid function and adding a bit of noise:

```
>>> n = 100  
      x = np.linspace(-10., 10., n)  
      y_model = f(x, a, b, c, d)  
      y = y_model + a * .2 * np.random.randn(n)
```

5. The following is a plot of the data points, with the particular sigmoid used for their generation (in dashed black):

```
>>> fig, ax = plt.subplots(1, 1, figsize=(6, 4))
    ax.plot(x, y_model, '--k')
    ax.plot(x, y, 'o')
```

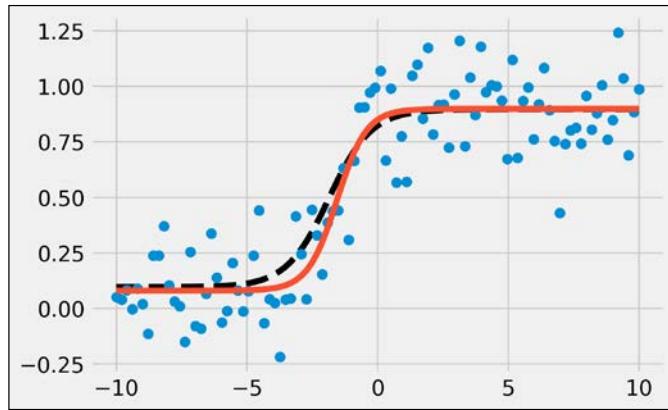


6. We now assume that we only have access to the data points and not the underlying generative function. These points could have been obtained during an experiment. By looking at the data, the points appear to approximately follow a sigmoid, so we may want to try to fit such a curve to the points. That's what **curve fitting** is about. SciPy's `curve_fit()` function allows us to fit a curve defined by an arbitrary Python function to the data:

```
>>> (a_, b_, c_, d_), _ = opt.curve_fit(f, x, y)
```

7. Now, let's take a look at the fitted sigmoid curve:

```
>>> y_fit = f(x, a_, b_, c_, d_)
>>> fig, ax = plt.subplots(1, 1, figsize=(6, 4))
    ax.plot(x, y_model, '--k')
    ax.plot(x, y, 'o')
    ax.plot(x, y_fit, '-')
```



The fitted sigmoid appears to be reasonably close to the original sigmoid used for data generation.

How it works...

In SciPy, nonlinear least squares curve fitting works by minimizing the following cost function:

$$S(\beta) = \sum_{i=1}^n (y_i - f_\beta(x_i))^2$$

Here, β is the vector of parameters (in our example, $\beta = (a, b, c, d)$).

Nonlinear least squares is really similar to linear least squares for linear regression. Whereas the function f is *linear* in the parameters with the linear least squares method, it is *not linear* here. Therefore, the minimization of $S(\beta)$ cannot be done analytically by solving the derivative of S with respect to β . SciPy implements an iterative method called the **Levenberg-Marquardt algorithm** (an extension of the Gauss–Newton algorithm).

Here are further references:

- ▶ Reference documentation of curvefit, available at http://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html
- ▶ Nonlinear least squares on Wikipedia, available at https://en.wikipedia.org/wiki/Non-linear_least_squares
- ▶ The Levenberg-Marquardt algorithm on Wikipedia, available at https://en.wikipedia.org/wiki/Levenberg%E2%80%93Marquardt_algorithm

See also

- ▶ The *Minimizing a mathematical function* recipe

Finding the equilibrium state of a physical system by minimizing its potential energy

In this recipe, we will give an application example of the function minimization algorithms described earlier. We will try to numerically find the equilibrium state of a physical system by minimizing its potential energy.

More specifically, we'll consider a structure made of masses and springs, attached to a vertical wall and subject to gravity. Starting from an initial position, we'll search for the equilibrium configuration where the gravity and elastic forces compensate.

How to do it...

1. Let's import NumPy, SciPy, and matplotlib:

```
>>> import numpy as np
      import scipy.optimize as opt
      import matplotlib.pyplot as plt
      %matplotlib inline
```

2. We define a few constants in the International System of Units:

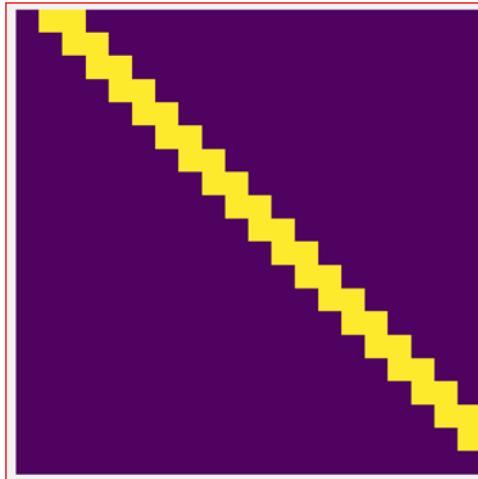
```
>>> g = 9.81 # gravity of Earth
      m = .1 # mass, in kg
      n = 20 # number of masses
      e = .1 # initial distance between the masses
      l = e # relaxed length of the springs
      k = 10000 # spring stiffness
```

3. We define the initial positions of the masses. They are arranged on a two-dimensional grid with two lines and $n/2$ columns:

```
>>> P0 = np.zeros((n, 2))
      P0[:, 0] = np.repeat(e * np.arange(n // 2), 2)
      P0[:, 1] = np.tile((0, -e), n // 2)
```

4. Now, let's define the connectivity matrix between the masses. Coefficient (i, j) is 1 if masses i and j are connected by a spring, 0 otherwise:

```
>>> A = np.eye(n, n, 1) + np.eye(n, n, 2)
      # We display a graphic representation of
      # the matrix.
      f, ax = plt.subplots(1, 1)
      ax.imshow(A)
      ax.set_axis_off()
```



5. We also specify the spring stiffness of each spring. It is l , except for *diagonal* springs where it is $l\sqrt{2}$:

```
>>> L = l * (np.eye(n, n, 1) + np.eye(n, n, 2))
      for i in range(n // 2 - 1):
          L[2 * i + 1, 2 * i + 2] *= np.sqrt(2)
```

6. We get the indices of the spring connections:

```
>>> I, J = np.nonzero(A)
```

7. This `dist()` function computes the distance matrix (the distance between any pair of masses):

```
>>> def dist(P):
      return np.sqrt((P[:, 0] - P[:, 0][:, np.newaxis])**2 +
                     (P[:, 1] - P[:, 1][:, np.newaxis])**2)
```

8. We define a function that displays the system. The springs are colored according to their tension:

```
>>> def show_bar(P):
    fig, ax = plt.subplots(1, 1, figsize=(5, 4))

    # Wall.
    ax.axvline(0, color='k', lw=3)

    # Distance matrix.
    D = dist(P)

    # Get normalized elongation in [-1, 1].
    elong = np.array([D[i, j] - L[i, j]
                     for i, j in zip(I, J)])
    elong_max = np.abs(elong).max()

    # The color depends on the spring tension, which
    # is proportional to the spring elongation.
    colors = np.zeros((len(elong), 4))
    colors[:, -1] = 1 # alpha channel is 1

    # Use two different sequential colormaps for
    # positive and negative elongations, to show
    # compression and extension in different colors.
    if elong_max > 1e-10:
        # We don't use colors if all elongations are
        # zero.
        elong /= elong_max
        pos, neg = elong > 0, elong < 0
        colors[pos] = plt.cm.copper(elong[pos])
        colors[neg] = plt.cm.bone(-elong[neg])

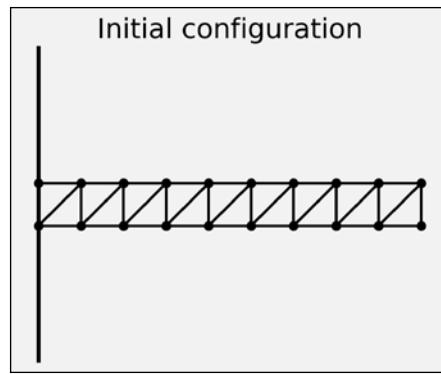
    # We plot the springs.
    for i, j, c in zip(I, J, colors):
        ax.plot(P[[i, j], 0],
                P[[i, j], 1],
                lw=2,
                color=c,
                )

    # We plot the masses.
    ax.plot(P[[I, J], 0], P[[I, J], 1], 'ok',)
```

```
# We configure the axes.  
ax.axis('equal')  
ax.set_xlim(P[:, 0].min() - e / 2,  
            P[:, 0].max() + e / 2)  
ax.set_ylim(P[:, 1].min() - e / 2,  
            P[:, 1].max() + e / 2)  
ax.set_axis_off()  
  
return ax
```

9. Here is the system in its initial configuration:

```
>>> ax = show_bar(P0)  
ax.set_title("Initial configuration")
```



10. To find the equilibrium state, we need to minimize the total potential energy of the system. The following function computes the energy of the system given the positions of the masses. This function is explained in the *How it works...* section of this recipe:

```
>>> def energy(P):  
    # The argument P is a vector (flattened matrix).  
    # We convert it to a matrix here.  
    P = P.reshape((-1, 2))  
    # We compute the distance matrix.  
    D = dist(P)  
    # The potential energy is the sum of the  
    # gravitational and elastic potential energies.  
    return (g * m * P[:, 1].sum() +  
           .5 * (k * A * (D - L)**2).sum())
```

11. Let's compute the potential energy of the initial configuration:

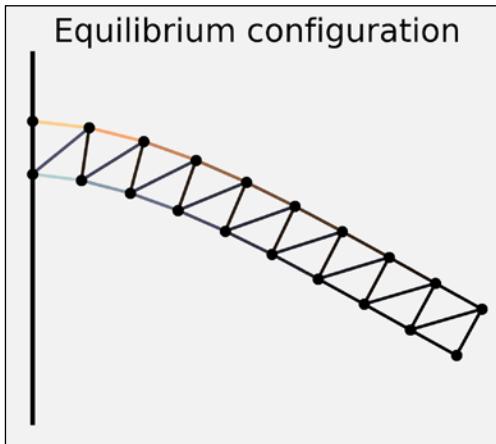
```
>>> energy(P0.ravel())
-0.981
```

12. Now, let's minimize the potential energy with a function minimization method. We need a **constrained optimization algorithm**, because we make the assumption that the first two masses are fixed to the wall. Therefore, their positions cannot change. The **L-BFGS-B algorithm**, a variant of the BFGS algorithm, accepts bound constraints. Here, we force the first two points to stay at their initial positions, whereas there are no constraints on the other points. The `minimize()` function accepts a `bounds` list containing, for each dimension, a pair of `[min, max]` values:

```
>>> bounds = np.c_[P0[:2, :].ravel(),
                  P0[:2, :].ravel()].tolist() + \
                  [[None, None]] * (2 * (n - 2))
>>> P1 = opt.minimize(energy, P0.ravel(),
                      method='L-BFGS-B',
                      bounds=bounds).x.reshape((-1, 2))
```

13. Let's display the stable configuration:

```
>>> ax = show_bar(P1)
ax.set_title("Equilibrium configuration")
```



The springs near the wall are maximally extended (top) or compressed (bottom).

How it works...

This example is conceptually simple. The state of the system is only described by the positions of the masses. If we can write a Python function that returns the total energy of the system, finding the equilibrium is just a matter of minimizing this function. This is the **principle of minimum total potential energy**, due to the second law of thermodynamics.

Here, we give an expression of the total energy of the system. Since we are only interested in the *equilibrium*, we omit any kinetic aspect and we only consider potential energy due to gravity (**gravitational force**) and spring forces (**elastic potential energy**).

Letting U be the total potential energy of the system, U can be expressed as the sum of the gravitational potential energies of the masses and the elastic potential energies of the springs. Therefore:

$$U = \sum_{i=1}^n mgy_i + \frac{1}{2} \sum_{i,j=1}^n ka_{ij} (||\mathbf{p}_i - \mathbf{p}_j|| - l_{ij})^2$$

Here:

- ▶ m is the mass
- ▶ g is the gravity of Earth
- ▶ k is the stiffness of the springs
- ▶ $p_i = (x_i, y_i)$ is the position of mass i ,
- ▶ a_{ij} is 1 if masses i and j are attached by a spring, 0 otherwise
- ▶ l_{ij} is the relaxed length of spring (i, j) , or 0 if masses i and j are not attached

The `energy()` function implements this formula using vectorized computations on NumPy arrays.

There's more...

The following references contain details about the physics behind this formula:

- ▶ Potential energy on Wikipedia, available at https://en.wikipedia.org/wiki/Potential_energy
- ▶ Elastic potential energy on Wikipedia, available at https://en.wikipedia.org/wiki/Elastic_potential_energy

- ▶ Hooke's law, which is the linear approximation of the springs' response, described at https://en.wikipedia.org/wiki/Hooke%27s_law
- ▶ The principle of minimum energy on Wikipedia, available at https://en.wikipedia.org/wiki/Minimum_total_potential_energy_principle
Here is a reference about the optimization algorithm:
- ▶ The L-BFGS-B algorithm on Wikipedia, available at https://en.wikipedia.org/wiki/Limited-memory_BFGS#L-BFGS-B

See also

- ▶ The *Minimizing a mathematical function* recipe

10

Signal Processing

In this chapter, we will cover the following topics:

- ▶ Analyzing the frequency components of a signal with a Fast Fourier Transform
- ▶ Applying a linear filter to a digital signal
- ▶ Computing the autocorrelation of a time series

Introduction

Signals are mathematical functions that describe the variation of a quantity across time or space. Time-dependent signals are often called **time series**. Examples of time series include share prices, which are typically presented as successive points in time spaced at uniform time intervals. In physics or biology, experimental devices record the evolution of variables such as electromagnetic waves or biological processes.

In signal processing, a general objective consists of extracting meaningful and relevant information from raw, noisy measurements. Signal processing topics include signal acquisition, transformation, compression, filtering, and feature extraction, among others. When dealing with a complex dataset, it can be beneficial to clean it before applying more advanced mathematical analysis methods (such as machine learning, for instance).

In this concise chapter, we will illustrate and explain the main foundations of signal processing. In the next chapter, *Chapter 11, Image and Audio Processing*, we will see particular signal processing methods adapted to images and sounds.

First, we will give some important definitions in this introduction.

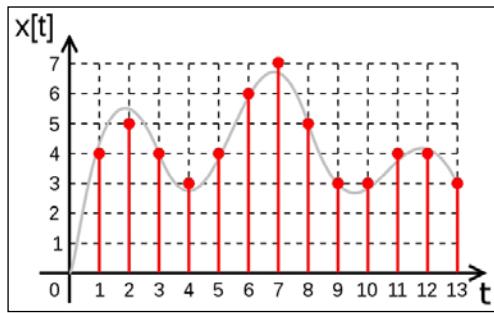
Analog and digital signals

Signals can be time-dependent or space-dependent. In this chapter, we will focus on time-dependent signals.

Let $x(t)$ be a time-varying signal. We say that:

- ▶ This signal is **analog** if t is a continuous variable and $x(t)$ is a real number
- ▶ This signal is **digital** if t is a discrete variable (**discrete-time signal**) and $x(t)$ can only take a finite number of values (**quantified signal**)

The following figure shows the difference between an analog signal (the continuous curve) and a digital signal (dots):



Analog and digital signals (https://en.wikipedia.org/wiki/Digital_signal#/media/File:Digital.signal.discret.svg)

Analog signals are found in mathematics and in most physical systems such as electric circuits. Yet, computers being discrete machines, they can only understand digital signals. This is why computational science especially deals with digital signals.

A digital signal recorded by an experimental device is typically characterized by two important quantities:

- ▶ **The sampling rate:** The number of values (or samples) recorded every second (in Hertz)
- ▶ **The resolution:** The precision of the quantization, usually in bits per sample (also known as **bit depth**)

Digital signals with high sampling rates and bit depths are more accurate, but they require more memory and processing power. These two parameters are limited by the experimental devices that record the signals.

The Nyquist–Shannon sampling theorem

Let's consider a continuous (analog) time-varying signal $x(t)$. We record this physical signal with an experimental device, and we obtain a digital signal with a sampling rate of f_s . The original analog signal has an infinite precision, whereas the recorded signal has a finite precision. Therefore, we expect to lose information in the analog-to-digital process.

The **Nyquist–Shannon sampling theorem** states that under certain conditions on the analog signal and the sampling rate, it is possible not to lose any information in the process. In other words, under these conditions, we can recover the exact original continuous signal from the sampled digital signal. For more details, refer to https://en.wikipedia.org/wiki/Nyquist%E2%80%93Shannon_sampling_theorem.

Let's define these conditions. The **Fourier transform** $\hat{x}(f)$ of $x(t)$ is defined by:

$$\hat{x}(f) = \int_{-\infty}^{+\infty} x(t)e^{-2i\pi ft} dt$$

Here, the Fourier transform is a representation of a time-dependent signal in the frequency domain. The **Nyquist criterion** states that:

there exists $B < f_s/2$ such that, for all $|f| > B$, $\hat{x}(f) = 0$.

In other words, the signal must be **bandlimited**, meaning that it must not contain any frequency higher than a certain cutoff frequency B . Additionally, the sampling rate f_s needs to be at least twice as large as this frequency B . Here are a couple of definitions:

- ▶ The **Nyquist rate** is $2B$. For a given bandlimited analog signal, it is the minimal sampling rate required to sample the signal without loss.
- ▶ The **Nyquist frequency** is $f_s/2$. For a given sampling rate, it is the maximal frequency that the signal can contain in order to be sampled without loss.

Under these conditions, we can theoretically reconstruct the original analog signal from the sampled digital signal.

Compressed sensing

Compressed sensing is a recent and important approach to signal processing. It acknowledges that many real-world signals are intrinsically low dimensional. For example, speech signals have a very specific structure depending on the general physical constraints of the human vocal tract.

Even if a speech signal has many frequencies in the Fourier domain, it may be well approximated by a **sparse decomposition** on an adequate basis (dictionary). By definition, a decomposition is sparse if most of the coefficients are zero. If the dictionary is chosen well, every signal is a combination of a small number of the basis signals.

This dictionary contains elementary signals that are specific to the signals considered in a given problem. This is different from the Fourier transform that decomposes a signal on a universal basis of sine functions. In other words, with sparse representations, the Nyquist condition can be circumvented. We can precisely reconstruct a continuous signal from a sparse representation containing fewer samples than what the Nyquist condition requires.

Sparse decompositions can be found with sophisticated algorithms. In particular, these problems may be turned into convex optimization problems that can be tackled with specific numerical optimization methods.

Compressed sensing has many applications in signal compression, image processing, computer vision, biomedical imaging, and many other scientific and engineering areas.

Here are further references about compressed sensing:

- ▶ https://en.wikipedia.org/wiki/Compressed_sensing
- ▶ https://en.wikipedia.org/wiki/Sparse_approximation
- ▶ Compressed sensing in Python at <http://www.pyrunner.com/weblog/2016/05/26/compressed-sensing-python/>

References

Here are a few references:

- ▶ *Understanding Digital Signal Processing*, Richard G. Lyons, Pearson Education, (2010).
- ▶ For good coverage of compressed sensing, refer to the book *A Wavelet Tour of Signal Processing: The Sparse Way*, Mallat Stéphane, Academic Press, (2008).
- ▶ Harmonic Analysis Lectures on Awesome Math, at <https://github.com/rossant/awesome-math/#harmonic-analysis>
- ▶ The book *Python for Signal Processing*, Jose Unpingco, Springer International Publishing contains many more details than what we can cover in this chapter. The code is available as Jupyter notebooks on GitHub (<http://python-for-signal-processing.blogspot.com>).

- ▶ *Digital Signal Processing* on WikiBooks available at http://en.wikibooks.org/wiki/Digital_Signal_Processing.
- ▶ Numerical Tours in Python, available at <http://www.numerical-tours.com/python/>

Analyzing the frequency components of a signal with a Fast Fourier Transform

In this recipe, we will show how to use a **Fast Fourier Transform (FFT)** to compute the spectral density of a signal. The spectrum represents the energy associated to frequencies (encoding periodic fluctuations in a signal). It is obtained with a Fourier transform, which is a frequency representation of a time-dependent signal. A signal can be transformed back and forth from one representation to the other with no information loss.

In this recipe, we will illustrate several aspects of the Fourier transform. We will apply this tool to weather data spanning 20 years in France obtained from the US National Climatic Data Center.

How to do it...

1. Let's import the packages, including `scipy.fftpack`, which includes many FFT-related routines:

```
>>> import datetime
      import numpy as np
      import scipy as sp
      import scipy.fftpack
      import pandas as pd
      import matplotlib.pyplot as plt
      %matplotlib inline
```

2. We import the data from the CSV file (it has been obtained at <http://www.ncdc.noaa.gov/cdo-web/datasets#GHCND>). The number -9999 is used for N/A values. The pandas can easily handle this. In addition, we tell pandas to parse dates contained in the DATE column:

```
>>> df0 = pd.read_csv('https://github.com/ipython-books/'
                      'cookbook-2nd-data/blob/master/'
                      'weather.csv?raw=true',
                      na_values=(-9999),
```

```

        parse_dates=['DATE'])
>>> df = df0[df0['DATE'] >= '19940101']
>>> df.head()

```

| | STATION | DATE | PRCP | TMAX | TMIN |
|-----|-----------------|------------|------|-------|------|
| 365 | GHCND:FR0130... | 1994-01-01 | 0.0 | 104.0 | 72.0 |
| 366 | GHCND:FR0130... | 1994-01-02 | 4.0 | 128.0 | 49.0 |
| 367 | GHCND:FR0130... | 1994-01-03 | 0.0 | 160.0 | 87.0 |
| 368 | GHCND:FR0130... | 1994-01-04 | 0.0 | 118.0 | 83.0 |
| 369 | GHCND:FR0130... | 1994-01-05 | 34.0 | 133.0 | 55.0 |

3. Each row contains the precipitation and extreme temperatures recorded each day by one weather station in France. For every date in the calendar, we want to get a single average temperature for the whole country. The `groupby()` method provided by pandas lets us do this easily. We also remove any N/A value with `dropna()`:

```

>>> df_avg = df.dropna().groupby('DATE').mean()
>>> df_avg.head()

```

| | PRCP | TMAX | TMIN |
|------------|------------|------------|-----------|
| DATE | | | |
| 1994-01-01 | 178.666667 | 127.388889 | 70.333333 |
| 1994-01-02 | 122.000000 | 152.421053 | 81.736842 |
| 1994-01-03 | 277.333333 | 157.666667 | 95.555556 |
| 1994-01-04 | 177.105263 | 142.210526 | 95.684211 |
| 1994-01-05 | 117.944444 | 130.222222 | 75.444444 |

4. Now, we get the list of dates and the list of corresponding temperatures. The unit is in tenths of a degree, and we get the average value between the minimal and maximal temperature, which explains why we divide by 20.

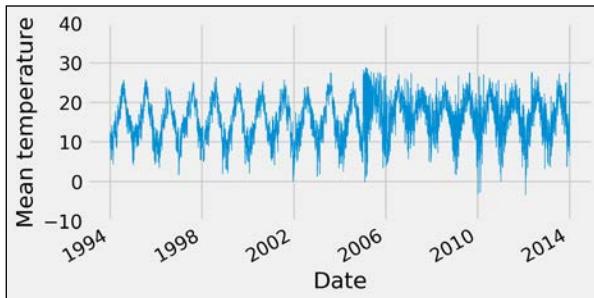
```

>>> date = df_avg.index.to_datetime()
temp = (df_avg['TMAX'] + df_avg['TMIN']) / 20.
N = len(temp)

```

5. Let's take a look at the evolution of the temperature:

```
>>> fig, ax = plt.subplots(1, 1, figsize=(6, 3))
    temp.plot(ax=ax, lw=.5)
    ax.set_xlim(-10, 40)
    ax.set_xlabel('Date')
    ax.set_ylabel('Mean temperature')
```



6. We now compute the Fourier transform and the spectral density of the signal. The first step is to compute the FFT of the signal using the `fft()` function:

```
>>> temp_fft = sp.fftpack.fft(temp)
```

7. Once the FFT has been obtained, we need to take the square of its absolute value in order to get the **Power Spectral Density (PSD)**:

```
>>> temp_psd = np.abs(temp_fft) ** 2
```

8. The next step is to get the frequencies corresponding to the values of the PSD. The `fftfreq()` utility function does just that. It takes the length of the PSD vector as input as well as the frequency unit. Here, we choose an annual unit: a frequency of 1 corresponds to 1 year (365 days). We provide $1/365$ because the original unit is in days:

```
>>> fftfreq = sp.fftpack.fftfreq(len(temp_psd), 1. / 365)
```

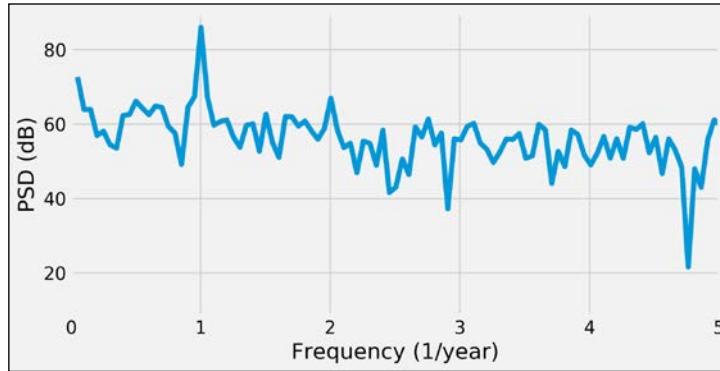
9. The `fftfreq()` function returns positive and negative frequencies. We are only interested in positive frequencies here, as we have a real signal:

```
>>> i = fftfreq > 0
```

10. We now plot the PSD of our signal, as a function of the frequency (in unit of 1/year).

We choose a logarithmic scale for the y axis (**decibels**):

```
>>> fig, ax = plt.subplots(1, 1, figsize=(8, 4))
    ax.plot(fftreq[i], 10 * np.log10(temp_psd[i]))
    ax.set_xlim(0, 5)
    ax.set_xlabel('Frequency (1/year)')
    ax.set_ylabel('PSD (dB)')
```



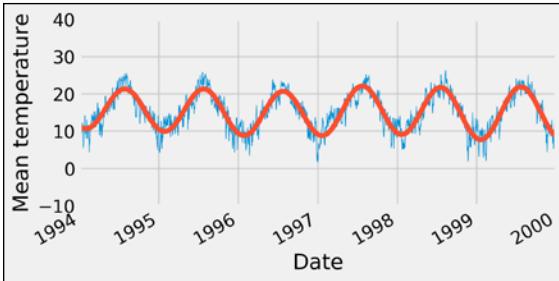
Because the fundamental frequency of the signal is the yearly variation of the temperature, we observe a peak for $f=1$.

11. Now, we cut out frequencies higher than the fundamental frequency:

```
>>> temp_fft_bis = temp_fft.copy()
    temp_fft_bis[np.abs(fftreq) > 1.1] = 0
```

12. Next, we perform an **inverse FFT** to convert the modified Fourier transform back to the temporal domain. This way, we recover a signal that mainly contains the fundamental frequency, as shown in the following figure:

```
>>> temp_slow = np.real(sp.fftpack.ifft(temp_fft_bis))
>>> fig, ax = plt.subplots(1, 1, figsize=(6, 3))
    temp.plot(ax=ax, lw=.5)
    ax.plot_date(date, temp_slow, '-')
    ax.set_xlim(datetime.date(1994, 1, 1),
                datetime.date(2000, 1, 1))
    ax.set_ylim(-10, 40)
    ax.set_xlabel('Date')
    ax.set_ylabel('Mean temperature')
```



We get a smoothed version of the signal, because the fast variations have been lost when we have removed the high frequencies in the Fourier transform.

How it works...

Broadly speaking, the Fourier transform is an alternative representation of a signal as a superposition of periodic components. It is an important mathematical result that any well-behaved function can be represented under this form. Whereas a time-varying signal is most naturally considered as a function of time, the Fourier transform represents it as a function of the frequency. A magnitude and a phase, which are both encoded in a single complex number, are associated to each frequency.

The discrete Fourier transform

Let's consider a digital signal x represented by a vector (x_0, \dots, x_{N-1}) . We assume that this signal is regularly sampled. The **Discrete Fourier Transform (DFT)** of x is $X = (X_0, \dots, X_{N-1})$ defined as:

$$\forall k \in \{0, \dots, N-1\}, \quad X_k = \sum_{n=0}^{N-1} x_n e^{-2i\pi kn/N}.$$

The DFT can be computed efficiently with the FFT, an algorithm that exploits symmetries and redundancies in this definition to considerably speed up the computation. The complexity of the FFT is $O(N \log N)$ instead of $O(N^2)$ for the naive DFT. The FFT is one of the most important algorithms of the digital universe.

Here is an intuitive explanation of what the DFT describes. Instead of representing our signal on a real line, let's represent it on a circle. We can play the whole signal by making 1, 2, or any number k of laps on the circle. Therefore, when k is fixed, we represent each value x_n of the signal with an angle $2\pi kn/N$ and a distance from the original equal to x_n .

In the following figure, the signal is a sine wave at the frequency $f = 3\text{ Hz}$. The points of this signal are in blue, positioned at an angle $2\pi kn/N$. Their algebraic sum in the complex plane is in red. These vectors represent the different coefficients of the signal's DFT.

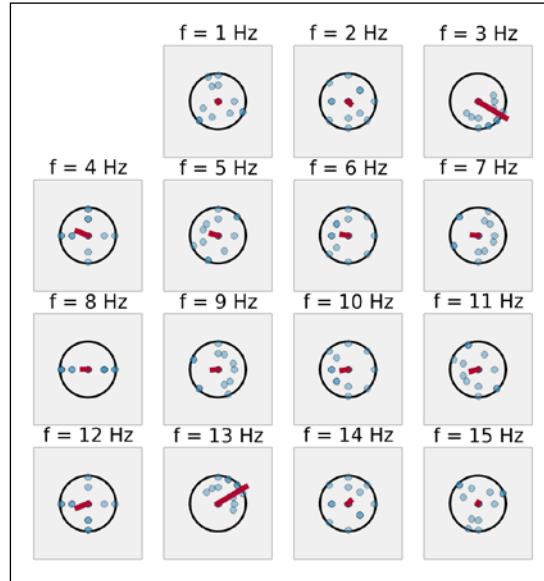
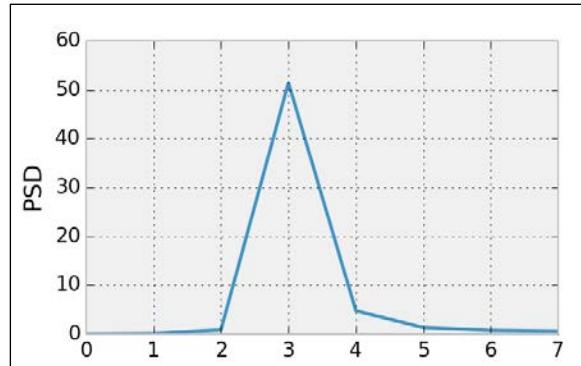


Illustration of the DFT.

The next figure represents the previous signal's PSD:



The PSD of the signal in the previous example.

Inverse Fourier transform

By considering all possible frequencies, we have an exact representation of our digital signal in the frequency domain. We can recover the initial signal with an **Inverse Fast Fourier Transform** that computes an **Inverse Discrete Fourier Transform**. The formula is very similar to the DFT:

$$\forall k \in \{0, \dots, N - 1\}, \quad x_k = \frac{1}{N} \sum_{n=0}^{N-1} X_n e^{2i\pi kn/N}.$$

The DFT is useful when periodic patterns are to be found. However, generally speaking, the Fourier transform cannot detect *transient* changes at specific frequencies. Local spectral methods are required, such as the **wavelet transform**.

There's more...

The following links contain more details about Fourier transforms:

- ▶ Introduction to the FFT with SciPy, available at <http://scipy-lectures.github.io/intro/scipy.html#fast-fourier-transforms-scipy-fftpack>
- ▶ Reference documentation for the `fftpack` in SciPy, available at <http://docs.scipy.org/doc/scipy/reference/fftpack.html>
- ▶ Fourier transform on Wikipedia, available at https://en.wikipedia.org/wiki/Fourier_transform
- ▶ DFT on Wikipedia, available at https://en.wikipedia.org/wiki/Discrete_Fourier_transform
- ▶ FFT on Wikipedia, available at https://en.wikipedia.org/wiki/Fast_Fourier_transform
- ▶ Decibel on Wikipedia, available at <https://en.wikipedia.org/wiki/Decibel>

See also

- ▶ The *Applying a linear filter to a digital signal* recipe
- ▶ The *Computing the autocorrelation of a time series* recipe

Applying a linear filter to a digital signal

Linear filters play a fundamental role in signal processing. With a linear filter, one can extract meaningful information from a digital signal.

In this recipe, we will show two examples using stock market data (the NASDAQ stock exchange). First, we will smooth out a very noisy signal with a low-pass filter to extract its slow variations. We will also apply a high-pass filter to the original time series to extract the fast variations. These are just two common examples among a wide variety of applications of linear filters.

How to do it...

1. Let's import the packages:

```
>>> import numpy as np
      import scipy as sp
      import scipy.signal as sg
      import pandas as pd
      import matplotlib.pyplot as plt
      %matplotlib inline
```

2. We load the NASDAQ data (obtained from <https://finance.yahoo.com/quote/%5EIXIC/history?period1=631148400&period2=1510786800&interval=1d&filter=history&frequency=1d>) with pandas:

```
>>> nasdaq_df = pd.read_csv(
      'https://github.com/ipython-books/'
      'cookbook-2nd-data/blob/master/'
      'nasdaq.csv?raw=true',
      index_col='Date',
      parse_dates=['Date'])
>>> nasdaq_df.head()
```

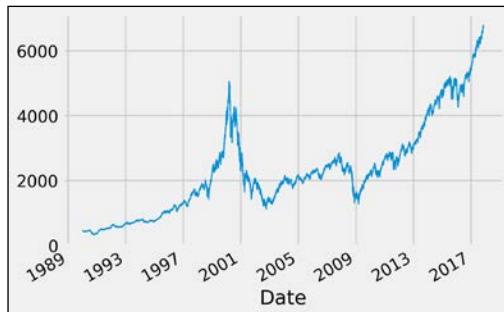
| Date | Open | High | Low | Close | Adj Close | Volume |
|------------|------------|------------|------------|------------|------------|-----------|
| 1990-01-02 | 452.899994 | 459.299988 | 452.700012 | 459.299988 | 459.299988 | 110720000 |
| 1990-01-03 | 461.100006 | 461.600006 | 460.000000 | 460.899994 | 460.899994 | 152660000 |
| 1990-01-04 | 460.399994 | 460.799988 | 456.899994 | 459.399994 | 459.399994 | 147950000 |
| 1990-01-05 | 457.899994 | 459.399994 | 457.799988 | 458.200012 | 458.200012 | 137230000 |
| 1990-01-08 | 457.100006 | 458.700012 | 456.500000 | 458.700012 | 458.700012 | 115500000 |

3. Let's extract two columns: the date and the daily closing value:

```
>>> date = nasdaq_df.index
       nasdaq = nasdaq_df['Close']
```

4. Let's take a look at the raw signal:

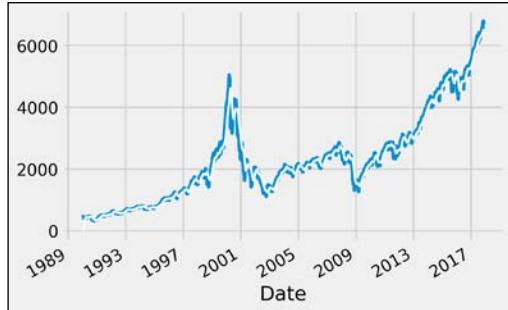
```
>>> fig, ax = plt.subplots(1, 1, figsize=(6, 4))
       nasdaq.plot(ax=ax, lw=1)
```



5. Now, we will follow the first approach to get the slow variations of the signal. We will convolve the signal with a triangular window, which corresponds to a **FIR filter**. We will explain the idea behind this method in the *How it works...* section of this recipe. For now, let's just say that we replace each value with a weighted mean of the signal around this value:

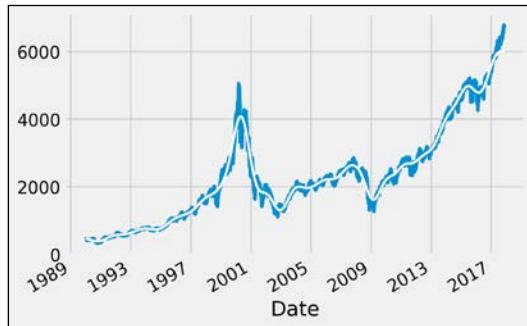
```
>>> # We get a triangular window with 60 samples.
       h = sg.get_window('triang', 60)
       # We convolve the signal with this window.
       fil = sg.convolve(nasdaq, h / h.sum())
```

```
>>> fig, ax = plt.subplots(1, 1, figsize=(6, 4))
# We plot the original signal...
nasdaq.plot(ax=ax, lw=3)
# ... and the filtered signal.
ax.plot_date(date, fil[:len(nasdaq)],
              '-w', lw=2)
```



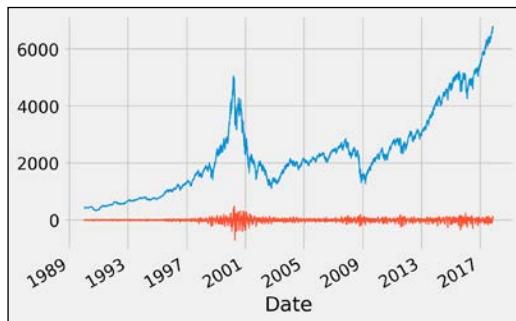
6. Now, let's use another method. We create an IIR Butterworth low-pass filter to extract the slow variations of the signal. The `filtfilt()` method allows us to apply a filter forward and backward in order to avoid phase delays:

```
>>> fig, ax = plt.subplots(1, 1, figsize=(6, 4))
nasdaq.plot(ax=ax, lw=3)
# We create a 4-th order Butterworth low-pass filter.
b, a = sg.butter(4, 2. / 365)
# We apply this filter to the signal.
ax.plot_date(date, sg.filtfilt(b, a, nasdaq),
              '-w', lw=2)
```



7. Finally, we use the same method to create a high-pass filter and extract the *fast* variations of the signal:

```
>>> fig, ax = plt.subplots(1, 1, figsize=(6, 4))
nasdaq.plot(ax=ax, lw=1)
b, a = sg.butter(4, 2 * 5. / 365, btype='high')
ax.plot_date(date, sg.filtfilt(b, a, nasdaq),
              '--', lw=1)
```



The fast variations around 2000 correspond to the **dot-com bubble burst**, reflecting the high-market volatility and the fast fluctuations of the stock market indices at that time. For more details, refer to https://en.wikipedia.org/wiki/Dot-com_bubble.

How it works...

In this section, we explain the very basics of linear filters in the context of digital signals.

A **digital signal** is a discrete sequence (x_n) indexed by $n \geq 0$. Although we often assume infinite sequences, in practice, a signal is represented by a vector of the finite size N .

In the continuous case, we would rather manipulate time-dependent functions $f(t)$. Loosely stated, we can go from continuous signals to discrete signals by discretizing time and transforming integrals into sums.

What are linear filters?

A **linear filter** F transforms an input signal $x = (x_n)$ to an output signal $y = (y_n)$. This transformation is linear—the transformation of the sum of two signals is the sum of the transformed signals: $F(x + y) = F(x) + F(y)$.

In addition to this, multiplying the input signal by a constant yields the same output as multiplying the original output signal by the same constant: $F(\lambda x) = \lambda F(x)$.

A **Linear Time-Invariant (LTI)** filter has an additional property: if the signal (x_n) is transformed to (y_n) , then the shifted signal (x_{n-k}) is transformed to (y_{n-k}) , for any fixed k . In other words, the system is time-invariant because the output does not depend on the particular time the input is applied.



From now on, we will only consider LTI filters.



Linear filters and convolutions

A very important result in the LTI system theory is that LTI filters can be described by a single signal: the impulse response h . This is the output of the filter in response to an impulse signal. For digital filters, the impulse signal is $(1, 0, 0, 0, \dots)$.

It can be shown that $x = (x_n)$ is transformed to $y = (y_n)$ defined by the **convolution** of the impulse response h with the signal x :

$$\mathbf{y} = \mathbf{h} * \mathbf{x}, \quad \text{or} \quad y_n = \sum_{k=0}^n h_k x_{n-k}$$

The convolution is a fundamental mathematical operation in signal processing. Intuitively, and considering a convolution function peaking around zero, the convolution is equivalent to taking a local average of the signal (x here), weighted by a given window (h here).

It is implied, by our notations, that we restrict ourselves to **causal filters** ($h_n = 0$ for $n < 0$). This property means that the output of the signal only depends on the present and the past of the input, not the future. This is a natural property in many situations.

The FIR and IIR filters

The **support** of a signal (h_n) is the set of n such that $h_n \neq 0$. LTI filters can be classified into two categories:

- ▶ A **Finite Impulse Response (FIR)** filter has an impulse response with finite support
- ▶ A **Infinite Impulse Response (IIR)** filter has an impulse response with infinite support

A FIR filter can be described by a finite impulse response of size N (a vector). It works by convolving a signal with its impulse response. Let's define $b_n = h_n$ for $n \leq N$. Then, y_n is a linear combination of the last $N + 1$ values of the input signal:

$$y_n = \sum_{k=0}^N b_k x_{n-k}$$

On the other hand, an IIR filter is described by an infinite impulse response that cannot be represented exactly under this form. For this reason, we often use an alternative representation:

$$y_n = \frac{1}{a_0} \left(\sum_{k=0}^N b_k x_{n-k} - \sum_{l=1}^M a_l y_{n-l} \right)$$

This **difference equation** expresses y_n as a linear combination of the last $N + 1$ values of the *input* signal (the **feedforward term**, like for a FIR filter) and a linear combination of the last M values of the *output* signal (**feedback term**). The feedback term makes the IIR filter more complex than a FIR filter in that the output depends not only on the input but also on the previous values of the output (dynamics).

Filters in the frequency domain

We only described filters in the temporal domain. Alternate representations in other domains exist such as Laplace transforms, Z-transforms, and Fourier transforms.

In particular, the *Fourier transform* has a very convenient property: it transforms convolutions into multiplications in the frequency domain. In other words, in the frequency domain, an LTI filter multiplies the Fourier transform of the input signal by the Fourier transform of the impulse response.

The low-, high-, and band-pass filters

Filters can be characterized by their effects on the amplitude of the input signal's frequencies. They are as follows:

- ▶ A **low-pass filter** attenuates the components of the signal at frequencies *higher* than a **cutoff frequency**
- ▶ A **high-pass filter** attenuates the components of the signal at frequencies *lower* than a cutoff frequency
- ▶ A **band-pass filter** passes the components of the signal at frequencies within a certain range and attenuates those outside

In this recipe, we first convolved the input signal with a triangular window (with finite support). It can be shown that this operation corresponds to a low-pass FIR filter. It is a particular case of the **moving average method**, which computes a local weighted average of every value in order to smooth out the signal.

Then, we applied two instances of the **Butterworth filter**, a particular kind of IIR filter that can act as a low-pass, high-pass, or band-pass filter. In this recipe, we first used it as a low-pass filter to smooth out the signal, before using it as a high-pass filter to extract fast variations of the signal.

There's more...

Here are some general references about digital signal processing and linear filters:

- ▶ Digital signal processing on Wikipedia, available at https://en.wikipedia.org/wiki/Digital_signal_processing
- ▶ Linear filters on Wikipedia, available at https://en.wikipedia.org/wiki/Linear_filter
- ▶ LTI filters on Wikipedia, available at https://en.wikipedia.org/wiki/LTI_system_theory

See also

- ▶ The *Analyzing the frequency components of a signal with a Fast Fourier Transform* recipe

Computing the autocorrelation of a time series

The autocorrelation of a time series can inform us about repeating patterns or serial correlation. The latter refers to the correlation between the signal at a given time and at a later time. The analysis of the autocorrelation can thereby inform us about the timescale of the fluctuations. Here, we use this tool to analyze the evolution of baby names in the US, based on data provided by the United States Social Security Administration.

How to do it...

1. We import the following packages:

```
>>> import os  
      import numpy as np  
      import pandas as pd  
      import matplotlib.pyplot as plt  
      %matplotlib inline
```

2. We download the *Babies* dataset (available on the GitHub data repository of the book) using the *requests* third-party package. The dataset was obtained initially from the `data.gov` website (<https://catalog.data.gov/dataset/baby-names-from-social-security-card-applications-national-level-data>). We extract the archive locally in the `babies` subdirectory. There is one CSV file per year. Each file contains all baby names given that year with the respective frequencies.

```
>>> import io  
      import requests
```

```
import zipfile
>>> url = ('https://github.com/ipython-books/'
           'cookbook-2nd-data/blob/master/'
           'babies.zip?raw=true')
       r = io.BytesIO(requests.get(url).content)
       zipfile.ZipFile(r).extractall('babies')
>>> %ls babies
yob1902.txt
yob1903.txt
yob1904.txt
...
yob2014.txt
yob2015.txt
yob2016.txt
```

3. We read the data with pandas. We load the data in a dictionary, containing one DataFrame per year:

```
>>> files = [file for file in os.listdir('babies')
             if file.startswith('yob')]
>>> years = np.array(sorted([int(file[3:7])
                           for file in files]))
>>> data = {year:
             pd.read_csv('babies/yob%d.txt' % year,
                         index_col=0, header=None,
                         names=['First name',
                                'Gender',
                                'Number'])
           for year in years}
>>> data[2016].head()
```

| Gender Number | | |
|---------------|--------|--------|
| First name | Gender | Number |
| Emma | F | 19414 |
| Olivia | F | 19246 |
| Ava | F | 16237 |
| Sophia | F | 16070 |
| Isabella | F | 14722 |

-
4. We write functions to retrieve the frequencies of baby names as a function of the name, gender, and birth year:

```
>>> def get_value(name, gender, year):  
    """Return the number of babies born a given year,  
    with a given gender and a given name."""  
    dy = data[year]  
    try:  
        return dy[dy['Gender'] ==  
                  gender]['Number'][name]  
    except KeyError:  
        return 0  
>>> def get_evolution(name, gender):  
    """Return the evolution of a baby name over  
    the years."""  
    return np.array([get_value(name, gender, year)  
                   for year in years])
```

5. Let's define a function that computes the autocorrelation of a signal. This function is essentially based on NumPy's `correlate()` function.

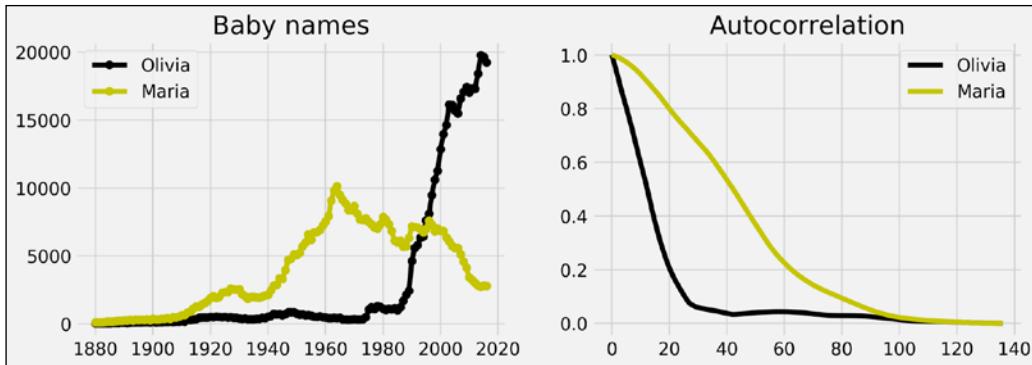
```
>>> def autocorr(x):  
    result = np.correlate(x, x, mode='full')  
    return result[result.size // 2:]
```

6. Now, we create a function that displays the evolution of a baby name as well as its (normalized) autocorrelation:

```
>>> def autocorr_name(name, gender, color, axes=None):  
    x = get_evolution(name, gender)  
    z = autocorr(x)  
  
    # Evolution of the name.  
    axes[0].plot(years, x, '-o' + color,  
                 label=name)  
    axes[0].set_title("Baby names")  
    axes[0].legend()  
  
    # Autocorrelation.  
    axes[1].plot(z / float(z.max()),  
                 '--' + color, label=name)  
    axes[1].legend()  
    axes[1].set_title("Autocorrelation")
```

7. Let's take a look at two female names:

```
>>> fig, axes = plt.subplots(1, 2, figsize=(12, 4))  
autocorr_name('Olivia', 'F', 'k', axes=axes)  
autocorr_name('Maria', 'F', 'y', axes=axes)
```



The autocorrelation of Olivia is decaying much faster than Maria's. This is mainly because of the steep increase of the name Olivia at the end of the twentieth century. By contrast, the name Maria is varying more slowly globally, and its autocorrelation is decaying slower.

How it works...

A **time series** is a sequence indexed by time. Important applications include stock markets, product sales, weather forecasting, biological signals, and many others. Time series analysis is an important part of statistical data analysis, signal processing, and machine learning.

There are various definitions of the autocorrelation. Here, we define the autocorrelation of a time series (x_n) as:

$$R(k) = \frac{1}{N} \sum_n x_n x_{n+k}$$

In the previous plot, we normalized the autocorrelation by its maximum so as to compare the autocorrelation of two signals. The autocorrelation quantifies the average similarity between the signal and a shifted version of the same signal, as a function of the delay between the two. In other words, the autocorrelation can give us information about repeating patterns as well as the timescale of the signal's fluctuations. The faster the autocorrelation decays to zero, the faster the signal varies.

There's more...

Here are a few references:

- ▶ NumPy's correlation function documentation, available at <http://docs.scipy.org/doc/numpy/reference/generated/numpy.correlate.html>
- ▶ Autocorrelation function in statsmodels, documented at <http://statsmodels.sourceforge.net/stable/tsa.html>

Signal Processing —————

- ▶ Time series on Wikipedia, available at https://en.wikipedia.org/wiki/Time_series
- ▶ Serial dependence on Wikipedia, available at https://en.wikipedia.org/wiki/Serial_dependence
- ▶ Autocorrelation on Wikipedia, available at <https://en.wikipedia.org/wiki/Autocorrelation>

See also

- ▶ The *Analyzing the frequency components of a signal with a Fast Fourier Transform* recipe

11

Image and Audio Processing

In this chapter, we will cover the following topics:

- ▶ Manipulating the exposure of an image
- ▶ Applying filters on an image
- ▶ Segmenting an image
- ▶ Finding points of interest in an image
- ▶ Detecting faces in an image with OpenCV
- ▶ Applying digital filters to speech sounds
- ▶ Creating a sound synthesizer in the Notebook

Introduction

In the previous chapter, we covered signal processing techniques for one-dimensional, time-dependent signals. In this chapter, we will see signal processing techniques for images and sounds.

Generic signal processing techniques can be applied to images and sounds, but many image or audio processing tasks require specialized algorithms. For example, we will see algorithms for segmenting images, detecting points of interest in an image, or detecting faces. We will also hear the effect of linear filters on speech sounds.

The `scikit-image` package is one of the main image processing packages in Python. We will use it in most of the image processing recipes in this chapter. For more on `scikit-image`, refer to <http://scikit-image.org>.

We will also use **OpenCV** (<http://opencv.org>), a computer vision library in C++ that has a Python wrapper.

In this introduction, we will discuss the particularities of images and sounds from a signal processing point of view.

Images

A **grayscale image** is a bidimensional signal represented by a function, f , that maps each pixel to an **intensity**. For example, the intensity could be a real value between 0 (dark) and 1 (light). In a colored image, this function maps each pixel to a triplet of intensities—generally, the red, green, and blue (RGB) components.

On a computer, images are digitally sampled. The intensities are not real values, but integers or floating point numbers. On one hand, the mathematical formulation of continuous functions allows us to apply analytical tools such as derivatives and integrals. On the other hand, we need to take into account the digital nature of the images we deal with.

Sounds

From a signal processing perspective, a sound is a time-dependent signal that has sufficient power in the hearing frequency range (about 20 Hz to 20 kHz). Then, according to the Nyquist-Shannon theorem (introduced in *Chapter 10, Signal Processing*), the sampling rate of a digital sound signal needs to be at least 40 kHz. A sampling rate of 44100 Hz is frequently chosen.

References

Here are a few references:

- ▶ Image processing on Wikipedia, available at https://en.wikipedia.org/wiki/Image_processing
- ▶ Numerical Tours, advanced image processing algorithms available at <http://www.numerical-tours.com/python/>
- ▶ Audio signal processing on Wikipedia, available at https://en.wikipedia.org/wiki/Audio_signal_processing
- ▶ Particularities of the 44100 Hz sampling rate explained at https://en.wikipedia.org/wiki/44,100_Hz

Manipulating the exposure of an image

The **exposure** of an image tells us whether the image is too dark, too light, or balanced. It can be measured with a histogram of the intensity values of all pixels. Improving the exposure of an image is a basic image-editing operation. As we will see in this recipe, it can be done easily with scikit-image.

Getting ready

The `scikit-image` command should be included by default in Anaconda. Otherwise, you can always install it manually with `conda install scikit-image`.

How to do it...

1. Let's import the packages:

```
>>> import numpy as np  
      import matplotlib.pyplot as plt  
      import skimage.exposure as skie  
      %matplotlib inline
```

2. We open an image with Matplotlib. We only take a single RGB component to have a grayscale image (it is a very crude way of doing it, we give much better ways at the end of this recipe):

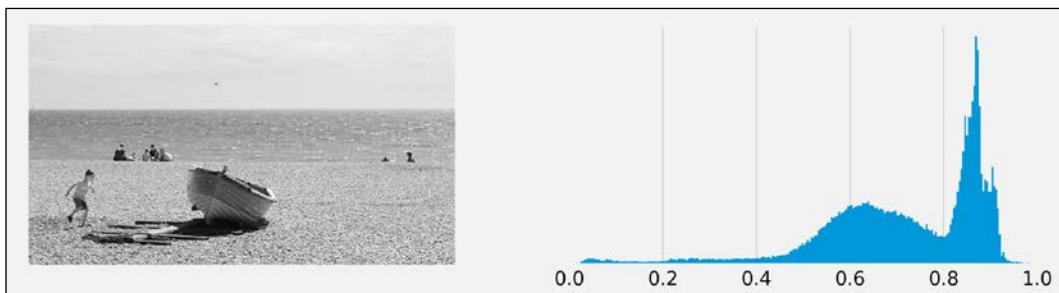
```
>>> img = plt.imread('https://github.com/ipython-books/'  
'cookbook-2nd-data/blob/master/'  
'beach.png?raw=true')[..., 0]
```

3. We create a function that displays the image along with its histogram of the intensity values (that is, the exposure):

```
>>> def show(img):  
      # Display the image.  
      fig, (ax1, ax2) = plt.subplots(1, 2,  
                                    figsize=(12, 3))  
  
      ax1.imshow(img, cmap=plt.cm.gray)  
      ax1.set_axis_off()  
  
      # Display the histogram.  
      ax2.hist(img.ravel(), lw=0, bins=256)  
      ax2.set_xlim(0, img.max())  
      ax2.set_yticks([])  
  
      plt.show()
```

4. Let's display the image along with its histogram:

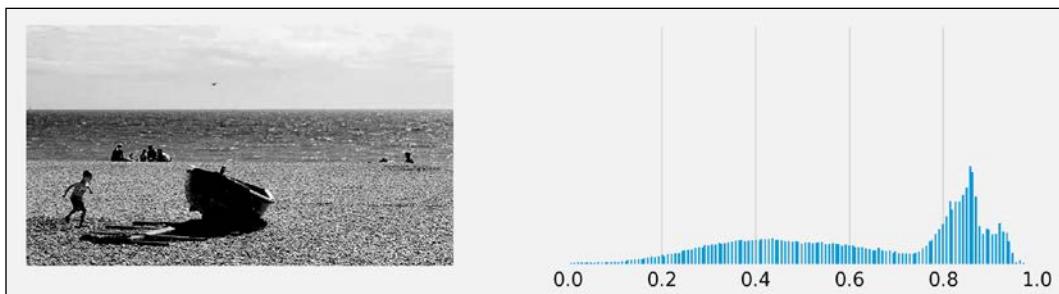
```
>>> show(img)
```



The histogram is unbalanced and the image appears overexposed (many pixels are too bright).

5. Now, we rescale the intensity of the image using scikit-image's `rescale_intensity` function. The `in_range` and `out_range` parameters define a linear mapping from the original image to the modified image. The pixels that are outside `in_range` are clipped to the extremal values of `out_range`. Here, the darkest pixels (intensity less than 100) become completely black (0), whereas the brightest pixels (>240) become completely white (255):

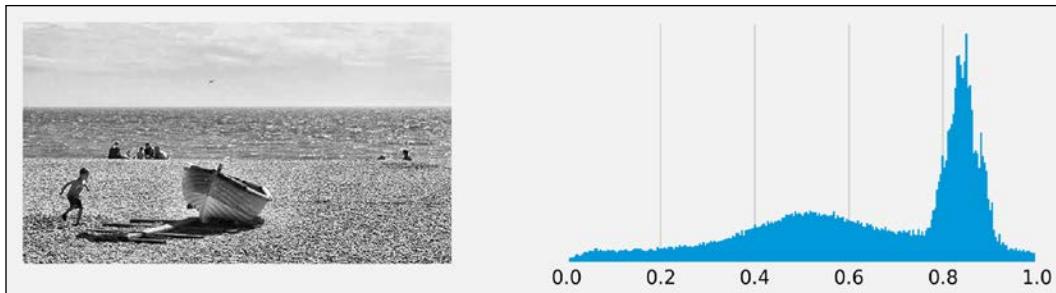
```
>>> show(skie.rescale_intensity(  
    img, in_range=(0.4, .95), out_range=(0, 1)))
```



Many intensity values seem to be missing in the histogram, which reflects the poor quality of this crude exposure correction technique.

6. We now use a more advanced exposure correction technique called **Contrast Limited Adaptive Histogram Equalization (CLAHE)**:

```
>>> show(skie.equalize_adapthist(img))
```



The histogram seems more balanced, and the image now appears more contrasted.

How it works...

An image's histogram represents the distribution of the pixels' intensity values. It is a central tool in image editing, image processing, and computer vision.

The `rescale_intensity()` function stretches or shrinks the intensity levels of the image. One use case is to ensure that the whole range of values allowed by the data type is used by the image.

The `equalize_adapthist()` function works by splitting the image into rectangular sections and computing the histogram for each section. Then, the intensity values of the pixels are redistributed to improve the contrast and enhance the details.

The `skimage.color.rgb2gray()` function converts a colored image to a grayscale image using a special weighting of the color channels that preserves luminance.

There's more...

Here are some references:

- ▶ Transforming image data in the scikit-image documentation, at http://scikit-image.org/docs/dev/user_guide/transforming_image_data.html
- ▶ Histogram equalization in the scikit-image documentation, at http://scikit-image.org/docs/dev/auto_examples/color_exposure/plot_equalize.html
- ▶ Image histogram on Wikipedia, available at https://en.wikipedia.org/wiki/Image_histogram
- ▶ Histogram equalization on Wikipedia, available at https://en.wikipedia.org/wiki/Histogram_equalization

- ▶ Adaptive histogram equalization on Wikipedia, available at https://en.wikipedia.org/wiki/Adaptive_histogram_equalization
- ▶ Contrast on Wikipedia, available at [https://en.wikipedia.org/wiki/Contrast_\(vision\)](https://en.wikipedia.org/wiki/Contrast_(vision))

See also

- ▶ The *Applying filters on an image* recipe

Applying filters on an image

In this recipe, we apply filters on an image for various purposes: blurring, denoising, and edge detection.

How it works...

1. Let's import the packages:

```
>>> import numpy as np
      import matplotlib.pyplot as plt
      import skimage
      import skimage.color as skic
      import skimage.filters as skif
      import skimage.data as skid
      import skimage.util as sku
      %matplotlib inline
```

2. We create a function that displays a grayscale image:

```
>>> def show(img):
      fig, ax = plt.subplots(1, 1, figsize=(8, 8))
      ax.imshow(img, cmap=plt.cm.gray)
      ax.set_axis_off()
      plt.show()
```

3. Now, we load the *Astronaut* image (bundled in scikit-image). We convert it to a grayscale image with the `rgb2gray()` function:

```
>>> img = skic.rgb2gray(skid.astronaut())
>>> show(img)
```



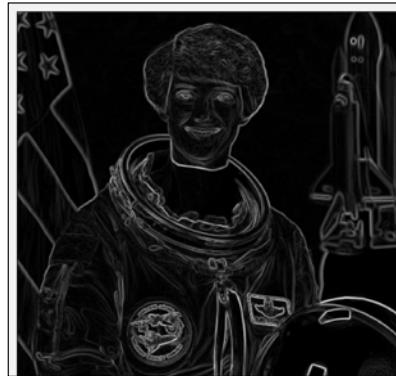
4. Let's apply a blurring **Gaussian filter** to the image:

```
>>> show(skif.gaussian(img, 5.))
```



5. We now apply a **Sobel filter** that enhances the edges in the image:

```
>>> sobimg = skif.sobel(img)
       show(sobimg)
```



6. We can threshold the filtered image to get a *sketch* effect. We obtain a binary image that only contains the edges. We use a notebook widget to find an adequate thresholding value; by adding the `@interact` decorator, we display a slider on top of the image. This widget lets us control the threshold dynamically.

```
>>> from ipywidgets import widgets
```

```
@widgets.interact(x=(0.01, .2, .005))
def edge(x):
    show(sobimg < x)
```



7. Finally, we add some noise to the image to illustrate the effect of a denoising filter:

```
>>> img = skimage.img_as_float(skid.astronaut())  
  
    # We take a portion of the image to show the details.  
    img = img[50:200, 150:300]  
  
    # We add Gaussian noise.  
    img_n = sku.random_noise(img)  
    show(img_n)
```



The `denoise_tv_bregman()` function implements total-variation denoising using the Split Bregman method:

```
>>> img_r = skimage.restoration.denoise_tv_bregman(  
        img_n, 5.)  
  
fig, (ax1, ax2, ax3) = plt.subplots(  
    1, 3, figsize=(12, 8))  
  
ax1.imshow(img_n)  
ax1.set_title('With noise')  
ax1.set_axis_off()  
  
ax2.imshow(img_r)  
ax2.set_title('Denoised')  
ax2.set_axis_off()
```

```
ax3.imshow(img)
ax3.set_title('Original')
ax3.set_axis_off()
```



How it works...

Many filters used in image processing are linear filters. These filters are very similar to those seen in *Chapter 10, Signal Processing*; the only difference is that they work in two dimensions. Applying a linear filter to an image amounts to performing a discrete **convolution** of the image with a particular function. The Gaussian filter applies a convolution with a Gaussian function to blur the image.

The Sobel filter computes an approximation of the gradient of the image. Therefore, it can detect fast-varying spatial changes in the image, which generally correspond to edges.

Image denoising refers to the process of removing noise from an image. **Total variation denoising** works by finding a regular image close to the original (noisy) image. Regularity is quantified by the **total variation** of the image:

$$V(x) = \sum_{i,j} \sqrt{|x_{i+1,j} - x_{i,j}|^2 + |x_{i,j+1} - x_{i,j}|^2}$$

The **Split Bregman method** is a variant based on the L1 norm. It is an instance of **compressed sensing**, which aims to find regular and sparse approximations of real-world noisy measurements.

There's more...

Here are a few references:

- ▶ API reference of the `skimage.filter` module available at <http://scikit-image.org/docs/dev/api/skimage.filters.html>
- ▶ Noise reduction on Wikipedia, available at https://en.wikipedia.org/wiki/Noise_reduction

- ▶ Gaussian filter on Wikipedia, available at https://en.wikipedia.org/wiki/Gaussian_filter
- ▶ Sobel filter on Wikipedia, available at https://en.wikipedia.org/wiki/Sobel_operator
- ▶ The Split Bregman algorithm explained at http://www.ece.rice.edu/~tag7/Tom_Goldstein/Split_Bregman.html

See also

- ▶ The *Manipulating the exposure of an image* recipe

Segmenting an image

Image segmentation consists of partitioning an image into different regions that share certain characteristics. This is a fundamental task in computer vision, facial recognition, and medical imaging. For example, an image segmentation algorithm can automatically detect the contours of an organ in a medical image.

The scikit-image provides several segmentation methods. In this recipe, we will demonstrate how to segment an image containing different objects. This recipe is inspired by a scikit-image example available at http://scikit-image.org/docs/dev/user_guide/tutorial_segmentation.html

How to do it...

1. Let's import the packages:

```
>>> import numpy as np
      import matplotlib.pyplot as plt
      from skimage.data import coins
      from skimage.filters import threshold_otsu
      from skimage.segmentation import clear_border
      from skimage.morphology import label, closing, square
      from skimage.measure import regionprops
      from skimage.color import lab2rgb
      %matplotlib inline
```

2. We create a function that displays a grayscale image:

```
>>> def show(img, cmap=None):
      cmap = cmap or plt.cm.gray
      fig, ax = plt.subplots(1, 1, figsize=(8, 6))
      ax.imshow(img, cmap=cmap)
      ax.set_axis_off()
      plt.show()
```

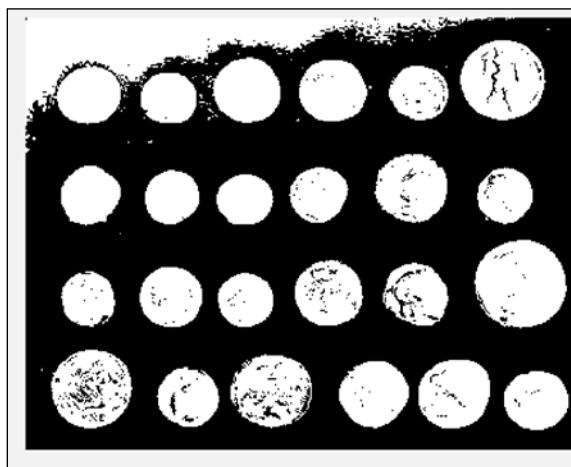
3. We get a test image bundled in scikit-image, showing various coins on a plain background:

```
>>> img = coins()  
>>> show(img)
```



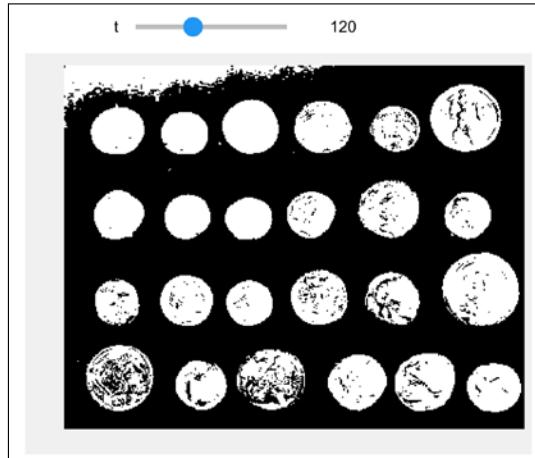
4. The first step to segment the image is finding an intensity threshold separating the (bright) coins from the (dark) background. **Otsu's method** defines a simple algorithm to automatically find such a threshold.

```
>>> threshold_otsu(img)  
107  
>>> show(img > 107)
```



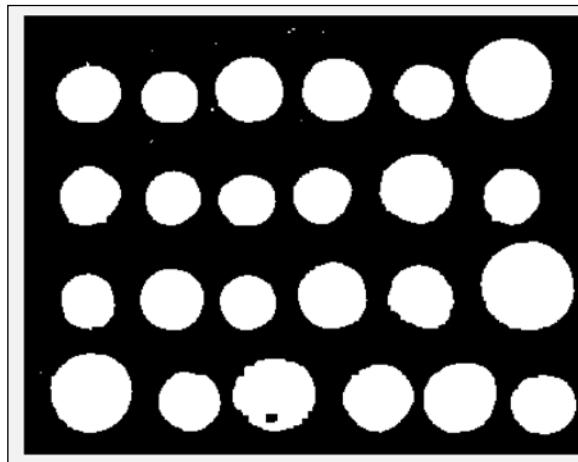
5. There appears to be a problem in the top-left corner of the image, with part of the background being too bright. Let's use a Notebook widget to find a better threshold:

```
>>> from ipywidgets import widgets  
  
@widgets.interact(t=(50, 240))  
def threshold(t):  
    show(img > t)
```



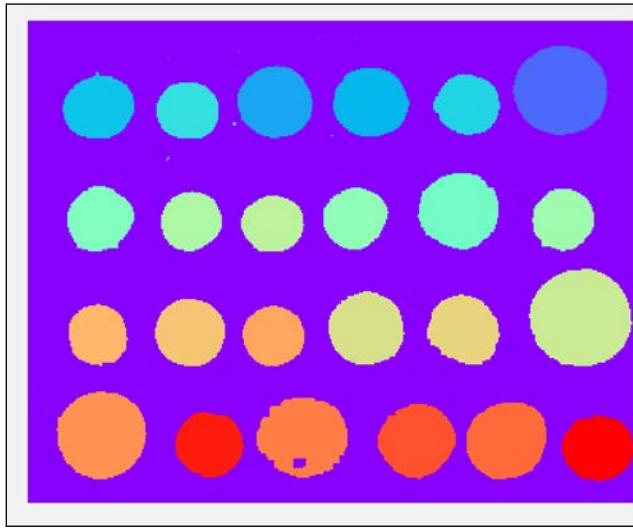
6. The threshold 120 looks better. The next step consists of cleaning the binary image by smoothing the coins and removing the border. The `scikit-image` library contains a few functions for these purposes.

```
>>> img_bin = closing(closing(img > 120, square(5)))  
show(img_bin)
```



7. Next, we perform the segmentation task itself with the `label()` function. This function detects the connected components in the image and attributes a unique label to every component. Here, we color code the labels in the binary image:

```
>>> labels = label(img_bin)
      show(labels, cmap=plt.cm.rainbow)
```



8. Small artifacts in the image result in spurious labels that do not correspond to coins. Therefore, we only keep components with more than 100 pixels. The `regionprops()` function allows us to retrieve specific properties of the components (here, the area and the bounding box):

```
>>> regions = regionprops(labels)
      boxes = np.array([label['BoundingBox']
                        for label in regions
                        if label['Area'] > 100])
      print(f"There are {len(boxes)} coins.")
There are 24 coins.
```

9. Finally, we show the label number on top of each component in the original image:

```
>>> fig, ax = plt.subplots(1, 1, figsize=(8, 6))
      ax.imshow(img, cmap=plt.cm.gray)
      ax.set_axis_off()

      # Get the coordinates of the boxes.
      xs = boxes[:, [1, 3]].mean(axis=1)
      ys = boxes[:, [0, 2]].mean(axis=1)
```

```

# We reorder the boxes by increasing
# column first, and row second.
for row in range(4):
    # We select the coins in each of the four rows.
    if row < 3:
        ind = ((ys[6 * row] <= ys) &
               (ys < ys[6 * row + 6]))
    else:
        ind = (ys[6 * row] <= ys)
    # We reorder by increasing x coordinate.
    ind = np.nonzero(ind)[0]
    reordered = ind[np.argsort(xs[ind])]
    xs_row = xs(reordered)
    ys_row = ys(reordered)
    # We display the coin number.
    for col in range(6):
        n = 6 * row + col
        ax.text(xs_row[col] - 5, ys_row[col] + 5,
                str(n),
                fontsize=20)

```



How it works...

To clean up the coins in the thresholded image, we used **mathematical morphology** techniques. These methods, based on set theory, geometry, and topology, allow us to manipulate shapes.

For example, let's explain **dilation** and **erosion**. First, if A is a set of pixels in an image, and b is a 2D vector, we denote A_b the set A translated by b as:

$$A_b = \{a + b \mid a \in A\}$$

Let B be a set of vectors with integer components. We call B the **structuring element** (here, we used a square). This set represents the neighborhood of a pixel. The dilation of A by B is:

$$A \oplus B = \bigcup_{b \in B} A_b$$

The erosion of A by B is:

$$A \ominus B = \{z \in E \mid B_z \subseteq A\}$$

A dilation extends a set by adding pixels close to its boundaries. An erosion removes the pixels of the set that are too close to the boundaries. The **closing** of a set is a dilation followed by an erosion. This operation can remove small dark spots and connect small bright cracks. In this recipe, we used a square structuring element.

There's more...

Here are a few references:

- ▶ SciPy lecture notes on image processing available at <http://scipy-lectures.github.io/packages/scikit-image/>
- ▶ Image segmentation on Wikipedia, available at https://en.wikipedia.org/wiki/Image_segmentation
- ▶ Otsu's method to find a threshold explained at https://en.wikipedia.org/wiki/Otsu%27s_method
- ▶ Segmentation tutorial with scikit-image (which inspired this recipe) available at http://scikit-image.org/docs/dev/user_guide/tutorial_segmentation.html
- ▶ Mathematical morphology on Wikipedia, available at https://en.wikipedia.org/wiki/Mathematical_morphology

- ▶ API reference of the `skimage.morphology` module available at <http://scikit-image.org/docs/dev/api/skimage.morphology.html>

See also

- ▶ The *Computing connected components in an image* recipe, in *Chapter 14, Graphs, Geometry, and Geographic Information Systems*.

Finding points of interest in an image

In an image, **points of interest** are positions where there might be edges, corners, or interesting objects. For example, in a landscape picture, points of interest can be located near a house or a person. Detecting points of interest is useful in image recognition, computer vision, or medical imaging.

In this recipe, we will find points of interest in an image with scikit-image. This will allow us to crop an image around the subject of the picture, even when this subject is not in the center of the image.

How to do it...

1. Let's import the packages:

```
>>> import numpy as np
      import matplotlib.pyplot as plt
      import skimage
      import skimage.feature as sf
      %matplotlib inline
```

2. We create a function to display a colored or grayscale image:

```
>>> def show(img, cmap=None):
      cmap = cmap or plt.cm.gray
      fig, ax = plt.subplots(1, 1, figsize=(8, 6))
      ax.imshow(img, cmap=cmap)
      ax.set_axis_off()
      return ax
```

3. We load an image:

```
>>> img = plt.imread('https://github.com/ipython-books/'  
'cookbook-2nd-data/blob/master/'  
'child.png?raw=true')  
>>> show(img)
```



4. Let's find salient points in the image with the Harris corner method. The first step consists of computing the **Harris corner measure response image** with the `corner_harris()` function (we will explain this measure in the *How it works...* section of this recipe). This function requires a grayscale image, thus we select the first RGB component:

```
>>> corners = sf.corner_harris(img[:, :, 0])  
>>> show(corners)
```



We see that the patterns in the child's coat are well detected by this algorithm.

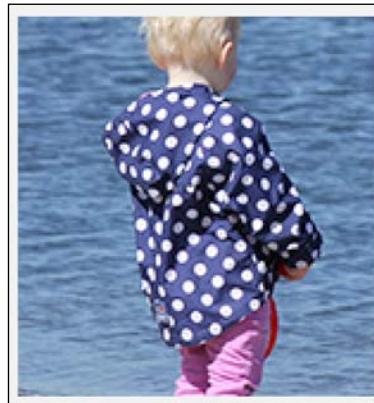
5. The next step is to detect corners from this measure image, using the `corner_peaks()` function:

```
>>> peaks = sf.corner_peaks(corners)  
>>> ax = show(img)  
    ax.plot(peaks[:, 1], peaks[:, 0], 'or', ms=4)
```



6. Finally, we create a box around the median position of the corner points to define our region of interest:

```
>>> # The median defines the approximate position of
# the corner points.
ym, xm = np.median(peaks, axis=0)
# The standard deviation gives an estimation
# of the spread of the corner points.
ys, xs = 2 * peaks.std(axis=0)
xm, ym = int(xm), int(ym)
xs, ys = int(xs), int(ys)
show(img[ym - ys:ym + ys, xm - xs:xm + xs])
```



How it works...

Let's explain the method used in this recipe. The first step consists of computing the **structure tensor** (or **Harris matrix**) of the image:

$$A = \begin{bmatrix} \langle I_x^2 \rangle & \langle I_x I_y \rangle \\ \langle I_x I_y \rangle & \langle I_y^2 \rangle \end{bmatrix}$$

Here, $I(x, y)$ is the image, I_x and I_y are the partial derivatives, and the brackets denote the local spatial average around neighboring values.

This tensor associates a $(2, 2)$ positive symmetric matrix at each point. This matrix is used to calculate a sort of autocorrelation of the image at each point.

Let λ and μ be the two eigenvalues of this matrix (the matrix is diagonalizable because it is real and symmetric). Roughly, a corner is characterized by a large variation of the autocorrelation in all directions, or in large positive eigenvalues λ and μ . The corner measure image is defined as:

$$M = \det(A) - k \times \text{trace}(A)^2 = \lambda\mu - k(\lambda + \mu)^2$$

Here, k is a tunable parameter. M is large when there is a corner. Finally, `corner_peaks()` finds corner points by looking at local maxima in the corner measure image.

There's more...

Here are a few references:

- ▶ A corner detection example with scikit-image available at http://scikit-image.org/docs/dev/auto_examples/features_detection/plot_corner.html
- ▶ An image processing tutorial with scikit-image available at <http://blog.yhat.com/posts/image-processing-with-scikit-image.html>
- ▶ Corner detection on Wikipedia, available at https://en.wikipedia.org/wiki/Corner_detection
- ▶ Structure tensor on Wikipedia, available at https://en.wikipedia.org/wiki/Structure_tensor
- ▶ API reference of the `skimage.feature` module available at <http://scikit-image.org/docs/dev/api/skimage.feature.html>

Detecting faces in an image with OpenCV

OpenCV (Open Computer Vision) is an open source C++ library for computer vision. It features algorithms for image segmentation, object recognition, augmented reality, face detection, and other computer vision tasks.

In this recipe, we will use OpenCV in Python to detect faces in a picture.

Getting ready

You need OpenCV and the Python wrapper. You can install them with the following command:

```
conda install -c conda-forge opencv
```

How to do it...

1. First, we import the packages:

```
>>> import io
      import zipfile
      import requests
      import numpy as np
      import cv2
      import matplotlib.pyplot as plt
      %matplotlib inline
```

2. We download and extract the dataset in the data/ subfolder:

```
>>> url = ('https://github.com/ipython-books/'
          'cookbook-2nd-data/blob/master/'
          'family.zip?raw=true')
r = io.BytesIO(requests.get(url).content)
zipfile.ZipFile(r).extractall('data')
```

3. We open the JPG image with OpenCV:

```
>>> img = cv2.imread('data/family.jpg')
```

4. Then, we convert it to a grayscale image using OpenCV's `cvtColor()` function. For face detection, it is sufficient and faster to use grayscale images.

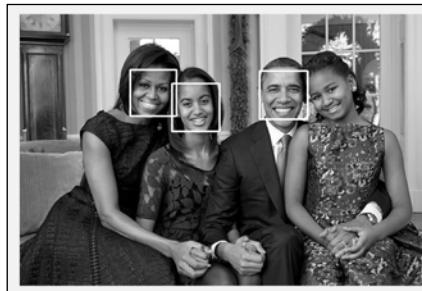
```
>>> gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

5. To detect faces, we will use the **Viola–Jones object detection framework**. A cascade of Haar-like classifiers has been trained on many images to detect faces (more details are given in the next section). The result of the training is stored in an XML file that is part of the archive that was downloaded in step 2. We load this cascade from this XML file with OpenCV's `CascadeClassifier` class:

```
>>> path = 'data/haarcascade_frontalface_default.xml'  
face_cascade = cv2.CascadeClassifier(path)
```

6. Finally, the `detectMultiScale()` method of the classifier detects the objects on a grayscale image and returns a list of rectangles around these objects:

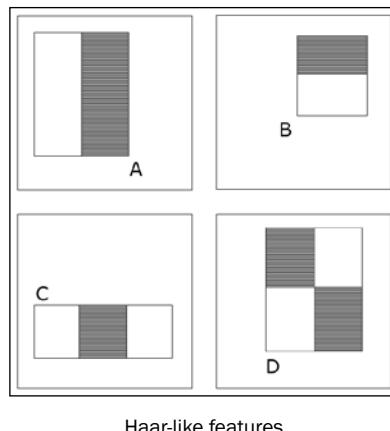
```
>>> for x, y, w, h in face_cascade.detectMultiScale(  
    gray, 1.3):  
    cv2.rectangle(  
        gray, (x, y), (x + w, y + h), (255, 0, 0), 2)  
fig, ax = plt.subplots(1, 1, figsize=(8, 6))  
ax.imshow(gray, cmap=plt.cm.gray)  
ax.set_axis_off()
```



We see that, although all detected objects are indeed faces, one face out of four is not detected. This is probably due to the fact that this face is not perfectly facing the camera, whereas the faces in the training set were. This shows that the efficacy of this method is limited by the quality and generality of the training set.

How it works...

The Viola–Jones object detection framework works by training a cascade of boosted classifiers with Haar-like features. First, we consider a set of features:



Haar-like features

A feature is positioned at a particular location and size in the image. It covers a small window in the image (for example, 24 x 24 pixels). The sum of all pixels in the black area is subtracted to the sum of the pixels in the white area. This operation can be done efficiently with integral images.

Then, the set of all classifiers is trained with a boosting technique; only the best features are kept for the next stage during training. The training set contains positive and negative images (with and without faces). Although the classifiers yield poor performance individually, the cascade of boosted classifiers is both efficient and fast. This method is therefore well-adapted to real-time processing.

The XML file has been obtained in OpenCV's package. There are multiple files corresponding to different training sets. You can also train your own cascade with your own training set.

There's more...

Here are a few references:

- ▶ A cascade tutorial with OpenCV (C++) available at http://docs.opencv.org/doc/tutorials/objdetect/cascade_classifier/cascade_classifier.html
- ▶ Documentation to train a cascade, available at http://docs.opencv.org/doc/user_guide/ug_traincascade.html
- ▶ Haar cascades library, available at <https://github.com/Itseez/opencv/tree/master/data/haarcascades>

- ▶ OpenCV's cascade classification API reference available at http://docs.opencv.org/modules/objdetect/doc/cascade_classification.html
 - ▶ The Viola-Jones object detection framework on Wikipedia, available at https://en.wikipedia.org/wiki/Viola%20%26%20Jones_object_detection_framework
 - ▶ Boosting, or how to create one strong classifier from many weak classifiers, explained at https://en.wikipedia.org/wiki/Boosting_%28machine_learning%29

Applying digital filters to speech sounds

In this recipe, we will show how to play sounds in the Notebook. We will also illustrate the effect of simple digital filters on speech sounds.

Getting ready

You need the pydub package. You can install it with `pip install pydub` or download it from <https://github.com/jiaaro/pydub/>.

This package requires the open source multimedia library FFmpeg for the decompression of MP3 files, available at <http://www.ffmpeg.org>.

How to do it

- ## 1. Let's import the packages:

```
>>> from io import BytesIO
      import tempfile
      import requests
      import numpy as np
      import scipy.signal as sg
      import pydub
      import matplotlib.pyplot as plt
      from IPython.display import Audio, display
      %matplotlib inline
```

2. We create a Python function that loads an MP3 sound and returns a NumPy array with the raw sound data:

```
>>> def speak(data):
    # We convert the mp3 bytes to wav.
    audio = pydub.AudioSegment.from_mp3(BytesIO(data))
```

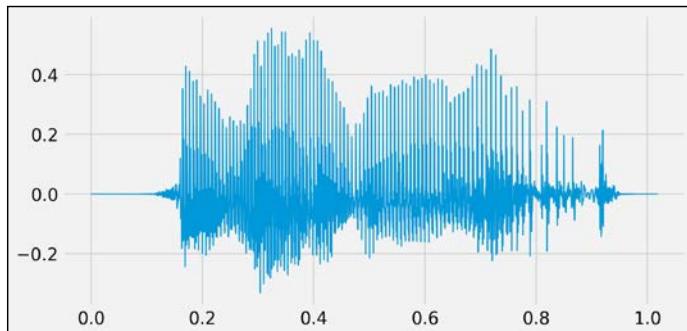
```
with tempfile.TemporaryFile() as fn:  
    wavef = audio.export(fn, format='wav')  
    wavef.seek(0)  
    wave = wavef.read()  
# We get the raw data by removing the 24 first  
# bytes of the header.  
x = np.frombuffer(wave, np.int16)[24:] / 2.**15  
return x, audio.frame_rate
```

3. We create a function that plays a sound (represented by a NumPy vector) in the Notebook, using IPython's Audio class:

```
>>> def play(x, fr, autoplay=False):  
    display(Audio(x, rate=fr, autoplay=autoplay))
```

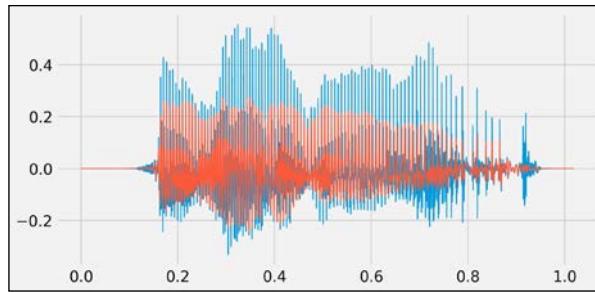
4. Let's play a sound that had been obtained from <http://www.fromtexttospeech.com>:

```
>>> url = ('https://github.com/ipython-books/'  
         'cookbook-2nd-data/blob/master/'  
         'voice.mp3?raw=true')  
voice = requests.get(url).content  
>>> x, fr = speak(voice)  
play(x, fr)  
fig, ax = plt.subplots(1, 1, figsize=(8, 4))  
t = np.linspace(0., len(x) / fr, len(x))  
ax.plot(t, x, lw=1)
```



5. Now, we will hear the effect of a Butterworth low-pass filter applied to this sound (500 Hz cutoff frequency):

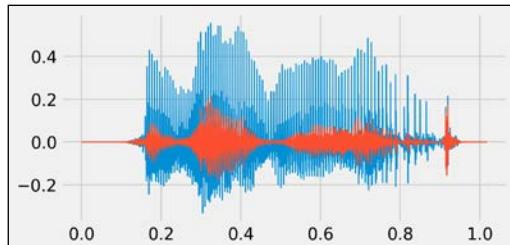
```
>>> b, a = sg.butter(4, 500. / (fr / 2.), 'low')
      x_fil = sg.filtfilt(b, a, x)
>>> play(x_fil, fr)
fig, ax = plt.subplots(1, 1, figsize=(8, 4))
ax.plot(t, x, lw=1)
ax.plot(t, x_fil, lw=1)
```



We hear a muffled voice.

6. Now, with a high-pass filter (1000 Hz cutoff frequency):

```
>>> b, a = sg.butter(4, 1000. / (fr / 2.), 'high')
      x_fil = sg.filtfilt(b, a, x)
>>> play(x_fil, fr)
fig, ax = plt.subplots(1, 1, figsize=(6, 3))
ax.plot(t, x, lw=1)
ax.plot(t, x_fil, lw=1)
```

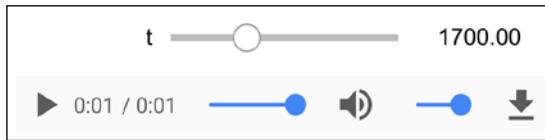


It sounds like a phone call.

- Finally, we can create a simple widget to quickly test the effect of a high-pass filter with an arbitrary cutoff frequency: we get a slider that lets us change the cutoff frequency and hear the effect in real-time.

```
>>> from ipywidgets import widgets

@widgets.interact(t=(100., 5000., 100.))
def highpass(t):
    b, a = sg.butter(4, t / (fr / 2.), 'high')
    x_fil = sg.filtfilt(b, a, x)
    play(x_fil, fr, autoplay=True)
```



How it works...

The human ear can hear frequencies up to 20 kHz. The human voice frequency band ranges from approximately 300 Hz to 3,000 Hz.

Digital filters were described in *Chapter 10, Signal Processing*. The example given here allows us to hear the effect of low- and high-pass filters on sounds.

There's more...

Here are a few references:

- ▶ Audio signal processing on Wikipedia, available at https://en.wikipedia.org/wiki/Audio_signal_processing
- ▶ Audio filters on Wikipedia, available at [https://en.wikipedia.org/wiki/Filter_\(signal_processing\)](https://en.wikipedia.org/wiki/Filter_(signal_processing))
- ▶ Voice frequency on Wikipedia, available at https://en.wikipedia.org/wiki/Voice_frequency
- ▶ PyAudio, an audio Python package that uses the PortAudio library, available at <http://people.csail.mit.edu/hubert/pyaudio/>

See also

- ▶ The *Creating a sound synthesizer in the Notebook* recipe

Creating a sound synthesizer in the Notebook

In this recipe, we will create a small electronic piano in the Notebook. We will synthesize sinusoidal sounds with NumPy instead of using recorded tones.

How to do it...

1. We import the modules:

```
>>> import numpy as np
      import matplotlib.pyplot as plt
      from IPython.display import (
          Audio, display, clear_output)
      from ipywidgets import widgets
      from functools import partial
      %matplotlib inline
```

2. We define the sampling rate and the duration of the notes:

```
>>> rate = 16000.
      duration = .25
      t = np.linspace(
          0., duration, int(rate * duration))
```

3. We create a function that generates and plays the sound of a note (sine function) at a given frequency, using NumPy and IPython's audio class:

```
>>> def synth(f):
      x = np.sin(f * 2. * np.pi * t)
      display(Audio(x, rate=rate, autoplay=True))
```

4. Here is the fundamental 440 Hz note:

```
>>> synth(440)
```



5. Now, we generate the note frequencies of our piano. The chromatic scale is obtained by a geometric progression with the common ratio $2^{1/12}$:

```
>>> notes = 'C,C#,D,D#,E,F,F#,G,G#,A,A#,B,C'.split(',')
      freqs = 440. * 2**np.arange(3, 3 + len(notes)) / 12.)
      notes = list(zip(notes, freqs))
```

- Finally, we create the piano with the Notebook widgets. Each note is a button, and all buttons are contained in a horizontal box container. Clicking on one note plays a sound at the corresponding frequency.

```
>>> layout = widgets.Layout(
    width='30px', height='60px',
    border='1px solid black')

buttons = []
for note, f in notes:
    button = widgets.Button(
        description=note, layout=layout)

    def on_button_clicked(f, b):
        # When a button is clicked, we play the sound
        # in a dedicated Output widget.
        with widgets.Output():
            synth(f)

    button.on_click(partial(on_button_clicked, f))
    buttons.append(button)

# We place all buttons horizontally.
widgets.Box(children=buttons)
```



How it works...

A **pure tone** is a tone with a sinusoidal waveform. It is the simplest way of representing a musical note. A note generated by a musical instrument is typically much more complex. Although the sound contains many frequencies, we generally perceive a musical tone (**fundamental frequency**).

By generating another periodic function instead of a sinusoidal waveform, we would hear the same tone, but a different **timbre**. Electronic music synthesizers are based on this idea.

There's more...

Here are a few references:

- ▶ Synthesizer on Wikipedia, available at <https://en.wikipedia.org/wiki/Synthesizer>
- ▶ Equal temperament on Wikipedia, available at https://en.wikipedia.org/wiki/Equal_temperament
- ▶ Chromatic scale on Wikipedia, available at https://en.wikipedia.org/wiki/Chromatic_scale

See also

- ▶ The *Applying digital filters to speech sounds* recipe

12

Deterministic Dynamical Systems

In this chapter, we will cover the following topics:

- ▶ Plotting the bifurcation diagram of a chaotic dynamical system
- ▶ Simulating an elementary cellular automaton
- ▶ Simulating an ordinary differential equation with SciPy
- ▶ Simulating a partial differential equation — reaction-diffusion systems and Turing patterns

Introduction

The previous chapters dealt with classical approaches in data science: statistics, machine learning, and signal processing. In this chapter and the next chapter, we will cover a different type of approach. Instead of analyzing data directly, we will simulate mathematical models that represent how our data was generated. A representative model gives us an explanation of the real-world processes underlying our data.

Specifically, we will cover a few examples of **dynamical systems**. These mathematical equations describe the evolution of quantities over time and space. They can represent a wide variety of real-world phenomena in physics, chemistry, biology, economics, social sciences, computer science, engineering, and other disciplines.

In this chapter, we will consider *deterministic* dynamical systems. This term is used in contrast to *stochastic* systems, which incorporate randomness in their rules. We will cover stochastic systems in the next chapter.

Types of dynamical systems

The types of deterministic dynamical systems we will consider here are:

- ▶ Discrete-time dynamical systems (iterated functions)
- ▶ Cellular automata
- ▶ **Ordinary Differential Equations (ODEs)**
- ▶ **Partial Differential Equations (PDEs)**

In these models, the quantities of interest depend on one or several **independent variables**. Often, these variables include time and/or space. The independent variables can be discrete or continuous, resulting in different types of models and different analysis and simulation techniques.

A **discrete-time dynamical system** is described by the iterative application of a function on an initial point: $f(x), f(f(x)), f(f(f(x))),$ and so on. This type of system can lead to complex and **chaotic** behaviors.

A **cellular automaton** is represented by a discrete grid of cells that can be in a finite number of states. Rules describe how the state of a cell evolves according to the states of the neighboring cells. These simple models can lead to highly sophisticated behaviors.

An **ODE** describes the dependence of a continuous function on its derivative with respect to the independent variable. In differential equations, the unknown variable is a *function* instead of a *number*. ODEs notably arise when the rate of change of a quantity depends on the current value of this quantity. For example, in classical mechanics, the laws of motion (including the movements of planets and satellites) can be described by ODEs.

PDEs are similar to ODEs, but they involve several independent variables (for example, time and space). These equations contain **partial derivatives** of the function with respect to the different independent variables. For example, PDEs describe the propagation of waves (acoustic, electromagnetic, or mechanical waves) and fluids (**fluid dynamics**). They are also important in quantum mechanics.

Differential equations

ODEs and PDEs can be one-dimensional or multidimensional, depending on the dimensionality of the target space. Systems of multiple differential equations can be seen as multidimensional equations.

The **order** of an ODE or a PDE refers to the maximal derivative order in the equation. For example, a first-order equation only involves simple derivatives, a second-order equation also involves second-order derivatives (the derivatives of the derivatives), and so on.

Ordinary or partial differential equations come with additional rules: **initial** and **boundary conditions**. These formulas describe the behavior of the sought functions on the spatial and temporal domain boundaries. For example, in classical mechanics, boundary conditions include the initial position and initial speed of a physical body subject to forces.

Dynamical systems are often classified between **linear** and **nonlinear systems**, depending on whether the rules are linear or not (with respect to the unknown function). Nonlinear equations are typically much harder to study mathematically and numerically than linear equations. They can lead to extremely complex behaviors.

For example, the **Navier–Stokes equations**, a set of nonlinear PDEs that describe the motion of fluid substances, can lead to **turbulence**, a highly chaotic behavior seen in many fluid flows. Despite their high importance in meteorology, medicine, and engineering, fundamental properties of the Navier-Stokes equations remain unknown at this time. For example, the existence and smoothness problem in three dimensions is one of the seven Clay Mathematics Institute's Millennium Prize Problems. One million dollars is offered to anyone who comes up with a solution.

References

Here are a few references:

- ▶ Overview of dynamical systems on Wikipedia, available at https://en.wikipedia.org/wiki/Dynamical_system
- ▶ Mathematical definition of dynamical systems available at https://en.wikipedia.org/wiki/Dynamical_system_%28definition%29
- ▶ List of dynamical systems topics available at https://en.wikipedia.org/wiki/List_of_dynamical_systems_and_differential_equations_topics
- ▶ Navier-Stokes equations on Wikipedia, available at https://en.wikipedia.org/wiki/Navier%E2%80%93Stokes_equations
- ▶ A course on Computational Fluid Dynamics by Prof. Lorena Barba, written in the Jupyter Notebook, available at <https://github.com/barbagroup/CFDPython>
- ▶ PyDynamical, a Python package for modeling and visualizing discrete dynamical systems, available at <https://pydynamical.readthedocs.io/en/latest/>

Plotting the bifurcation diagram of a chaotic dynamical system

A **chaotic dynamical** system is highly sensitive to initial conditions; small perturbations at any given time yield completely different trajectories. The trajectories of a chaotic system tend to have complex and unpredictable behaviors.

Many real-world phenomena are chaotic, particularly those that involve nonlinear interactions among many agents (complex systems). Examples can be found in meteorology, economics, biology, and other disciplines.

In this recipe, we will simulate a famous chaotic system: the **logistic map**. This is an archetypal example of how chaos can arise from a very simple nonlinear equation. The logistic map models the evolution of a population, taking into account both reproduction and density-dependent mortality (starvation).

We will draw the system's **bifurcation diagram**, which shows the possible long-term behaviors (equilibria, fixed points, periodic orbits, and chaotic trajectories) as a function of the system's parameter. We will also compute an approximation of the system's **Lyapunov exponent**, characterizing the model's sensitivity to initial conditions.

How to do it...

1. Let's import NumPy and Matplotlib:

```
>>> import numpy as np  
      import matplotlib.pyplot as plt  
      %matplotlib inline
```

2. We define the logistic function by:

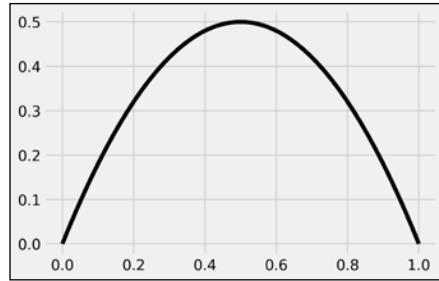
$$f_r(x) = rx(1 - x)$$

Here is the implementation of this function in Python:

```
>>> def logistic(r, x):  
      return r * x * (1 - x)
```

3. Here is a graphic representation of this function

```
>>> x = np.linspace(0, 1)  
      fig, ax = plt.subplots(1, 1)  
      ax.plot(x, logistic(2, x), 'k')
```



4. Our discrete dynamical system is defined by the recursive application of the logistic function:

$$x_{n+1}^{(r)} = f_r(x_n^{(r)}) = rx_n^{(r)}(1 - x_n^{(r)})$$

Let's simulate a few iterations of this system with two different values of r :

```
>>> def plot_system(r, x0, n, ax=None):
    # Plot the function and the
    # y=x diagonal line.
    t = np.linspace(0, 1)
    ax.plot(t, logistic(r, t), 'k', lw=2)
    ax.plot([0, 1], [0, 1], 'k', lw=2)

    # Recursively apply y=f(x) and plot two lines:
    # (x, x) -> (x, y)
    # (x, y) -> (y, y)
    x = x0
    for i in range(n):
        y = logistic(r, x)
        # Plot the two lines.
        ax.plot([x, x], [x, y], 'k', lw=1)
        ax.plot([x, y], [y, y], 'k', lw=1)
        # Plot the positions with increasing
        # opacity.
        ax.plot([x], [y], 'ok', ms=10,
                alpha=(i + 1) / n)
```

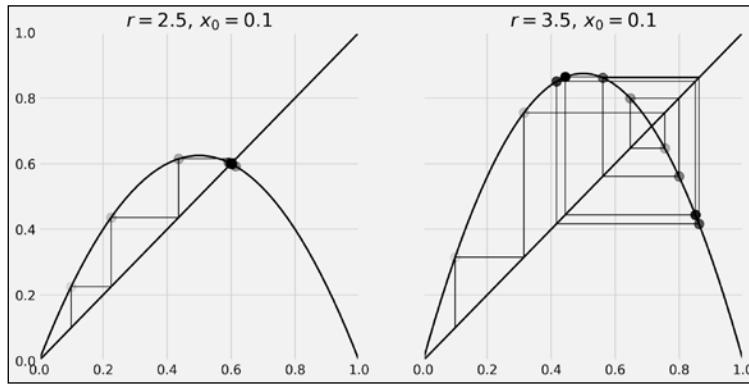
```

x = y

ax.set_xlim(0, 1)
ax.set_ylim(0, 1)
ax.set_title(f"$r={r:.1f}, \ x_0={x0:.1f}$")

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6),
                               sharey=True)
plot_system(2.5, .1, 10, ax=ax1)
plot_system(3.5, .1, 10, ax=ax2)

```



On the left panel, we can see that our system converges to the intersection point of the curve and the diagonal line (fixed point). On the right panel, however, using a different value for r , we observe a seemingly chaotic behavior of the system.

- Now, we simulate this system for 10000 values of r linearly spaced between 2.5 and 4, and vectorize the simulation with NumPy by considering a vector of independent systems (one dynamical system per parameter value):

```
>>> n = 10000
      r = np.linspace(2.5, 4.0, n)
```

- We use 1000 iterations of the logistic map and keep the last 100 iterations to display the bifurcation diagram:

```
>>> iterations = 1000
      last = 100
```

- We initialize our system with the same initial condition $x_0 = 0.00001$:

```
>>> x = 1e-5 * np.ones(n)
```

8. We also compute an approximation of the Lyapunov exponent for every value of r .
The Lyapunov exponent is defined by:

$$\lambda(r) = \lim_{n \rightarrow \infty} \frac{1}{n} \sum_{i=0}^{n-1} \log \left| \frac{df_r}{dx} (x_i^{(r)}) \right|$$

We first initialize the Lyapunov vector:

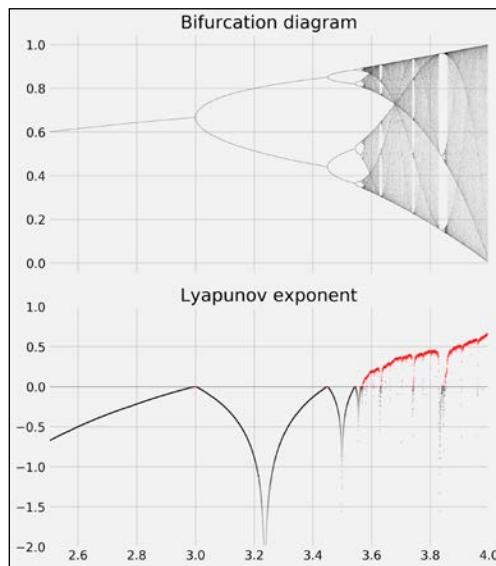
```
>>> lyapunov = np.zeros(n)
```

9. Now, we simulate the system and plot the bifurcation diagram. The simulation only involves the iterative evaluation of the `logistic()` function on our vector x . Then, to display the bifurcation diagram, we draw one pixel per point $x_n^{(r)}$ during the last 100 iterations:

```
>>> fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(8, 9),
                                 sharex=True)
for i in range(iterations):
    x = logistic(r, x)
    # We compute the partial sum of the
    # Lyapunov exponent.
    lyapunov += np.log(abs(r - 2 * r * x))
    # We display the bifurcation diagram.
    if i >= (iterations - last):
        ax1.plot(r, x, ',k', alpha=.25)
ax1.set_xlim(2.5, 4)
ax1.set_title("Bifurcation diagram")

# We display the Lyapunov exponent.
# Horizontal line.
ax2.axhline(0, color='k', lw=.5, alpha=.5)
# Negative Lyapunov exponent.
ax2.plot(r[lyapunov < 0],
          lyapunov[lyapunov < 0] / iterations,
          '.k', alpha=.5, ms=.5)
# Positive Lyapunov exponent.
ax2.plot(r[lyapunov >= 0],
          lyapunov[lyapunov >= 0] / iterations,
          '.r', alpha=.5, ms=.5)
ax2.set_xlim(2.5, 4)
```

```
ax2.set_ylim(-2, 1)
ax2.set_title("Lyapunov exponent")
plt.tight_layout()
```



The bifurcation diagram brings out the existence of a fixed point for $r < 3$, then two and four equilibria, and a chaotic behavior when r belongs to certain areas of the parameter space.

We observe an important property of the Lyapunov exponent: it is positive when the system is chaotic (in red as shown in the preceding diagram).

There's more...

Here are some references:

- ▶ Chaos theory on Wikipedia, available at https://en.wikipedia.org/wiki/Chaos_theory
- ▶ Complex systems on Wikipedia, available at https://en.wikipedia.org/wiki/Complex_system
- ▶ The logistic map on Wikipedia, available at https://en.wikipedia.org/wiki/Logistic_map
- ▶ Iterated functions (discrete dynamical systems) on Wikipedia, available at https://en.wikipedia.org/wiki/Iterated_function
- ▶ Bifurcation diagrams on Wikipedia, available at https://en.wikipedia.org/wiki/Bifurcation_diagram

- ▶ Lyapunov exponent on Wikipedia, available at https://en.wikipedia.org/wiki/Lyapunov_exponent

See also

- ▶ The *Simulating an ordinary differential equation with SciPy* recipe

Simulating an elementary cellular automaton

Cellular automata are discrete dynamical systems evolving on a grid of cells. These cells can be in a finite number of states (for example, on/off). The evolution of a cellular automaton is governed by a set of rules, describing how the state of a cell changes according to the state of its neighbors.

Although extremely simple, these models can initiate highly complex and chaotic behaviors. Cellular automata can model real-world phenomena such as car traffic, chemical reactions, propagation of fire in a forest, epidemic propagations, and much more. Cellular automata are also found in nature. For example, the patterns of some seashells are generated by natural cellular automata.



By Richard Ling (wikipedia@rling.com) - Own work; Location: Cod Hole, Great Barrier Reef, Australia, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=293495>

An **elementary cellular automaton** is a binary, one-dimensional automaton, where the rules concern the immediate left and right neighbors of every cell.

In this recipe, we will simulate elementary cellular automata with NumPy using their Wolfram code.

How to do it...

1. We import NumPy and Matplotlib:

```
>>> import numpy as np
      import matplotlib.pyplot as plt
      %matplotlib inline
```

2. We will use the following vector to obtain numbers written in binary representation:

```
>>> u = np.array([[4], [2], [1]])
```

3. Let's write a function that performs an iteration on the grid, updating all cells at once according to the given rule in binary representation (we will give all explanations in the *How it works...* section of this recipe). The first step consists of stacking circularly-shifted versions of the grid to get the **LCR (left, center, right)** triplets of each cell (y). Then, we convert these triplets into 3-bit numbers (z). Finally, we compute the next state of every cell using the specified rule:

```
>>> def step(x, rule_b):
    """Compute a single stet of an elementary cellular
    automaton."""
    # The columns contains the L, C, R values
    # of all cells.
    y = np.vstack((np.roll(x, 1), x,
                   np.roll(x, -1))).astype(np.int8)
    # We get the LCR pattern numbers between 0 and 7.
    z = np.sum(y * u, axis=0).astype(np.int8)
    # We get the patterns given by the rule.
    return rule_b[7 - z]
```

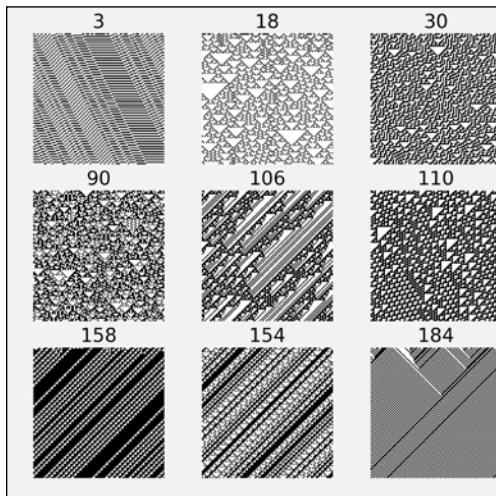
4. We now write a function that simulates any elementary cellular automaton. First, we compute the binary representation of the rule (**Wolfram Code**). Then, we initialize the first row of the grid with random values. Finally, we apply the function `step()` iteratively on the grid:

```
>>> def generate(rule, size=100, steps=100):
    """Simulate an elementary cellular automaton given
    its rule (number between 0 and 255)."""
    # Compute the binary representation of the rule.
    rule_b = np.array(
        [int(_) for _ in np.binary_repr(rule, 8)],
        dtype=np.int8)
    x = np.zeros((steps, size), dtype=np.int8)
    # Random initial state.
    x[0, :] = np.random.rand(size) < .5
    # Apply the step function iteratively.
```

```
for i in range(steps - 1):
    x[i + 1, :] = step(x[i, :], rule_b)
return x
```

5. Now, we simulate and display nine different automata:

```
>>> fig, axes = plt.subplots(3, 3, figsize=(8, 8))
rules = [3, 18, 30,
         90, 106, 110,
         158, 154, 184]
for ax, rule in zip(axes.flat, rules):
    x = generate(rule)
    ax.imshow(x, interpolation='none',
              cmap=plt.cm.binary)
    ax.set_axis_off()
    ax.set_title(str(rule))
```



How it works...

Let's consider an elementary cellular automaton in one dimension. Every cell C has two neighbors (L and R), and it can be either off (0) or on (1). Therefore, the future state of a cell depends on the current state of L , C , and R . This triplet can be encoded as a number between 0 and 7 (three digits in binary representation).

A particular elementary cellular automaton is entirely determined by the outcome of each of these eight configurations. Therefore, there are 256 different elementary cellular automata (2^8). Each of these automata is identified by a number between 0 and 255.

We consider all eight LCR states in order: 111, 110, 101, ..., 001, 000. Each of the eight digits in the binary representation of the automaton's number corresponds to a LCR state (using the same order). For example, in the **Rule 110 automaton** (01101110 in binary representation), the state 111 yields a new state of 0 for the center cell, 110 yields 1, 101 yields 1, and so on. It has been shown that this particular automaton is **Turing complete** (or **universal**); it can theoretically simulate any computer program.

There's more...

Other types of cellular automata include **Conway's Game of Life**, in two dimensions. This famous system yields various dynamic patterns. It is also Turing complete.

Here are a few references:

- ▶ Cellular automata on Wikipedia, available at https://en.wikipedia.org/wiki/Cellular_automaton
- ▶ Elementary cellular automata on Wikipedia, available at https://en.wikipedia.org/wiki/Elementary_cellular_automaton
- ▶ Rule 110, described at https://en.wikipedia.org/wiki/Rule_110
- ▶ The Wolfram code, explained at https://en.wikipedia.org/wiki/Wolfram_code, assigns a 1D elementary cellular automaton to any number between 0 and 255
- ▶ Conway's Game of Life on Wikipedia, available at https://en.wikipedia.org/wiki/Conway's_Game_of_Life
- ▶ A computer implemented in Conway's Game of Life, at <https://codegolf.stackexchange.com/questions/11880/build-a-working-game-of-tetris-in-conways-game-of-life>

Simulating an ordinary differential equation with SciPy

Ordinary Differential Equations (ODEs) describe the evolution of a system subject to internal and external dynamics. Specifically, an ODE links a quantity depending on a single independent variable (time, for example) to its derivatives. In addition, the system can be under the influence of external factors. A first-order ODE can typically be written as:

$$y'(t) = f(t, y(t))$$

More generally, an n -th order ODE involves successive derivatives of y until the order n . The ODE is said to be linear or nonlinear depending on whether f is linear in y or not.

ODEs naturally appear when the rate of change of a quantity depends on its value. Therefore, ODEs are found in many scientific disciplines such as mechanics (evolution of a body subject to dynamic forces), chemistry (concentration of reacting products), biology (spread of an epidemic), ecology (growth of a population), economics, and finance, among others.

Whereas simple ODEs can be solved analytically, many ODEs require a numerical treatment. In this recipe, we will simulate a simple linear second-order autonomous ODE, describing the evolution of a particle in the air subject to gravity and viscous resistance. Although this equation could be solved analytically, here we will use SciPy to simulate it numerically.

How to do it...

1. Let's import NumPy, SciPy (the **integrate** package), and Matplotlib:

```
>>> import numpy as np
      import scipy.integrate as spi
      import matplotlib.pyplot as plt
      %matplotlib inline
```

2. We define a few parameters appearing in our model:

```
>>> m = 1. # particle's mass
      k = 1. # drag coefficient
      g = 9.81 # gravity acceleration
```

3. We have two variables: x and y (two dimensions). We note $u = (x, y)$. The ODE that we are going to simulate is:

$$u'' = -\frac{k}{m}u' + g$$

Here, g is the gravity acceleration vector.

In order to simulate this second-order ODE with SciPy, we can convert it to a first-order ODE (another option would be to solve u' first before integrating the solution). To do this, we consider two 2D variables: u and u' . We note $v = (u, u')$. We can express v' as a function of v . Now, we create the initial vector v_0 at time $t = 0$: it has four components.

```
>>> # The initial position is (0, 0).
      v0 = np.zeros(4)
      # The initial speed vector is oriented
      # to the top right.
      v0[2] = 4.
      v0[3] = 10.
```

4. Let's create a Python function f that takes the current vector $v(t_0)$ and a time t_0 as arguments (with optional parameters) and that returns the derivative $v'(t_0)$:

```
>>> def f(v, t0, k):
    # v has four components: v=[u, u'].
    u, udot = v[:2], v[2:]
    # We compute the second derivative u'' of u.
    uddot = -k / m * udot
    uddot[1] -= g
    # We return v'=[u', u''].
    return np.r_[udot, uddot]
```

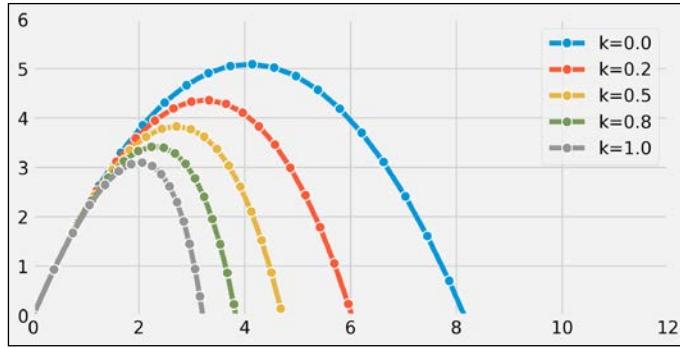
5. Now, we simulate the system for different values of k . We use the SciPy `odeint()` function, defined in the `scipy.integrate` package.

 Starting with SciPy 1.0, the generic `scipy.integrate.solve_ivp()` function can be used instead of the old function `odeint()`

```
>>> fig, ax = plt.subplots(1, 1, figsize=(8, 4))

# We want to evaluate the system on 30 linearly
# spaced times between t=0 and t=3.
t = np.linspace(0., 3., 30)

# We simulate the system for different values of k.
for k in np.linspace(0., 1., 5):
    # We simulate the system and evaluate $v$ on the
    # given times.
    v = spi.odeint(f, v0, t, args=(k,))
    # We plot the particle's trajectory.
    ax.plot(v[:, 0], v[:, 1], 'o-', mew=1, ms=8,
            mec='w', label=f'k={k:.1f}')
ax.legend()
ax.set_xlim(0, 12)
ax.set_ylim(0, 6)
```



In the preceding figure, the most outward trajectory (blue) corresponds to drag-free motion (without air resistance). It is a parabola. In the other trajectories, we can observe the increasing effect of air resistance, parameterized with k .

How it works...

Let's explain how we obtained the differential equation from our model. Let $u = (x, y)$ encode the 2D position of our particle with mass m . This particle is subject to two forces: gravity $mg = (0, -9.81 \cdot m)$ and air drag $F = -ku'$. This last term depends on the particle's speed and is only valid at low speed. With higher speeds, we need to use more complex nonlinear expressions.

Now, we use **Newton's second law of motion** in classical mechanics. This law states that, in an inertial reference frame, the mass multiplied by the acceleration of the particle is equal to the sum of all forces applied to that particle. Here, we obtain:

$$m \cdot u'' = F + mg$$

We immediately obtain our second-order ODE:

$$u'' = -\frac{k}{m}u' + g$$

We transform it into a single-order system of ODEs, with $v = (u, u')$:

$$v' = (u', u'') = (u', -\frac{k}{m}u' + g)$$

The last term can be expressed as a function of v only.

The SciPy `odeint()` function is a black-box solver; we simply specify the function that describes the system, and SciPy solves it automatically. This function leverages the Fortran library ODEPACK, which contains well-tested code that has been used for decades by many scientists and engineers.

The newer `solve_ivb()` function offers a common API for Python implementations of various ODE solvers.

An example of a simple numerical solver is the **Euler method**. To numerically solve the autonomous ODE $y' = f(y)$, the method consists of discretizing time with a time step dt and replacing y' with a first-order approximation:

$$y'(t) \simeq \frac{y(t + dt) - y(t)}{dt}$$

Then, starting from an initial condition $y_0 = y(t_0)$, the method evaluates y successively with the following recurrence relation:

$$y_{n+1} = y_n + dt \cdot f(y_n) \quad \text{with } t = n \cdot dt, \quad y_n = y(n \cdot dt)$$

There's more...

Here are a few references:

- ▶ The documentation of the `integrate` package in SciPy available at <http://docs.scipy.org/doc/scipy/reference/integrate.html>
- ▶ The new `solve_ivp()` function, available in SciPy 1.0 and later, at https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html
- ▶ ODEs on Wikipedia, available at https://en.wikipedia.org/wiki/Ordinary_differential_equation
- ▶ ODEs lectures on Awesome Math, at <https://github.com/rossant/awesome-math/#ordinary-differential-equations>
- ▶ Newton's laws of motion on Wikipedia, available at https://en.wikipedia.org/wiki/Newton's_laws_of_motion
- ▶ Air resistance on Wikipedia, available at https://en.wikipedia.org/wiki/Drag_%28physics%29
- ▶ Some numerical methods for ODEs described at https://en.wikipedia.org/wiki/Numerical_methods_for_ordinary_differential_equations
- ▶ The Euler method on Wikipedia, available at https://en.wikipedia.org/wiki/Euler_method

- ▶ Documentation of the ODEPACK package in Fortran available at <http://www.netlib.org/odepack/opks-sum>

See also

- ▶ The *Plotting the bifurcation diagram of a chaotic dynamical system* recipe

Simulating a partial differential equation — reaction-diffusion systems and Turing patterns

Partial Differential Equations (PDEs) describe the evolution of dynamical systems involving both time and space. Examples in physics include sound, heat, electromagnetism, fluid flow, and elasticity, among others. Examples in biology include tumor growth, population dynamics, and epidemic propagations.

PDEs are hard to solve analytically. Therefore, PDEs are often studied via numerical simulations.

In this recipe, we will illustrate how to simulate a **reaction-diffusion system** described by a PDE called the **FitzHugh–Nagumo equation**. A reaction-diffusion system models the evolution of one or several variables subject to two processes: reaction (transformation of the variables into each other) and diffusion (spreading across a spatial region). Some chemical reactions can be described by this type of model, but there are other applications in physics, biology, ecology, and other disciplines.

Here, we simulate a system that has been proposed by Alan Turing as a model of animal coat pattern formation. Two chemical substances influencing skin pigmentation interact according to a reaction-diffusion model. This system is responsible for the formation of patterns that are reminiscent of the pelage of zebras, jaguars, and giraffes.

We will simulate this system with the finite difference method. This method consists of discretizing time and space and replacing the derivatives with their discrete equivalents.

How to do it...

1. Let's import the packages:

```
>>> import numpy as np
      import matplotlib.pyplot as plt
      %matplotlib inline
```

2. We will simulate the following system of partial differential equations on the domain $E = [-1, 1]^2$:

$$\begin{aligned}\frac{\partial u}{\partial t} &= a\Delta u + u - u^3 - v + k \\ \tau \frac{\partial v}{\partial t} &= b\Delta v + u - v\end{aligned}$$

The variable u represents the concentration of a substance favoring skin pigmentation, whereas v represents another substance that reacts with the first and impedes pigmentation.

At initialization time, we assume that u and v contain independent random numbers on every grid point. We also take Neumann boundary conditions: we require the spatial derivatives of the variables with respect to the normal vectors to be null on the domain's boundaries.

3. Let's define the four parameters of the model:

```
>>> a = 2.8e-4
      b = 5e-3
      tau = .1
      k = -.005
```

4. We discretize time and space. The time step dt must be small enough to ensure the stability of the numerical simulation:

```
>>> size = 100 # size of the 2D grid
      dx = 2. / size # space step
>>> T = 9.0 # total time
      dt = .001 # time step
      n = int(T / dt) # number of iterations
```

5. We initialize the variables u and v . The matrices U and V contain the values of these variables on the vertices of the 2D grid. These variables are initialized with a uniform noise between 0 and 1:

```
>>> U = np.random.rand(size, size)
      V = np.random.rand(size, size)
```

6. Now, we define a function that computes the discrete Laplace operator of a 2D variable on the grid, using a five-point stencil finite difference method. This operator is defined by:

$$\Delta u(x, y) \simeq \frac{u(x+h, y) + u(x-h, y) + u(x, y+h) + u(x, y-h) - 4u(x, y)}{dx^2}$$

We can compute the values of this operator on the grid using vectorized matrix operations. Because of side effects on the edges of the matrix, we need to remove the borders of the grid in the computation:

```
>>> def laplacian(Z):
    Ztop = Z[0:-2, 1:-1]
    Zleft = Z[1:-1, 0:-2]
    Zbottom = Z[2:, 1:-1]
    Zright = Z[1:-1, 2:]
    Zcenter = Z[1:-1, 1:-1]
    return (Ztop + Zleft + Zbottom + Zright -
           4 * Zcenter) / dx**2
```

7. We define a function that displays the matrix:

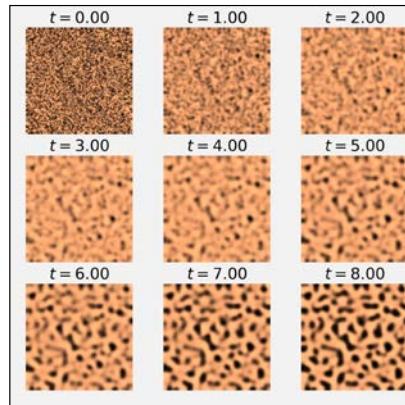
```
>>> def show_patterns(U, ax=None):
    ax.imshow(U, cmap=plt.cm.copper,
              interpolation='bilinear',
              extent=[-1, 1, -1, 1])
    ax.set_axis_off()
```

8. Now, we simulate the system of equations using the finite difference method. At each time step, we compute the right-hand sides of the two equations on the grid using discrete spatial derivatives (Laplacians). Then, we update the variables using a discrete time derivative. We also show the evolution of the system at 9 different steps:

```
>>> fig, axes = plt.subplots(3, 3, figsize=(8, 8))
step_plot = n // 9
# We simulate the PDE with the finite difference
# method.
for i in range(n):
    # We compute the Laplacian of u and v.
    deltaU = laplacian(U)
    deltaV = laplacian(V)
    # We take the values of u and v inside the grid.
    Uc = U[1:-1, 1:-1]
    Vc = V[1:-1, 1:-1]
    # We update the variables.
    U[1:-1, 1:-1], V[1:-1, 1:-1] = \
        Uc + dt * (a * deltaU + Uc - Uc**3 - Vc + k), \
        Vc + dt * (b * deltaV + Uc - Vc) / tau
    # Neumann conditions: derivatives at the edges
    # are null.
    for Z in (U, V):
        Z[0, :] = Z[1, :]
        Z[-1, :] = Z[-2, :]
```

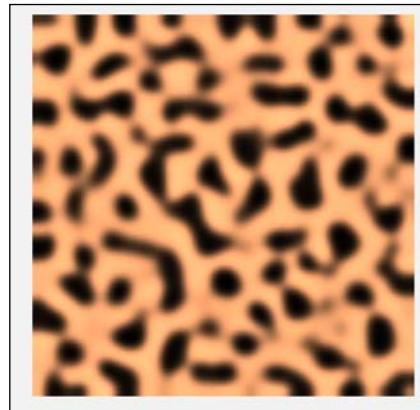
```
Z[:, 0] = Z[:, 1]
Z[:, -1] = Z[:, -2]

# We plot the state of the system at
# 9 different times.
if i % step_plot == 0 and i < 9 * step_plot:
    ax = axes.flat[i // step_plot]
    show_patterns(U, ax=ax)
    ax.set_title(f't={i * dt:.2f}')
```



9. Finally, we show the state of the system at the end of the simulation:

```
>>> fig, ax = plt.subplots(1, 1, figsize=(8, 8))
      show_patterns(U, ax=ax)
```



Whereas the variables were completely random at initialization time, we observe the formation of patterns after a sufficiently long simulation time.

How it works...

Let's explain how the finite difference method allowed us to implement the update step. We start from the following system of equations:

$$\begin{aligned}\frac{\partial u}{\partial t}(t; x, y) &= a\Delta u(t; x, y) + u(t; x, y) - u(t; x, y)^3 - v(t; x, y) + k \\ \tau \frac{\partial v}{\partial t}(t; x, y) &= b\Delta v(t; x, y) + u(t; x, y) - v(t; x, y)\end{aligned}$$

We first use the following scheme for the discrete Laplace operator:

$$\Delta u(x, y) \simeq \frac{u(x+h, y) + u(x-h, y) + u(x, y+h) + u(x, y-h) - 4u(x, y)}{dx^2}$$

We also use this scheme for the time derivative of u and v :

$$\frac{\partial u}{\partial t}(t; x, y) \simeq \frac{u(t+dt; x, y) - u(t; x, y)}{dt}$$

We end up with the following iterative update step:

$$\begin{aligned}u(t+dt; x, y) &= u(t; x, y) + dt(a\Delta u(t; x, y) + u(t; x, y) - u(t; x, y)^3 - v(t; x, y) + k) \\ v(t+dt; x, y) &= v(t; x, y) + \frac{dt}{\tau}(b\Delta v(t; x, y) + u(t; x, y) - v(t; x, y))\end{aligned}$$

Here, our Neumann boundary conditions state that the spatial derivatives with respect to the normal vectors are null on the boundaries of the domain E :

$$\begin{aligned}\forall w \in \{u, v\}, \forall t \geq 0, \forall x, y \in \partial E : \\ \frac{\partial w}{\partial x}(t; -1, y) = \frac{\partial w}{\partial x}(t; 1, y) = \frac{\partial w}{\partial y}(t; x, -1) = \frac{\partial w}{\partial y}(t; x, 1) = 0\end{aligned}$$

We implement these boundary conditions by duplicating values in matrices U and V on the edges (see the preceding code).

There's more...

Here are further references on partial differential equations, reaction-diffusion systems, and numerical simulations of those systems:

- ▶ Partial differential equations on Wikipedia, available at https://en.wikipedia.org/wiki/Partial_differential_equation

- ▶ Partial differential equations lectures on Awesome Math, at <https://github.com/rossant/awesome-math/#partial-differential-equations>
- ▶ Reaction-diffusion systems on Wikipedia, available at https://en.wikipedia.org/wiki/Reaction-diffusion_system
- ▶ FitzHugh-Nagumo system on Wikipedia, available at https://en.wikipedia.org/wiki/FitzHugh-Nagumo_equation
- ▶ Neumann boundary conditions on Wikipedia, available at https://en.wikipedia.org/wiki/Neumann_boundary_condition
- ▶ A course on *Computational Fluid Dynamics* by Prof. Lorena Barba, written in the Jupyter Notebook, available at <https://github.com/barbagroup/CFDPython>

13

Stochastic Dynamical Systems

In this chapter, we will cover the following topics:

- ▶ Simulating a discrete-time Markov chain
- ▶ Simulating a Poisson process
- ▶ Simulating a Brownian motion
- ▶ Simulating a stochastic differential equation

Introduction

Stochastic dynamical systems are dynamical systems subjected to the effect of noise. The randomness brought by the noise takes into account the variability observed in real-world phenomena. For example, the evolution of a share price typically exhibits long-term behaviors along with faster, smaller-amplitude oscillations, reflecting day-to-day or hour-to-hour variations.

Applications of stochastic systems to data science include methods for statistical inference (such as Markov chain Monte Carlo) and stochastic modeling for time series or geospatial data.

Stochastic discrete-time systems include discrete-time **Markov chains**. The **Markov property** means that the state of a system at time $n + 1$ only depends on its state at time n . **Stochastic cellular automata**, which are stochastic extensions of cellular automata, are particular Markov chains.

As far as continuous-time systems are concerned, Ordinary Differential Equations with noise yield **Stochastic Differential Equations (SDEs)**. Partial Differential Equations with noise yield **Stochastic Partial Differential Equations (SPDEs)**.

Point processes are another type of stochastic process. These processes model the random occurrence of instantaneous events over time (arrival of customers in a queue or action potentials in the nervous system) or space (locations of trees in a forest, cities in a territory, or stars in the sky).

Mathematically, the theory of stochastic dynamical systems is based on probability theory and measure theory. The study of continuous-time stochastic systems builds upon stochastic calculus, an extension of infinitesimal calculus (including derivatives and integrals) to stochastic processes.

In this chapter, we will see how to simulate different kinds of stochastic systems with Python.

References

Here are a few references on the subject:

- ▶ An overview of stochastic dynamical systems, available at http://www.scholarpedia.org/article/Stochastic_dynamical_systems
- ▶ The Markov property on Wikipedia, available at https://en.wikipedia.org/wiki/Markov_property
- ▶ Stochastic processes on awesome Math, at <https://github.com/rossant/awesome-math/#stochastic-processes>

Simulating a discrete-time Markov chain

Discrete-time Markov chains are stochastic processes that undergo transitions from one state to another in a state space. Transitions occur at every time step. Markov chains are characterized by their lack of memory in that the probability to undergo a transition from the current state to the next depends only on the current state, not the previous ones. These models are widely used in scientific and engineering applications.

Continuous-time Markov processes also exist and we will cover particular instances later in this chapter.

Markov chains are relatively easy to study mathematically and to simulate numerically. In this recipe, we will simulate a simple Markov chain modeling the evolution of a population.

How to do it...

1. Let's import NumPy and Matplotlib:

```
>>> import numpy as np  
      import matplotlib.pyplot as plt  
      %matplotlib inline
```

2. We consider a population that cannot comprise more than $N = 100$ individuals, and define the birth and death rates:

```
>>> N = 100 # maximum population size  
a = .5 / N # birth rate  
b = .5 / N # death rate
```

3. We simulate a Markov chain on the finite space $0, 1, \dots, N$. Each state represents a population size. The x vector will contain the population size at each time step. We set the initial state to $x_0 = 25$ (that is, there are 25 individuals in the population at initialization time):

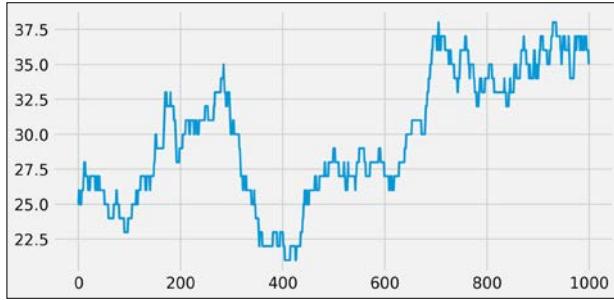
```
>>> nsteps = 1000  
x = np.zeros(nsteps)  
x[0] = 25
```

4. Now we simulate our chain. At each time step t , there is a new birth with probability ax_t , and independently, there is a new death with probability bx_t . These probabilities are proportional to the size of the population at that time. If the population size reaches 0 or N , the evolution stops:

```
>>> for t in range(nsteps - 1):  
    if 0 < x[t] < N - 1:  
        # Is there a birth?  
        birth = np.random.rand() <= a * x[t]  
        # Is there a death?  
        death = np.random.rand() <= b * x[t]  
        # We update the population size.  
        x[t + 1] = x[t] + 1 * birth - 1 * death  
    # The evolution stops if we reach $0$ or $N$.  
    else:  
        x[t + 1] = x[t]
```

5. Let's look at the evolution of the population size:

```
>>> fig, ax = plt.subplots(1, 1, figsize=(8, 4))
ax.plot(x, lw=2)
```



We see that, at every time step, the population size can stay stable, increase, or decrease by 1.

6. Now, we will simulate many independent trials of this Markov chain. We could run the previous simulation with a loop, but it would be very slow (two nested `for` loops). Instead, we vectorize the simulation by considering all independent trials at once. There is a single loop over time. At every time step, we update all trials simultaneously with vectorized operations on vectors. The `x` vector now contains the population size of all trials, at a particular time. At initialization time, the population sizes are set to random numbers between 0 and N:

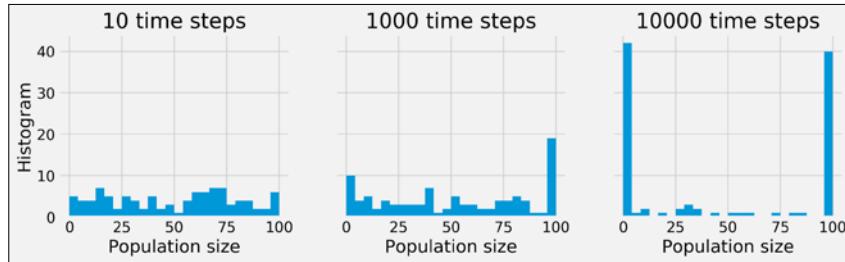
```
>>> ntrials = 100
x = np.random.randint(size=ntrials,
                      low=0, high=N)
```

7. We define a function that performs the simulation. At every time step, we find the trials that undergo births and deaths by generating random vectors, and we update the population sizes with vector operations:

```
>>> def simulate(x, nsteps):
    """Run the simulation."""
    for _ in range(nsteps - 1):
        # Which trials to update?
        upd = (0 < x) & (x < N - 1)
        # In which trials do births occur?
        birth = 1 * (np.random.rand(ntrials) <= a * x)
        # In which trials do deaths occur?
        death = 1 * (np.random.rand(ntrials) <= b * x)
        # We update the population size for all trials
        x[upd] += birth[upd] - death[upd]
```

8. Now, let's look at the histograms of the population size at different times. These histograms represent the probability distribution of the Markov chain, estimated with independent trials (the Monte Carlo method):

```
>>> bins = np.linspace(0, N, 25)
>>> nsteps_list = [10, 1000, 10000]
    fig, axes = plt.subplots(1, len(nsteps_list),
                           figsize=(12, 3),
                           sharey=True)
    for i, nsteps in enumerate(nsteps_list):
        ax = axes[i]
        simulate(x, nsteps)
        ax.hist(x, bins=bins)
        ax.set_xlabel("Population size")
        if i == 0:
            ax.set_ylabel("Histogram")
        ax.set_title(f"{nsteps} time steps")
```



Whereas, initially, the population sizes look uniformly distributed between 0 and N , they appear to converge to 0 or N after a sufficiently long time. This is because the states 0 and N are absorbing; once reached, the chain cannot leave these states. Furthermore, these states can be reached from any other state.

How it works...

Mathematically, a discrete-time Markov chain on a space E is a sequence of random variables X_1, X_2, \dots that satisfy the Markov property:

$$\forall n \geq 1, \quad P(X_{n+1} | X_1, X_2, \dots, X_n) = P(X_{n+1} | X_n)$$

A (stationary) Markov chain is characterized by the probability of transitions $P(X_j | X_i)$. These values form a matrix called the **transition matrix**. This matrix is the adjacency matrix of a directed graph called the **state diagram**. Every node is a state, and the node i is connected to the node j if the chain has a non-zero probability of transition between these nodes.

There's more...

Simulating a single Markov chain in Python is not particularly efficient because we need a `for` loop. However, simulating many independent chains following the same process can be made efficient with vectorization and parallelization (all tasks are independent, thus the problem is **embarrassingly parallel**). This is useful when we are interested in statistical properties of the chain (example of the Monte Carlo method).

There is vast literature on Markov chains. Many theoretical results can be established with linear algebra and probability theory.

Many generalizations of discrete-time Markov chains exist. Markov chains can be defined on infinite state spaces, or with a continuous time. Also, the Markov property is important in a broad class of stochastic processes.

Here are a few references:

- ▶ Markov chains on Wikipedia, available at https://en.wikipedia.org/wiki/Markov_chain
- ▶ Absorbing Markov chains on Wikipedia, available at https://en.wikipedia.org/wiki/Absorbing_Markov_chain
- ▶ Monte Carlo methods on Wikipedia, available at https://en.wikipedia.org/wiki/Monte_Carlo_method

See also

- ▶ The *Simulating a Brownian motion* recipe

Simulating a Poisson process

A **Poisson process** is a particular type of **point process**, a stochastic model that represents random occurrences of instantaneous events. Roughly speaking, the Poisson process is the least structured, or the most random, point process.

The Poisson process is a particular continuous-time Markov process.

Point processes, and notably Poisson processes, can model random instantaneous events such as the arrival of clients in a queue or on a server, telephone calls, radioactive disintegrations, action potentials of nerve cells, and many other phenomena.

In this recipe, we will show different methods to simulate a homogeneous stationary Poisson process.

How to do it...

1. Let's import NumPy and Matplotlib:

```
>>> import numpy as np  
      import matplotlib.pyplot as plt  
      %matplotlib inline
```

2. Let's specify the `rate` value, that is, the average number of events per second:

```
>>> rate = 20. # average number of events per second
```

3. First, we will simulate the process using small time bins of 1 millisecond:

```
>>> dt = .001 # time step  
n = int(1. / dt) # number of time steps
```

4. On every time bin, the probability that an event occurs is about `rate * dt` if `dt` is small enough. Besides, as the Poisson process has no memory, the occurrence of an event is independent from one bin to another. Therefore, we can sample Bernoulli random variables (either 1 or 0, respectively representing an experiment's success or failure) in a vectorized way in order to simulate our process:

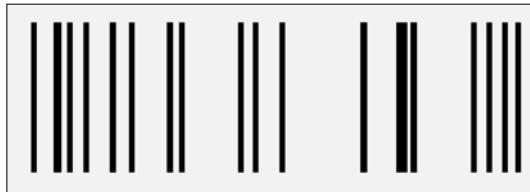
```
>>> x = np.zeros(n)  
x[np.random.rand(n) <= rate * dt] = 1
```

The `x` vector contains zeros and ones on all time bins, 1 corresponding to the occurrence of an event:

```
>>> x[:10]  
array([ 1.,  0.,  ...,  0.,  0.])
```

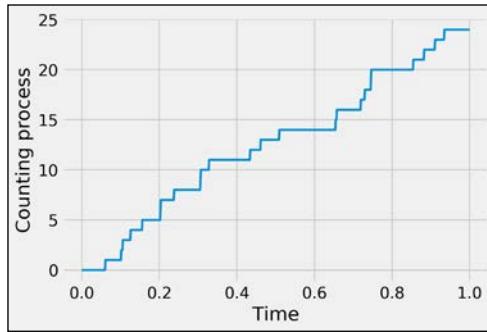
5. Let's display the simulated process. We draw a vertical line for each event:

```
>>> fig, ax = plt.subplots(1, 1, figsize=(6, 2))  
ax.vlines(np.nonzero(x)[0], 0, 1)  
ax.set_axis_off()
```



6. Another way of representing that same object is by considering the associated **counting process** $N(t)$, which is the number of events that have occurred until time t . Here, we can display this process using the `cumsum()` function:

```
>>> fig, ax = plt.subplots(1, 1, figsize=(6, 4))
    ax.plot(np.linspace(0., 1., n),
            np.cumsum(x), lw=2)
    ax.set_xlabel("Time")
    ax.set_ylabel("Counting process")
```



7. The other (and more efficient) way of simulating the homogeneous Poisson process is to use the property that the time intervals between two successive events follow an exponential distribution. Furthermore, these intervals are independent. Thus, we can sample them in a vectorized way. Finally, we get our process by cumulatively summing all of these intervals:

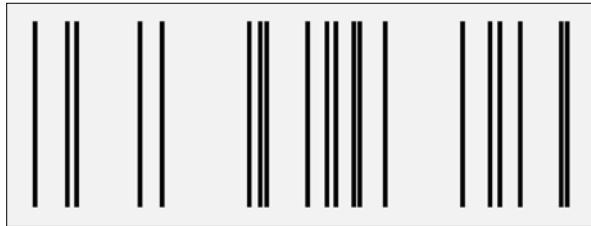
```
>>> y = np.cumsum(np.random.exponential(1. / rate,
                                         size=int(rate)))
```

The `y` vector contains another realization of our Poisson process, but the data structure is different. Every component of the vector is an event time:

```
>>> y[:10]
array([ 0.021,  0.072,  0.087,  0.189,  0.224,
        0.365,  0.382,  0.392,  0.458,  0.489])
```

8. Finally, let's display the simulated process:

```
>>> fig, ax = plt.subplots(1, 1, figsize=(8, 3))
    ax.vlines(y, 0, 1)
    ax.set_axis_off()
```



How it works...

For a Poisson process with rate λ , the number of events in a time window of length τ follows a Poisson distribution:

$$\forall k \geq 0, \quad P[N(t + \tau) - N(t) = k] = e^{-\lambda\tau} \frac{(\lambda\tau)^k}{k!}$$

When $\tau = dt$ is small, we can show that, at first order, this probability is about $\lambda\tau$.

Also, the **holding times** (delays between two consecutive events) are independent and follow an exponential distribution. The Poisson process satisfies other useful properties, such as the independent and stationary increments. This property justifies the first simulation method used in this recipe.

There's more...

In this recipe, we only considered homogeneous time-dependent Poisson processes. Other types of Poisson processes include inhomogeneous (or non-homogeneous) processes that are characterized by a time-varying rate, and multidimensional spatial Poisson processes.

Here are further references:

- ▶ The Poisson process on Wikipedia, available at https://en.wikipedia.org/wiki/Poisson_process
- ▶ Point processes on Wikipedia, available at https://en.wikipedia.org/wiki/Point_process
- ▶ Renewal theory on Wikipedia, available at https://en.wikipedia.org/wiki/Renewal_theory
- ▶ Spatial Poisson processes on Wikipedia, available at https://en.wikipedia.org/wiki/Spatial_Poisson_process

See also

- ▶ The *Simulating a discrete-time Markov chain* recipe

Simulating a Brownian motion

The **Brownian motion** (or **Wiener process**) is a fundamental object in mathematics, physics, and many other scientific and engineering disciplines. This model describes the movement of a particle suspended in a fluid resulting from random collisions with the quick molecules in the fluid (diffusion). More generally, the Brownian motion models a continuous-time random walk, where a particle evolves in space by making independent random steps in all directions.

Mathematically, the Brownian motion is a particular Markov continuous stochastic process. The Brownian motion is at the core of mathematical domains such as stochastic calculus and the theory of stochastic processes, but it is also central in applied fields such as quantitative finance, ecology, and neuroscience.

In this recipe, we will show how to simulate and plot a Brownian motion in two dimensions.

How to do it...

1. Let's import NumPy and Matplotlib:

```
>>> import numpy as np
      import matplotlib.pyplot as plt
      %matplotlib inline
```

2. We simulate Brownian motions with 5000 time steps:

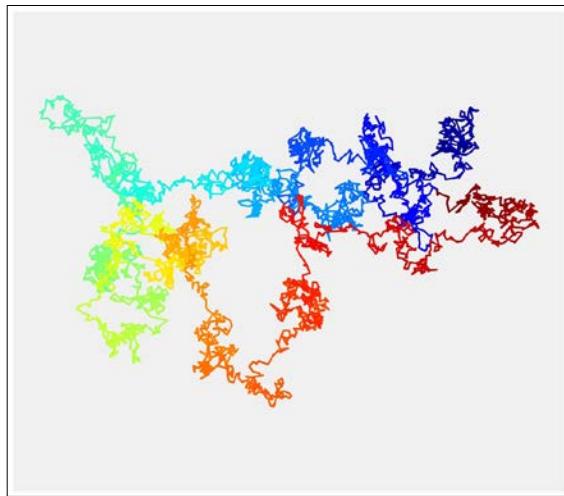
```
>>> n = 5000
```

3. We simulate two independent one-dimensional Brownian processes to form a single two-dimensional Brownian process. The (discrete) Brownian motion makes independent Gaussian jumps at each time step. Therefore, we merely have to compute the cumulative sum of independent normal random variables (one for each time step):

```
>>> x = np.cumsum(np.random.randn(n))
      y = np.cumsum(np.random.randn(n))
```

4. Now, to display the Brownian motion, we could just use `plot(x, y)`. However, the result would be monochromatic and a bit boring. We would like to use a gradient of color to illustrate the progression of the motion in time (the hue is a function of time). matplotlib does not support this feature natively, so instead we use `scatter()`. This function allows us to assign a different color to each point at the expense of dropping out line segments between points. To work around this issue, we linearly interpolate the process to give the illusion of a continuous line:

```
>>> # We add 10 intermediary points between two
# successive points. We interpolate x and y.
k = 10
x2 = np.interp(np.arange(n * k), np.arange(n) * k, x)
y2 = np.interp(np.arange(n * k), np.arange(n) * k, y)
>>> fig, ax = plt.subplots(1, 1, figsize=(8, 8))
# Now, we draw our points with a gradient of colors.
ax.scatter(x2, y2, c=range(n * k), linewidths=0,
marker='o', s=3, cmap=plt.cm.jet,)
ax.axis('equal')
ax.set_axis_off()
```



How it works...

The Brownian motion $W(t)$ has several important properties. First, it gives rise (almost surely) to continuous trajectories. Second, its increments $W(t + \tau) - W(t)$ are independent on non-overlapping intervals. Third, these increments are Gaussian random variables. More precisely:

$$\forall t, \tau > 0, \quad W(t + \tau) - W(t) \sim \mathcal{N}(0, \tau)$$

In particular, the density of $W(t)$ is a normal distribution with variance t .

Additionally, the Brownian motion, and stochastic processes in general, have deep connections with partial differential equations. Here, the density of $W(t)$ is a solution of the **heat equation**, a particular diffusion equation. More generally, the **Fokker-Planck equation** is a partial differential equation satisfied by the density of solutions of a stochastic differential equation.

There's more...

The Brownian motion is a limit of a random walk with an infinitesimal step size. We used this property here to simulate the process.

Here are a few references:

- ▶ The Brownian motion (physical phenomenon) described at https://en.wikipedia.org/wiki/Brownian_motion
- ▶ The Wiener process (mathematical object) explained at https://en.wikipedia.org/wiki/Wiener_process
- ▶ The Brownian motion is a particular type of the Lévy process; refer to https://en.wikipedia.org/wiki/L%C3%A9vy_process
- ▶ The Fokker-Planck equation links stochastic processes to partial differential equations; refer to https://en.wikipedia.org/wiki/Fokker%E2%80%93Planck_equation

See also

- ▶ The *Simulating a stochastic differential equation* recipe

Simulating a stochastic differential equation

Stochastic Differential Equations (SDEs) model dynamical systems that are subject to noise. They are widely used in physics, biology, finance, and other disciplines.

In this recipe, we simulate an **Ornstein-Uhlenbeck process**, which is a solution of the Langevin equation. This model describes the stochastic evolution of a particle in a fluid under the influence of friction. The particle's movement is due to collisions with the molecules of the fluid (diffusion). The difference with the Brownian motion is the presence of friction.

The Ornstein-Uhlenbeck process is stationary, Gaussian, and Markov, which makes it a good candidate to represent stationary random noise.

We will simulate this process with a numerical method called the **Euler-Maruyama method**. It is a simple generalization to SDEs of the Euler method for ODEs.

How to do it...

1. Let's import NumPy and Matplotlib:

```
>>> import numpy as np  
      import matplotlib.pyplot as plt  
      %matplotlib inline
```

2. We define a few parameters for our model:

```
>>> sigma = 1. # Standard deviation.  
      mu = 10. # Mean.  
      tau = .05 # Time constant.
```

3. Let's define a few simulation parameters:

```
>>> dt = .001 # Time step.  
      T = 1. # Total time.  
      n = int(T / dt) # Number of time steps.  
      t = np.linspace(0., T, n) # Vector of times.
```

4. We also define renormalized variables (to avoid recomputing these constants at every time step):

```
>>> sigma_bis = sigma * np.sqrt(2. / tau)  
      sqrt_dt = np.sqrt(dt)
```

5. We create a vector that will contain all successive values of our process during the simulation:

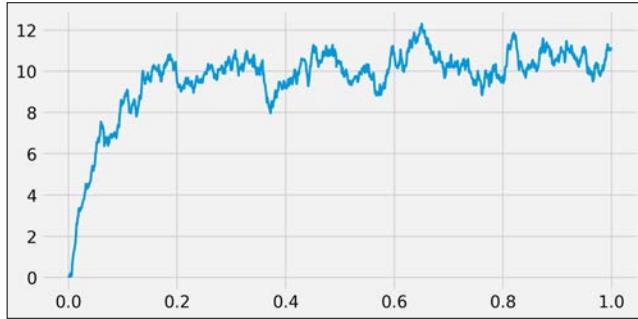
```
>>> x = np.zeros(n)
```

6. Now, let's simulate the process with the Euler-Maruyama method. It is really like the standard Euler method for ODEs, but with an extra stochastic term (which is just a scaled normal random variable). We will give the equation of the process along with the details of this method in the *How it works...* section of this recipe:

```
>>> for i in range(n - 1):  
      x[i + 1] = x[i] + dt * (- (x[i] - mu) / tau) + \  
                  sigma_bis * sqrt_dt * np.random.randn()
```

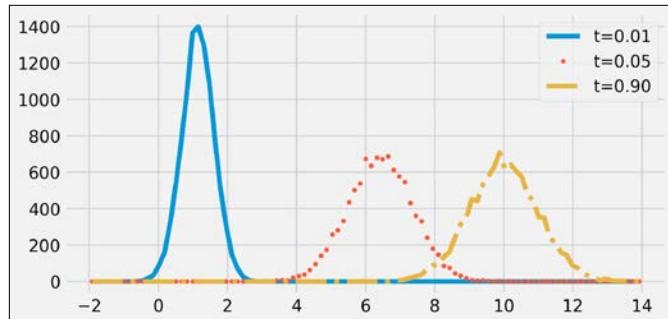
7. Let's display the evolution of the process:

```
>>> fig, ax = plt.subplots(1, 1, figsize=(8, 4))
ax.plot(t, x, lw=2)
```



8. Now, we are going to take a look at the time evolution of the distribution of the process. To do this, we will simulate many independent realizations of the same process in a vectorized way. We define a vector X that will contain all realizations of the process at a given time (that is, we do not keep all realizations at all times in memory). This vector will be overwritten at every time step. We will show the estimated distribution (histograms) at several points in time:

```
>>> ntrials = 10000
X = np.zeros(ntrials)
>>> # We create bins for the histograms.
bins = np.linspace(-2., 14., 100)
fig, ax = plt.subplots(1, 1, figsize=(8, 4))
for i in range(n):
    # We update the process independently for
    # all trials
    X += dt * (-X - mu) / tau + \
        sigma_bis * sqrt(dt) * np.random.randn(ntrials)
    # We display the histogram for a few points in
    # time
    if i in (5, 50, 900):
        hist, _ = np.histogram(X, bins=bins)
        ax.plot((bins[1:] + bins[:-1]) / 2, hist,
                {5: '-', 50: '.', 900: '-.'}, [i],
                label=f't={i * dt:.2f}')
ax.legend()
```



The distribution of the process tends to a Gaussian distribution with mean $\mu = 10$ and standard deviation $\sigma = 1$. The process would be stationary if the initial distribution was also a Gaussian with the adequate parameters.

How it works...

The Langevin equation that we use in this recipe is the following stochastic differential equation:

$$dx = -\frac{(x - \mu)}{\tau} dt + \sigma \sqrt{\frac{2}{\tau}} dW$$

Here, $x(t)$ is our stochastic process, dx is the infinitesimal increment, μ is the mean, σ is the standard deviation, and τ is the time constant. Also, W is a Brownian motion (or the Wiener process) that underlies our SDE.

The first term on the right-hand side is the deterministic term (in dt), while the second term is the stochastic term. Without that last term, the equation would be a regular deterministic ODE.

The infinitesimal step of a Brownian motion is a Gaussian random variable. Specifically, the derivative (in a certain sense) of a Brownian motion is a **white noise**, a sequence of independent Gaussian random variables.

The Euler-Maruyama method involves discretizing time and adding infinitesimal steps to the process at every time step. This method involves a deterministic term (like in the standard Euler method for ODEs) and a stochastic term (random Gaussian variable). Specifically, for an equation:

$$dx = a(t, x)dt + b(t, x)dW$$

The numerical scheme is (with $t = n * dt$):

$$x_{n+1} = x_n + dx = x_n + a(t, x_n)dt + b(t, x_n)\sqrt{dt}\xi, \quad \xi \sim N(0, 1)$$

Here, ξ is a random Gaussian variable with variance 1 (independent at each time step). The normalization factor \sqrt{dt} comes from the fact that the infinitesimal step for a Brownian motion has the standard deviation \sqrt{dt} .

There's more...

The mathematics of SDEs comprises the theory of stochastic calculus, Itô calculus, martingales, and other topics. Although these theories are quite involved, simulating stochastic processes numerically can be relatively straightforward, as we have seen in this recipe.

The error of the Euler-Maruyama method is of order \sqrt{dt} . The Milstein method is a more precise numerical scheme, of order dt .

Here are a few references on these topics:

- ▶ Stochastic differential equations on Wikipedia, available at https://en.wikipedia.org/wiki/Stochastic_differential_equation
- ▶ White noise, described at https://en.wikipedia.org/wiki/White_noise
- ▶ The Langevin equation on Wikipedia, available at https://en.wikipedia.org/wiki/Langevin_equation
- ▶ The Ornstein-Uhlenbeck process described at https://en.wikipedia.org/wiki/Ornstein-Uhlenbeck_process
- ▶ Itô calculus, described at https://en.wikipedia.org/wiki/It%C5%8D_calculus
- ▶ The Euler-Maruyama method, explained at https://en.wikipedia.org/wiki/Euler-Maruyama_method
- ▶ The Milstein method on Wikipedia, available at https://en.wikipedia.org/wiki/Milstein_method

See also

- ▶ The *Simulating a Brownian motion* recipe

14

Graphs, Geometry, and Geographic Information Systems

In this chapter, we will cover the following topics:

- ▶ Manipulating and visualizing graphs with NetworkX
- ▶ Drawing flight routes with NetworkX
- ▶ Resolving dependencies in a directed acyclic graph with a topological sort
- ▶ Computing connected components in an image
- ▶ Computing the Voronoi diagram of a set of points
- ▶ Manipulating geospatial data with Cartopy
- ▶ Creating a route planner for a road network

Introduction

In this chapter, we will cover Python's capabilities in graph theory, geometry, and geography.

Graphs are mathematical objects describing relations between items. They are ubiquitous in science and engineering, as they can represent many kinds of real-world relations: friends in a social network, atoms in a molecule, website links, cells in a neural network, neighboring pixels in an image, and so on. Graphs are also classical data structures in computer science. Finally, many domain-specific problems may be re-expressed as graph problems, and then solved with well-known algorithms.

We will also see a few recipes related to **geometry** and **Geographic Information Systems (GIS)**, which refers to the processing and analysis of any kind of spatial, geographical, or topographical data.

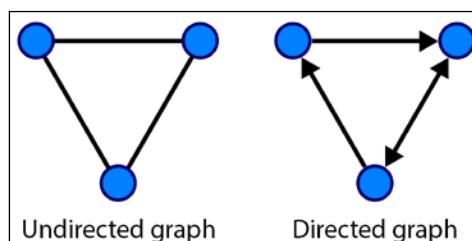
In this introduction, we will give a brief overview of these topics.

Graphs

Mathematically, a **graph** $G = (V, E)$ is defined by a set V of **vertices** or **nodes**, and a set E of **edges** (two-element subsets of V). Two nodes v and v' are said to be **connected** if (v, v') is an edge (element of E).

- ▶ If the edges are *unordered* (meaning that $(v, v') = (v', v)$), the graph is said to be **undirected**
- ▶ If the edges are *ordered* (meaning that $(v, v') \neq (v', v)$), the graph is said to be **directed**

An edge in an undirected graph is represented by a line segment between the two nodes. In a directed graph, it is represented by an arrow.



Undirected and directed graphs

A graph can be represented by different data structures, such as an **adjacency list** (for each vertex, a list of adjacent vertices) or an **adjacency matrix** (matrix of connections between vertices).

Problems in graph theory

Here are a few examples of classical graph problems:

- ▶ **Graph traversal:** How to walk through a graph, discussed at https://en.wikipedia.org/wiki/Graph_traversal
- ▶ **Graph coloring:** How to color nodes in a graph such that no two adjacent vertices share the same color, discussed at https://en.wikipedia.org/wiki/Graph_coloring

- ▶ **Connected components:** How to find connected components in a graph, explained at https://en.wikipedia.org/wiki/Connected_component_%28graph_theory%29
- ▶ **Shortest paths:** What is the shortest path from one node to another in a given graph?, discussed at https://en.wikipedia.org/wiki/Shortest_path_problem
- ▶ **Hamiltonian paths:** Does a graph include a Hamiltonian path, visiting every vertex exactly once?, explained at https://en.wikipedia.org/wiki/Hamiltonian_path
- ▶ **Eulerian paths:** Does a graph include an Eulerian path, visiting every edge exactly once?, discussed at https://en.wikipedia.org/wiki/Eulerian_path
- ▶ **Traveling salesman problem:** What is the shortest route visiting every node exactly once (Hamiltonian path)?, explained at https://en.wikipedia.org/wiki/Traveling_salesman_problem

Random graphs

Random graphs are particular kinds of graphs defined with probabilistic rules. They are useful for understanding the structure of large real-world graphs such as social graphs.

In particular, **small-world networks** have sparse connections, but most nodes can be reached from every other node in a small number of steps. This property is due to the existence of a small number of **hubs** that have a high number of connections.

Graphs in Python

Although graphs can be manipulated with native Python structures, it is more convenient to use a dedicated library implementing specific data structures and manipulation routines. In this chapter, we will use `NetworkX`, a pure Python library. An alternative library is `graph-tool`, largely written in C++.

`NetworkX` implements a flexible data structure for graphs, and it contains many algorithms. `NetworkX` also lets us draw graphs easily with `matplotlib`.

Geometry in Python

`Shapely` is a Python library used to manipulate 2D geometrical shapes such as points, lines, and polygons. It is most notably useful in geographic information systems.

Geographical information systems in Python

There are several Python modules used to manipulate geographical data and plotting maps.

In this chapter, we will use `Cartopy` and `Shapely` to handle GIS files.

The ESRI **shapefile** is a popular geospatial vector data format. It can be read by `Cartopy` and `NetworkX`.

`Cartopy` is a Python library that provides cartographic tools for Python. We can use it to perform map projections and draw maps with `matplotlib`. It relies on `Shapely`.

The `geoplot` is a young high-level geospatial data visualization library in Python that builds on top of `Cartopy` and `matplotlib`.

We will also use the `OpenStreetMap` service, a free, open source, collaborative service providing maps of the world.

Other GIS/mapping systems in Python that we couldn't cover in this chapter include `GeoPandas` and `Kartograph`.

References

Here are a few references about graphs:

- ▶ Graph theory on Wikipedia, available at https://en.wikipedia.org/wiki/Graph_theory
- ▶ Graph theory lectures on AwesomeMath, available at <https://github.com/rossant/awesome-math/#graph-theory>
- ▶ Data structures for graphs, described at https://en.wikipedia.org/wiki/Graph_%28abstract_data_type%29
- ▶ Random graphs on Wikipedia, available at https://en.wikipedia.org/wiki/Random_graph
- ▶ Small-world graphs on Wikipedia, available at https://en.wikipedia.org/wiki/Small-world_network
- ▶ NetworkX package, available at <http://networkx.github.io>
- ▶ The graph-tool package, available at <http://graph-tool.skewed.de>

Here are a few references about geometry and maps in Python:

- ▶ Cartopy at <http://scitools.org.uk/cartopy/>
- ▶ Shapely at <https://github.com/Toblerity/Shapely>
- ▶ Shapefile at <https://en.wikipedia.org/wiki/Shapefile>

- ▶ geoplot at <https://github.com/ResidentMario/geoplot>
- ▶ Folium at <https://github.com/wrobstory/folium>
- ▶ GeoPandas at <http://geopandas.org>
- ▶ Kartograph at <http://kartograph.org>
- ▶ OpenStreetMap at <http://www.openstreetmap.org>

Manipulating and visualizing graphs with NetworkX

In this recipe, we will show how to create, manipulate, and visualize graphs with NetworkX.

Getting ready

NetworkX is installed by default in Anaconda. If needed, you can also install it manually with `conda install networkx`.

How to do it...

1. Let's import NumPy, NetworkX, and matplotlib:

```
>>> import numpy as np
      import networkx as nx
      import matplotlib.pyplot as plt
      %matplotlib inline
```

2. There are many ways of creating a graph. Here, we create a list of edges (pairs of node indices):

```
>>> n = 10 # Number of nodes in the graph.
      # Each node is connected to the two next nodes,
      # in a circular fashion.
      adj = [(i, (i + 1) % n) for i in range(n)]
      adj += [(i, (i + 2) % n) for i in range(n)]
```

3. We instantiate a `Graph` object with our list of edges:

```
>>> g = nx.Graph(adj)
```

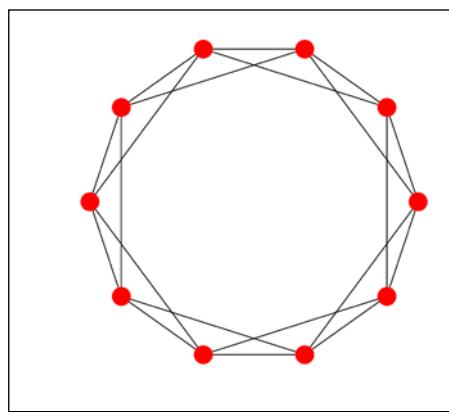
4. Let's check the list of nodes and edges of the graph, and its adjacency matrix:

```
>>> print(g.nodes())
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print(g.edges())
```

```
[ (0, 1), (0, 9), (0, 2), (0, 8), (1, 2), ...,
  (6, 8), (7, 8), (7, 9), (8, 9)]
>>> print(nx.adjacency_matrix(g))
(0, 1)    1
(0, 2)    1
(0, 8)    1
(0, 9)    1
(1, 0)    1
...
(8, 9)    1
(9, 0)    1
(9, 1)    1
(9, 7)    1
(9, 8)    1
```

5. Let's display this graph. NetworkX comes with a variety of drawing functions. We can either specify the nodes' positions explicitly, or we can use an algorithm to automatically compute an interesting layout. Here, we use the `draw_circular()` function that simply positions nodes linearly on a circle:

```
>>> fig, ax = plt.subplots(1, 1, figsize=(6, 6))
nx.draw_circular(g, ax=ax)
```



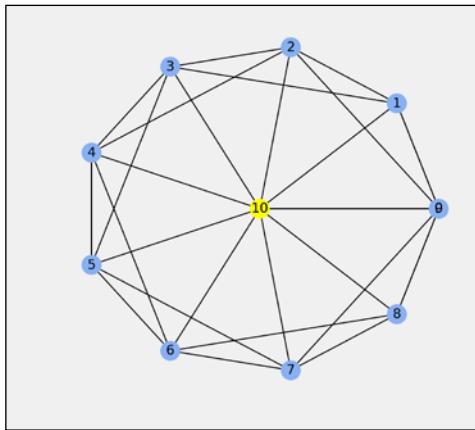
6. Graphs can be modified easily. Here, we add a new node connected to all existing nodes. We also specify a `color` attribute to this node. In NetworkX, every node and edge comes with a Python dictionary containing arbitrary attributes.

```
>>> g.add_node(n, color='#fcff00')
# We add an edge from every existing
```

```
# node to the new node.  
for i in range(n):  
    g.add_edge(i, n)
```

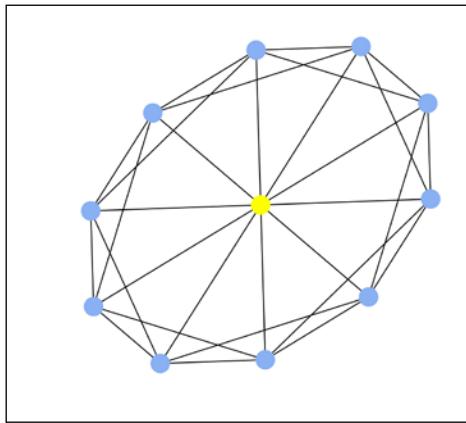
7. Now, let's draw the modified graph again. This time, we specify the nodes' positions and colors explicitly:

```
>>> # We define custom node positions on a circle  
# except the last node which is at the center.  
t = np.linspace(0., 2 * np.pi, n)  
pos = np.zeros((n + 1, 2))  
pos[:n, 0] = np.cos(t)  
pos[:n, 1] = np.sin(t)  
  
# A node's color is specified by its 'color'  
# attribute, or a default color if this attribute  
# doesn't exist.  
color = [g.node[i].get('color', '#88b0f3')  
         for i in range(n + 1)]  
  
# We now draw the graph with matplotlib.  
fig, ax = plt.subplots(1, 1, figsize=(6, 6))  
nx.draw_networkx(g, pos=pos, node_color=color, ax=ax)  
ax.set_axis_off()
```



8. Let's also use an automatic layout algorithm:

```
>>> fig, ax = plt.subplots(1, 1, figsize=(6, 6))
nx.draw_spectral(g, node_color=color, ax=ax)
ax.set_axis_off()
```



There's more...

In NetworkX, nodes are not necessarily integers. They can be numbers, strings, tuples, or instances of any hashable Python class.

In addition, every node and edge comes with optional attributes (which form a dictionary).

A few layout algorithms are implemented in NetworkX. The `draw_spectral()` function uses the eigenvectors of the graph's **Laplacian matrix**.

The `draw_spring()` function implements the **Fruchterman-Reingold force-directed** algorithm. Nodes are considered as masses subject to edge-dependent forces. A force-directed graph drawing algorithm minimizes the system's energy so as to find an equilibrium configuration. This results in an aesthetically appealing layout with as few crossing edges as possible.

Here are a few references:

- ▶ Graph drawing, described at https://en.wikipedia.org/wiki/Graph_drawing
- ▶ Laplacian matrix on Wikipedia, available at https://en.wikipedia.org/wiki/Laplacian_matrix
- ▶ Force-directed graph drawing, described at https://en.wikipedia.org/wiki/Force-directed_graph_drawing

See also

- ▶ The *Drawing flight routes with NetworkX* recipe

Drawing flight routes with NetworkX

In this recipe, we load and visualize a dataset containing many flight routes and airports around the world (obtained from the OpenFlights website at <https://openflights.org/data.html>).

Getting ready

To draw the graph on a map, you need Cartopy, available at <http://scitools.org.uk/cartopy/>. You can install it with `conda install -c conda-forge cartopy`.

How to do it...

1. Let's import a few packages:

```
>>> import math
     import json
     import numpy as np
     import pandas as pd
     import networkx as nx
     import cartopy.crs as ccrs
     import matplotlib.pyplot as plt
     from IPython.display import Image
     %matplotlib inline
```

2. We load the first dataset containing many flight routes:

```
>>> names = ('airline,airline_id,'  
           'source,source_id,'  
           'dest,dest_id,'  
           'codeshare,stops,equipment').split(',')  
>>> routes = pd.read_csv(  
    'https://github.com/ipython-books/'  
    'cookbook-2nd-data/blob/master/'  
    'routes.dat?raw=true',  
    names=names,  
    header=None)  
routes
```

| | airline | airline_id | source | source_id | dest | dest_id | codeshare | stops | equipment |
|-------|---------|------------|--------|-----------|------|---------|-----------|-------|-----------|
| 0 | 2B | 410 | AER | 2965 | KZN | 2990 | NaN | 0 | CR2 |
| 1 | 2B | 410 | ASF | 2966 | KZN | 2990 | NaN | 0 | CR2 |
| 2 | 2B | 410 | ASF | 2966 | MRV | 2962 | NaN | 0 | CR2 |
| 3 | 2B | 410 | CEK | 2968 | KZN | 2990 | NaN | 0 | CR2 |
| 4 | 2B | 410 | CEK | 2968 | OVB | 4078 | NaN | 0 | CR2 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 67658 | ZL | 4178 | WYA | 6334 | ADL | 3341 | NaN | 0 | SF3 |
| 67659 | ZM | 19016 | DME | 4029 | FRU | 2912 | NaN | 0 | 734 |
| 67660 | ZM | 19016 | FRU | 2912 | DME | 4029 | NaN | 0 | 734 |
| 67661 | ZM | 19016 | FRU | 2912 | OSS | 2913 | NaN | 0 | 734 |
| 67662 | ZM | 19016 | OSS | 2913 | FRU | 2912 | NaN | 0 | 734 |

67663 rows × 9 columns

3. We load the second dataset with details about the airports, and we only keep the airports from the United States:

```
>>> names = ('id,name,city,country,iata,icao,lat,lon,'  
           'alt,timezone,dst,tz,type,source').split(',')  
>>> airports = pd.read_csv(  
    'https://github.com/ipython-books/'  
    'cookbook-2nd-data/blob/master/'  
    'airports.dat?raw=true',  
    header=None,  
    names=names,  
    index_col=4,  
    na_values='\\N')  
airports_us = airports[airports['country'] ==  
                      'United States']  
airports_us
```

| | id | name | city | country | icao | ... | timezone | dst | tz | type | source |
|-------------|-----------|----------------------|---------------|----------------|-------------|------------|-----------------|------------|-------------------|-------------|---------------|
| iata | | | | | | | | | | | |
| BTI | 3411 | Barter Island LR... | Barter Island | United States | PABA | ... | -9.0 | A | America/Anchorage | airport | OurAirports |
| LUR | 3413 | Cape Lisburne L... | Cape Lisburne | United States | PALU | ... | -9.0 | A | America/Anchorage | airport | OurAirports |
| PIZ | 3414 | Point Lay LRRS ... | Point Lay | United States | PPIZ | ... | -9.0 | A | America/Anchorage | airport | OurAirports |
| ITO | 3415 | Hilo Internationa... | Hilo | United States | PHTO | ... | -10.0 | N | Pacific/Honolulu | airport | OurAirports |
| ORL | 3416 | Orlando Executiv... | Orlando | United States | KORL | ... | -5.0 | A | America/New_York | airport | OurAirports |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| XMR | 11866 | Cape Canaveral... | Cocoa Beach | United States | KXMR | ... | NaN | NaN | NaN | airport | OurAirports |
| NaN | 11867 | Homey (Area 51)... | Groom Lake | United States | KXTA | ... | NaN | NaN | NaN | airport | OurAirports |
| ZZV | 11868 | Zanesville Munic... | Zanesville | United States | KZZV | ... | NaN | NaN | NaN | airport | OurAirports |
| ENN | 11918 | Nenana Municip... | Nenana | United States | PANN | ... | NaN | NaN | NaN | airport | OurAirports |
| WWA | 11919 | Wasilla Airport | Wasilla | United States | PAWS | ... | NaN | NaN | NaN | airport | OurAirports |

1435 rows × 13 columns

The DataFrame index is the IATA code, a 3-character code identifying the airports.

- Let's keep all national US flight routes—that is, those for which the source and the destination airports belong to the list of US airports:

```
>>> routes_us = routes[
        routes['source'].isin(airports_us.index) &
        routes['dest'].isin(airports_us.index)]
routes_us
```

| | airline | airline_id | source | source_id | dest | dest_id | codeshare | stops | equipment |
|--------------|----------------|-------------------|---------------|------------------|-------------|----------------|------------------|--------------|------------------|
| 172 | 2O | 146 | ADQ | 3531 | KLN | 7162 | NaN | 0 | BNI |
| 177 | 2O | 146 | KLN | 7162 | KYK | 7161 | NaN | 0 | BNI |
| 260 | 3E | 10739 | BRL | 5726 | ORD | 3830 | NaN | 0 | CNC |
| 261 | 3E | 10739 | BRL | 5726 | STL | 3678 | NaN | 0 | CNC |
| 262 | 3E | 10739 | DEC | 4042 | ORD | 3830 | NaN | 0 | CNC |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 67565 | ZK | 2607 | SHR | 5769 | DEN | 3751 | NaN | 0 | EM2 |
| 67566 | ZK | 2607 | SOW | 7078 | FMN | 3743 | NaN | 0 | BE1 |
| 67567 | ZK | 2607 | SOW | 7078 | PHX | 3462 | NaN | 0 | BE1 |
| 67569 | ZK | 2607 | VIS | 7121 | LAX | 3484 | NaN | 0 | BE1 |
| 67570 | ZK | 2607 | WRL | 5777 | CYS | 3804 | NaN | 0 | BEH BE1 |

10507 rows × 9 columns

5. We construct the list of edges representing our graph, where nodes are airports, and two airports are connected if there exists a route between them (flight network):

```
>>> edges = routes_us[['source', 'dest']].values
      edges
array( [ ['ADQ', 'KLN'],
        ['KLN', 'KYK'],
        ['BRL', 'ORD'],
        ...,
        ['SOW', 'PHX'],
        ['VIS', 'LAX'],
        ['WRL', 'CYS']], dtype=object)
```

6. We create the networkX graph from the edges array:

```
>>> g = nx.from_edgelist(edges)
```

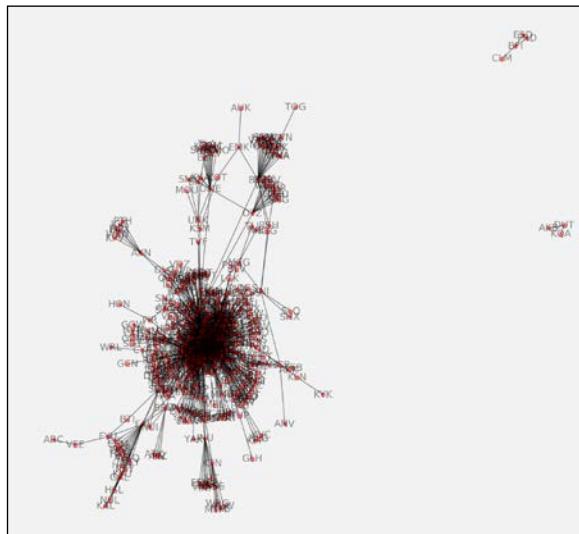
7. Let's take a look at the graph's statistics:

```
>>> len(g.nodes()), len(g.edges())
(546, 2781)
```

There are 546 US airports and 2781 routes in the dataset.

8. Let's plot the graph:

```
>>> fig, ax = plt.subplots(1, 1, figsize=(6, 6))
      nx.draw_networkx(g, ax=ax, node_size=5,
                        font_size=6, alpha=.5,
                        width=.5)
      ax.set_axis_off()
```

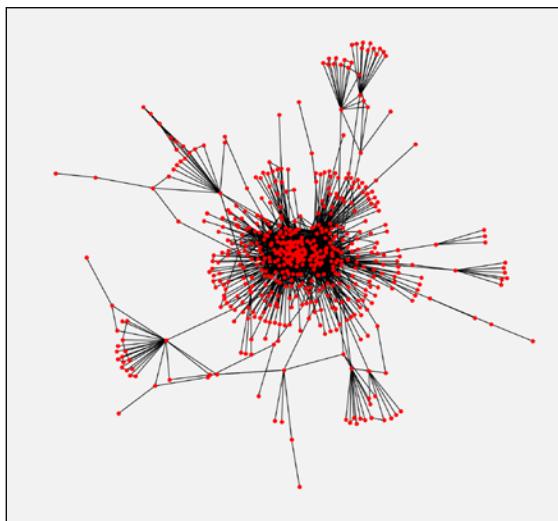


9. There are a few airports that are not connected to the rest of the airports. We keep the largest connected component of the graph as follows (the subgraphs returned by `connected_component_subgraphs()` are sorted by decreasing size):

```
>>> sg = next(nx.connected_component_subgraphs(g))
```

10. Now, we plot the largest connected component subgraph:

```
>>> fig, ax = plt.subplots(1, 1, figsize=(6, 6))
    nx.draw_networkx(sg, ax=ax, with_labels=False,
                      node_size=5, width=.5)
    ax.set_axis_off()
```



The graph encodes only the *topology* (connections between the airports) and not the *geometry* (actual positions of the airports on a map). Airports at the center of the graph are the largest US airports.

11. We're going to draw the graph on a map, using the geographical coordinates of the airports. First, we need to create a dictionary where the keys are the airports IATA codes, and the values are the coordinates:

```
>>> pos = {airport: (v['lon'], v['lat'])
            for airport, v in
            airports_us.to_dict('index').items()}
```

12. The node sizes will depend on the degree of the nodes—that is, the number of airports connected to every node:

```
>>> deg = nx.degree(sg)
    sizes = [5 * deg[iata] for iata in sg.nodes]
```

13. We will also show the airport altitude as the node color:

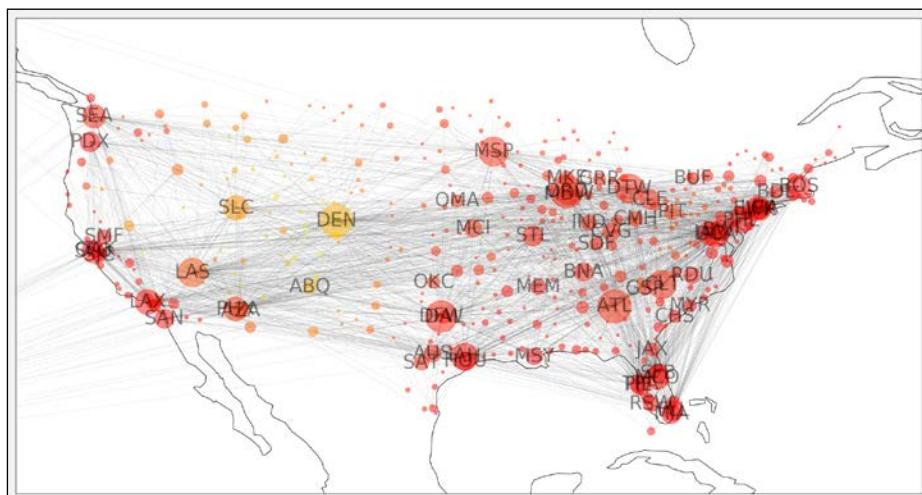
```
>>> altitude = airports_us['alt']
      altitude = [altitude[iata] for iata in sg.nodes]
```

14. We will display the labels of the largest airports only (at least 20 connections to other US airports):

```
>>> labels = {iata: iata if deg[iata] >= 20 else ''
              for iata in sg.nodes}
```

15. Finally, we use Cartopy to project the points on the map:

```
>>> # Map projection
      crs = ccrs.PlateCarree()
      fig, ax = plt.subplots(
          1, 1, figsize=(12, 8),
          subplot_kw=dict(projection=crs))
      ax.coastlines()
      # Extent of continental US.
      ax.set_extent([-128, -62, 20, 50])
      nx.draw_networkx(sg, ax=ax,
                      font_size=16,
                      alpha=.5,
                      width=.075,
                      node_size=sizes,
                      labels=labels,
                      pos=pos,
                      node_color=altitude,
                      cmap=plt.cm.autumn)
```



See also

- ▶ The *Manipulating and visualizing graphs with NetworkX* recipe
- ▶ The *Manipulating geospatial data with Cartopy* recipe

Resolving dependencies in a directed acyclic graph with a topological sort

In this recipe, we will show an application of a well-known graph algorithm: **topological sorting**. Let's consider a directed graph describing dependencies between items.

For example, in a package manager, before we can install a given package P , we may need to install *dependent* packages.

The set of dependencies forms a directed graph. With topological sorting, the package manager can resolve the dependencies and find the right installation order of the packages.

Topological sorting has many other applications. Here, we will illustrate this notion on real data from the JavaScript package manager `npm`. We will find the installation order of the required packages for the `react` JavaScript package.

How to do it...

1. We import a few packages:

```
>>> import io
     import json
     import requests
     import numpy as np
     import networkx as nx
     import matplotlib.pyplot as plt
     %matplotlib inline
```

2. We download the dataset (a GraphML file stored on GitHub, that we created using a script at <https://github.com/graphcommons/npm-dependency-network>) and we load it with the NetworkX function `read_graphml()`:

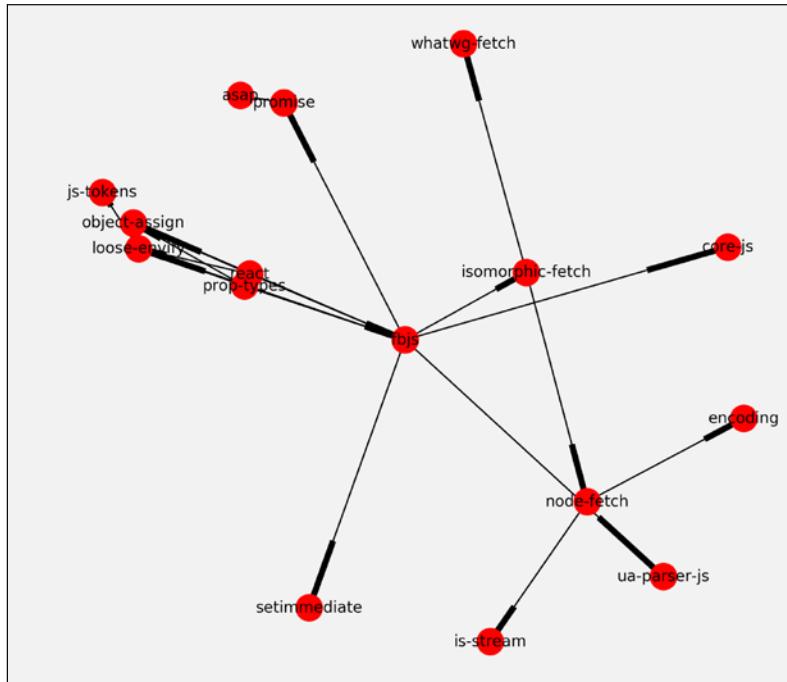
```
>>> url = ('https://github.com/ipython-books/'
           'cookbook-2nd-data/blob/master/'
           'react.graphml?raw=true')
f = io.BytesIO(requests.get(url).content)
graph = nx.read_graphml(f)
```

3. The graph is a directed graph (DiGraph) with few nodes and edges:

```
>>> graph
<networkx.classes.digraph.DiGraph at 0x7f69ac6dfdd8>
>>> len(graph.nodes), len(graph.edges)
(16, 20)
```

4. Let's draw this graph:

```
>>> fig, ax = plt.subplots(1, 1, figsize=(8, 8))
nx.draw_networkx(graph, ax=ax, font_size=10)
ax.set_axis_off()
```



5. A topological sort only exists when the graph is a **Directed Acyclic Graph (DAG)**. This means that there is no cycle in the graph—that is, no circular dependency. Is our graph a DAG? Let's see:

```
>>> nx.is_directed_acyclic_graph(graph)
True
```

6. We can perform the topological sort, thereby obtaining a linear installation order satisfying all dependencies:

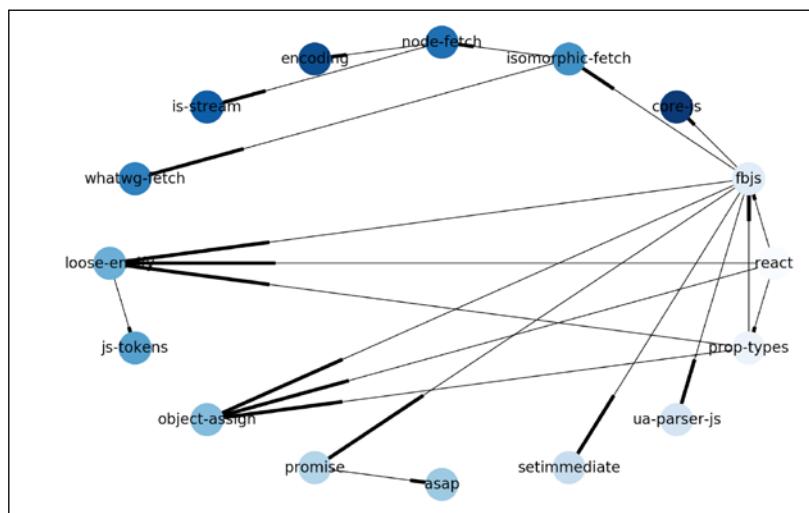
```
>>> ts = list(nx.topological_sort(graph))
ts
['react',
 'prop-types',
 'loose-envify',
 'object-assign',
 'js-tokens',
 'promise-asap',
 'whatwg-fetch',
 'isomorphic-fetch',
 'core-js',
 'encoding',
 'ua-parser-js',
 'node-fetch',
 'is-stream',
 'setimmediate']
```

```
'fbjs',
'ua-parser-js',
'setimmediate',
'promise',
'asap',
'object-assign',
'loose-envify',
'js-tokens',
'isomorphic-fetch',
'whatwg-fetch',
'node-fetch',
'is-stream',
'encoding',
'core-js']
```

Since we used the convention that A directs to B if B needs to be installed before A (A depends on B), the installation order is the reversed order here.

7. Finally, we draw our graph with a shell layout algorithm, and we display the dependence order using the node colors (darker nodes need to be installed before lighter ones):

```
>>> # Each node's color is the index of the node in the
       # topological sort.
colors = [ts.index(node) for node in graph.nodes]
>>> nx.draw_shell(graph,
                  node_color=colors,
                  cmap=plt.cm.Blues,
                  font_size=8,
                  width=.5
                 )
```



How it works...

We used the following code (adapted from <https://github.com/graphcommons/npm-dependency-network>) to obtain the dependency graph of the react npm package:

```
>>> from lxml.html import fromstring
     import cssselect # Need to do: pip install cssselect
     from requests.packages import urllib3

     urllib3.disable_warnings()
     fetched_packages = set()

     def import_package_dependencies(graph, pkg_name,
                                     max_depth=3, depth=0):
         if pkg_name in fetched_packages:
             return
         if depth > max_depth:
             return
         fetched_packages.add(pkg_name)
         url = f'https://www.npmjs.com/package/{pkg_name}'
         response = requests.get(url, verify=False)
         doc = fromstring(response.content)
         graph.add_node(pkg_name)
         for h3 in doc.cssselect('h3'):
             content = h3.text_content()
             if content.startswith('Dependencies'):
                 for dep in h3.getnext().cssselect('a'):
                     dep_name = dep.text_content()
                     print('-' * depth * 2, dep_name)
                     graph.add_node(dep_name)
                     graph.add_edge(pkg_name, dep_name)
                     import_package_dependencies(
                         graph,
                         dep_name,
                         depth=depth + 1
                     )

         graph = nx.DiGraph()
         import_package_dependencies(graph, 'react')
         nx.write_graphml(graph, 'react.graphml')
```

You can use that code to obtain the dependency graph of any other npm package. The script may take a few minutes to complete.

There's more...

Directed acyclic graphs are found in many applications. They can represent causal relations, influence diagrams, dependencies, and other concepts. For example, the version history of a distributed revision control system such as Git is described with a DAG.

Topological sorting is useful in any scheduling task in general (project management and instruction scheduling).

Here are a few references:

- ▶ Directed acyclic graphs on NetworkX, at <https://networkx.github.io/documentation/latest/reference/algorithms/dag.html>
- ▶ Topological sort documentation on NetworkX, available at https://networkx.github.io/documentation/latest/reference/algorithms/generated/networkx.algorithms.dag.topological_sort.html
- ▶ Topological sorting on Wikipedia, available at https://en.wikipedia.org/wiki/Topological_sorting
- ▶ Directed acyclic graphs, described at https://en.wikipedia.org/wiki/Directed_acyclic_graph

Computing connected components in an image

In this recipe, we will show an application of graph theory in image processing. We will compute **connected components** in an image. This method will allow us to label contiguous regions of an image, similar to the bucket fill tool of paint programs.

Finding connected components is also useful in many puzzle video games such as Minesweeper, bubble shooters, and others. In these games, contiguous sets of items with the same color need to be automatically detected.

How to do it...

1. Let's import the packages:

```
>>> import itertools
      import numpy as np
      import networkx as nx
      import matplotlib.colors as col
      import matplotlib.pyplot as plt
      %matplotlib inline
```

2. We create a 10×10 image where each pixel can take one of three possible labels (or colors):

```
>>> n = 10
>>> img = np.random.randint(size=(n, n),
                           low=0, high=3)
```

3. Now, we create the underlying 2D grid graph encoding the structure of the image. Each node is a pixel, and a node is connected to its nearest neighbors. NetworkX defines a `grid_2d_graph()` function to generate this graph:

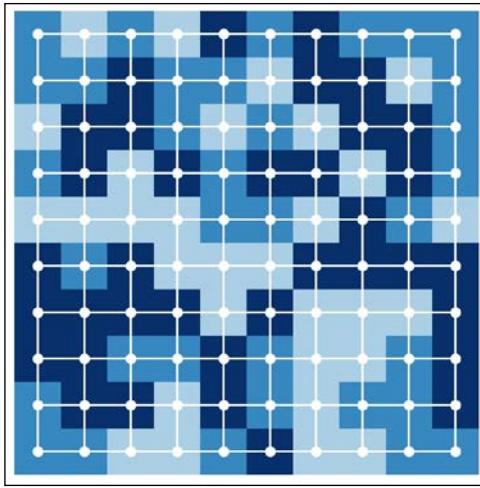
```
>>> g = nx.grid_2d_graph(n, n)
```

4. Let's create two functions to display the image and the corresponding graph:

```
>>> def show_image(img, ax=None, **kwargs):
    ax.imshow(img, origin='lower',
              interpolation='none',
              **kwargs)
    ax.set_axis_off()
>>> def show_graph(g, ax=None, **kwargs):
    pos = {(i, j): (j, i) for (i, j) in g.nodes()}
    node_color = [img[i, j] for (i, j) in g.nodes()]
    nx.draw_networkx(g,
                     ax=ax,
                     pos=pos,
                     node_color='w',
                     linewidths=3,
                     width=2,
                     edge_color='w',
                     with_labels=False,
                     node_size=50,
                     **kwargs)
>>> cmap = plt.cm.Blues
```

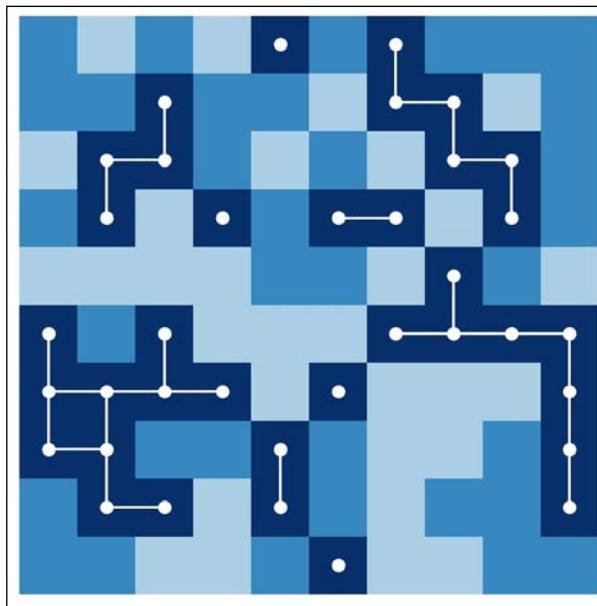
5. Here is the original image superimposed with the underlying graph:

```
>>> fig, ax = plt.subplots(1, 1, figsize=(8, 8))
show_image(img, ax=ax, cmap=cmap, vmin=-1)
show_graph(g, ax=ax, cmap=cmap, vmin=-1)
```



6. Now, we are going to find all contiguous dark blue regions containing more than three pixels. First, we consider the *subgraph* corresponding to all dark blue pixels:

```
>>> g2 = g.subgraph(zip(*np.nonzero(img == 2)))
>>> fig, ax = plt.subplots(1, 1, figsize=(8, 8))
    show_image(img, ax=ax, cmap=cmap, vmin=-1)
    show_graph(g2, ax=ax, cmap=cmap, vmin=-1)
```

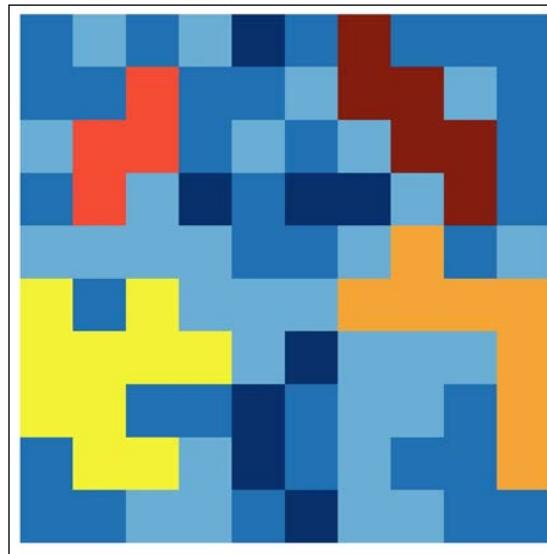


7. The requested contiguous regions correspond to the *connected components* containing more than three nodes in the subgraph. We can use the `connected_components()` function of NetworkX to find those components:

```
>>> components = [np.array(list(comp))
                  for comp in nx.connected_components(g2)
                  if len(comp) >= 3]
len(components)
4
```

8. Finally, we assign a new color to each of these components, and we display the new image:

```
>>> # We copy the image, and assign a new label
# to each found component.
img_bis = img.copy()
for i, comp in enumerate(components):
    img_bis[comp[:, 0], comp[:, 1]] = i + 3
>>> # We create a new discrete color map extending
# the previous map with new colors.
colors = [cmap(.5), cmap(.75), cmap(1.),
          '#f4f235', '#f4a535', '#f44b35',
          '#821d10']
cmap2 = col.ListedColormap(colors, 'indexed')
>>> fig, ax = plt.subplots(1, 1, figsize=(8, 8))
show_image(img_bis, ax=ax, cmap=cmap2)
```



How it works...

The problem we solved is called **connected-component labeling**. It is also closely related to the **flood-fill algorithm**.

The idea to associate a grid graph to an image is quite common in image processing. Here, contiguous color regions correspond to **connected components** of subgraphs. A connected component can be defined as an equivalence class of the **reachability** relation. Two nodes are connected in the graph if there is a path from one node to the other. An equivalence class contains nodes that can be reached from one another.

Finally, the simple approach described here is only adapted to basic tasks on small images. More advanced algorithms are covered in *Chapter 11, Image and Audio Processing*.

There's more...

Here are a few references:

- ▶ Connected components on Wikipedia, available at https://en.wikipedia.org/wiki/Connected_component_%28graph_theory%29
- ▶ Connected-component labeling on Wikipedia, at https://en.wikipedia.org/wiki/Connected-component_labeling
- ▶ Flood-fill algorithm on Wikipedia, available at https://en.wikipedia.org/wiki/Flood_fill

Computing the Voronoi diagram of a set of points

The **Voronoi diagram** of a set of seed points divides space into several regions. Each region contains all points closer to one seed point than to any other seed point.

The Voronoi diagram is a fundamental structure in computational geometry. It is widely used in computer science, robotics, geography, and other disciplines. For example, the Voronoi diagram of a set of metro stations gives us the closest station from any point in the city.

In this recipe, we compute the Voronoi diagram of the set of metro stations in Paris using SciPy.

Getting ready

You need the Smopy module to display the OpenStreetMap map of Paris. You can install this package with `pip install git+https://github.com/rossant/smopy.git`.

How to do it...

- Let's import the packages:

```
>>> import numpy as np
     import pandas as pd
     import scipy.spatial as spatial
     import matplotlib.pyplot as plt
     import matplotlib.path as path
     import matplotlib as mpl
     import smopy
%matplotlib inline
```

- Let's load the dataset with pandas (which had been obtained on the RATP open data website, the public transport operator in Paris, at <http://data.ratp.fr>):

```
>>> df = pd.read_csv('https://github.com/ipython-books/'
                     'cookbook-2nd-data/blob/master/'
                     'ratp.csv?raw=true',
                     sep='#', header=None)
>>> df[df.columns[1:]].tail(3)
```

| | 1 | 2 | 3 | 4 | 5 |
|-------|----------|-----------|-----------------|---------------|------|
| 11608 | 2.350173 | 48.937238 | THEATRE GERA... | SAINT-DENIS | tram |
| 11609 | 2.301197 | 48.933118 | TIMBAUD | GENNEVILLIERS | tram |
| 11610 | 2.230144 | 48.913708 | VICTOR BASCH | COLOMBES | tram |

- The DataFrame object contains the coordinates, name, city, district, and type of station. Let's select all metro stations:

```
>>> metro = df[(df[5] == 'metro')]
>>> metro[metro.columns[1:]].tail(3)
```

| | 1 | 2 | 3 | 4 | 5 |
|-----|----------|-----------|---------------------|-------------|-------|
| 305 | 2.308041 | 48.841697 | Volontaires | PARIS-15EME | metro |
| 306 | 2.379884 | 48.857876 | Voltaire (Léon B... | PARIS-11EME | metro |
| 307 | 2.304651 | 48.883874 | Wagram | PARIS-17EME | metro |

- We are going to extract the district number of Paris' stations. With pandas, we can use vectorized string operations using the `str` attribute of the corresponding column.

```
>>> # We only extract the district from stations in Paris.
     paris = metro[4].str.startswith('PARIS').values
>>> # We create a vector of integers with the district
```

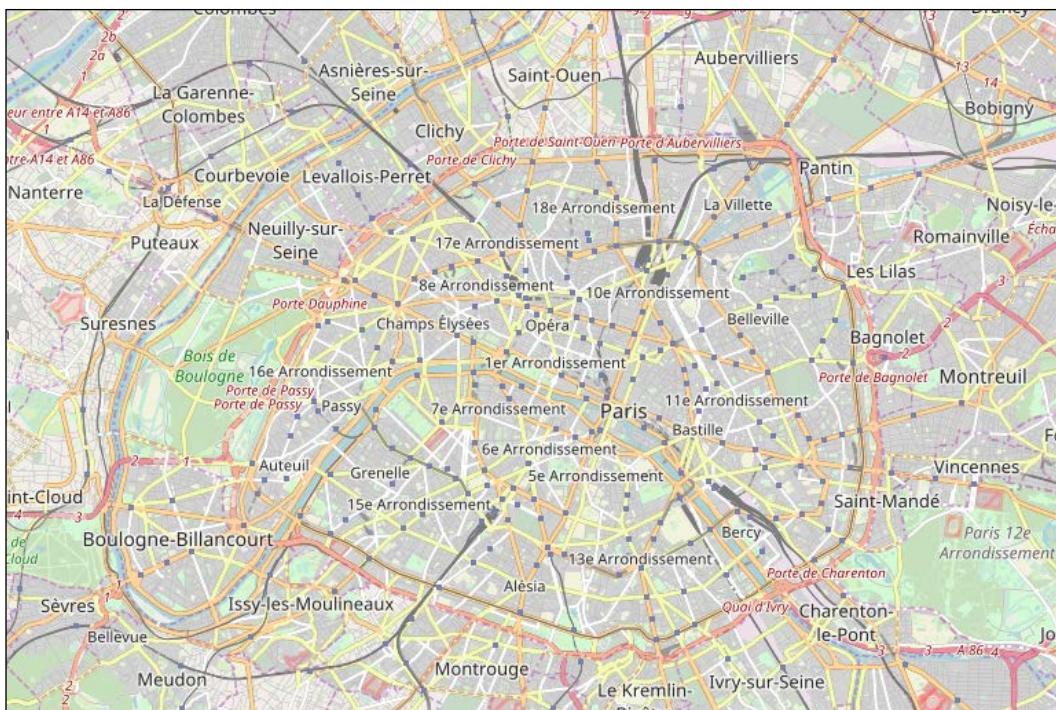
```
# number of the corresponding station, or 0 if the
# station is not in Paris.
districts = np.zeros(len(paris), dtype=np.int32)
districts[paris] = metro[4][paris].str.slice(6, 8) \
    .astype(np.int32)
districts[~paris] = 0
ndistricts = districts.max() + 1
```

5. We also extract the coordinates of all metro stations:

```
>>> lon = metro[1]
      lat = metro[2]
```

6. Now, let's retrieve Paris' map with OpenStreetMap. We specify the map's boundaries with the extreme latitude and longitude coordinates of all our metro stations. We use Smopy to generate the map:

```
>>> box = (lat[paris].min(), lon[paris].min(),
          lat[paris].max(), lon[paris].max())
m = smopy.Map(box, z=12)
m.show_ipython()
```



7. We now compute the Voronoi diagram of the stations using SciPy. A Voronoi object is created with the points coordinates. It contains several attributes we will use for display:

```
>>> vor = spatial.Voronoi(np.c_[lat, lon])
```

8. We create a generic function to display a Voronoi diagram. SciPy already implements such a function, but this function does not take infinite points into account. The implementation we will use is available at <http://stackoverflow.com/a/20678647/1595060>:

```
>>> def voronoi_finite_polygons_2d(vor, radius=None):
    """Reconstruct infinite Voronoi regions in a
    2D diagram to finite regions.
    Source:
    https://stackoverflow.com/a/20678647/1595060
    """
    if vor.points.shape[1] != 2:
        raise ValueError("Requires 2D input")
    new_regions = []
    new_vertices = vor.vertices.tolist()
    center = vor.points.mean(axis=0)
    if radius is None:
        radius = vor.points.ptp().max()
    # Construct a map containing all ridges for a
    # given point
    all_ridges = {}
    for (p1, p2), (v1, v2) in zip(vor.ridge_points,
                                   vor.ridge_vertices):
        all_ridges.setdefault(
            p1, []).append((p2, v1, v2))
        all_ridges.setdefault(
            p2, []).append((p1, v1, v2))
    # Reconstruct infinite regions
    for p1, region in enumerate(vor.point_region):
        vertices = vor.regions[region]
        if all(v >= 0 for v in vertices):
            # finite region
            new_regions.append(vertices)
            continue
        # reconstruct a non-finite region
        ridges = all_ridges[p1]
        new_region = [v for v in vertices if v >= 0]
        for p2, v1, v2 in ridges:
            if v2 < 0:
                v1, v2 = v2, v1
```

```

if v1 >= 0:
    # finite ridge: already in the region
    continue
# Compute the missing endpoint of an
# infinite ridge
t = vor.points[p2] - \
    vor.points[p1] # tangent
t /= np.linalg.norm(t)
n = np.array([-t[1], t[0]]) # normal
midpoint = vor.points[[p1, p2]]. \
    mean(axis=0)
direction = np.sign(
    np.dot(midpoint - center, n)) * n
far_point = vor.vertices[v2] + \
    direction * radius
new_region.append(len(new_vertices))
new_vertices.append(far_point.tolist())
# Sort region counterclockwise.
vs = np.asarray([new_vertices[v]
                 for v in new_region])
c = vs.mean(axis=0)
angles = np.arctan2(
    vs[:, 1] - c[1], vs[:, 0] - c[0])
new_region = np.array(new_region)[
    np.argsort(angles)]
new_regions.append(new_region.tolist())
return new_regions, np.asarray(new_vertices)

```

9. The `voronoi_finite_polygons_2d()` function returns a list of regions and a list of vertices. Every region is a list of vertex indices. The coordinates of all vertices are stored in `vertices`. From these structures, we can create a list of cells. Every cell represents a polygon as an array of vertex coordinates. We also use the `to_pixels()` method of the `smopy.Map` instance. This function converts latitude and longitude geographical coordinates to pixels in the image.

```

>>> regions, vertices = voronoi_finite_polygons_2d(vor)
>>> cells = [m.to_pixels(vertices[region])
            for region in regions]

```

10. Now, we compute the color of every polygon:

```

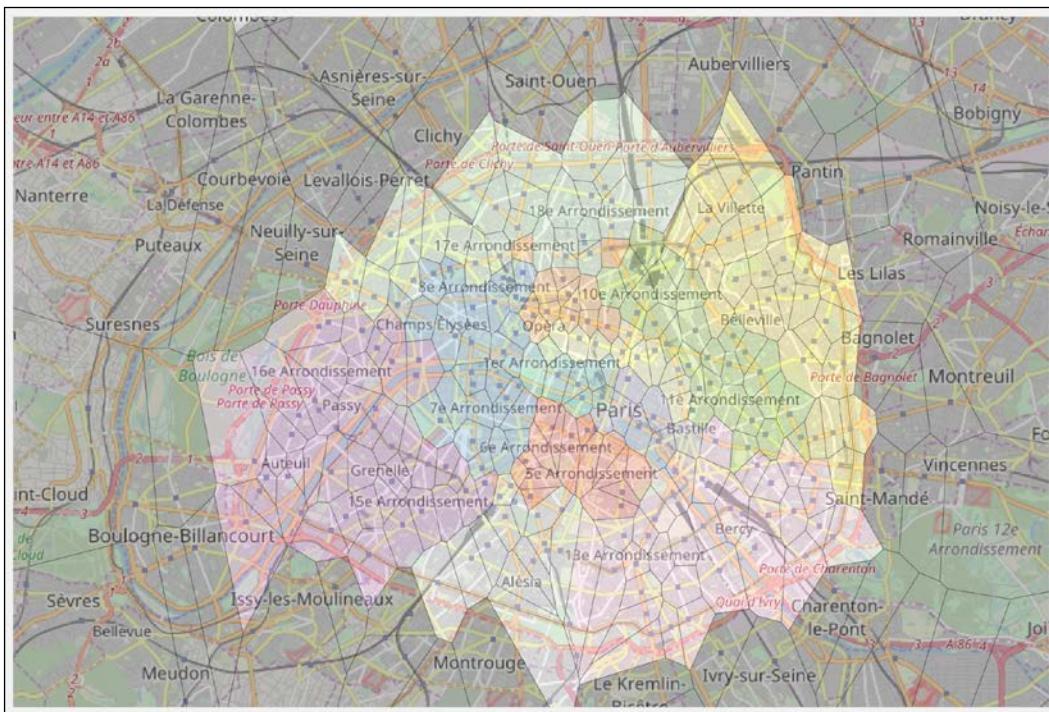
>>> cmap = plt.cm.Set3
# We generate colors for districts using a color map.
colors_districts = cmap(
    np.linspace(0., 1., ndistricts))[:, :3]

```

```
# The color of every polygon, grey by default.  
colors = .25 * np.ones((len(districts), 3))  
# We give each polygon in Paris the color of  
# its district.  
colors[paris] = colors_districts[districts[paris]]
```

11. Finally, we display the map with the Voronoi diagram, using the `show_mpl()` method of the `Map` instance:

```
>>> ax = m.show_mpl(figsize=(12, 8))
      ax.add_collection(
          mpl.collections.PolyCollection(
              cells, facecolors=colors,
              edgecolors='k', alpha=.35))
```



How it works...

Let's give the mathematical definition of the Voronoi diagram in a Euclidean space. If (x_i) is a set of points, the Voronoi diagram of this set of points is the collection of subsets V_i (called **cells** or **regions**) defined by:

$$V_i = \{\mathbf{x} \in \mathbb{R}^d \mid \forall j \neq i, \quad \|\mathbf{x} - \mathbf{x}_i\| \leq \|\mathbf{x} - \mathbf{x}_j\|\}$$

The dual graph of the Voronoi diagram is the **Delaunay triangulation**. This geometrical object covers the convex hull of the set of points with triangles.

SciPy computes Voronoi diagrams with `Qhull`, a computational geometry library in C++.

There's more...

Here are further references:

- ▶ Voronoi diagram on Wikipedia, available at https://en.wikipedia.org/wiki/Voronoi_diagram
- ▶ Delaunay triangulation on Wikipedia, available at https://en.wikipedia.org/wiki/Delaunay_triangulation
- ▶ The documentation of `scipy.spatial.voronoi` available at <http://docs.scipy.org/doc/scipy-dev/reference/generated/scipy.spatial.Voronoi.html>
- ▶ The `Qhull` library available at <http://www.qhull.org>

See also

- The *Manipulating geospatial data with Cartopy* recipe

Manipulating geospatial data with Cartopy

In this recipe, we will show how to load and display geographical data in the Shapefile format. Specifically, we will use data from **Natural Earth** (<http://www.naturalearthdata.com>) to display the countries of Africa, color coded with their population and **Gross Domestic Product (GDP)**. This type of graph is called a **choropleth map**.

Shapefile (<https://en.wikipedia.org/wiki/Shapefile>) is a popular geospatial vector data format for GIS software. It can be read by Cartopy, a GIS package in Python.

Getting ready

You need **Cartopy**, available at <http://scitools.org.uk/cartopy/>. You can install it with conda install -c conda-forge cartopy.

How to do it...

1. Let's import the packages:

```
>>> import io
     import requests
     import zipfile
     import numpy as np
     import matplotlib.pyplot as plt
     import matplotlib.collections as col
     from matplotlib.colors import Normalize
     import cartopy.crs as ccrs
     from cartopy.feature import ShapelyFeature
     import cartopy.io.shapereader as shpreader
     %matplotlib inline
```

2. We download and load the Shapefile that contains geometric and administrative information about all countries in the world (it had been obtained from Natural Earth's website at <http://www.naturalearthdata.com/downloads/10m-cultural-vectors/10m-admin-0-countries/>):

```
>>> url = ('https://github.com/ipython-books/'
           'cookbook-2nd-data/blob/master/'
           'africa.zip?raw=true')
r = io.BytesIO(requests.get(url).content)
zipfile.ZipFile(r).extractall('data')
countries = shpreader.Reader(
    'data/ne_10m_admin_0_countries.shp')
```

3. We keep the African countries:

```
>>> africa = [c for c in countries.records()
              if c.attributes['CONTINENT'] == 'Africa']
```

4. Let's write a function that draws the borders of Africa:

```
>>> crs = ccrs.PlateCarree()
     extent = [-23.03, 55.20, -37.72, 40.58]
>>> def draw_africa(ax):
     ax.set_extent(extent)
     ax.coastlines()
```

```
>>> fig, ax = plt.subplots(  
    1, 1, figsize=(6, 8),  
    subplot_kw=dict(projection=crs))  
draw_africa(ax)
```

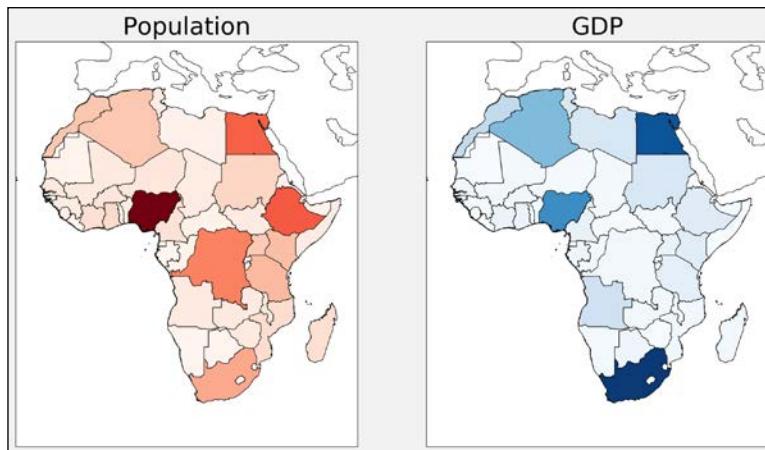


5. Now, we write a function that displays the countries of Africa with a color that depends on a specific attribute, like the population or GDP:

```
>>> def choropleth(ax, attr, cmap_name):  
    # We need to normalize the values before we can  
    # use the colormap.  
    values = [c.attributes[attr] for c in africa]  
    norm = Normalize(  
        vmin=min(values), vmax=max(values))  
    cmap = plt.cm.get_cmap(cmap_name)  
    for c in africa:  
        v = c.attributes[attr]  
        sp = ShapelyFeature(c.geometry, crs,  
                            edgecolor='k',  
                            facecolor=cmap(norm(v)))  
        ax.add_feature(sp)
```

- Finally, we display two choropleth maps with the population and GDP of all African countries:

```
>>> fig, (ax1, ax2) = plt.subplots(  
    1, 2, figsize=(10, 16),  
    subplot_kw=dict(projection=crs))  
draw_africa(ax1)  
choropleth(ax1, 'POP_EST', 'Reds')  
ax1.set_title('Population')  
  
draw_africa(ax2)  
choropleth(ax2, 'GDP_MD_EST', 'Blues')  
ax2.set_title('GDP')
```



There's more...

The geoplot package, available at <https://github.com/ResidentMario/geoplot>, provides high-level tools to draw choropleth maps and other geospatial figures.

See also

- ▶ The *Creating a route planner for a road network* recipe

Creating a route planner for a road network

In this recipe, we build upon several techniques described in the previous recipes in order to create a simple GPS-like route planner in Python. We will retrieve California's road network data from the United States Census Bureau in order to find shortest paths in the road network graph. This will allow us to display road itineraries between any two locations in California.

Getting ready

You need Smopy for this recipe. You can install it with `pip install git+https://github.com/rossant/smopy`. In order for NetworkX to read Shapefile datasets, you also need GDAL/OGR. You can install it with `conda install gdal`.



At the time of this writing, gdal does not appear to work well with conda and Python 3.6. You may need to downgrade Python to Python 3.5 with `conda install python=3.5`.



How to do it...

1. Let's import the packages:

```
>>> import io
      import zipfile
      import requests
      import networkx as nx
      import numpy as np
      import pandas as pd
      import json
      import smopy
      import matplotlib.pyplot as plt
      %matplotlib inline
```

2. We load the data (a Shapefile dataset) with NetworkX. This dataset contains detailed information about the primary roads in California. NetworkX's `read_shp()` function returns a graph, where each node is a geographical position, and each edge contains information about the road linking the two nodes. The data comes from the United States Census Bureau website at <http://www.census.gov/geo/maps-data/data/tiger.html>.

```
>>> url = ('https://github.com/ipython-books/'
          'cookbook-2nd-data/blob/master/'
          'road.zip?raw=true')
r = io.BytesIO(requests.get(url).content)
zipfile.ZipFile(r).extractall('data')
g = nx.read_shp('data/tl_2013_06_prisecroads.shp')
```

3. This graph is not necessarily connected, but we need a connected graph in order to compute shortest paths. Here, we take the largest connected subgraph using the `connected_component_subgraphs()` function:

```
>>> sgs = list(nx.connected_component_subgraphs(  
...     g.to_undirected()))  
i = np.argmax([len(sg) for sg in sgs])  
sg = sgs[i]  
len(sg)
```

464

4. We define two positions (with the latitude and longitude) and find the shortest path between these two positions:

```
>>> pos0 = (36.6026, -121.9026)  
pos1 = (34.0569, -118.2427)
```

5. Each edge in the graph contains information about the road, including a list of points along this road. We first create a function that returns this array of coordinates, for any edge in the graph:

```
>>> def get_path(n0, n1):  
...     """If n0 and n1 are connected nodes in the graph,  
...     this function returns an array of point  
...     coordinates along the road linking these two  
...     nodes."""  
...     return np.array(json.loads(sg[n0][n1]['Json']))  
...         ['coordinates'])
```

6. We can notably use the road path to compute its length. We first need to define a function that computes the distance between any two points in geographical coordinates:

```
>>> # from https://stackoverflow.com/a/8859667/1595060  
EARTH_R = 6372.8  
  
def geocalc(lat0, lon0, lat1, lon1):  
    """Return the distance (in km) between two points  
    in geographical coordinates."""  
    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

```

7. Now, we define a function computing a path's length:

```

>>> def get_path_length(path):
    return np.sum(geocalc(path[1:, 1], path[1:, 0],
                          path[:-1, 1], path[:-1, 0]))

```

8. We update our graph by computing the distance between any two connected nodes. We add this information with the `distance` attribute of the edges:

```

>>> # Compute the length of the road segments.
for n0, n1 in sg.edges:
    path = get_path(n0, n1)
    distance = get_path_length(path)
    sg.edges[n0, n1]['distance'] = distance

```

9. The last step before we can find the shortest path in the graph is to find the two nodes in the graph that are closest to the two requested positions:

```

>>> nodes = np.array(sg.nodes())
# Get the closest nodes in the graph.
pos0_i = np.argmin(
    np.sum((nodes[:, ::-1] - pos0)**2, axis=1))
pos1_i = np.argmin(
    np.sum((nodes[:, ::-1] - pos1)**2, axis=1))

```

10. Now, we use NetworkX's `shortest_path()` function to compute the shortest path between our two positions. We specify that the weight of every edge is the length of the road between them:

```

>>> # Compute the shortest path.
path = nx.shortest_path(
    sg,
    source=tuple(nodes[pos0_i]),
    target=tuple(nodes[pos1_i]),
    weight='distance')
len(path)

```

19

11. The itinerary has been computed. The `path` variable contains the list of edges that form the shortest path between our two positions. Now we can get information about the itinerary with pandas. The dataset has a few fields of interest, including the name and type (State, Interstate, and so on) of the roads:

```

>>> roads = pd.DataFrame(
    [sg.edges[path[i], path[i + 1]]
     for i in range(len(path) - 1)],

```

```
columns= ['FULLNAME', 'MTFCC',
          'RTTYP', 'distance'])
roads
```

| | FULLNAME | MTFCC | RTTYP | distance |
|-----|---------------|-------|-------|------------|
| 0 | State Rte 1 | S1200 | S | 100.658130 |
| 1 | State Rte 1 | S1200 | S | 33.419556 |
| 2 | Cabrillo Hwy | S1200 | M | 4.399051 |
| 3 | State Rte 1 | S1200 | S | 12.400382 |
| 4 | Cabrillo Hwy | S1200 | M | 36.693272 |
| ... | ... | ... | ... | ... |
| 13 | US Hwy 101 | S1200 | U | 75.852281 |
| 14 | Ventura Fwy | S1200 | M | 49.045475 |
| 15 | Hollywood Fwy | S1200 | M | 0.885826 |
| 16 | Hollywood Fwy | S1200 | M | 14.087603 |
| 17 | Hollywood Fwy | S1200 | M | 0.010107 |

18 rows × 4 columns

Here is the total length of this itinerary:

```
>>> roads['distance'].sum()
508.664
```

12. Finally, let's display the itinerary on the map. We first retrieve the map with Smopy:

```
>>> m = smopy.Map(pos0, pos1, z=7, margin=.1)
```

13. Our path contains connected nodes in the graph. Every edge between two nodes is characterized by a list of points (constituting a part of the road). Therefore, we need to define a function that concatenates the positions along every edge in the path. We have to concatenate the positions in the right order along our path. We choose the order based on the fact that the last point in an edge needs to be close to the first point in the next edge:

```
>>> def get_full_path(path):
        """Return the positions along a path."""
        p_list = []
        curp = None
        for i in range(len(path) - 1):
            p = get_path(path[i], path[i + 1])
            if curp is None:
                curp = p
            else:
                curp = curp + p
```

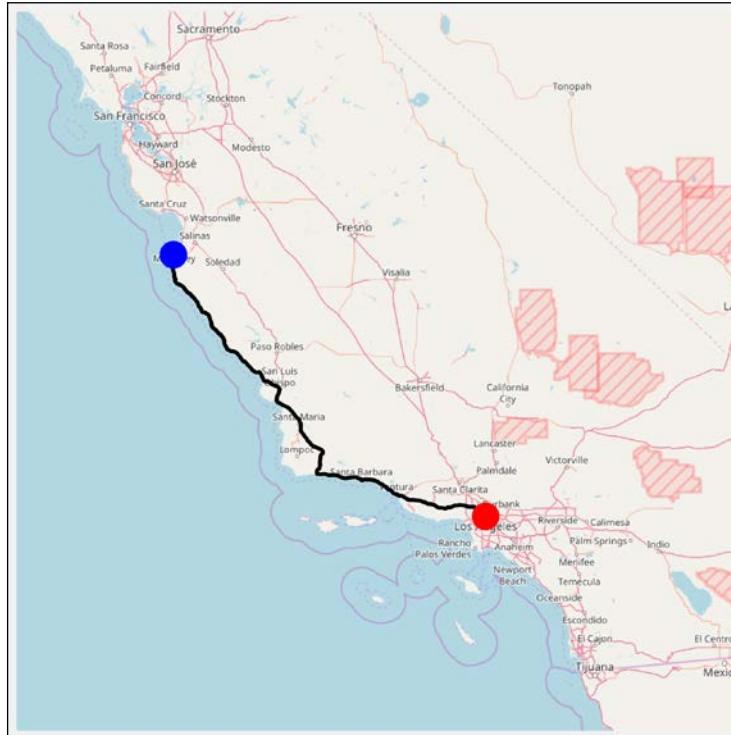
```
if (np.sum((p[0] - curp) ** 2) >
       np.sum((p[-1] - curp) ** 2)):
    p = p[::-1, :]
p_list.append(p)
curp = p[-1]
return np.vstack(p_list)
```

14. We convert the path into pixels in order to display it on the Smopy map:

```
>>> linepath = get_full_path(path)
x, y = m.to_pixels(linepath[:, 1], linepath[:, 0])
```

15. Finally, let's display the map, with our two positions and the computed itinerary between them:

```
>>> ax = m.show_mpl(figsize=(8, 8))
# Plot the itinerary.
ax.plot(x, y, '-k', lw=3)
# Mark our two positions.
ax.plot(x[0], y[0], 'ob', ms=20)
ax.plot(x[-1], y[-1], 'or', ms=20)
```



How it works...

We computed the shortest path with NetworkX's `shortest_path()` function. Here, this function used **Dijkstra's algorithm**. This algorithm has a wide variety of applications, for example in network routing protocols.

There are different ways to compute the geographical distance between two points. Here, we used a relatively precise formula: the **orthodromic distance** (also called **great-circle distance**), which assumes that the Earth is a perfect sphere. We could also have used a simpler formula since the distance between two successive points on a road is small.

There's more...

You can find more information about shortest path problems and Dijkstra's algorithm in the following references:

- ▶ Shortest paths in the NetworkX documentation, https://networkx.github.io/documentation/stable/reference/algorithms/shortest_paths.html
- ▶ *What algorithms compute directions from point A to point B on a map?* on StackOverflow, at <https://stackoverflow.com/q/430142/1595060>
- ▶ Shortest paths on Wikipedia, available at https://en.wikipedia.org/wiki/Shortest_path_problem
- ▶ Dijkstra's algorithm, described at https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Here are a few references about geographical distances:

- ▶ Geographical distance on Wikipedia, at https://en.wikipedia.org/wiki/Geographical_distance
- ▶ Great circles on Wikipedia, at https://en.wikipedia.org/wiki/Great_circle
- ▶ Great-circle distance on Wikipedia, at https://en.wikipedia.org/wiki/Great-circle_distance

15

Symbolic and Numerical Mathematics

In this chapter, we will cover the following topics:

- ▶ Diving into symbolic computing with SymPy
- ▶ Solving equations and inequalities
- ▶ Analyzing real-valued functions
- ▶ Computing exact probabilities and manipulating random variables
- ▶ A bit of number theory with SymPy
- ▶ Finding a Boolean propositional formula from a truth table
- ▶ Analyzing a nonlinear differential system — Lotka-Volterra (predator-prey) equations
- ▶ Getting started with Sage

Introduction

In this chapter, we will introduce **SymPy**, a Python library for symbolic mathematics. Whereas most of the book deals with numerical methods, we will see examples here where symbolic computations are more suitable.

SymPy is to symbolic computing what NumPy is to numerical computing. For example, SymPy can help us analyze a mathematical model before we run a simulation.

Although quite powerful, SymPy may be slower than other computer algebra systems. The main reason is that SymPy is written in pure Python. A faster and more complete mathematics system is **Sage** (see also the *Getting started with Sage* recipe in this chapter). Sage is a heavy standalone program that has many dependencies (including SymPy), and it uses only Python 2 at the time of writing. It is essentially meant for interactive use. Sage can be used with the Jupyter Notebook.

LaTeX

LaTeX is a document markup language widely used to write publication-quality mathematical equations. Equations written in LaTeX can be displayed in the browser with the **MathJax** JavaScript library. SymPy uses this system to display equations in the Jupyter Notebook.

LaTeX equations can also be used in matplotlib. In this case, it is recommended to have a LaTeX installation on your local computer.

Here are a few references:

- ▶ LaTeX on Wikipedia, at <https://en.wikipedia.org/wiki/LaTeX>
- ▶ LaTeX in matplotlib, described at <http://matplotlib.org/users/usetex.html>
- ▶ Documentation for displaying equations with SymPy, available at <http://docs.sympy.org/latest/tutorial/printing.html>
- ▶ To install LaTeX on your computer, refer to <http://latex-project.org/ftp.html>

Diving into symbolic computing with SymPy

In this recipe, we will give a brief introduction to symbolic computing with SymPy. We will see more advanced features of SymPy in the next recipes.

Getting ready

Anaconda should come with SymPy by default, but you can always install it with `conda install sympy`.

How to do it...

SymPy can be used from a Python module, or interactively in Jupyter/IPython. In the Notebook, all mathematical expressions are displayed with LaTeX, thanks to the MathJax JavaScript library.

Here is an introduction to SymPy:

1. First, we import SymPy and enable LaTeX printing in the Jupyter Notebook:

```
>>> from sympy import *
      init_printing()
```

2. To deal with symbolic variables, we first need to declare them:

```
>>> var('x y')
```

$$(x, y)$$

3. The `var()` function creates symbols and injects them into the namespace. This function should only be used in the interactive mode. In a Python module, it is better to use the `symbols()` function that returns the symbols:

```
>>> x, y = symbols('x y')
```

4. We can create mathematical expressions with these symbols:

```
>>> expr1 = (x + 1) ** 2
      expr2 = x**2 + 2 * x + 1
```

5. Are these expressions equal?

```
>>> expr1 == expr2
      False
```

6. These expressions are mathematically equal, but not syntactically identical. To test whether they are mathematically equal, we can ask SymPy to simplify the difference algebraically:

```
>>> simplify(expr1 - expr2)
```

$$0$$

7. A very common operation with symbolic expressions is the substitution of a symbol by another symbol, expression, or a number, using the `subs()` method of a symbolic expression:

```
>>> expr1.subs(x, expr1)
```

$$\left((x + 1)^2 + 1\right)^2$$

```
>>> expr1.subs(x, pi)
```

$$(1 + \pi)^2$$

-
8. A rational number cannot be written simply as $1/2$ as this Python expression evaluates to 0.5. A possibility is to convert the number 1 into a SymPy integer object, for example by using the `s()` function:

```
>>> expr1.subs(x, s(1) / 2)
```

$$\frac{9}{4}$$

9. Exactly represented numbers can be evaluated numerically with `evalf()`:

```
>>> _.evalf()
```

2.25

10. We can easily create a Python function from a SymPy symbolic expression using the `lambdify()` function. The resulting function can notably be evaluated on NumPy arrays. This is quite convenient when we need to go from the symbolic world to the numerical world:

```
>>> f = lambdify(x, expr1)
>>> import numpy as np
      f(np.linspace(-2., 2., 5))
array([ 1.,  0.,  1.,  4.,  9.])
```

How it works...

A core idea in SymPy is to use the standard Python syntax to manipulate exact expressions. Although this is very convenient and natural, there are a few caveats. Symbols such as `x`, which represent mathematical variables, cannot be used in Python before being instantiated (otherwise, a `NameError` exception is thrown by the interpreter). This is in contrast to most other computer algebra systems. For this reason, SymPy offers ways to declare symbolic variables beforehand.

Another example is integer division; as $1/2$ evaluates to 0.5 (or 0 in Python 2), SymPy has no way to know that the user intended to write a fraction instead. We need to convert the numerical integer 1 to the symbolic integer 1 before dividing it by 2.

Also, the Python equality refers to the equality between syntax trees rather than between mathematical expressions.

See also

- ▶ Solving equations and inequalities
- ▶ Getting started with Sage

Solving equations and inequalities

Sympy offers several ways to solve linear and nonlinear equations and systems of equations. Of course, these functions do not always succeed in finding closed-form exact solutions. In this case, we can fall back to numerical solvers and obtain approximate solutions.

How to do it...

1. Let's define a few symbols:

```
>>> from sympy import *
      init_printing()
>>> var('x y z a')
```

$$(x, y, z, a)$$

2. We use the `solve()` function to solve equations (the right-hand side is 0 by default):

```
>>> solve(x**2 - a, x)
```

$$[-\sqrt{a}, \sqrt{a}]$$

3. We can also solve inequalities. Here, we need to use the `solve_univariate_inequality()` function to solve this univariate inequality in the real domain:

```
>>> x = Symbol('x')
      solve_univariate_inequality(x**2 > 4, x)
```

$$(-\infty < x \wedge x < -2) \vee (2 < x \wedge x < \infty)$$

4. The `solve()` function also accepts systems of equations (here, a linear system):

```
>>> solve([x + 2*y + 1, x - 3*y - 2], x, y)
```

$$\left\{ x : \frac{1}{5}, y : -\frac{3}{5} \right\}$$

5. Nonlinear systems are also handled:

```
>>> solve([x**2 + y**2 - 1, x**2 - y**2 - S(1) / 2], x, y)
```

$$\left[\left(-\frac{\sqrt{3}}{2}, -\frac{1}{2} \right), \left(-\frac{\sqrt{3}}{2}, \frac{1}{2} \right), \left(\frac{\sqrt{3}}{2}, -\frac{1}{2} \right), \left(\frac{\sqrt{3}}{2}, \frac{1}{2} \right) \right]$$

6. Singular linear systems can also be solved (here, there is an infinite number of solutions because the two equations are collinear):

```
>>> solve([x + 2*y + 1, -x - 2*y - 1], x, y)
```

$$\{x : -2y - 1\}$$

7. Now, let's solve a linear system using matrices containing symbolic variables:

```
>>> var('a b c d u v')
```

$$(a, b, c, d, u, v)$$

8. We create the **augmented matrix**, which is the horizontal concatenation of the system's matrix with the linear coefficients and the right-hand side vector. This matrix corresponds to the following system in x, y : $ax + by = u, cx + dy = v$:

```
>>> M = Matrix([[a, b, u], [c, d, v]])
M
```

$$\begin{bmatrix} a & b & u \\ c & d & v \end{bmatrix}$$

```
>>> solve_linear_system(M, x, y)
```

$$\left\{ x : \frac{-bv + du}{ad - bc}, \quad y : \frac{av - cu}{ad - bc} \right\}$$

9. This system needs to be nonsingular in order to have a unique solution, which is equivalent to saying that the determinant of the system's matrix needs to be nonzero (otherwise the denominators in the preceding fractions are equal to zero):

```
>>> det(M[:, :2])
```

$$ad - bc$$

There's more...

Matrix support in SymPy is quite rich; we can perform a large number of operations and decompositions (see the reference guide at <http://docs.sympy.org/latest/modules/matrices/matrices.html>).

Here are more references about linear algebra:

- ▶ Linear algebra on Wikipedia, at https://en.wikipedia.org/wiki/Linear_algebra#Further_reading
- ▶ Linear algebra on Wikibooks, at http://en.wikibooks.org/wiki/Linear_Algebra
- ▶ Linear algebra lectures on Awesome Math, at <https://github.com/rossant/awesome-math/#linear-algebra>

Analyzing real-valued functions

SymPy contains a rich calculus toolbox to analyze real-valued functions: limits, power series, derivatives, integrals, Fourier transforms, and so on. In this recipe, we will show the very basics of these capabilities.

How to do it...

1. Let's define a few symbols and a function (which is just an expression depending on x):

```
>>> from sympy import *
>>> init_printing()
>>> var('x z')

(x, z)
```

```
>>> f = 1 / (1 + x**2)
```

2. Let's evaluate this function at 1:

```
>>> f.subs(x, 1)
```

$$\frac{1}{2}$$

3. We can compute the derivative of this function:

```
>>> diff(f, x)
```

$$-\frac{2x}{(x^2 + 1)^2}$$

4. What is f 's limit to infinity? (Note the double o (∞) for the infinity symbol):

```
>>> limit(f, x, oo)
```

$$0$$

5. Here's how to compute a Taylor series (here, around 0, of order 9). The **Big O** can be removed with the `removeO()` method.

```
>>> series(f, x0=0, n=9)
```

$$1 - x^2 + x^4 - x^6 + x^8 + \mathcal{O}(x^9)$$

6. We can compute definite integrals (here, over the entire real line):

```
>>> integrate(f, (x, -oo, oo))
```

$$\pi$$

7. SymPy can also compute indefinite integrals:

```
>>> integrate(f, x)
```

$$\text{atan}(x)$$

8. Finally, let's compute f 's Fourier transforms:

```
>>> fourier_transform(f, x, z)
```

$$\pi e^{-2\pi z}$$

There's more...

SymPy includes a large number of other integral transforms besides the Fourier transform (<http://docs.sympy.org/latest/modules/integrals/integrals.html>). However, SymPy will not always be able to find closed-form solutions.

Here are a few general references about real analysis and calculus:

- ▶ Real analysis on Wikipedia, at https://en.wikipedia.org/wiki/Real_analysis#Bibliography
- ▶ Calculus on Wikibooks, at <http://en.wikibooks.org/wiki/Calculus>
- ▶ Real analysis on Awesome Math, at <https://github.com/rossant/awesome-math/#real-analysis>

Computing exact probabilities and manipulating random variables

Sympy includes a module named `stats` that lets us create and manipulate random variables. This is useful when we work with probabilistic or statistical models; we can compute symbolic expectancies, variances, probabilities, and densities of random variables.

How to do it...

1. Let's import SymPy and the `stats` module:

```
>>> from sympy import *
      from sympy.stats import *
      init_printing()
```

2. Let's roll two dice, `X` and `Y`, with six faces each:

```
>>> X, Y = Die('X', 6), Die('Y', 6)
```

3. We can compute probabilities defined by equalities (with the `Eq` operator) or inequalities:

```
>>> P(Eq(X, 3))
```

$$\frac{1}{6}$$

```
>>> P(X > 3)
```

$$\frac{1}{2}$$

4. Conditions can also involve multiple random variables:

```
>>> P(X > Y)
```

$$\frac{5}{12}$$

5. We can compute conditional probabilities:

```
>>> P(X + Y > 6, X < 5)
```

$$\frac{5}{12}$$

6. We can also work with arbitrary discrete or continuous random variables:

```
>>> Z = Normal('Z', 0, 1) # Gaussian variable  
>>> P(Z > pi)
```

$$\frac{\sqrt{2}\sqrt{\pi}}{4} \left(-\frac{\sqrt{2}}{\sqrt{\pi}} \operatorname{erf} \left(\frac{\sqrt{2}\pi}{2} \right) + \frac{\sqrt{2}}{\sqrt{\pi}} \right)$$

7. We can compute expectancies and variances:

```
>>> E(Z**2), variance(Z**2)
```

$$(1, 2)$$

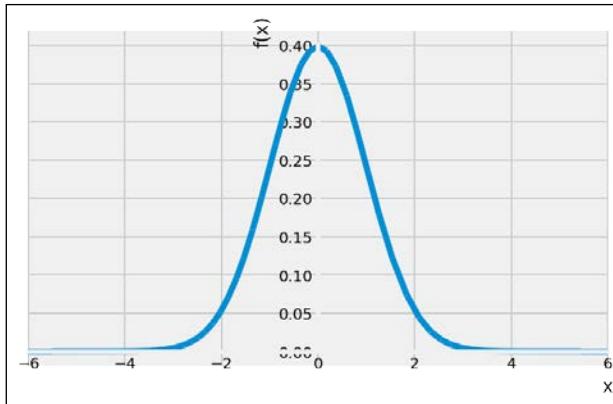
8. We can also compute densities:

```
>>> f = density(Z)  
>>> var('x')  
f(x)
```

$$\frac{\sqrt{2}e^{-\frac{x^2}{2}}}{2\sqrt{\pi}}$$

9. We can plot these densities:

```
>>> %matplotlib inline
plot(f(x), (x, -6, 6))
```



How it works...

Sympy's **stats** module contains many functions to define random variables with classical laws (binomial, exponential, and so on), discrete or continuous. It works by leveraging Sympy's powerful integration algorithms to compute exact probabilistic quantities as integrals of probability distributions. For example, $P(Z > \pi)$ is:

```
>>> Eq(Integral(f(x), (x, pi, oo)),
      simplify(integrate(f(x), (x, pi, oo))))
```

$$\int_{\pi}^{\infty} \frac{\sqrt{2}e^{-\frac{x^2}{2}}}{2\sqrt{\pi}} dx = -\frac{1}{2} \operatorname{erf}\left(\frac{\sqrt{2}\pi}{2}\right) + \frac{1}{2}$$

Note that the equality condition is written using the `Eq` operator rather than the more standard `==` Python syntax. This is a general feature in SymPy; `==` means equality between Python variables, whereas `Eq` is the mathematical operation between symbolic expressions.

There's more...

Here are a few references:

- ▶ SymPy stats module documentation at <http://docs.sympy.org/latest/modules/stats.html>
- ▶ Probability lectures on Awesome Math, at <https://github.com/rossant/awesome-math/#probability-theory>

- ▶ Statistics lectures on Awesome Math, at <https://github.com/rossant/awesome-math/#statistics>

A bit of number theory with SymPy

SymPy contains many number-theory-related routines: obtaining prime numbers, integer decompositions, and much more. We will show a few examples here.

Getting ready

To display legends using LaTeX in matplotlib, you will need an installation of LaTeX on your computer (see this chapter's introduction).

How to do it...

1. Let's import SymPy and the number theory package:

```
>>> from sympy import *
      import sympy.ntheory as nt
      init_printing()
```

2. We can test whether a number is prime:

```
>>> nt.isprime(2017)
True
```

3. We can find the next prime after a given number:

```
>>> nt.nextprime(2017)
2027
```

4. What is the 1000th prime number?

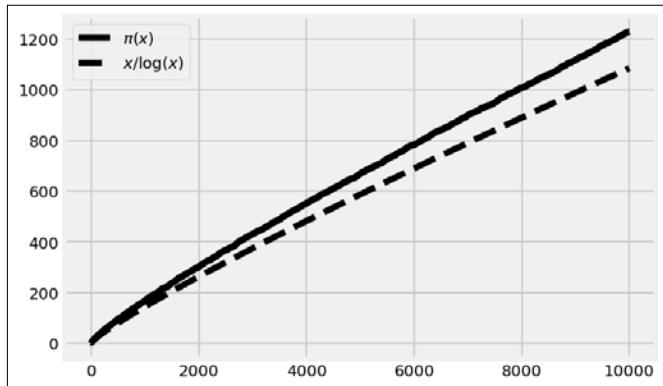
```
>>> nt.prime(1000)
7919
```

5. How many primes less than 2017 are there?

```
>>> nt.primepi(2017)
306
```

6. We can plot $\pi(x)$, the **prime-counting function** (the number of prime numbers less than or equal to some number x). The **prime number theorem** states that this function is asymptotically equivalent to $x / \log(x)$. This expression approximately quantifies the distribution of prime numbers among all integers:

```
>>> import numpy as np
      import matplotlib.pyplot as plt
      %matplotlib inline
      x = np.arange(2, 10000)
      fig, ax = plt.subplots(1, 1, figsize=(6, 4))
      ax.plot(x, list(map(nt.primepi, x)), '-k',
              label='$\pi(x)$')
      ax.plot(x, x / np.log(x), '--k',
              label='$x/\log(x)$')
      ax.legend(loc=2)
```



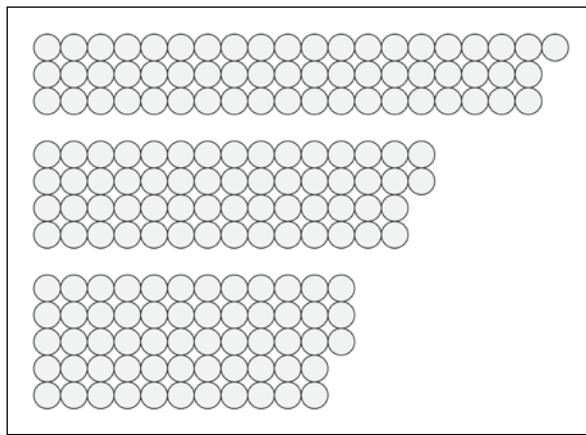
7. Let's compute the integer factorization of a number:

```
>>> nt.factorint(1998)
{2: 1, 3: 3, 37: 1}
```

```
>>> 2 * 3**3 * 37
```

1998

8. Finally, a small problem. A lazy mathematician is counting his marbles. When they are arranged in three rows, the last column contains one marble. When they form four rows, there are two marbles in the last column, and there are three with five rows. How many marbles are there? (Hint: The lazy mathematician has fewer than 100 marbles.)



Marbles

The Chinese Remainder Theorem gives us the answer:

```
>>> from sympy.ntheory.modular import solve_congruence  
solve_congruence((1, 3), (2, 4), (3, 5))  
  
(58, 60)
```

There are infinitely many solutions: 58 plus any multiple of 60. Since there are less than 100 marbles, 58 is the right answer.

How it works...

Sympy contains many number-theory-related functions. Here, we used the **Chinese Remainder Theorem** to find the solutions of the following system of arithmetic equations:

$$\begin{aligned} n &\equiv a_1 \pmod{m_1} \\ &\vdots \\ n &\equiv a_k \pmod{m_k} \end{aligned}$$

The triple bar is the symbol for modular congruence. Here, it means that m_i divides $a_i - n$. In other words, n and a_i are equal up to a multiple of m_i . Reasoning with congruences is very convenient when periodic scales are involved. For example, operations involving 12-hour clocks are done modulo 12. The numbers 11 and 23 are equivalent modulo 12 (they represent the same hour on the clock) because their difference is a multiple of 12.

In this recipe's example, three congruences have to be satisfied: the remainder of the number of marbles in the division with 3 is 1 (there's one extra marble in that arrangement), it is 2 in the division with 4, and 3 in the division with 5. With SymPy, we simply specify these values in the `solve_congruence()` function to get the solutions.

The theorem states that solutions exist as soon as the m_i are pairwise co-prime (any two distinct numbers among them are co-prime). All solutions are congruent modulo the product of the m_i . This fundamental theorem in number theory has several applications, notably in cryptography.

There's more...

Here are a few textbooks about number theory:

- ▶ Undergraduate level: Elementary Number Theory, Gareth A. Jones, Josephine M. Jones, Springer, (1998)
- ▶ Graduate level: A Classical Introduction to Modern Number Theory, Kenneth Ireland, Michael Rosen, Springer, (1982)

Here are a few references:

- ▶ Documentation on SymPy's number-theory module, available at <http://docs.sympy.org/latest/modules/nttheory.html>
- ▶ The Chinese Remainder Theorem on Wikipedia, at https://en.wikipedia.org/wiki/Chinese_remainder_theorem
- ▶ Applications of the Chinese Remainder Theorem, given at <http://mathoverflow.net/questions/10014/applications-of-the-chinese-remainder-theorem>
- ▶ Number theory lectures on Awesome Math, at <https://github.com/rossant/awesome-math/#number-theory>

Finding a Boolean propositional formula from a truth table

The logic module in SymPy lets us manipulate complex Boolean expressions, also known as **propositional formulas**.

This recipe will show an example where this module can be useful. Let's suppose that, in a program, we need to write a complex `if` statement depending on three Boolean variables. We can think about each of the eight possible cases (true, true and false, and so on) and evaluate what the outcome should be. SymPy offers a function to generate a compact logic expression that satisfies our truth table.

How to do it...

1. Let's import SymPy:

```
>>> from sympy import *
init_printing()
```

2. Let's define a few symbols:

```
>>> var('x y z')
```

$$(x, y, z)$$

3. We can define propositional formulas with symbols and a few operators:

```
>>> P = x & (y | ~z)
P
```

$$x \wedge (y \vee \neg z)$$

4. We can use `subs()` to evaluate a formula on actual Boolean values:

```
>>> P.subs({x: True, y: False, z: True})
```

```
False
```

5. Now, we want to find a propositional formula depending on x, y, and z, with the following truth table:

| x | y | z | ?? |
|---|---|---|----|
| T | T | T | * |
| T | T | F | * |
| T | F | T | T |
| T | F | F | T |
| F | T | T | F |
| F | T | F | F |
| F | F | T | F |
| F | F | F | T |

A truth table

6. Let's write down all combinations that we want to evaluate to True, and those for which the outcome does not matter:

```
>>> minterms = [[1, 0, 1], [1, 0, 0], [0, 0, 0]]  
dontcare = [[1, 1, 1], [1, 1, 0]]
```

7. Now, we use the `SOPform()` function to derive an adequate formula:

```
>>> Q = SOPform(['x', 'y', 'z'], minterms, dontcare)  
Q
```

$$x \vee (\neg y \wedge \neg z)$$

8. Let's test that this proposition works:

```
>>> Q.subs({x: True, y: False, z: False}), Q.subs(  
{x: False, y: True, z: True})  
(True, False)
```

How it works...

The `SOPform()` function generates a full expression corresponding to a truth table and simplifies it using the **Quine-McCluskey algorithm**. It returns the smallest *Sum of Products* form (or disjunction of conjunctions). Similarly, the `POSform()` function returns a Product of Sums.

The given truth table can occur in this case: suppose that we want to write a file if it doesn't already exist (z), or if the user wants to force the writing (x). In addition, the user can prevent the writing (y). The expression evaluates to True if the file is to be written. The resulting SOP formula works if we explicitly forbid x and y in the first place (forcing and preventing the writing at the same time is forbidden).

There's more...

Here are a few references:

- ▶ SymPy logic module documentation at <http://docs.sympy.org/latest/modules/logic.html>
- ▶ The propositional formula on Wikipedia, at https://en.wikipedia.org/wiki/Propositional_formula
- ▶ Sum of Products on Wikipedia, at https://en.wikipedia.org/wiki/Canonical_normal_form

- ▶ The Quine–McCluskey algorithm on Wikipedia, at https://en.wikipedia.org/wiki/Quine%E2%80%93McCluskey_algorithm
- ▶ Logic lectures on Awesome Math, at <https://github.com/rossant/awesome-math/#logic>

Analyzing a nonlinear differential system — Lotka-Volterra (predator-prey) equations

Here, we will conduct a brief analytical study of a famous nonlinear differential system: the **Lotka-Volterra equations**, also known as predator-prey equations. These equations are first-order differential equations that describe the evolution of two interacting populations (for example, sharks and sardines), where the predators eat the prey. This example illustrates how to obtain exact expressions and results about fixed points and their stability with SymPy.

Getting ready

For this recipe, knowing the basics of linear and nonlinear systems of differential equations is recommended.

How to do it...

1. Let's create some symbols:

```
>>> from sympy import *
init_printing(pretty_print=True)

var('x y')
var('a b c d', positive=True)

(a, b, c, d)
```

2. The variables x and y represent the populations of the prey and predators, respectively. The parameters a , b , c , and d are strictly positive parameters (described more precisely in the *How it works...* section of this recipe). The equations are:

$$\frac{dx}{dt} = f(x) = x(a - by) \quad (1)$$

$$\frac{dy}{dt} = g(x) = -y(c - dx) \quad (2)$$

```
>>> f = x * (a - b * y)
g = -y * (c - d * x)
```

3. Let's find the fixed points of the system (solving $f(x, y) = g(x, y) = 0$). We call them (x_0, y_0) and (x_1, y_1) :

```
>>> solve([f, g], (x, y))
```

$$\left[(0, 0), \left(\frac{c}{d}, \frac{a}{b} \right) \right]$$

```
>>> (x0, y0), (x1, y1) = _
```

4. Let's write the 2D vector with the two equations:

```
>>> M = Matrix((f, g))
M
```

$$\begin{bmatrix} x(a - by) \\ -y(c - dx) \end{bmatrix}$$

5. Now, we can compute the **Jacobian** of the system, as a function of (x, y) :

```
>>> J = M.jacobian((x, y))
J
```

$$\begin{bmatrix} a - by & -bx \\ dy & -c + dx \end{bmatrix}$$

6. Let's study the stability of the first fixed point by looking at the eigenvalues of the Jacobian at this point. The first fixed point corresponds to extinct populations:

```
>>> M0 = J.subs(x, x0).subs(y, y0)
M0
```

$$\begin{bmatrix} a & 0 \\ 0 & -c \end{bmatrix}$$

```
>>> M0.eigenvals()
```

$$\{a : 1, -c : 1\}$$

The parameters a and c are strictly positive, so the eigenvalues are real and of opposite signs, and this fixed point is a saddle point. As this point is unstable, the extinction of both populations is unlikely in this model.

7. Let's consider the second fixed point now:

```
>>> M1 = J.subs(x, x1).subs(y, y1)
M1
```

$$\begin{bmatrix} 0 & -\frac{bc}{d} \\ \frac{ad}{b} & 0 \end{bmatrix}$$

```
>>> M1.eigenvals()
```

$$\{-i\sqrt{a}\sqrt{c}: 1, i\sqrt{a}\sqrt{c}: 1\}$$

The eigenvalues are purely imaginary: thus, this fixed point is not hyperbolic. Therefore, we cannot draw conclusions from this linear analysis about the qualitative behavior of the system around this fixed point. However, we could show with other methods that oscillations occur around this point.

How it works...

The Lotka-Volterra equations model the growth of the predator and prey populations, taking into account their interactions. In the first equation, the ax term represents the exponential growth of the prey, and $-bxy$ represents death by predators. Similarly, in the second equation, $-yc$ represents the natural death of the predators, and dxy represents their growth as they eat more and more prey.

To find the **equilibrium points** of the system, we need to find the values x, y such that $dx/dt = dy/dt = 0$, that is, $f(x, y) = g(x, y) = 0$, so that the variables do not evolve anymore. Here, we were able to obtain analytical values for these equilibrium points with the `solve()` function.

To analyze their stability, we need to perform a linear analysis of the nonlinear equations, by taking the **Jacobian matrix** at these equilibrium points. This matrix represents the linearized system, and its eigenvalues tell us about the stability of the system near the equilibrium point. The **Hartman-Grobman theorem** states that the behavior of the original system qualitatively matches the behavior of the linearized system around an equilibrium point if this point is **hyperbolic** (meaning that no eigenvalues of the matrix have a real part equal to 0). Here, the first equilibrium point is hyperbolic as $a, c > 0$, but the second is not.

Here, we were able to compute symbolic expressions for the Jacobian matrix and its eigenvalues at the equilibrium points.

There's more...

Even when a differential system is not solvable analytically (as is the case here), a mathematical analysis can still give us qualitative information about the behavior of the system's solutions. A purely numerical analysis is not always relevant when we are interested in qualitative results, as numerical errors and approximations can lead to wrong conclusions about the system's behavior.

Here are a few references:

- ▶ Matrix documentation in SymPy, available at <http://docs.sympy.org/latest/modules/matrices/matrices.html>
- ▶ Dynamical systems on Wikipedia, at https://en.wikipedia.org/wiki/Dynamical_system
- ▶ Equilibrium points on Scholarpedia, at <http://www.scholarpedia.org/article/Equilibrium>
- ▶ Bifurcation theory on Wikipedia, at https://en.wikipedia.org/wiki/Bifurcation_theory
- ▶ Chaos theory on Wikipedia, at https://en.wikipedia.org/wiki/Chaos_theory
- ▶ Further reading on dynamical systems, at https://en.wikipedia.org/wiki/Dynamical_system#Further_reading
- ▶ Lectures on ordinary differential equations on Awesome Math, at <https://github.com/rossant/awesome-math/#ordinary-differential-equations>

Getting started with Sage

Sage (<http://www.sagemath.org>) is a standalone mathematics software based on Python. It is an open source alternative to commercial products such as Mathematica, Maple, or MATLAB. Sage provides a unified interface to many open source mathematical libraries. These libraries include SciPy, SymPy, NetworkX, and other Python scientific packages, but also non-Python libraries such as ATLAS, BLAS, GSL, LAPACK, Singular, and many others.

In this recipe, we will give a brief introduction to Sage.

Getting ready

You can either:

- ▶ Install Sage on your local computer (<http://www.sagemath.org/doc/installation/>)
- ▶ Create Sage notebooks remotely in the cloud (<https://cloud.sagemath.com/>)

Being based on so many libraries, Sage is heavy and hard to compile from source. On Ubuntu, you can use the system's package manager (see <http://www.sagemath.org/download-linux.html>). Binaries exist for most systems except Windows, where you generally have to use VirtualBox (a virtualization solution: <http://www.virtualbox.org>).

Alternatively, you can use Sage in a browser with a Jupyter notebook running in the cloud.

Once Sage is installed, you can use it with Jupyter by typing the following command in a Terminal: `sage -n jupyter`.

How to do it...

Here, we will create a new Sage notebook and introduce the most basic features:

1. Sage accepts mathematical expressions as we would expect:

```
>>> 3 * 4  
12
```

2. Being based on Python, Sage's syntax is almost Python, but there are a few differences. For example, the power exponent is the more classical `^` symbol:

```
>>> 2 ^ 3  
8
```

3. Like in SymPy, symbolic variables need to be declared beforehand with the `var()` function. However, the `x` variable is always predefined. Here, we define a new mathematical function:

```
>>> f = 1 - sin(x) ^ 2
```

4. Let's simplify the expression of `f`:

```
>>> f.simplify_trig()  
cos(x)^2
```

5. Let's evaluate f on a given point:

```
>>> f(x=pi)  
1
```

6. Functions can be differentiated and integrated:

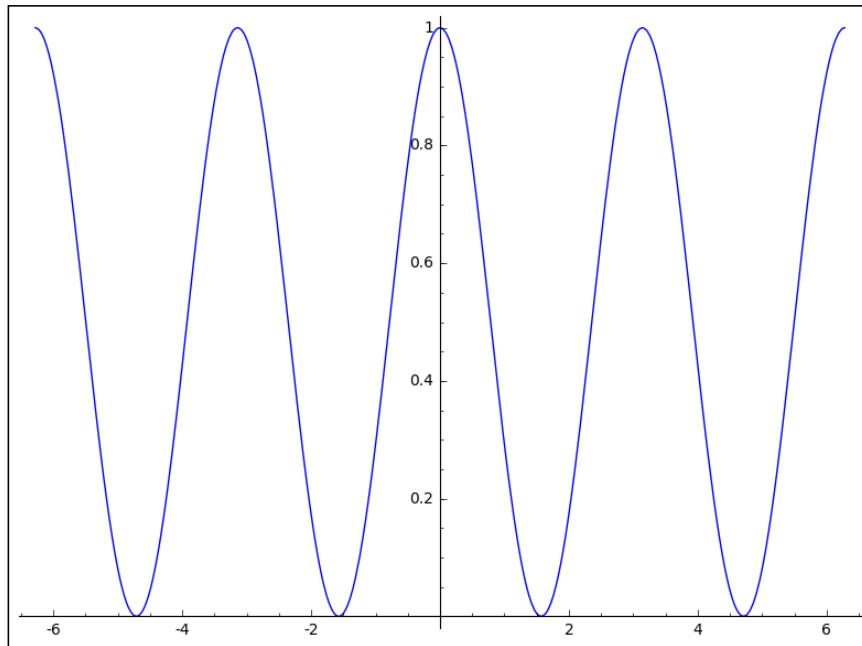
```
>>> f.diff(x)  
-2*cos(x)*sin(x)  
>>> f.integrate(x)  
1/2*x + 1/4*sin(2*x)
```

7. Sage also supports numerical computations in addition to symbolic computations:

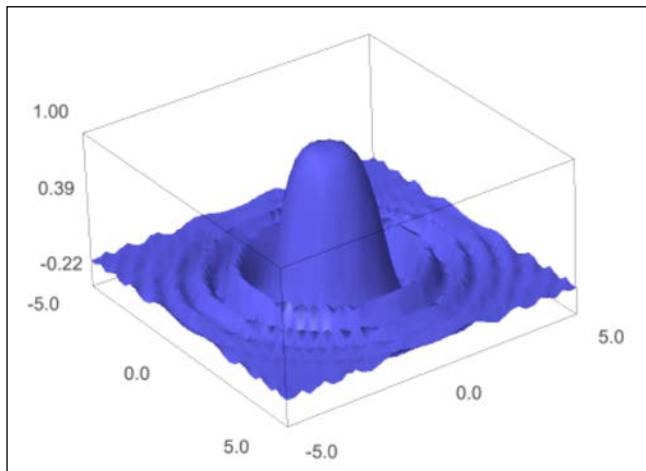
```
>>> find_root(f - x, 0, 2)  
0.6417143708729726
```

8. Sage also comes with rich plotting capabilities (including interactive plotting widgets):

```
>>> f.plot((x, -2 * pi, 2 * pi))
```



```
>>> x, y = var('x,y')
      plot3d(sin(x ^ 2 + y ^ 2) / (x ^ 2 + y ^ 2),
              (x, -5, 5), (y, -5, 5))
```



There's more...

This (too) short recipe cannot do justice to the huge list of possibilities offered by Sage. Many aspects of mathematics are covered: algebra, combinatorics, numerical mathematics, number theory, calculus, geometry, graph theory, and many others. Here are a few references:

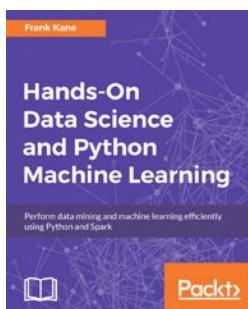
- ▶ An in-depth tutorial on Sage, available at <http://doc.sagemath.org/html/en/tutorial/index.html>
- ▶ The Sage reference manual, available at <http://doc.sagemath.org/html/en/reference/index.html>
- ▶ Videos on Sage, available at <http://www.sagemath.org/help-video.html>

See also

- ▶ The *Diving into symbolic computing with SymPy* recipe

Another Book You May Enjoy

If you enjoyed this book, you may be interested in another book by Packt



Hands-On Data Science and Python Machine Learning

Frank Kane

ISBN: 978-1-78728-074-8

- ▶ Learn how to clean your data and ready it for analysis
- ▶ Implement the popular clustering and regression methods in Python
- ▶ Train efficient machine learning models using decision trees and random forests
- ▶ Visualize the results of your analysis using Python's Matplotlib library
- ▶ Use Apache Spark's MLlib package to perform machine learning on large datasets

Leave a review – let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

A

accelerate.profiling
reference 131
adaptive histogram equalization
URL 386
adjacency matrix 450
Air resistance
URL 426
Altair
about 217
plots, creating 234-239
references 239
Anaconda
about 4
URL 4, 161
analog signals 360
array computations
accelerating, with NumExpr 165-167
asynchronous parallel tasks
interacting, in IPython 194-196
AsyncResult
references 197
audio filters
URL 407
audio signal processing
URL 382, 407
augmented matrix 492
AutoHotKey
URL 68
Awesome Math
references 497

B

bagging 319
ball trees 308
band-pass filter 375
basin-hopping algorithm 344, 347
Bayesian methods
about 244, 253-255
Bayes' theorem 255
conjugate distributions 257
credible interval 257
Maximum a posteriori (MAP) 256, 257
non-informative (objective)
prior distributions 257
posterior distribution, computation 256
Bayesian model
applying, from a posterior distribution with
MCMC 273-278
Bayes' theorem 255
benchmarking 127
Bernoulli distribution
URL 250
Bernoulli Naive Bayes classifier 311
bias-variance dilemma
URL 289
bias-variance tradeoff
URL 272
bifurcation diagram
plotting, of chaotic dynamical
system 414-418
URL 418
Bifurcation theory
URL 507

Big O 494
binder
 URL 97
binomial distribution 254
Birnbaum-Sanders distribution
 about 266
 URL 266
bisection method
 reference 341
Bitbucket
 URL 52
bivariate method 243
Blinn-Phong shading model
 URL 182
block 188
blocking mode 192
Bokeh
 interactive web visualizations,
 creating 218-224
 URL 218
Boolean propositional formula
 searching, from truth table 502, 503
 URL 503
bootstrap aggregating
 URL 323
boundary conditions 413
bqplot 229
branching
 references 62
 workflow 58-62
Brent's method
 reference 341
broadcasting 22
Brownian motion
 references 444
 simulating 442-444
Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm 343
B-tree 152
Butterworth filter 375

C

calculus
 URL 448, 495
Cameron Davidson-Pilon
 URL 278

cardinal sine
 URL 342
Cartopy
 geospatial data, manipulating 478-480
 URL 478
cascade
 references 403
cascade classification API
 URL 404
causal filters 374
cells 477
cellular automaton 412
Chaos theory
 URL 418, 507
chaotic dynamical system
 about 413
 bifurcation diagram, plotting 414-418
Chinese Remainder Theorem
 about 500
 references 501
chi-squared test
 about 261
 references 262
 used, for estimating correlation between two variables 258-261
choropleth map 477
chromatic scale
 URL 410
chunks 152
classification 287
Classification and Regression Trees (CART) algorithm 322
C library
 wrapping, in Python with ctypes 167-171
client 84
clustering
 hidden structures, detecting
 in dataset 328-332
 URL 332
clusters 328
CMA-ES algorithm
 URL 348
Codeship
 URL 78
column-major order 140
Comma-separated Values (CSV) 14

CommonMark
URL 67

complex systems
URL 418

components 286

Comprehensive R Archive Network (CRAN)
URL 283

compressed sensing
about 361, 362
references 362

Computational Fluid Dynamics
URL 413

concurrent programming 155

conda
URL 72

conditional probability distribution 255

connected-component labeling
URL 471

connected components
computing, in image 467-471
URL 451, 471

constrained optimization 337

constrained optimization algorithm 356

contiguous block 152

contingency table
about 261
URL 262
used, for estimating correlation between two variables 258-261

continuous functions 336

continuous integration 78

continuous optimization 335

contrast
URL 386

Contrast Limited Adaptive Histogram Equalization (CLAHE) 384

convex functions 336

convex optimization 336

convolution 374

Conway's Game of Life
references 422

corner detection
URL 400

counting process 440

Covariance Matrix Adaptation Evolution Strategy (CMA-ES) algorithm 348

coverage.py module
URL 78

cProfile
about 127
reference 131
used, for code profiling in IPython 127-130

CPython 155

credible intervals
about 257
URL 257

cross-validation
references 298

ctypes
C library, wrapping in Python 167-171
URL 171

CUDA
massively parallel code, writing for NVIDIA
graphics cards 184-190
references 190

CUDA cores 188

Cumulative Distribution Function (CDF) 266

curve fitting 350

custom Jupyter Notebook widgets
creating, in HTML 104-106
creating, in JavaScript 104-106
creating, in Python 104-106

custom magic commands
IPython extension, creating 24-27

custom widget
URK 107
URL 103

Cython
code, optimizing 175-182
Python code, accelerating 171-175
references 175, 182
used, for releasing GIL 182-184

D

D3.js
NetworkX graph, visualizing
in Notebook 224-229
URL 224

Task
out-of-core computations, performing on large arrays 197-201

references 202
URL 68

data buffer 139

data dimensionality
reducing, with principal component analysis 324-327

data manipulation
references 19

data normalization 288

dataset
about 151
exploring, with Matplotlib 245-248
exploring, with pandas 245-248

Datashader
URL 224

data type (dtype) 22

data visualization 324

decision tree learning
URL 323

decision trees 319

decompositions
URL 327

deep learning
references 291

defensive programming 70

Delaunay triangulation
URL 477

dependencies
functional dependency 193
graph dependency 193

design patterns 70

deterministic algorithm 338

dichotomy method 339

differentiable functions 336

digital filters
applying, to speech sounds 404-407

digital signals
about 360, 373
linear filter, applying 370-373
resolution 360
sampling rate 360

Dijkstra's algorithm
URL 486

dilation 396

dill
URL 68

dimensionality 286

Directed Acyclic Graph (DAG)
about 464
references 467

direct interface 192

Discrete Fourier Transform (DFT) 367

discrete optimization 335

discrete-time dynamical system 412

discrete-time Markov chain
simulating 434-437

distributed version control system 52-57

Docker
URL 68

dot-com bubble burst 373

dynamical systems
about 411
differential equations 412, 413
references 413
types 412
URL 507

Dynamic Random Access Memory (DRAM) 190

E

Eclipse/PyDev 65

elastic potential energy
about 357
URL 357

elementary cellular automaton
simulating 419-422

embarrassingly parallel
URL 183

empirical distribution function 266

ensemble learning
about 319
URL 323

equal temperament
URL 410

equilibrium points
URL 507

erosion 396

estimation 243

Eulerian paths
URL 451

Euler-Maruyama method
URL 448

- Euler method**
URL 426
- expectation-maximization algorithm**
about 332
URL 332
- exploratory data analysis**
about 18
using, in Jupyter Notebook 13-17
- exploratory methods** 242
- F**
- Fast Fourier Transform (FFT)**
discrete Fourier transform 367, 368
Inverse Fourier Transform 369
references 369
used, for analyzing frequency components 363-367
- feature** 286
- feature extraction** 288
- feature scaling** 288
- feature selection**
URL 289
- filters**
applying, on image 386-390
- Finite Impulse Response (FIR)** 374
- FitzHugh-Nagumo equation** 427
- FitzHugh-Nagumo system**
URL 432
- fixtures** 77
- Flake8**
URL 71
- flight routes**
drawing, with NetworkX 457-461
- flood-fill algorithm**
URL 471
- fluid dynamics** 412
- Fokker-Planck equation**
URL 444
- Force-directed graph drawing**
URL 457
- forking** 62
- Fourier transform** 361
- frequentist and Bayesian methods** 244
- frequentist methods**
about 244
URL 244
- Fruchterman-Reingold force-directed algorithm** 456
- f-strings** 48
- fundamental frequency** 409
- G**
- Gaussian filter**
URL 391
- Gaussian kernel** 272
- General Purpose Programming on Graphics Processing Units (GPGPU)** 184
- geodetic coordinate system** 270
- geographical distances**
references 486
- Geographic Information Systems (GIS)**
about 450
using, in Python 452
- geometry**
references 452
using, in Python 451
- geospatial data**
manipulating, with Cartopy 477-480
- Git**
references 58
- git-flow**
references 62
- GitHub**
URL 52
- GitLab**
URL 52
- Git Large File Storage (Git LFS)**
URL 55
- Global Interpreter Lock (GIL)**
references 155
- global minimum** 336
- gradient**
URL 348
- gradient descent** 347
- graph** 450
- graph coloring**
URL 450
- Graph drawing**
URL 457
- Graphics Processing Units (GPUs)** 154, 184
- graph, problems**
connected components 451

Eulerian paths 451
graph coloring 450
graph traversal 450
Hamiltonian paths 451
traveling salesman problem 451

graphs
about 449, 450
manipulating, with NetworkX 453-456
random graphs 451
references 452
using, in Python 451
visualizing, with NetworkX 453-456

graph traversal
URL 450

Graphviz
URL 321

gravitational force 357

grayscale image 382

great-circle distance 486

grid 188

grid search
URL 299

Gross Domestic Product (GDP) 477

groups 151, 328

H

h5py
reference 152

Haar cascades library
URL 403

Hamiltonian paths
URL 451

handwritten digit recognition 287

Hartman-Grobman theorem 506

HDF5 chunking
reference 152

heat equation 444

Hessian 347

Hierarchical Data Format (HDF5)
limitations, reference 152
used, for manipulating large arrays 150-152

high-pass filter 375

histogram equalization
URL 385

holding times 441

HoloViews
interactive web visualizations,
creating 218-224

Hooke's law
URL 358

hyperbolic 506

Hyper-Threading Technology (HTT) 166

I

image
about 382
faces, detecting with OpenCV 401-403
filters, applying 386-390
points of interest, searching 397-400
segmenting 391-396

image denoising 390

image exposure
manipulating 383-385

image histogram
URL 385

image processing
URL 382, 400

image segmentation
URL 396

independent variables 412

Infinite Impulse Response (IIR) 374

in-place operation
and implicit-copy operation, differences 140

instance-based learning
URL 309

Integrated Development Environments (IDEs) 63

Intel Math Kernel Library (MKL) 139

intensity 382

interactive computing
about 41
workflow, with IPython 63

InteractiveShell class 27

interactive visualization libraries
discovering, in Notebook 229-233
references 233

interactive web visualizations
creating, with Bokeh 218-224
creating, with HoloViews 218-224

intermediate value theorem
reference 341

Inverse Discrete Fourier Transform 369
Inverse Fast Fourier Transform 369
ipyleaflet 229
ipynd module
 URL 65
ipyparallel
 about 68, 190
 references 194
IPython
 about 2, 6-12
 asynchronous parallel tasks,
 interacting 194-196
 code, debugging 79-81
 command time, evaluating 126, 127
 cProfile, used for code profiling 127-130
 debugging 80
 IDEs, references 65
 Integrated Development
 Environments (IDEs) 65
 interactive computing workflows 63
 Jupyter Notebook 64, 65
 post-mortem mode 79
 Python code, distributing across multiple
 cores 190-193
 Terminal 64
 text editor 64
 URL 64
IPython Blocks
 URL 86
 used, as programming tutorial
 in Notebook 86-90
IPython configuration system
 configurable class 31, 32
 configuration file 31
 configuration object 31
 HasTraits class 31
 Magics class 32
 references 32
 user profile 31
IPython extension
 creating, with custom magic
 commands 24-27
 InteractiveShell class 27
 loading 28
 references 28
IPython Notebook 2
IPython's configuration system
 mastering 29-31
ipyvolume 229
ipywidgets
 about 97-103
 URL 103
Iris dataset
 URL 327
Iris flower dataset
 URL 327
IRkernel
 URL 279
iterated functions
 URL 418

J

Jacobian 505
Jacobian matrix 506
JavaScript Object Notation (JSON) 83
Jeffreys prior
 URL 257
Jinja2
 URL 97
Joblib
 URL 68
Julia
 references 207
 URL 202
 using, in Jupyter Notebook 202-206
Jupyter
 about 2
 kernel, creating 33-39
JupyterHub
 about 85
 URL 85
JupyterLab
 about 65, 84, 111-123
 references 123
Jupyter Notebook
 about 6-12
 architecture 84
 clients, connecting to kernel 84
 configuring 107-110
 converting, with nbconvert 91-97
 data, analyzing with R 278-283

exploratory data analysis 13-18
Julia, using 202-206
JupyterHub 85
programming tutorial,
 with IPython Blocks 86-90
references 12, 111
security 85
widgets 97-103

Just-In-Time (JIT) compilation
Python code, accelerating 161-164

K

Kaggle
 references 299
 URL 309

K-D trees 308

kernel
 about 84, 188
 clients, connecting 84
 creating, for Jupyter 33-39
 references 39

Kernel Density Estimation (KDE)
 URL 273

K-means clustering algorithm
 URL 332

K-nearest neighbors (K-NN) classifier
 handwritten digits, recognizing 305-308

K-NN algorithm
 references 309

Kolmogorov-Smirnov test
 about 266
 URL 267

L

L² norm 297

Langevin equation
 about 444
 URL 448

Laplacian matrix
 about 456
 URL 457

large arrays
 manipulating, with Hierarchical Data Format
 (HDF5) 150-152

LaTeX
 references 488
 URL 91

L-BFGS-B algorithm
 URL 358

least squares method
 references 283

Leave-one-out cross-validation (LOOCV) 298

left singular vectors 327

Levenberg-Marquardt algorithm 351

Lévi function 345

linear algebra
 references 493

linear combination 143

linear filter
 about 373
 applying, to digital signal 370-373
 band-pass filter 375
 convolution 374
 filters, using in frequency domain 375
 FIR 374, 375
 high-pass filter 375
 IIR filters 374, 375
 low-pass filter 375
 references 376

Linear Time-Invariant (LTI) 374

line_profiler
 reference 133
 used, for line-by-line code profiling 131-133

Lloyd's algorithm 332

load-balanced interface 192

locality of reference 139

local minimum 336

logic lectures
 URL 504

logic module
 URL 503

logistic map
 URL 418

logistic regression
 references 304
 using, for prediction 299-303

loss function 296

Lotka-Volterra (predator-prey) equations
 implementing 504-506

- Low Level Virtual Machine (LLVM)** [163](#)
- low-pass filter** [375](#)
- Lyapunov exponent**
URL [419](#)
- M**
- machine learning**
references [290](#)
- maps**
references [452](#)
- Markov chain Monte Carlo (MCMC) method** [273](#)
Bayesian model, applying from posterior distribution [273-278](#)
URL [278](#)
- Markov chains**
references [438](#)
- Markov property**
URL [434](#)
- mathematical function**
minimizing [341-347](#)
references, for root finding [341](#)
root, finding [338-341](#)
- mathematical morphology**
URL [396](#)
- mathematical morphology techniques** [395](#)
- mathematical optimization**
about [335](#)
constrained optimization [337](#)
deterministic algorithm [338](#)
global minima [336, 337](#)
local minima [336, 337](#)
objective function [336](#)
references [338](#)
stochastic algorithm [338](#)
unconstrained optimization [337](#)
URL [348](#)
- MathJax** [488](#)
- Matplotlib**
about [3](#)
dataset, exploring [245-248](#)
URL [4](#)
- Matplotlib styles**
references [213](#)
using [209-212](#)
- matrix** [22](#)
- matrix documentation**
URL [507](#)
- Maximum a posteriori (MAP)** [256, 257](#)
- maximum likelihood method**
data, probability distribution
applying [262-267](#)
URL [267](#)
- memory mapping**
used, for processing NumPy arrays [148, 149](#)
- memory_profiler**
reference [135](#)
used, for memory profiling [134, 135](#)
- Metaheuristics**
URL [348](#)
- Metropolis-Hastings algorithm**
about [273, 278](#)
URL [278](#)
- Microsoft Visual Studio**
URL [156](#)
- Milstein method**
URL [448](#)
- model evaluation**
URL [290](#)
- model selection**
about [289](#)
URL [290](#)
- Monte Carlo methods**
URL [438](#)
- multidimensional array**
using, in NumPy for array computations [19-23](#)
- multiprocessing module** [190](#)
- multivariate method** [243](#)
- N**
- Naive Bayes**
for Natural Language Processing [309](#)
- Naive Bayes classifier**
references [312](#)
- Natural Earth**
URL [477](#)
- natural language processing**
references [313](#)
- Navier-Stokes equations**
URL [413](#)

nbconvert
Jupyter notebook, converting 92-97
URL 12, 97

nbconvert
Jupyter notebook, converting 91
URL 91

nbformat
URL 91

nbviewer
URL 12, 97

NetworkX
flight routes, drawing 457-461
graphs, manipulating 453-456
graphs, visualizing 453-456

NetworkX graph
references 229
visualizing, in Notebook
with D3.js 224-229

Neumann boundary conditions
URL 432

Newton's laws of motion
URL 426

Newton's method
about 341
reference 341
URL 348

nodes 450

noise reduction
URL 390

non-contiguous 137

nonlinear kernels 316

nonlinear least squares
function, fitting to data 349-351
references 351

nonlinear least squares curve fitting 349

nonparametric estimation 268

nonparametric model 245

Notebook
sound synthesizer, creating 408, 409

NPY file format
reference 152

interact 12

null hypothesis 249

Numba
about 154
Python code, accelerating 161-164

references 165
URL 161

number-theory module
references 501

Numerical Tours
URL 382

NumExpr
about 154
array computations, accelerating 165-167
references 167

NumPy
about 4
broadcasting rules 141
multidimensional array, using for array
computations 19-23
references 24
stride tricks, using 142-144
unnecessary array copies, avoiding 135-138
URL 4

NumPy arrays
about 149
efficiency 139
processing, with memory mapping 148
reshaping, without copy 140, 141

NumPy broadcasting rules
about 141
references 141

NVIDIA graphics cards (GPUs)
massively parallel code,
writing with CUDA 184-190

Nyquist criterion 361

Nyquist-Shannon sampling theorem 361

O

objective function 336

observation 286

offset 136

Online Python Tutor
reference 133

OpenCL 184

OpenCV (Open Computer Vision)
URL 382
used, for detecting faces in image 401-403

OpenFlights
URL 457

OpenMP
used, for releasing GIL 182-184

ordinary differential equations
URL 507

Ordinary Differential Equations (ODEs)
about 412
references 426
simulating, with SciPy 422-425

Ordinary Least Squares (OLS) regression 296

Ornstein-Uhlenbeck process 444

orthodromic distance 486

Otsu's method
URL 396

outer product 138

out-of-core computations 148

overfitting
about 294
URL 289

P

pandas
about 3
dataset, exploring 245-248
references 249

pandas 0.21
URL 4

pandoc
URL 91

parameter vector 296

parametric estimation method 268

parametric method 244

partial derivatives 412

Partial Differential Equations (PDEs)
about 412
references 431
simulating 427-431

partitions 287

Pearson correlation coefficient
about 261
URL 261

physical system
equilibrium state, finding by potential energy
minimization 352-357

pip 5

pipes 46

plate carrée 270

Plotly
URL 224

podoc module
URL 65

point process 438

points of interest
searching, of image 397-400

Poisson process
about 275
references 441
simulating 438-441
URL 275

polynomial interpolation
with linear regression 297

potential energy
URL 357

Power Spectral Density 365

prediction 243

prime-counting function 499

prime number theorem 499

principal component 327

principal component analysis (PCA)
about 324
data dimensionality, reducing 324-327

principle of minimum energy
URL 358

principle of minimum total
potential energy 357

prior probability distribution 253

probabilistic model 244

probability distribution
applying, to data with maximum likelihood
method 262-267

probability distribution nonparametrically
estimating, with kernel density
estimation 268-273

Probability Mass Function (PMF) 254

probit model
URL 287

profile 125

profiling 127

pstats
reference 131

pull request 62

pure tone 409

PyCall 207

PyCharm 65

pydot 323
pyjulia 207
Pylint
 URL 71
PyMC3
 references 273
PyPy
 URL 154
PyTables optimization guide
 reference 152
py.test
 unit tests, writing 73-77
Python
 about 2
 C library, wrapping with ctypes 167-171
 compilers, using 156
 installing 4, 5
 profiling tools, reference 131
 references 5, 155
Python 3
 features, using 46-51
 references 51, 52
 URL 161
Python code
 accelerating, with Cython 171-174
 accelerating, with Just-In-Time
 | compilation 161-164
 accelerating, with Numba 161-164
 distributing, across multiple cores with
 IPython 190-193
 improving 156-160
 references 73
 writing 70-72
Python debuggers
 URL 81
**Python Enhancement Proposal
 number 8 (PEP8)**
 URL 71
Python Package Index (PyPI)
 URL 5
Python Tools for Visual Studio (PTVS) 65
pythreejs 229

Q

Quasi-Newton methods
 about 347
 URL 348
Quine-McCluskey algorithm
 about 503
 URL 504

R

R
 data, analyzing in Jupyter
 Notebook 278-283
 references 283
 URL 278
Rackspace
 URL 97
Radial Basis Function (RBF) 316
Random Access Memory (RAM) 139
random forest
 about 319
 used, for selecting features
 for regression 319-322
 URL 323
RandomForestRegressor
 URL, for API 323
random graphs 451
random subspace method 323
 URL 323
random variable 253
Ray tracing
 URL 182
reachability relation 471
reaction-diffusion systems
 about 427
 references 432
 URL 432
Read-Evaluate-Print Loop (REPL) 84
real analysis
 references 495
real-valued functions
 analyzing 493, 494
rebasing 61

red, green, and blue (RGB) 382
regions 477
regression 287
regression analysis
 references 283
regularization
 URL 289
regularization term 294
reproducible interactive computing experiments
 conducting 66-69
 references 69
RequireJS
 URL 107
reStructuredText (reST)
 URL 67
ridge regression 294, 297
RISE
 URL 97
robust model 289
Rodeo 65
rolling average algorithm
 about 145
 implementing, with stride tricks 145-147
rolling mean 17
route planner
 creating, for road network 481-486
row-major order 140
rpy2
 URL 279
Rule
 about 110
 URL 422
Rule 110 automaton 422
rule of thumb 272

S

Sage
 about 488, 507-510
 references 508-510
 URL 507
sample 286
Scalable Vector Graphics (SVG) 10
scientific Python
 references 6

scikit-learn
 about 291-296
 cross-validation 298
 grid search 298
 Ordinary Least Squares regression 296
 polynomial interpolation,
 with linear regression 297
 references 299
 ridge regression 297
 scikit-learn API 296
 text data, handling 309-312
 URL 289-308
SciPy
 ODEs, simulating 422-425
 URL 3
SciPy 1.0
 URL 4
SciPy ecosystem 3, 4
Scott's Rule 272
seaborn
 references 217, 218
 statistical plots, creating 214-217
sequential locality 139
Shapefile
 URL 477
shortest paths
 in NetworkX, references 486
 URL 451
sigmoid function 303
SIMD paradigm 184
Single Instruction, Multiple Data (SIMD) 135, 164
Singular Value Decomposition (SVD) 327
SnakeViz
 reference 131
Sobel filter
 URL 391
sounds 382
sound synthesizer
 creating, in Notebook 408, 409
spam filtering 287
sparse decomposition 362
sparse matrices
 about 149
 reference 149
sparse matrix 310

spatial locality 139
Sphinx
 URL 67
Split Bregman algorithm
 URL 391
Split Bregman method 390
Spyder 65
state diagram 437
statistical data analysis 242
statistical hypothesis testing
 about 249-252
 URL 252
statistical inference 243
Statistical Learning
 URL 290
statistical plots
 creating, with seaborn 214-217
statistics
 references 245
stats module
 URL 497
Stochastic cellular automata 433
Stochastic Differential Equations (SDEs)
 about 434
 simulating 444-447
 URL 448
stochastic dynamical systems
 URL 434
Stochastic Partial Differential Equations (SPDEs) 434
Stochastic processes
 URL 434
stream processors 188
strided indexing scheme 143
strides 140
stride tricks
 used, for implementing rolling average
 algorithm 145-147
 using, in NumPy 142-144
structure tensor
 about 400
 URL 400
structuring element 396
Sum of Products
 URL 503
supervised learning 286
Support Vector Classifier (SVC) 314

support vector machines (SVM)
 references 318
 URL 313
 using, for classification tasks 313-318
SVD decomposition
 URL 327
symbolic computing
 exploring, with SymPy 488-491
Sympy
 about 487
 equations, solving 491, 492
 inequalities, solving 491, 492
 number theory 498-501
 probabilities, computing 495-497
 random variables, manipulating 495-497
 used, for symbolic computing
 exploration 488-491
synthesizer
 URL 410

T

t-Distributed Stochastic Neighbor Embedding (t-SNE)
 references 333
term frequency-inverse document frequency (tf-idf) 312
 URL 312
test-driven development (TDD)
 about 78
 URL 79
test functions for optimization
 URL 345
test set 286
test statistic 249
text feature extraction
 URL 312
thread 188
Tikhonov regularization
 URL 297
timbre 409
time series
 about 359, 379
 autocorrelation, computing 376-379
 references 379, 380
topological sort documentation
 URL 467

topological sorting
dependencies, resolving in directed acyclic graph 463-467
URL 467

total variation 390

total variation denoising 390

trace module
reference 133

training set 286

traitlets package
URL 29

transition matrix 437

Travis CI
URL 78

two-dimensional array 22

U

underfitting 289

Uniform Manifold Approximation and Projection (UMAP)
URL 332

uninformative priors
URL 257

unit tests
test coverage 78
workflows 78
writing, with py.test 73-77

univariate method 243

Unix shell
about 42-46
references 46

unsupervised learning
about 286, 287
clustering 288
density estimation 288
dimension reduction 288
manifold learning 288
methods 324
URL 328

V

Vandermonde matrix
URL 297

variable 286

variance 289

vector 22

vectorized instructions 139

vectorizer
URL 312

Vega 217, 234

Vega-Lite
plots, creating 234-239
references 239

vertices 450

views 140

Viola-Jones object detection framework
about 402
URL 404

VirtualBox
URL 508

vocabulary 310

voice frequency
URL 407

Voronoi diagram
computing, of set of points 471-477
URL 477

W

wavelet transform 369

white box model 323

white noise

URL 448

Wiener process
URL 444

Windows compilers
URL 156

Windows operating system
URL 42

Wolfram code
URL 422

X

xarray library
URL 224

Z

Zachary's Karate Club graph 225

ZeroMQ (ZMQ)
URL 84