



Quick answers to common problems

NumPy Cookbook

Over 70 interesting recipes for learning the Python open source mathematical library, NumPy

Ivan Idris

[PACKT] open source*
PUBLISHING community experience distilled

NumPy Cookbook

Over 70 interesting recipes for learning the Python open source mathematical library, NumPy

Ivan Idris

[PACKT] open source 
PUBLISHING community experience distilled

BIRMINGHAM - MUMBAI

NumPy Cookbook

Copyright © 2012 Packt Publishing

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

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

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

First published: October 2012

Production Reference: 1181012

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

ISBN 978-1-849518-92-5

www.packtpub.com

Cover Image by Avishek Roy (roy007avishek88@gmail.com)

Credits

Author

Ivan Idris

Project Coordinator

Vishal Bodwani

Reviewers

Alexandre Devert

Ludovico Fischer

Ryan R. Rosario

Proofreader

Clyde Jenkins

Indexer

Monica Ajmera Mehta

Acquisition Editor

Usha Iyer

Production Coordinators

Arvindkumar Gupta

Manu Joseph

Lead Technical Editor

Ankita Shashi

Cover Work

Arvindkumar Gupta

Manu Joseph

Technical Editors

Merin Jose

Rohit Rajgor

Farhaan Shaikh

Nitee Shetty

Copy Editor

Insiya Morbiwala

About the Author

Ivan Idris has an MSc in Experimental Physics. His graduation thesis had a strong emphasis on Applied Computer Science. After graduating, he worked for several companies as a Java Developer, Data Warehouse Developer, and QA Analyst. His main professional interests are business intelligence, big data, and cloud computing. He enjoys writing clean, testable code, and interesting technical articles. He is the author of *NumPy 1.5 Beginner's Guide*. You can find more information and a blog with a few NumPy examples at ivanidris.net.

I would like to dedicate this book to my family and friends. I would like to take this opportunity to thank the reviewers and the team at Packt for making this book possible. Thanks also goes to my teachers, professors, and colleagues, who taught me about science and programming. Last but not least, I would like to acknowledge my parents, family, and friends for their support.

About the Reviewers

Alexandre Devert is a computer scientist. To put his happy obsessions to good use, he decided to solve optimization problems, in both academic and industrial contexts. This included all kinds of optimization problems, such as civil engineering problems, packing problems, logistics problems, biological engineering problems—you name it. It involved throwing lots of science on the wall and seeing what sticks. To do so, he had to analyze and visualize large amounts of data quickly, for which Python, NumPy, Scipy, and Matplotlib excel. Thus, the latter are among the daily tools he has been using for a couple of years. He also lectures on Data mining at the University of Science and Technology of China, and uses those very same tools for demonstration purposes and to enlighten his students with graphics glittering of anti-aliased awesomeness.

I would like to thank my significant other for her understanding my usually hefty work schedule, and my colleagues, for their patience with my shallow interpretation of concepts such as a "deadline".

Ludovico Fischer is a software developer working in the Netherlands. By day, he builds enterprise applications for large multinational companies. By night, he cultivates his academic interests in mathematics and computer science, and plays with mathematical and scientific software.

Ryan R. Rosario is a Doctoral Candidate at the University of California, Los Angeles. He works at Riot Games as a Data Scientist, and he enjoys turning large quantities of massive, messy data into gold. He is heavily involved in the open source community, particularly with R, Python, Hadoop, and Machine Learning, and has also contributed code to various Python and R projects. He maintains a blog dedicated to Data Science and related topics at <http://www.bytemining.com>. He has also served as a technical reviewer for *NumPy 1.5 Beginner's Guide*.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

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.



<http://PacktLib.PacktPub.com>

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

Why Subscribe?

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

Free Access for Packt account holders

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

Table of Contents

| | |
|---|-----------|
| Preface | 1 |
| Chapter 1: Winding Along with IPython | 5 |
| Introduction | 5 |
| Installing IPython | 6 |
| Using IPython as a shell | 8 |
| Reading manual pages | 10 |
| Installing Matplotlib | 11 |
| Running a web notebook | 12 |
| Exporting a web notebook | 14 |
| Importing a web notebook | 16 |
| Configuring a notebook server | 20 |
| Exploring the SymPy profile | 23 |
| Chapter 2: Advanced Indexing and Array Concepts | 25 |
| Introduction | 25 |
| Installing SciPy | 26 |
| Installing PIL | 28 |
| Resizing images | 29 |
| Creating views and copies | 32 |
| Flipping Lena | 34 |
| Fancy indexing | 36 |
| Indexing with a list of locations | 38 |
| Indexing with booleans | 40 |
| Stride tricks for Sudoku | 42 |
| Broadcasting arrays | 45 |
| Chapter 3: Get to Grips with Commonly Used Functions | 49 |
| Introduction | 50 |
| Summing Fibonacci numbers | 50 |
| Finding prime factors | 54 |

| | |
|---|------------|
| Finding palindromic numbers | 56 |
| The steady state vector determination | 58 |
| Discovering a power law | 64 |
| Trading periodically on dips | 67 |
| Simulating trading at random | 70 |
| Sieving integers with the Sieve of Erasthothenes | 72 |
| Chapter 4: Connecting NumPy with the Rest of the World | 75 |
| Introduction | 75 |
| Using the buffer protocol | 76 |
| Using the array interface | 79 |
| Exchanging data with MATLAB and Octave | 80 |
| Installing RPy2 | 82 |
| Interfacing with R | 82 |
| Installing JPy2 | 84 |
| Sending a NumPy array to JPy2 | 84 |
| Installing Google App Engine | 86 |
| Deploying NumPy code in the Google cloud | 88 |
| Running NumPy code in a Python Anywhere web console | 90 |
| Setting up PiCloud | 92 |
| Chapter 5: Audio and Image Processing | 95 |
| Introduction | 95 |
| Loading images into memory map | 96 |
| Combining images | 100 |
| Blurring images | 104 |
| Repeating audio fragments | 108 |
| Generating sounds | 110 |
| Designing an audio filter | 114 |
| Edge detection with the Sobel filter | 117 |
| Chapter 6: Special Arrays and Universal Functions | 121 |
| Introduction | 121 |
| Creating a universal function | 121 |
| Finding Pythagorean triples | 122 |
| Performing string operations with chararray | 124 |
| Creating a masked array | 125 |
| Ignoring negative and extreme values | 128 |
| Creating a scores table with recarray | 131 |
| Chapter 7: Profiling and Debugging | 135 |
| Introduction | 135 |
| Profiling with timeit | 135 |
| Profiling with IPython | 139 |

| | |
|--|------------|
| Installing line_profiler | 142 |
| Profiling code with line_profiler | 143 |
| Profiling code with the cProfile extension | 144 |
| Debugging with IPython | 146 |
| Debugging with pdb | 148 |
| Chapter 8: Quality Assurance | 151 |
| <hr/> | |
| Introduction | 151 |
| Installing Pyflakes | 151 |
| Performing static analysis with Pyflakes | 152 |
| Analyzing code with Pylint | 153 |
| Performing static analysis with Pychecker | 155 |
| Testing code with docstrings | 156 |
| Writing unit tests | 158 |
| Testing code with mocks | 162 |
| Testing the BDD way | 164 |
| Chapter 9: Speed Up Code with Cython | 169 |
| <hr/> | |
| Introduction | 169 |
| Installing Cython | 170 |
| Building a Hello World program | 170 |
| Using Cython with NumPy | 172 |
| Calling C functions | 173 |
| Profiling Cython code | 175 |
| Approximating factorials with Cython | 178 |
| Chapter 10: Fun with Scikits | 183 |
| <hr/> | |
| Introduction | 183 |
| Installing scikits-learn | 184 |
| Loading an example dataset | 184 |
| Clustering Dow Jones stocks with scikits-learn | 185 |
| Installing scikits-statsmodels | 189 |
| Performing a normality test with scikits-statsmodels | 190 |
| Installing scikits-image | 191 |
| Detecting corners | 191 |
| Detecting edges | 193 |
| Installing Pandas | 194 |
| Estimating stock returns correlation with Pandas | 195 |
| Loading data as pandas objects from statsmodels | 198 |
| Resampling time series data | 200 |
| Index | 205 |

Preface

We, NumPy users, live in exciting times. New NumPy-related developments seem to come to our attention every week or maybe even daily. When this book was being written, NumPy Foundation of Open Code for Usable Science was created. The Numba project—NumPy-aware, dynamic Python compiler using LLVM—was announced. Also, Google added support to their Cloud product Google App Engine.

In the future, we can expect improved concurrency support for clusters of GPUs and CPUs. OLAP-like queries will be possible with NumPy arrays.

This is wonderful news, but we have to keep reminding ourselves that NumPy is not alone in the scientific (Python) software ecosystem. There is Scipy, Matplotlib (a very useful Python plotting library), IPython (an interactive shell), and Scikits. Outside of the Python ecosystem, languages such as R, C, and Fortran are pretty popular. We will go into the details of exchanging data with these environments.

What this book covers

Chapter 1, Winding Along with IPython, covers IPython that is a toolkit, mostly known for its shell. The web-based notebook is a new and exciting feature, which we will cover in detail. Think of Matlab and Mathematica, but in your browser, that is open source and free.

Chapter 2, Advanced Indexing and Array Concepts, describes some of NumPy's more advanced and tricky indexing techniques. NumPy has very efficient arrays that are easy to use due to their powerful indexing mechanism.

Chapter 3, Get to Grips with Commonly Used Functions, makes an attempt to document the most essential functions that every NumPy user should know. NumPy has many functions, too many to even mention in this book.

Chapter 4, Connecting NumPy with the Rest of the World, shows us that the number of programming languages, libraries, and tools that one encounters in the real world is mind-boggling. Some of the software runs on the Cloud, and some of it lives on your local machine or a remote server. Being able to fit and connect NumPy in such an environment is just as important as being able to write standalone NumPy code.

Chapter 5, Audio and Image Processing, shows you a different view of NumPy. So when you think of NumPy after reading this chapter, you'll probably think of sounds or images too.

Chapter 6, Special Arrays and Universal Functions, covers technical topics, such as special arrays and universal functions. It will help us learn how to perform string operations, ignore illegal values, and store heterogeneous data.

Chapter 7, Profiling and Debugging, will demonstrate several convenient profiling and debugging tools necessary to produce a great software application.

Chapter 8, Quality Assurance, will discuss common methods and techniques such as unit testing, mocking, and BDD, including the NumPy testing utilities, as quality assurance deserves a lot of attention.

Chapter 9, Speed Up Code with Cython, shows how Cython works from the NumPy perspective. Cython tries to combine the speed of C and the strengths of Python.

Chapter 10, Fun with Scikits, gives us a quick tour through some of the most useful Scikits projects. Scikits are a yet another part of the fascinating, scientific Python ecosystem.

What you need for this book

To try out the code samples in this book, you will need a recent build of NumPy. This means that you will need to have one of the Python versions supported by NumPy as well. Recipes to install other relevant software packages are provided throughout the book.

Who this book is for

This book is for scientists, engineers, programmers, or analysts, with a basic knowledge of Python and NumPy, who want to go to the next level. Also, some affinity for or at least interest in mathematics and statistics is required.

Conventions

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

Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
[default]
exten => s,1,Dial(Zap/1|30)
exten => s,2,Voicemail(u100)
exten => s,102,Voicemail(b100)
exten => i,1,Voicemail(s0)
```

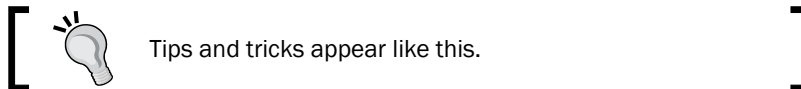
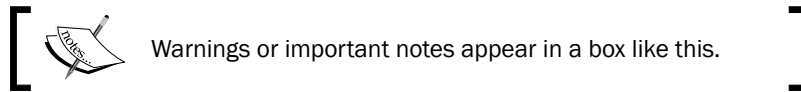
When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
[default]
exten => s,1,Dial(Zap/1|30)
exten => s,2,Voicemail(u100)
exten => s,102,Voicemail(b100)
exten => i,1,Voicemail(s0)
```

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
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking the **Next** button moves you to the next screen".



Reader feedback

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

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title through the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

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

Downloading the example code

You can download the example code files for all Packt books you have purchased 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 e-mailed directly to you.

Errata

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

Piracy

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

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

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

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Winding Along with IPython

In this chapter, we will cover the following topics:

- ▶ Installing IPython
- ▶ Using IPython as a shell
- ▶ Reading manual pages
- ▶ Installing Matplotlib
- ▶ Running a web notebook
- ▶ Exporting a web notebook
- ▶ Importing a web notebook
- ▶ Configuring a notebook server
- ▶ Exploring the SymPy profile

Introduction

IPython, which is available at <http://ipython.org/>, is a free, open source project available for Linux, Unix, Mac OS X, and Windows. The IPython authors only request that you cite IPython in any scientific work where IPython was used. It provides the following components, among others:

- ▶ Interactive Python shells (terminal-based and Qt application)
- ▶ A web notebook (available in IPython 0.12 and later) with support for rich media and plotting
- ▶ IPython is compatible with Python versions 2.5, 2.6, 2.7, 3.1, and 3.2

You can try IPython in cloud without installing it on your system, by going to the following URL: <http://www.pythonanywhere.com/try-ipython/>. There is a slight delay compared to locally installed software; so this is not as good as the real thing. However, most of the features available in the IPython interactive shell seem to be available. They also have a **Vi (m) editor**, which if you like vi, is of course great. You can save and edit files from your IPython sessions. The author of this book doesn't care much about other editors, such as the one that starts with E and ends with macs. This should, however, not be a problem.

Installing IPython

IPython can be installed in various ways depending on your operating system. For the terminal-based shell, there is a dependency on `readline`. The web notebook requires `tornado` and `zmq`.

In addition to installing IPython, we will install `setuptools`, which gives you the `easy_install` command. The `easy_install` command is the default, standard package manager for Python. `pip` can be installed once you have `easy_install` available. The `pip` command is similar to `easy_install`, and adds options such as uninstalling.

How to do it...

This section describes how IPython can be installed on Windows, Mac OS X, and Linux. It also describes how to install IPython and its dependencies with `easy_install` and `pip`, or from source.

- ▶ **Installing IPython and setup tools on Windows:** A binary Windows installer for Python 2 or Python 3 is available on the IPython website. Also see <http://ipython.org/ipython-doc/stable/install/install.html#windows>. Install `setuptools` with an installer from <http://pypi.python.org/pypi/setuptools#files>. Then install `pip`; for instance:

```
cd C:\Python27\scripts
python .\easy_install-27-script.py pip
```
- ▶ **Installing IPython On Mac OS X:** Install the **Apple Developer Tools (Xcode)** if necessary. Xcode can be found on the OSX DVDs that came with your Mac or App Store. Follow the `easy_install/pip` instructions, or the installing from source instructions provided later in this section.
- ▶ **Installing IPython On Linux:** Because there are so many Linux distributions, this section will not be exhaustive.
 - On Debian, type the following command:

```
su - aptitude install ipython python-setuptools
```

- On Fedora, the magic command is as follows:
`su - yum install ipython python-setuptools-devel`
 - The following command will install IPython on Gentoo:
`su - emerge ipython`
 - For Ubuntu, the install command is as follows:
`sudo apt-get install ipython python-setuptools`
- ▶ **Installing IPython with easy_install or pip:** Install IPython and all the dependencies required for the recipes in this chapter with `easy_install`, using the following command:

```
easy_install ipython pyzmq tornado readline
```

Alternatively, you can first install `pip` with `easy_install`, by typing the following command in your terminal:

```
easy_install pip
```

After that, install IPython using `pip`, with the following command:

```
sudo pip install ipython pyzmq tornado readline
```

- ▶ **Installing from source:** If you want to use the bleeding edge development version, then installing from source is for you.
1. Download the latest tarball from <https://github.com/ipython/ipython/downloads>.
 2. Unpack the source code from the archive:
`tar xzf ipython-<version>.tar.gz`
 3. If you have Git installed, you can clone the Git repository instead:
`$ git clone https://github.com/ipython/ipython.git`
 4. Go to the `ipython` directory:
`cd ipython`
 5. Run the `setup` script. This may require you to run the command with `sudo`, as follows:
`sudo setup.py install`

How it works...

We installed IPython using several methods. Most of these methods install the latest stable release, except when you install from source, which will install the development version.

Using IPython as a shell

Scientists and engineers are used to experimenting. IPython was created by scientists with experimentation in mind. The interactive environment that IPython provides is viewed by many as a direct answer to Matlab, Mathematica, and Maple and R.

Following is a list of features of the IPython shell:

- ▶ Tab completion
- ▶ History mechanism
- ▶ Inline editing
- ▶ Ability to call external Python scripts with `%run`
- ▶ Access to system commands
- ▶ The pylab switch
- ▶ Access to Python debugger and profiler

How to do it...

This section describes how to use the IPython shell.

- ▶ **The pylab switch:** The pylab switch automatically imports all the *Scipy*, *NumPy*, and *Matplotlib* packages. Without this switch, we would have to import these packages ourselves.

All we need to do is enter the following instruction on the command line:

```
$ ipython -pylab
```

```
Type "copyright", "credits" or "license" for more information.
```

```
IPython 0.12 -- An enhanced Interactive Python.
```

```
?          -> Introduction and overview of IPython's features.
```

```
%quickref -> Quick reference.
```

```
help       -> Python's own help system.
```

```
object?   -> Details about 'object', use 'object??' for extra details.
```

```
Welcome to pylab, a matplotlib-based Python environment [backend: MacOSX].
```

```
For more information, type 'help(pylab)'.  
In [1]: quit()
```

```
quit() or Ctrl + D quits the IPython shell.
```

- ▶ **Saving a session:** We might want to be able to go back to our experiments. In IPython, it is easy to save a session for later use, with the following command:

```
In [1]: %logstart
Activating auto-logging. Current session state plus future input
saved.
Filename      : ipython_log.py
Mode          : rotate
Output logging : False
Raw input log  : False
Timestamping  : False
State         : active
```

Logging can be switched off as follows:

```
In [9]: %logoff
Switching logging OFF
```

- ▶ **Executing system shell commands:** Execute system shell commands in the default IPython profile by prefixing the command with the ! symbol. For instance, the following input will get the current date:

```
In [1]: !date
```

In fact, any line prefixed with ! is sent to the system shell. Also, we can store the command output, as shown here:

```
In [2]: thedate = !date
In [3]: thedate
```

- ▶ **Displaying history:** We can show the history of commands with the %hist command () for example:

```
In [1]: a = 2 + 2
```

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

```
In [3]: %hist
a = 2 + 2
a
%hist
```

This is a common feature in **Command Line Interface (CLI)** environments. We can also search through the history with the `-g` switch

```
In [5]: %hist -g a = 2
1: a = 2 + 2
```



Downloading the example code

You can download the example code files for all Packt books you have purchased 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 e-mailed directly to you.

How it works...

We saw a number of so called "magic functions" in action. These functions start with the `%` character. If the magic function is used on a line by itself, the `%` prefix is optional.

Reading manual pages

When we are in IPython's pylab mode, we can open manual pages for NumPy functions with the `help` command. It is not necessary to know the name of a function. We can type a few characters and then let tab completion do its work. Let's, for instance, browse the available information for the `arange` function.

How to do it...

We can browse the available information, in either of the following two ways:

- ▶ **Calling the help function:** Call the `help` command. Type a few characters of the function and press the `Tab` key:

```
In [1]: help ar
arange      arcsin      arctan2     argmin      around      array_equal  array_split
arccos      arcsinh     arctanh     argsort     array        array_equiv  array_str
arccosh     arctan     arqmax      arqwhere    arrayZstring array_repr   arrow
```

- ▶ **Querying with a question mark:** Another option is to put a question mark behind the function name. You will then, of course, need to know the function name, but you don't have to type `help`:

```
In [3]: arange?
```

How it works...

Tab completion is dependent on `readline`, so you need to make sure it is installed. The question mark gives you information from docstrings.

Installing Matplotlib

Matplotlib is a very useful plotting library, which we will need for the next recipe. It depends on NumPy, but in all likelihood you already have NumPy installed.

How to do it...

We will see how Matplotlib can be installed in Windows, Linux, and Mac, and also how to install it from source.

- ▶ **Installing Matplotlib on Windows:** Install with the Enthought distribution (<http://www.enthought.com/products/epd.php>).
It might be necessary to put the `msvc71.dll` file in your `C:\Windows\system32` directory. You can get it from <http://www.dll-files.com/dllindex/dll-files.shtml?msvc71>.
- ▶ **Installing Matplotlib on Linux:** Let's see how Matplotlib can be installed in the various distributions of Linux:
 - The install command on Debian and Ubuntu is as follows:

```
sudo apt-get install python-matplotlib
```
 - The install command on Fedora/Redhat is as follows:

```
su - yum install python-matplotlib
```
- ▶ **Installing from source:** Download the latest source from the `tar.gz` release at Sourceforge (<http://sourceforge.net/projects/matplotlib/files/>) or from the Git repository using the following command:

```
git clone git://github.com/matplotlib/matplotlib.git
```


Once it has been downloaded, build and install as usual with the following command:

```
cd matplotlib  
python setup.py install
```
- ▶ **Installing Matplotlib on Mac:** Get the latest DMG file from <http://sourceforge.net/projects/matplotlib/files/matplotlib/>, and install it.

Running a web notebook

The newest release of IPython introduced a new exciting feature – the web notebook. A so called "notebook server" can serve notebooks over the web. We can now start a notebook server and have a web-based IPython environment. This environment has most of the features in the regular IPython environment. The new features include the following:

- ▶ Displaying images and inline plots
- ▶ Using HTML and Markdown in text cells
- ▶ Importing and exporting of notebooks

Getting ready

Before we start, we should make sure that all the required software is installed. There is a dependency on `tornado` and `zmq`. See the *Installing IPython* recipe in this chapter for more information.

How to do it...

- ▶ **Running a notebook:** We can start a notebook with the following code:

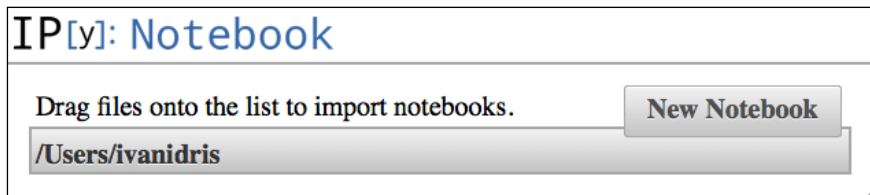
```
$ ipython notebook
```

```
[NotebookApp] Using existing profile dir: u'/Users/ivanidris/.ipython/profile_default'
```

```
[NotebookApp] The IPython Notebook is running at:  
http://127.0.0.1:8888
```

```
[NotebookApp] Use Control-C to stop this server and shut down  
all kernels.
```

As you can see, we are using the `default` profile. A server started on the local machine at port 8888. We will learn how to configure these settings later on in this chapter. The notebook is opened in your default browser; this is configurable as well:



IPython lists all the notebooks in the directory where you started the notebook. In this example no notebooks were found. The server can be stopped with *Ctrl + C*.

- ▶ **Running a notebook in the pylab mode:** Run a web notebook in the pylab mode with the following command:

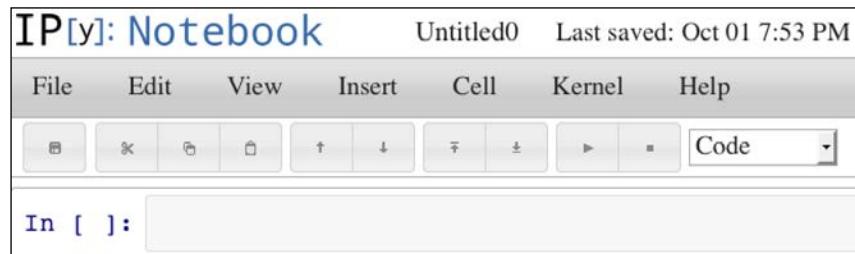
```
$ ipython notebook --pylab
```

This loads the Scipy, NumPy, and Matplotlib modules.

- ▶ **Running notebook with inline figures:** We can display inline Matplotlib plots with the inline directive, using the following command:

```
$ ipython notebook --pylab inline
```

1. **Create a notebook:** Click on the **New Notebook** button to create a new notebook:



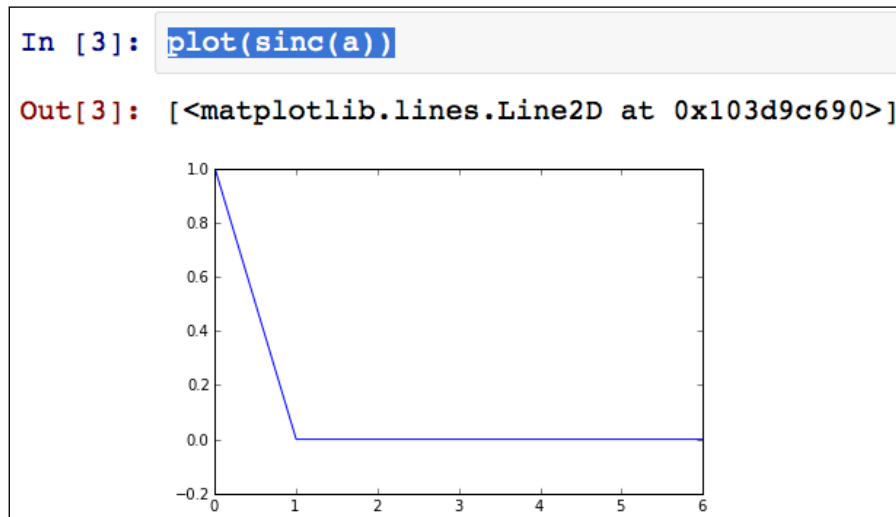
2. **Create an array:** Create an array with the `arange` function. Type the command in the following screenshot, and press *Enter*:

```
In [1]: a = arange(7)
```

Next, enter the following command and press *Enter*. You will see the output as shown in **Out [2]** in the following screenshot:

```
In [2]: a
Out[2]: array([0, 1, 2, 3, 4, 5, 6])
```


3. **Plot the sinc function:** Apply the `sinc` function to the array and plot the result, as shown in the following screenshot:



How it works...

The inline option lets you display inline Matplotlib plots. When combined with the `pylab` mode, you don't need to import the NumPy, SciPy, and Matplotlib packages.

See also

The *Installing IPython* recipe.

Exporting a web notebook

Sometimes you will want to exchange notebooks with friends or colleagues. The web notebook provides several methods to export your data.


How to do it...

A web notebook can be exported using the following options:

- ▶ **The Print option:** The **Print** button doesn't actually print the notebook, but allows you to export the notebook as PDF or HTML document.

- ▶ **Downloading the notebook:** Download your notebook to a location chosen by you, using the **Download** button. We can specify whether we want to download the notebook as `.py` file, which is just a normal Python program, or in the JSON format as a `.ipynb` file. The notebook we created in the previous recipe looks like the following, after exporting:

```
{
  "metadata": {
    "name": "Untitled1"
  },
  "nbformat": 2,
  "worksheets": [
    {
      "cells": [
        {
          "cell_type": "code",
          "collapsed": false,
          "input": [
            "plot(sinc(a))"
          ],
          "language": "python",
          "outputs": [
            {
              "output_type": "pyout",
              "prompt_number": 3,
              "text": [
                "[&lt;matplotlib.lines.Line2D at
                 0x103d9c690&gt;]"
              ]
            },
            {
              "output_type": "display_data",
              "png": "iVBORw0KGgoAAAANSUhEUgAAAXk
                AAAD9CAYAAABZVQdHAAAABHNCSVQICAgIf...
                mgkAAAAASUVORK5CYII=\n"
            }
          ],
          "prompt_number": 3
        }
      ]
    }
  ]
}
```

 Some of the text has been omitted for brevity. This file is not intended for editing or reading even, but it is pretty readable if you ignore the image representation part. For more information about JSON please see <https://en.wikipedia.org/wiki/JSON>.

- ▶ **Saving the notebook:** Save the notebook using the **Save** button. This will automatically export a notebook in the native JSON `.ipynb` format. The file will be stored in the directory where you started IPython initially.

Importing a web notebook

Python scripts can be imported as a web notebook. Obviously, we can also import previously exported notebooks.

How to do it...

The following steps show how a python script can be imported as a web notebook:

1. Import a python script by dragging it from Explorer or Finder into the notebook page. The following screenshot is an example of what we see after dragging the `vectorsum.py` from NumPy Beginner's Guide into the notebook page:



2. Click the **Upload** button to import the program. IPython does a decent job of importing the code. Unfortunately, as shown in the following screenshot, the code is all placed in one cell. At least that is how it worked at the time of writing:

```

File Edit View Insert Cell Kernel Help
Code
In [ ]: #!/usr/bin/env/python

import sys
from datetime import datetime
import numpy

"""
Chapter 1 of NumPy Beginners Guide.
This program demonstrates vector addition the Python way.
Run from the command line as follows

python vectorsum.py n

where n is an integer that specifies the size of the vectors.

The first vector to be added contains the squares of 0 up to n.
The second vector contains the cubes of 0 up to n.
The program prints the last 2 elements of the sum and the elapsed time.
"""

def numpysum(n):
    a = numpy.arange(n) ** 2
    b = numpy.arange(n) ** 3
    c = a + b

```

3. Tag the script for multiple cells.

In order to split the code into multiple cells we need to use special tags. These tags are in fact Python comments, but they look a bit like XML tags. The code has to start with the following tag:

```
# <nbformat>2</nbformat>
```

This indicates the format of the notebook. Each new code cell is indicated with the following tag:

```
# <codecell>
```

The following is the tagged code:

```
# <nbformat>2</nbformat>
#!/usr/bin/env/python
```

```
from datetime import datetime
import numpy
```

```
"""
Chapter 1 of NumPy Beginners Guide.
This program demonstrates vector addition the Python way.
Run from the command line as follows

    python vectorsum.py n

where n is an integer that specifies the size of the vectors.

The first vector to be added contains the squares of 0 up to n.
The second vector contains the cubes of 0 up to n.
The program prints the last 2 elements of the sum and the elapsed
time.
"""

def numpysum(n):
    a = numpy.arange(n) ** 2
    b = numpy.arange(n) ** 3
    c = a + b

    return c

def pythonsum(n):
    a = range(n)
    b = range(n)
    c = []

    for i in range(len(a)):
        a[i] = i ** 2
        b[i] = i ** 3
        c.append(a[i] + b[i])

    return c

# <codecell>
size = int(50)

# <codecell>
start = datetime.now()
```

```
c = pythonsum(size)
delta = datetime.now() - start
print "The last 2 elements of the sum", c[-2:]
print "PythonSum elapsed time in microseconds", delta.microseconds

# <codecell>
start = datetime.now()
c = numpysum(size)
delta = datetime.now() - start
print "The last 2 elements of the sum", c[-2:]
print "NumPySum elapsed time in microseconds", delta.microseconds
```

The code is split into several cells according to the tags, as shown in the following screenshot:

```
In [ ]: size = int(50)

In [ ]: start = datetime.now()

        c = pythonsum(size)

        delta = datetime.now() - start

        print "The last 2 elements of the sum", c[-2:]

        print "PythonSum elapsed time in microseconds", delta.microseconds

In [ ]: start = datetime.now()

        c = numpysum(size)

        delta = datetime.now() - start

        print "The last 2 elements of the sum", c[-2:]

        print "NumPySum elapsed time in microseconds", delta.microseconds
```

Configuring a notebook server

A public notebook server needs to be secure. You should set a password and use a SSL certificate to connect to it. We need the certificate to provide secure communication over https (for more information see https://en.wikipedia.org/wiki/Transport_Layer_Security).

How to do it...

The following steps describe how to configure a secure notebook server:

1. **Generate a password:** We can generate a password from IPython. Start a new IPython session, and type in the following commands:

```
In [1]: from IPython.lib import passwd
```

```
In [2]: passwd()
```

```
Enter password:
```

```
Verify password:
```

```
Out[2]: 'sha1:0e422dfccef2:84cfbcb  
b3ef95872fb8e23be3999c123f862d856'
```

At the second input line you will be prompted for a password. You need to remember this password. A long string is generated. Copy this string because we will need it later on.

2. **Create a SSL certificate:** To create a SSL certificate, you will need to have the `openssl` command in your path.

Setting up the `openssl` command is not rocket science, but can be tricky. Unfortunately, it is outside the scope of this book. On the bright side there are plenty of tutorials available online to help you further.

Execute the following command to create a certificate with the name `mycert.pem`:

```
$ openssl req -x509 -nodes -days 365 -newkey rsa:1024 -keyout  
mycert.pem -out mycert.pem
```

```
Generating a 1024 bit RSA private key
```

```
.....+++++
```

```
.....+++++
```

```
writing new private key to 'mycert.pem'
```

```
-----
```

```
You are about to be asked to enter information that will be  
incorporated
```

```
into your certificate request.
```

What you are about to enter is what is called a Distinguished Name or a DN.

There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.

```
Country Name (2 letter code) [AU]:
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (eg, YOUR name) []:
Email Address []:
```

The `openssl` utility prompts you to fill in some fields. For more information, check the relevant man page (short for manual page).

3. **Create a server profile:** Create a special profile for the server using the following command:

```
ipython profile create nbserver
```

4. **Edit the profile configuration file:** Edit the configuration file. In this example, it can be found in Edit in `~/ipython/profile_nbserver/ipython_notebook_config.py`.

The configuration file is pretty large, so we will omit many of the lines in it. The lines that we need to change at minimum are:

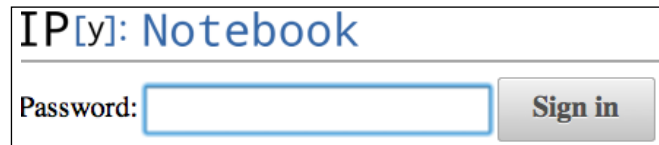
```
c.NotebookApp.certfile = u'/absolute/path/to/your/certificate'
c.NotebookApp.password = u'sha1:b...your password'
c.NotebookApp.port = 9999
```

Notice that we are pointing to the SSL certificate we created. We set a password and changed the port to 9999.

5. **Start the server:** Using the following command, start the server to check whether the changes worked.

```
ipython notebook --profile=nbserver
[NotebookApp] Using existing profile dir: u'/Users/ivanidris/.ipython/profile_nbserver'
[NotebookApp] The IPython Notebook is running at:
https://127.0.0.1:9999
[NotebookApp] Use Control-C to stop this server and shut down all kernels.
```


The server is running on port 9999, and you need to connect to it via https. If everything goes well, we should see a login page. Also, you would probably need to accept a security exception in your browser.

A screenshot of the IPython Notebook login interface. It features a header "IP[y]: Notebook" in blue text. Below the header, there is a "Password:" label followed by a text input field and a "Sign in" button.

How it works...

We created a special profile for our public server. There are some sample profiles that are already present, such as the default profile. Creating a profile adds a `profile_<profilename>` folder to the `.ipython` directory with, among others, a configuration file. The profile can then be loaded with the `--profile=<profile_name>` command-line option. We can list the profiles with the following command:

```
ipython profile list
```

Available profiles in IPython:

```
cluster
math
pysh
python3
```

The first request for a bundled profile will copy it into your IPython directory (`/Users/ivanidris/.ipython`), where you can customize it.

Available profiles in `/Users/ivanidris/.ipython`:

```
default
nbserver
sh
```

Exploring the SymPy profile

IPython has a sample SymPy profile. SymPy is a Python symbolic, mathematics library. For instance, we can simplify algebraic expressions or differentiate, similar to Mathematica and Maple. SymPy is obviously a fun piece of software, but is not directly necessary for our journey through the NumPy landscape. Consider this as an optional bonus recipe. Like dessert, feel free to skip, although you might miss out on the sweetest piece of this chapter.

Getting ready

Install SymPy using either `easy_install`, or `pip`:

```
easy_install sympy
sudo pip install sympy
```

How to do it...

1. Look at the configuration file, which can be found at `~/.ipython/profile_sympy/ipython_config.py`. The contents are as follows:

```
c = get_config()
app = c.InteractiveShellApp

# This can be used at any point in a config file to load a sub
# config
# and merge it into the current one.
load_subconfig('ipython_config.py', profile='default')

lines = """
from __future__ import division
from sympy import *
x, y, z, t = symbols('x y z t')
k, m, n = symbols('k m n', integer=True)
f, g, h = symbols('f g h', cls=Function)
"""

# You have to make sure that attributes that are containers
# already
# exist before using them. Simple assigning a new list will
# override
# all previous values.
```

```
if hasattr(app, 'exec_lines'):
    app.exec_lines.append(lines)
else:
    app.exec_lines = [lines]

# Load the sympy_printing extension to enable nice printing of
# sympy expr's.
if hasattr(app, 'extensions'):
    app.extensions.append('sympyprinting')
else:
    app.extensions = ['sympyprinting']
```

This code accomplishes the following:

- Loading the default profile
 - Importing the SymPy packages
 - Defining symbols
2. Start IPython with the SymPy profile using the following command:
`ipython --profile=sympy`
 3. Expand an algebraic expression using the command shown in the following screenshot:

```
In [1]: expand((x+y)**7)
Out[1]:
```

$$x^7 + 7 \cdot x^6 \cdot y + 21 \cdot x^5 \cdot y^2 + 35 \cdot x^4 \cdot y^3 + 35 \cdot x^3 \cdot y^4 + 21 \cdot x^2 \cdot y^5 + 7 \cdot x \cdot y^6 + y^7$$

2

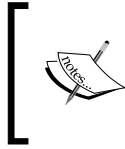
Advanced Indexing and Array Concepts

In this chapter, we will cover:

- ▶ Installing SciPy
- ▶ Installing PIL
- ▶ Resizing images
- ▶ Comparing views and copies
- ▶ Flipping Lena
- ▶ Fancy indexing
- ▶ Indexing with a list of locations
- ▶ Indexing with booleans
- ▶ Stride tricks for Sudoku
- ▶ Broadcasting arrays

Introduction

NumPy is famous for its efficient arrays. This fame is partly due to the ease of indexing. We will demonstrate advanced indexing tricks using images. Before diving into indexing, we will install the necessary software— SciPy and PIL.



The code for the recipes in this chapter can be found on the book website at <http://www.packtpub.com>. You can also visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Some of the examples in this chapter will involve manipulating images. In order to do that, we will require the **Python Image Library (PIL)**; but don't worry, instructions and pointers to help you install PIL and other necessary Python software are given throughout the chapter, when necessary.

Installing SciPy

SciPy is the scientific Python library and is closely related to NumPy. In fact, SciPy and NumPy used to be one and the same project many years ago. In this recipe, we will install SciPy.

Getting ready

In *Chapter 1, Winding Along with IPython*, we discussed how to install setup tools and pip. Reread the recipe if necessary.

How to do it...

In this recipe, we will go through the steps for installing SciPy.

- ▶ **Installing from source:** If you have Git installed, you can clone the SciPy repository using the following command:

```
git clone https://github.com/scipy/scipy.git
```

```
python setup.py build
```

```
python setup.py install --user
```

This installs to your home directory and requires Python 2.6 or higher.

Before building, you will also need to install the following packages on which SciPy depends:

- BLAS and LAPACK libraries
- C and Fortran compilers

There is a chance that you have already installed this software as a part of the NumPy installation.

- ▶ **Installing SciPy on Linux:** Most Linux distributions have SciPy packages. We will go through the necessary steps for some of the popular Linux distributions:
 - In order to install SciPy on Red Hat, Fedora, and CentOS, run the following instructions from the command line:

```
yum install python-scipy
```
 - In order to install SciPy on Mandriva, run the following command line instruction:

```
urpmi python-scipy
```
 - In order to install SciPy on Gentoo, run the following command line instruction:

```
sudo emerge scipy
```
 - On Debian or Ubuntu, we need to type the following:

```
sudo apt-get install python-scipy
```
- ▶ **Installing SciPy on Mac OS X: Apple Developer Tools (XCode)** is required, because it contains the *BLAS* and *LAPACK* libraries. It can be found either in the App Store, or in the installation DVD that came with your Mac, or you can get the latest version from Apple Developer's connection at <https://developer.apple.com/technologies/tools/>. Make sure that everything, including all the optional packages is installed.

You probably already have a Fortran compiler installed for NumPy. The binaries for *gfortran* can be found at <http://r.research.att.com/tools/>.
- ▶ **Installing SciPy using easy_install or pip:** Install with *either of* the following two commands:

```
sudo pip install scipy
```

```
easy_install scipy
```
- ▶ **Installing on Windows:** If you have Python installed already, the preferred method is to download and use the binary distribution. Alternatively, you may want to install the Enthought Python distribution, which comes with other scientific Python software packages.
- ▶ **Check your installation:** Check the SciPy installation with the following code:

```
import scipy
print scipy.__version__
print scipy.__file__
```

This should print the correct SciPy version.

How it works...

Most package managers will take care of any dependencies for you. However, in some cases, you will need to install them manually. Unfortunately, this is beyond the scope of this book. If you run into problems, you can ask for help at:

- ▶ The #scipy IRC channel of freenode, or
- ▶ The SciPy mailing lists at http://www.scipy.org/Mailing_Lists

Installing PIL

PIL, the Python imaging library, is a prerequisite for the image processing recipes in this chapter.

How to do it...

Let's see how to install PIL.

- ▶ **Installing PIL on Windows:** Install using the Windows executable from the PIL website <http://www.pythonware.com/products/pil/>.
- ▶ **Installing on Debian or Ubuntu:** On Debian or Ubuntu, install PIL using the following command:

```
sudo apt-get install python-imaging
```
- ▶ **Installing with easy_install or pip:** At the time of writing this book, it appeared that the package managers of Red Hat, Fedora, and CentOS did not have direct support for PIL. Therefore, please follow this step if you are using one of these Linux distributions.

Install with either of the following commands:

```
easy_install PIL
```

```
sudo pip install PIL
```

Resizing images

In this recipe, we will load a sample image of *Lena*, which is available in the SciPy distribution, into an array. This chapter is not about image manipulation, by the way; we will just use the image data as an input.



Lena Soderberg appeared in a 1972 Playboy magazine. For historical reasons, one of those images is often used in the field of image processing. Don't worry; the picture in question is completely safe for work.

We will resize the image using the `repeat` function. This function *repeats* an array, which in practice means resizing the image by a certain factor.

Getting ready

A prerequisite for this recipe is to have SciPy, Matplotlib, and PIL installed. Have a look at the corresponding recipes in this chapter and the previous chapter.

How to do it...

1. Load the Lena image into an array.

SciPy has a `lena` function, which can load the image into a NumPy array:

```
lena = scipy.misc.lena()
```

Some refactoring has occurred since version 0.10, so if you are using an older version, the correct code is:

```
lena = scipy.lena()
```

2. Check the shape.

Check the shape of the Lena array using the `assert_equal` function from the `numpy.testing` package—this is an optional sanity check test:

```
numpy.testing.assert_equal((LENA_X, LENA_Y), lena.shape)
```

3. Resize the Lena array.

Resize the Lena array with the `repeat` function. We give this function a resize factor in the x and y direction:

```
resized = lena.repeat(yfactor, axis=0).repeat(xfactor, axis=1)
```


4. Plot the arrays.

We will plot the Lena image and the resized image in two subplots that are a part of the same grid. Plot the Lena array in a subplot:

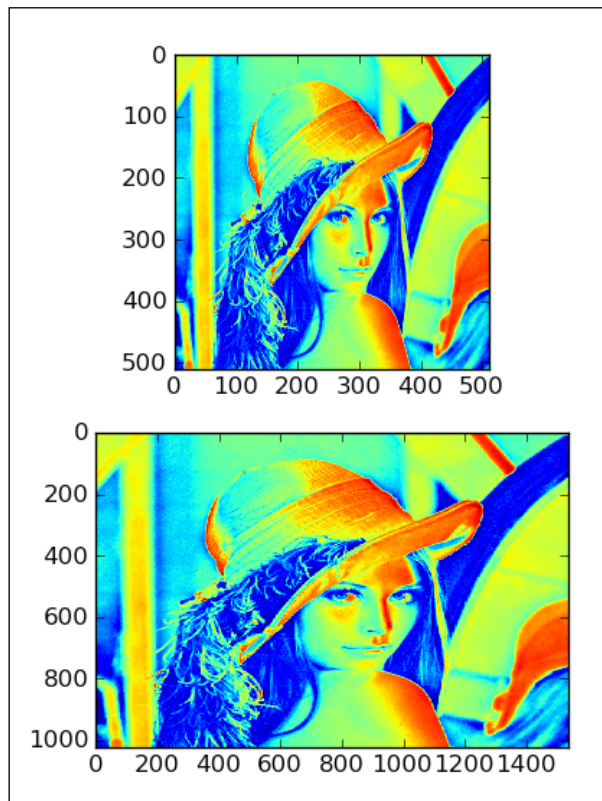
```
matplotlib.pyplot.subplot(211)
matplotlib.pyplot.imshow(lena)
```

The Matplotlib `subplot` function creates a subplot. This function accepts a 3-digit integer as the parameter, where the first digit is the number of rows, the second digit is the number of columns, and the last digit is the index of the subplot starting with 1. The `imshow` function shows images. Finally, the `show` function displays the end result.

Plot the resized array in another subplot and display it. The index is now 2:

```
matplotlib.pyplot.subplot(212)
matplotlib.pyplot.imshow(resized)
matplotlib.pyplot.show()
```

The following screenshot is the result with the original image (first) and the resized image (second):



The following is the complete code for this recipe:

```
import scipy.misc
import sys
import matplotlib.pyplot
import numpy.testing

# This script resizes the Lena image from Scipy.

if(len(sys.argv) != 3):
    print "Usage python %s yfactor xfactor" % (sys.argv[0])
    sys.exit()

# Loads the Lena image into an array
lena = scipy.misc.lena()
#Lena's dimensions
LENA_X = 512
LENA_Y = 512
#Check the shape of the Lena array
numpy.testing.assert_equal((LENA_X, LENA_Y), lena.shape)

# Get the resize factors
yfactor = float(sys.argv[1])
xfactor = float(sys.argv[2])

# Resize the Lena array
resized = lena.repeat(yfactor, axis=0).repeat(xfactor, axis=1)

#Check the shape of the resized array
numpy.testing.assert_equal((yfactor * LENA_Y, xfactor * LENA_X),
resized.shape)

# Plot the Lena array
matplotlib.pyplot.subplot(211)
matplotlib.pyplot.imshow(lena)

#Plot the resized array
matplotlib.pyplot.subplot(212)
matplotlib.pyplot.imshow(resized)
matplotlib.pyplot.show()
```

How it works...

The `repeat` function repeats arrays, which, in this case, resulted in changing the size of the original image. The Matplotlib `subplot` function creates a subplot. The `imshow` function shows images. Finally, the `show` function displays the end result.

See also

- ▶ The *Installing Matplotlib* recipe in *Chapter 1, Winding Along with IPython*
- ▶ The *Installing SciPy* recipe
- ▶ The *Installing PIL* recipe

Creating views and copies

It is important to know when we are dealing with a shared array view, and when we have a copy of the array data. A slice, for instance, will create a view. This means that if you assign the slice to a variable and then change the underlying array, the value of this variable will change. We will create an array from the famous Lena image, copy the array, create a view, and, at the end, modify the view.

Getting ready

The prerequisites are the same as in the previous recipe.

How to do it...

Let's create a copy and views of the Lena array:

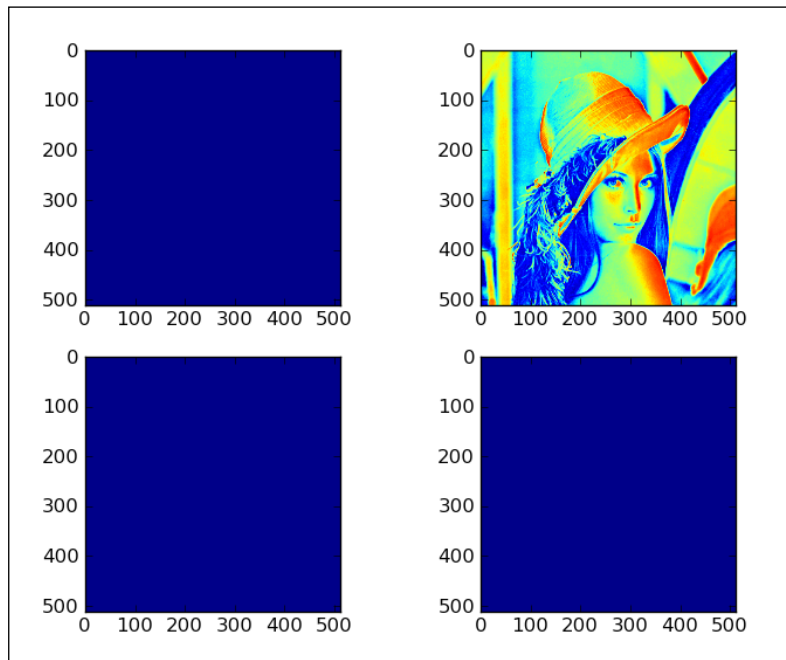
1. Create a copy of the Lena array:

```
acopy = lena.copy()
```
2. Create a view of the array:

```
aview = lena.view()
```
3. Set all the values of the view to 0 with a flat iterator:

```
aview.flat = 0
```

The end result is that only one of the images shows the Playboy model. The other ones get censored completely:



The following is the code of this tutorial showing the behavior of array views and copies:

```
import scipy.misc
import matplotlib.pyplot

lena = scipy.misc.lena()
acopy = lena.copy()
aview = lena.view()

# Plot the Lena array
matplotlib.pyplot.subplot(221)
matplotlib.pyplot.imshow(lena)

#Plot the copy
matplotlib.pyplot.subplot(222)
matplotlib.pyplot.imshow(acopy)

#Plot the view
matplotlib.pyplot.subplot(223)
matplotlib.pyplot.imshow(aview)
```

```
# Plot the view after changes
aview.flat = 0
matplotlib.pyplot.subplot(224)
matplotlib.pyplot.imshow(aview)

matplotlib.pyplot.show()
```

How it works...

As you can see, by changing the view at the end of the program, we changed the original Lena array. This resulted in having three blue (or black if you are looking at a black and white image) images—the copied array was unaffected. It is important to remember that views are not read-only.

Flipping Lena

We will be flipping the SciPy Lena image—all in the name of science, of course, or at least as a demo. In addition to flipping the image, we will slice it and apply a mask to it.

How to do it...

The steps to follow are listed below:

1. Plot the flipped image.

Flip the Lena array around the vertical axis using the following code:

```
matplotlib.pyplot.imshow(lena[:,::-1])
```

2. Plot a slice of the image.

Take a slice out of the image and plot it. In this step, we will have a look at the shape of the Lena array. The shape is a tuple representing the dimensions of the array. The following code effectively selects the left-upper quadrant of the Playboy picture.

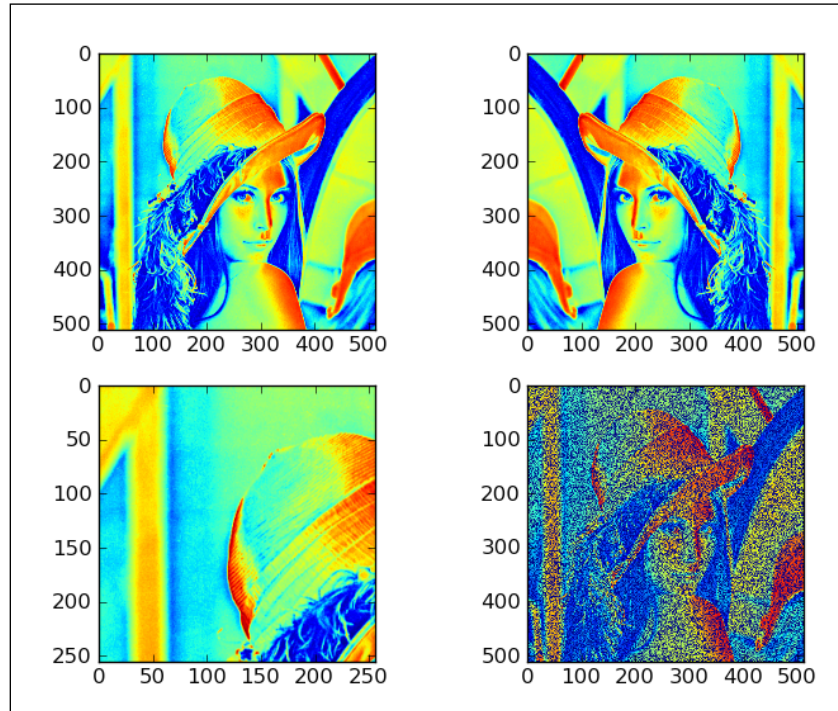
```
matplotlib.pyplot.imshow(lena[:lena.shape[0]/2,
                             :lena.shape[1]/2])
```

3. Apply a mask to the image.

Apply a mask to the image by finding all the values in the Lena array that are even (this is just arbitrary for demo purposes). Copy the array and change the even values to 0. This has the effect of putting lots of blue dots (dark spots if you are looking at a black and white image) on the image:

```
mask = lena % 2 == 0
masked_lena = lena.copy()
masked_lena[mask] = 0
```

All these efforts result in a 2 by 2 image grid, as shown in the following screenshot:



The following is the complete code for this recipe:

```
import scipy.misc
import matplotlib.pyplot

# Load the Lena array
lena = scipy.misc.lena()

# Plot the Lena array
matplotlib.pyplot.subplot(221)
matplotlib.pyplot.imshow(lena)

#Plot the flipped array
matplotlib.pyplot.subplot(222)
matplotlib.pyplot.imshow(lena[:,::-1])

#Plot a slice array
matplotlib.pyplot.subplot(223)
matplotlib.pyplot.imshow(lena[:lena.shape[0]/2,:lena.shape[1]/2])
```

```
# Apply a mask
mask = lena % 2 == 0
masked_lena = lena.copy()
masked_lena[mask] = 0
matplotlib.pyplot.subplot(224)
matplotlib.pyplot.imshow(masked_lena)

matplotlib.pyplot.show()
```

See also

- ▶ The *Installing Matplotlib* recipe in *Chapter 1, Winding Along with IPython*
- ▶ The *Installing SciPy* recipe
- ▶ The *Installing PIL* recipe

Fancy indexing

In this tutorial, we will apply fancy indexing to set the diagonal values of the Lena image to 0. This will draw black lines along the diagonals, crossing it through, not because there is something wrong with the image, but just as an exercise. Fancy indexing is indexing that does not involve integers or slices, which is *normal* indexing.

How to do it...

We will start with the first diagonal:

1. Set the values of the first diagonal to 0.

To set the diagonal values to 0, we need to define two different ranges for the x and y values:

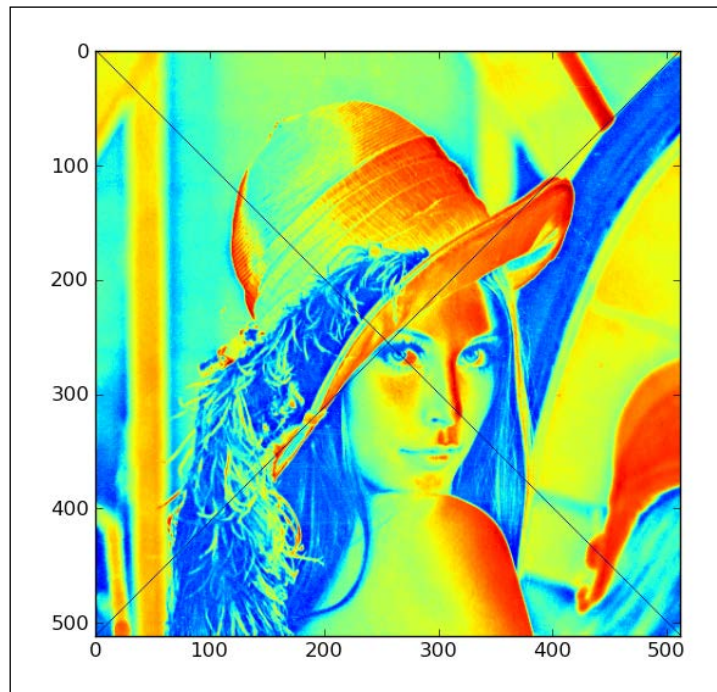
```
lena[range(xmax), range(ymax)] = 0
```

2. Set the values of the other diagonal to 0.

To set the values of the other diagonal, we require a different set of ranges, but the principles stay the same:

```
lena[range(xmax-1, -1, -1), range(ymax)] = 0
```

At the end, we get this image with the diagonals crossed off, as shown in the following screenshot:



The following is the complete code for this recipe:

```
import scipy.misc
import matplotlib.pyplot

# This script demonstrates fancy indexing by setting values
# on the diagonals to 0.

# Load the Lena array
lena = scipy.misc.lena()
xmax = lena.shape[0]
ymax = lena.shape[1]

# Fancy indexing
# Set values on diagonal to 0
# x 0-xmax
# y 0-ymax
lena[range(xmax), range(ymax)] = 0
```



```
# Set values on other diagonal to 0
# x xmax-0
# y 0-ymax
lena[range(xmax-1, -1, -1), range(ymax)] = 0

# Plot Lena with diagonal lines set to 0
matplotlib.pyplot.imshow(lena)
matplotlib.pyplot.show()
```

How it works...

We defined separate ranges for the x values and y values. These ranges were used to index the Lena array. Fancy indexing is performed based on an internal NumPy iterator object. The following three steps are performed:

1. The iterator object is created.
2. The iterator object gets bound to the array.
3. Array elements are accessed via the iterator.

Indexing with a list of locations

Let's use the `ix_` function to shuffle the Lena image. This function creates a mesh from multiple sequences.

How to do it...

We will start by randomly shuffling the array indices:

1. Shuffle array indices.

Create a random indices array with the `shuffle` function of the `numpy.random` module:

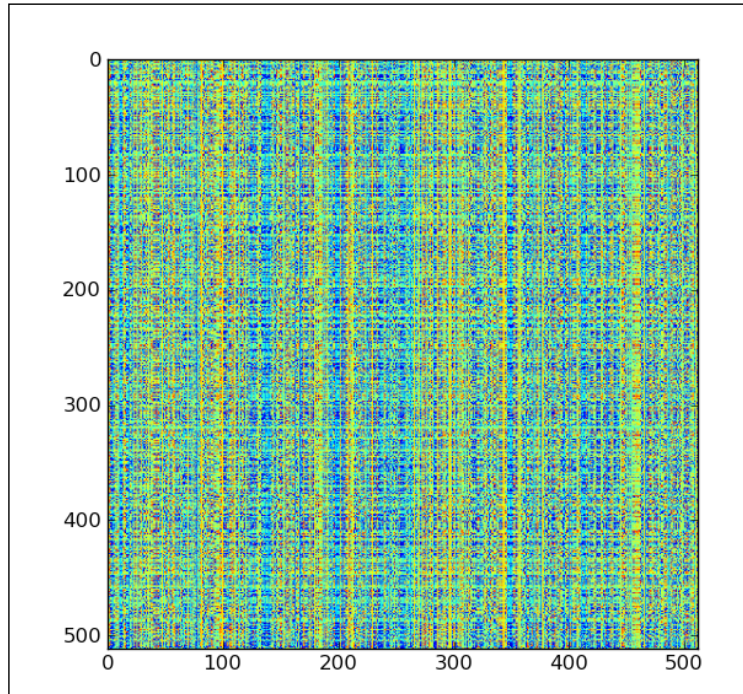
```
def shuffle_indices(size):
    arr = numpy.arange(size)
    numpy.random.shuffle(arr)

    return arr
```

2. Plot the shuffled indices:

```
matplotlib.pyplot.imshow(lena[numpy.ix_(xindices, yindices)])
```

What we get is a completely scrambled Lena image, as shown in the following screenshot:



The following is the complete code for the recipe:

```
import scipy.misc
import matplotlib.pyplot
import numpy.random
import numpy.testing

# Load the Lena array
lena = scipy.misc.lena()
xmax = lena.shape[0]
ymax = lena.shape[1]

def shuffle_indices(size):
    arr = numpy.arange(size)
    numpy.random.shuffle(arr)

    return arr
```

```
xindices = shuffle_indices(xmax)
numpy.testing.assert_equal(len(xindices), xmax)
yindices = shuffle_indices(ymax)
numpy.testing.assert_equal(len(yindices), ymax)

# Plot Lena
matplotlib.pyplot.imshow(lena[numpy.ix_(xindices, yindices)])
matplotlib.pyplot.show()
```

Indexing with booleans

Boolean indexing is indexing based on a boolean array and falls in the category *fancy indexing*.

How to do it...

We will apply this indexing technique to an image:

1. Image with dots on the diagonal.

This is in some way similar to the *Fancy indexing* recipe, in this chapter. This time we select modulo 4 points on the diagonal of the image:

```
def get_indices(size):
    arr = numpy.arange(size)
    return arr % 4 == 0
```

Then we just apply this selection and plot the points:

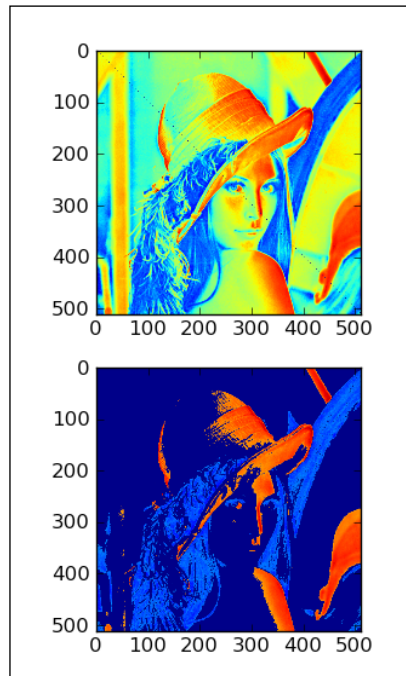
```
lena1 = lena.copy()
xindices = get_indices(lena.shape[0])
yindices = get_indices(lena.shape[1])
lena1[xindices, yindices] = 0
matplotlib.pyplot.subplot(211)
matplotlib.pyplot.imshow(lena1)
```

2. Set to 0 based on value.

Select array values between quarter and three-quarters of the maximum value and set them to 0:

```
lena2[(lena > lena.max()/4) &
      (lena < 3 * lena.max()/4)] = 0
```

The plot with the two new images will look like the following screenshot:



The following is the complete code for this recipe:

```
import scipy.misc
import matplotlib.pyplot
import numpy

# Load the Lena array
lena = scipy.misc.lena()

def get_indices(size):
    arr = numpy.arange(size)
    return arr % 4 == 0

# Plot Lena
lena1 = lena.copy()
xindices = get_indices(lena.shape[0])
yindices = get_indices(lena.shape[1])
lena1[xindices, yindices] = 0
matplotlib.pyplot.subplot(211)
matplotlib.pyplot.imshow(lena1)
```

```
lena2 = lena.copy()
# Between quarter and 3 quarters of the max value
lena2[(lena > lena.max()/4) & (lena < 3 * lena.max()/4)] = 0
matplotlib.pyplot.subplot(212)
matplotlib.pyplot.imshow(lena2)

matplotlib.pyplot.show()
```

How it works...

Because boolean indexing is a form of fancy indexing, the way it works is basically the same. This means that indexing happens with the help of a special iterator object.

See also

- ▶ The *Fancy Indexing* recipe

Stride tricks for Sudoku

The `ndarray` class has a `strides` field, which is a tuple indicating the number of bytes to step in each dimension when going through an array. Let's apply some stride tricks to the problem of splitting a Sudoku puzzle to the 3 by 3 squares of which it is composed.



Explaining the Sudoku rules is outside the scope of this book. For more information see <http://en.wikipedia.org/wiki/Sudoku>.

How to do it...

1. Define the Sudoku puzzle array

Let's define the Sudoku puzzle array. This one is filled with the contents of an actual, solved Sudoku puzzle:

```
sudoku = numpy.array([
    [2, 8, 7, 1, 6, 5, 9, 4, 3],
    [9, 5, 4, 7, 3, 2, 1, 6, 8],
    [6, 1, 3, 8, 4, 9, 7, 5, 2],
    [8, 7, 9, 6, 5, 1, 2, 3, 4],
    [4, 2, 1, 3, 9, 8, 6, 7, 5],
    [3, 6, 5, 4, 2, 7, 8, 9, 1],
    [1, 9, 8, 5, 7, 3, 4, 2, 6],
    [5, 4, 2, 9, 1, 6, 3, 8, 7],
    [7, 3, 6, 2, 8, 4, 5, 1, 9]
])
```

2. Calculate the strides. The `itemsize` field of `ndarray` gives us the number of bytes in an array. Using the `itemsize`, calculate the strides:

```
strides = sudoku.itemsize *  
    numpy.array([27, 3, 9, 1])
```

3. Split into squares.

Now we can split the puzzle into squares with the `as_strided` function of the `numpy.lib.stride_tricks` module:

```
squares = numpy.lib.stride_tricks.as_strided  
    (sudoku, shape=shape, strides=strides)  
print (squares)
```

This prints separate Sudoku squares:

```
[[[2 8 7]  
  [9 5 4]  
  [6 1 3]]  
  
 [[1 6 5]  
  [7 3 2]  
  [8 4 9]]  
  
 [[9 4 3]  
  [1 6 8]  
  [7 5 2]]]  
  
 [[8 7 9]  
  [4 2 1]  
  [3 6 5]]  
  
 [[6 5 1]  
  [3 9 8]  
  [4 2 7]]  
  
 [[2 3 4]  
  [6 7 5]  
  [8 9 1]]]  
  
 [[1 9 8]  
  [5 4 2]  
  [7 3 6]]]
```

```
[[5 7 3]
 [9 1 6]
 [2 8 4]]

[[4 2 6]
 [3 8 7]
 [5 1 9]]]
```

The following is the complete source code for this recipe:

```
import numpy

sudoku = numpy.array([
    [2, 8, 7, 1, 6, 5, 9, 4, 3],
    [9, 5, 4, 7, 3, 2, 1, 6, 8],
    [6, 1, 3, 8, 4, 9, 7, 5, 2],
    [8, 7, 9, 6, 5, 1, 2, 3, 4],
    [4, 2, 1, 3, 9, 8, 6, 7, 5],
    [3, 6, 5, 4, 2, 7, 8, 9, 1],
    [1, 9, 8, 5, 7, 3, 4, 2, 6],
    [5, 4, 2, 9, 1, 6, 3, 8, 7],
    [7, 3, 6, 2, 8, 4, 5, 1, 9]
])

shape = (3, 3, 3, 3)

strides = sudoku.itemsize *
    numpy.array([27, 3, 9, 1])

squares = numpy.lib.stride_tricks.as_strided
    (sudoku, shape=shape, strides=strides)
print(squares)
```

How it works...

We applied stride tricks to decompose a Sudoku puzzle in its constituent 3 by 3 squares. The strides tell us how many bytes we need to skip at each step when going through the Sudoku array.

Broadcasting arrays

Without knowing it, you might have broadcasted arrays. In a nutshell, NumPy tries to perform an operation even though the operands do not have the same shape. In this recipe, we will multiply an array and a scalar. The scalar is "extended" to the shape of the array operand and then the multiplication is performed. We will download an audio file and make a new version that is quieter.

How to do it...

Let's start by reading a WAV file:

1. Reading a WAV file.

We will use a standard Python code to download an audio file of Austin Powers called "Smashing, baby". SciPy has a `wavfile` module, which allows you to load sound data or generate WAV files. If SciPy is installed, then we should have this module already. The `read` function returns a data array and sample rate. In this example, we only care about the data:

```
sample_rate, data = scipy.io.wavfile.read(WAV_FILE)
```

2. Plot the original WAV data.

Plot the original WAV data with Matplotlib. Give the subplot the title `Original`.

```
matplotlib.pyplot.subplot(2, 1, 1)
matplotlib.pyplot.title("Original")
matplotlib.pyplot.plot(data)
```

3. Create a new array.

Now we will use NumPy to make a quieter audio sample. It's just a matter of creating a new array with smaller values by multiplying with a constant. This is where the magic of broadcasting occurs. At the end, we need to make sure that we have the same data type as in the original array, because of the WAV format:

```
newdata = data * 0.2
newdata = newdata.astype(numpy.uint8)
```

4. Write to a WAV file.

This new array can be written into a new WAV file as follows:

```
scipy.io.wavfile.write("quiet.wav",
                      sample_rate, newdata)
```

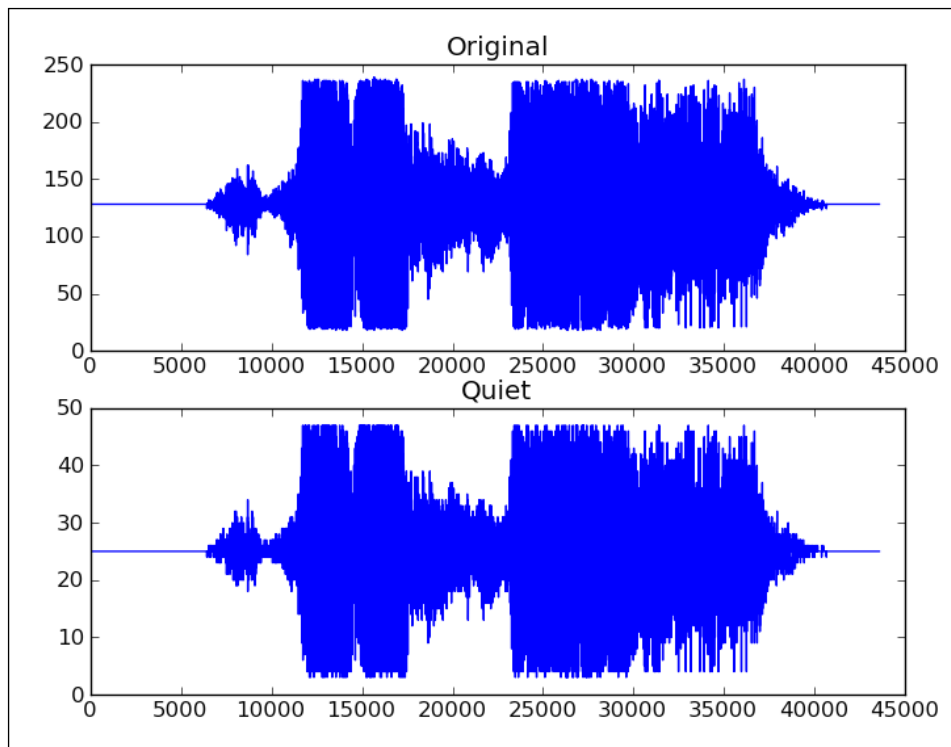

5. Plot the new WAV data.

Plot the new data array with Matplotlib:

```
matplotlib.pyplot.subplot(2, 1, 2)
matplotlib.pyplot.title("Quiet")
matplotlib.pyplot.plot(newdata)
```

```
matplotlib.pyplot.show()
```

The result is a plot of the original WAV file data and a new array with smaller values, as shown in the following screenshot:



The following is the complete code for this recipe:

```
import scipy.io.wavfile
import matplotlib.pyplot
import urllib2
import numpy
```

```
response = urllib2.urlopen('http://www.thesoundarchive.com/
austinpowers/smashingbaby.wav')
print response.info()
WAV_FILE = 'smashingbaby.wav'
filehandle = open(WAV_FILE, 'w')
filehandle.write(response.read())
filehandle.close()
sample_rate, data = scipy.io.wavfile.read(WAV_FILE)
print "Data type", data.dtype, "Shape", data.shape

matplotlib.pyplot.subplot(2, 1, 1)
matplotlib.pyplot.title("Original")
matplotlib.pyplot.plot(data)

newdata = data * 0.2
newdata = newdata.astype(numpy.uint8)
print "Data type", newdata.dtype, "Shape", newdata.shape

scipy.io.wavfile.write("quiet.wav",
    sample_rate, newdata)

matplotlib.pyplot.subplot(2, 1, 2)
matplotlib.pyplot.title("Quiet")
matplotlib.pyplot.plot(newdata)

matplotlib.pyplot.show()
```


3

Get to Grips with Commonly Used Functions

In this chapter, we will cover a number of commonly used functions:

- ▶ `sqrt`, `log`, `arange`, `astype`, and `sum`
- ▶ `ceil`, `modf`, `where`, `ravel`, and `take`
- ▶ `sort` and `outer`
- ▶ `diff`, `sign`, `eig`
- ▶ `histogram` and `polyfit`
- ▶ `compress` and `randint`

We will be discussing these functions through the following recipes:

- ▶ Summing Fibonacci numbers
- ▶ Finding prime factors
- ▶ Finding palindromic numbers
- ▶ The steady state vector determination
- ▶ Discovering a power law
- ▶ Trading periodically on dips
- ▶ Simulating trading at random
- ▶ Sieving integers with the Sieve of Eratosthenes

Introduction

This chapter is about the commonly used functions. These are the functions that you will be using on a daily basis. Obviously, the usage may differ for you. There are so many NumPy functions that it is virtually impossible to know all of them, but the functions in this chapter will be the bare minimum with which we must be familiar. You can download source code for this chapter from the book website <http://www.packtpub.com>.

Summing Fibonacci numbers

In this recipe, we will sum the even-valued terms in the Fibonacci sequence whose values do not exceed four million. The Fibonacci series is a sequence of integers starting with zero, where each number is the sum of the previous two; except, of course, the first two numbers zero and one.



For more information, read the Wikipedia article about Fibonacci numbers at http://en.wikipedia.org/wiki/Fibonacci_number.

This recipe uses a formula based on the golden ratio, which is an irrational number with special properties comparable to pi. It will use the `sqrt`, `log`, `arange`, `astype`, and `sum` functions.

How to do it...

The first thing to do is calculate the golden ratio (http://en.wikipedia.org/wiki/Golden_ratio), also called the golden section or golden mean.

1. Calculate the golden ratio.

We will be using the `sqrt` function to calculate the square root of five:

```
phi = (1 + numpy.sqrt(5))/2
print "Phi", phi
```

This prints the golden mean:

```
Phi 1.61803398875
```

2. Find the index below four million.

Next in the recipe, we need to find the index of the Fibonacci number below four million. A formula for this is given in the Wikipedia page, and we will compute it using that formula. All we need to do is convert log bases with the `log` function. We don't need to round the result down to the closest integer. This is automatically done for us in the next step of the recipe:

```
n = numpy.log(4 * 10 ** 6 * numpy.sqrt(5)
              + 0.5)/numpy.log(phi)
print n
```

The value for `n` is:

```
33.2629480359
```

3. Create an array of 1-n.

The `arange` function is a very basic function, which many people know. Still, we will mention it here for completeness:


```
n = numpy.arange(1, n)
```


4. Compute the Fibonacci numbers.

There is a convenient formula we can use to calculate the Fibonacci numbers. We will need the golden ratio and the array from the previous step in this recipe as input parameters.

Print the first nine Fibonacci numbers to check the result:

```
fib = (phi**n - (-1/phi)**n)/numpy.sqrt(5)
print "First 9 Fibonacci Numbers", fib[:9]
```

 I could have made a unit test instead of a print statement. A unit test is a test which tests a small unit of code, such as a function. This variation of the recipe is left as an exercise for the reader.

 Have a look at *Chapter 8, Quality Assurance* for pointers on how to write a unit test.

We are not starting with the number zero here, by the way. The aforementioned code gives us a series as expected:

```
First 9 Fibonacci Numbers [ 1.  1.  2.  3.  5.  8. 13. 21.
34.]
```

You can plug this right into a unit test, if you want.

5. Convert to integers.

This step is optional. I think it's nice to have an integer result at the end. OK, I actually wanted to show you the `astype` function:

```
fib = fib.astype(int)
print "Integers", fib
```

This code gives us the following result, after snipping a bit for brevity:

```
Integers [      1      1      2      3      5      8      13
21      34
... snip ... snip ...
317811 514229 832040 1346269 2178309 3524578]
```

6. Select even-valued terms.

The recipe demands that we select the even-valued terms now. This should be easy for you, if you followed the boolean indexing piece in the previous chapter:

```
eventerms = fib[fib % 2 == 0]
print eventerms
```

There we go:

```
[      2      8      34      144      610      2584      10946      46368
196418
      832040 3524578]
```

For completeness, following is the complete code for this recipe:

```
import numpy

#Each new term in the Fibonacci sequence is generated by adding the
previous two terms.
#By starting with 1 and 2, the first 10 terms will be:

#1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

#By considering the terms in the Fibonacci sequence whose values do
not exceed four million,
#find the sum of the even-valued terms.

#1. Calculate phi
phi = (1 + numpy.sqrt(5))/2
print "Phi", phi

#2. Find the index below 4 million
n = numpy.log(4 * 10 ** 6 * numpy.sqrt(5) + 0.5)/numpy.log(phi)
print n
```

```
#3. Create an array of 1-n
n = numpy.arange(1, n)
print n

#4. Compute Fibonacci numbers
fib = (phi**n - (-1/phi)**n)/numpy.sqrt(5)
print "First 9 Fibonacci Numbers", fib[:9]

#5. Convert to integers
# optional
fib = fib.astype(int)
print "Integers", fib

#6. Select even-valued terms
eventerms = fib[fib % 2 == 0]
print eventerms

#7. Sum the selected terms
print eventerms.sum()
```

How it works...

In this recipe, we used the functions `sqrt`, `log`, `arange`, `astype`, and `sum`; their description is as follows:

| Function | Description |
|---------------------|---|
| <code>sqrt</code> | Calculates the square root of array elements. |
| <code>log</code> | Calculates the natural log of array elements. |
| <code>arange</code> | Creates an array with a specified range. |
| <code>astype</code> | Converts array elements to a specified data type. |
| <code>sum</code> | Calculates the sum of array elements. |

See also

- ▶ The *Indexing with booleans* recipe in *Chapter 2, Advanced Indexing and Array Concepts*

Finding prime factors

Prime factors (http://en.wikipedia.org/wiki/Prime_factor) are prime numbers that divide an integer exactly without a remainder. Finding prime factors seems almost impossible to crack. However, using the right algorithm—Fermat's factorization method (http://en.wikipedia.org/wiki/Fermat%27s_factorization_method) and NumPy—it becomes very easy. The idea is to factor a number **N** into two numbers **c** and **d**, according to the following equation:

$$N = cd = (a + b)(a - b) = a^2 - b^2$$

We can apply the factorization recursively, until we get the required prime factors.

How to do it...

The algorithm requires us to try a number of trial values for **a**.

1. Create an array of trial values.

It makes sense to create a NumPy array and eliminate the need for loops. However, you should be careful to not create an array that is too big in terms of memory requirements. On my system, an array of a million elements seems to be just the right size:

```
a = numpy.ceil(numpy.sqrt(n))
lim = min(n, LIM)
a = numpy.arange(a, a + lim)
b2 = a ** 2 - n
```

We used the `ceil` function to return the ceiling of the input, element-wise.

2. Get the fractional part of the **b** array.

We are now supposed to check whether **b** is a square. Use the NumPy `modf` function to get the fractional part of the **b** array:

```
fractions = numpy.modf(numpy.sqrt(b2))[0]
```

3. Find 0 fractions.

Call the NumPy `where` function to find the indices of zero fractions, where the fractional part is 0:

```
indices = numpy.where(fractions == 0)
```

4. Find the first occurrence of a zero fraction.

Actually, we only need the first occurrence of a zero fraction. First, call the NumPy `take` function with the indices array from the previous step to get the values of zero fractions. Now we need to "flatten" this array with the NumPy `ravel` function:

```
a = numpy.ravel(numpy.take(a, indices))[0]
```

The following is the entire code needed to solve the problem of finding the largest prime factor of the number 600851475143:

```
import numpy

#The prime factors of 13195 are 5, 7, 13 and 29.

#What is the largest prime factor of the number 600851475143 ?

N = 600851475143
LIM = 10 ** 6

def factor(n):
    #1. Create array of trial values
    a = numpy.ceil(numpy.sqrt(n))
    lim = min(n, LIM)
    a = numpy.arange(a, a + lim)
    b2 = a ** 2 - n

    #2. Check whether b is a square
    fractions = numpy.modf(numpy.sqrt(b2))[0]

    #3. Find 0 fractions
    indices = numpy.where(fractions == 0)

    #4. Find the first occurrence of a 0 fraction
    a = numpy.ravel(numpy.take(a, indices))[0]
    a = int(a)
    b = numpy.sqrt(a ** 2 - n)
    b = int(b)
    c = a + b
    d = a - b

    if c == 1 or d == 1:
        return
```

```
print c, d
factor(c)
factor(d)

factor(N)
```

The output for this code is the following:

```
1234169 486847
1471 839
6857 71
```

How it works...

We applied the Fermat factorization recursively using the NumPy functions `ceil`, `modf`, `where`, `ravel`, and `take`. The description of these functions is as follows:

| Function | Description |
|--------------------|---|
| <code>ceil</code> | Calculates the ceiling of array elements. |
| <code>modf</code> | Returns the fractional and integral part of floating point numbers. |
| <code>where</code> | Returns array indices based on condition. |
| <code>ravel</code> | Returns a flattened array. |
| <code>take</code> | Takes element from an array. |

Finding palindromic numbers

A palindromic number reads the same both ways. The largest palindrome made from the product of two 2-digit numbers is $9009 = 91 \times 99$. Let's try to find the largest palindrome made from the product of two 3-digit numbers.

How to do it...

We will create an array to hold 3-digit numbers from 100 to 999 using our favorite NumPy function `arange`.

1. Create a 3-digit numbers array.

Check the first and last element of the array with the `assert_equal` function from the `numpy.testing` package:

```
a = numpy.arange(100, 1000)
numpy.testing.assert_equal(100, a[0])
numpy.testing.assert_equal(999, a[-1])
```

2. Create the products array

Now, we will create an array to hold all the possible products of the elements of the 3-digits array with itself. We can accomplish this with the `outer` function. The resulting array needs to be flattened with `ravel`, to be able to easily iterate over it. Call the `sort` method on the array to make sure the array is properly sorted. After that, we can do some sanity checks:

```
numbers = numpy.outer(a, a)
numbers = numpy.ravel(numbers)
numbers.sort()
numpy.testing.assert_equal(810000, len(numbers))
numpy.testing.assert_equal(10000, numbers[0])
numpy.testing.assert_equal(998001, numbers[-1])
```

The following is the complete program:

```
import numpy
import numpy.testing

#A palindromic number reads the same both ways.
#The largest palindrome made from the product of two 2-digit
numbers is 9009 = 91 x 99.

#Find the largest palindrome made from the product of two
3-digit numbers.

#1. Create 3-digits numbers array
a = numpy.arange(100, 1000)
numpy.testing.assert_equal(100, a[0])
numpy.testing.assert_equal(999, a[-1])

#2. Create products array
numbers = numpy.outer(a, a)
numbers = numpy.ravel(numbers)
numbers.sort()
numpy.testing.assert_equal(810000, len(numbers))
numpy.testing.assert_equal(10000, numbers[0])
numpy.testing.assert_equal(998001, numbers[-1])
```

```
#3. Find largest palindromic number
for i in xrange(-1, -1 * len(numbers), -1):
    s = str(numbers[i])
    if s == s[::-1]:
        print s
        break
```

The code prints 906609, which is a palindromic number.

How it works...

We saw the `outer` function in action. This function returns the outer product of two arrays (http://en.wikipedia.org/wiki/Outer_product). The `sort` function returns a sorted copy of an array.

There's more...

It might be a good idea to check the result. Find out which two 3-digit numbers produce our palindromic number by modifying the code a bit. Try implementing the last step in a NumPy way.

The steady state vector determination

A **Markov chain** is a system that has at least two states. For detailed information on Markov chains, please refer to http://en.wikipedia.org/wiki/Markov_chain. The state at time t depends on the state at time $t-1$, and only the state at $t-1$. The system switches at random between these states. I would like to define a Markov chain for a stock. Let's say that we have the states flat F , up U , and down D . We can determine the steady state based on end of day close prices.

Far into the distant future or in theory infinite time, the state of our Markov chain system will not change anymore. This is also called a steady state (http://en.wikipedia.org/wiki/Steady_state). The **stochastic matrix** (http://en.wikipedia.org/wiki/Stochastic_matrix) A , which contains the state transition probabilities, and when applied to the steady state, will yield the same state x . The mathematical notation for this will be as follows:

$$Ax = x$$

Another way to look at this is as the **eigenvector** (http://en.wikipedia.org/wiki/Eigenvalues_and_eigenvectors) for eigenvalue 1.

How to do it...

Now we need to obtain the data.

1. Obtain one year of data.

One way we can do this is with Matplotlib (refer to the *Installing Matplotlib* recipe in *Chapter 1, Winding Along with IPython*, if necessary). We will retrieve the data going back one year. Here is the code to do this:

```
today = date.today()
start = (today.year - 1, today.month, today.day)

quotes = quotes_historical_yahoo('AAPL', start, today)
```

2. Select the close price.

We now have historical data from Yahoo Finance. The data is represented as a list of tuples, but we are only interested in the close price. For example:

```
[(734744.0, 675.25, 673.4700000000003, 677.6699999999996,
 672.6000000000002, 7228500.0), ..., (734745.0, 670.6399999999999,
 663.87, 671.5499999999995, 662.8500000000002, 10799600.0)]
```

The first element in the tuple represents the *date*. It is followed by the *open*, *high*, *low*, and *close* prices. The last element is the *volume*. We can select the close prices as follows.

```
close = [q[4] for q in quotes]
```

The close price is the fifth number in each tuple. We should have a list of about 253 close prices now.

3. Determine the states.

We can determine the states by subtracting the price of sequential days with the NumPy `diff` function. The state is then given by the sign of the difference. The NumPy `sign` function returns `-1` for a negative, `1` for a positive number, or `0` otherwise.

```
states = numpy.sign(numpy.diff(close))
```

4. Initialize the stochastic matrix to 0 values.

We have three possible start states and three possible end states for each transition. For instance, if we start from a *U* state, we could switch to:

- U
- F
- D

Initialize the stochastic matrix with the NumPy `zeros` function:

```
NDIM = 3
SM = numpy.zeros((NDIM, NDIM))
```

5. For each sign, select the corresponding start state indices.

Now the code becomes a bit messy. We will have to use actual loops! We will loop over the possible signs, and select the start state indices corresponding to each sign. Select the indices with the NumPy, `where` function. `k` here is a smoothing constant that we will discuss later on:

```
signs = [-1, 0, 1]
k = int(sys.argv[2])

for i in xrange(len(signs)):
    #we start the transition from the state with the specified sign
    start_indices = numpy.where
        (states[:-1] == signs[i])[0]
```

6. Smoothing and the stochastic matrix.

We can now count the number of occurrences of each transition. Dividing by the total number of transitions for a given start state gives us the transition probabilities for our stochastic matrix. This is not the best method, by the way, since it could be over-fitting.

In real life, we could have a day that the close price does not change, although unlikely for liquid stock markets. One way to deal with zero occurrences is to apply additive smoothing (http://en.wikipedia.org/wiki/Additive_smoothing). The idea is to add a certain constant to the number of occurrences we find, getting rid of zeroes. The following code calculates the values of the stochastic matrix:

```
N = len(start_indices) + k * NDIM

# skip since there are no transitions possible
if N == 0:
    continue

#find the values of states at the end positions
end_values = states[start_indices + 1]

for j in xrange(len(signs)):
    # number of occurrences of this transition
    occurrences = len
        (end_values[end_values == signs[j]])
    SM[i][j] = (occurrences + k)/float(N)

print SM
```

What the aforementioned code does is compute the transition probabilities for each possible transition based on the number of occurrences and additive smoothing.

For **AAPL (Apple Inc.)** and smoothing constant $k = 1$, I got the following stochastic matrix:

```
[[ 0.50925926  0.00925926  0.48148148]
 [ 0.33333333  0.33333333  0.33333333]
 [ 0.35135135  0.00675676  0.64189189]]
```

7. Eigenvalues and eigenvectors.

To get the eigenvalues and eigenvectors, we will need the NumPy `linalg` module and the `eig` function:

```
eig_out = numpy.linalg.eig(SM)
print eig_out
```

The `eig` function returns an array containing the eigenvalues and an array containing the eigenvectors:

```
(array([ 1.          ,  0.15817566,  0.32630882]),
 array([[ 0.57735027,  0.74473695,  0.00297158],
        [ 0.57735027, -0.39841481, -0.99983179],
        [ 0.57735027, -0.53538072,  0.01809841]]))
```

8. Select the eigenvector for eigenvalue 1.

Currently, we are only interested in the eigenvector for eigenvalue 1. In reality, the eigenvalue might not be exactly one, so we should build in a margin for error. We can find the index for eigenvalue between 0.9 and 1.1 as follows:

```
idx_vec = numpy.where
    (numpy.abs(eig_out[0] - 1) < 0.1)
print "Index eigenvalue 1", idx_vec

x = eig_out[1][:,idx_vec].flatten()
```

The following is the complete code for the steady state vector example:

```
from matplotlib.finance
import quotes_historical_yahoo
from datetime import date
import numpy
import sys

if (len(sys.argv) != 3):
    print "Usage python %s SYMBOL
          k" % (sys.argv[0])
```


Get to Grips with Commonly Used Functions

```
print "For instance python %s
      AAPL 1" % (sys.argv[0])
sys.exit()

today = date.today()
start = (today.year - 1, today.month, today.day)

quotes = quotes_historical_yahoo
        (sys.argv[1], start, today)
close = [q[4] for q in quotes]

states = numpy.sign(numpy.diff(close))

NDIM = 3
SM = numpy.zeros((NDIM, NDIM))

signs = [-1, 0, 1]
k = int(sys.argv[2])

for i in xrange(len(signs)):
    #we start the transition from the state with the specified sign
    start_indices = numpy.where
        (states[:-1] == signs[i])[0]

    N = len(start_indices) + k * NDIM

    # skip since there are no transitions possible
    if N == 0:
        continue

    #find the values of states at the end positions
    end_values = states[start_indices + 1]

    for j in xrange(len(signs)):
        # number of occurrences of this transition
        occurrences = len
            (end_values[end_values == signs[j]])
        SM[i][j] = (occurrences + k)/float(N)

print SM
eig_out = numpy.linalg.eig(SM)
print eig_out
```

```

idx_vec = numpy.where
    (numpy.abs(eig_out[0] - 1) < 0.1)
print "Index eigenvalue 1", idx_vec

x = eig_out[1][:,idx_vec].flatten()
print "Steady state vector", x
print "Check", numpy.dot(SM, x)

```

The output for this code is as follows:

```

[[ 0.4952381  0.00952381  0.4952381 ]
 [ 0.33333333 0.33333333 0.33333333]
 [ 0.34210526 0.00657895 0.65131579]]
(array([ 1.          , 0.15328174, 0.32660547]),
 array([[ 0.57735027, 0.7424435 , 0.00144451],
        [ 0.57735027, -0.44112882, -0.99982343],
        [ 0.57735027, -0.50416566, 0.01873551]]))
Index eigenvalue 1 (array([0]),)
Steady state vector [ 0.57735027  0.57735027  0.57735027]
Check [ 0.57735027  0.57735027  0.57735027]

```

How it works...

The values for the eigenvector we get are not normalized. Because we are the dealing with probabilities, they should sum up to one. The `diff`, `sign`, and `eig` functions were introduced in this example. Their descriptions are as follows:

| Function | Description |
|-------------------|--|
| <code>diff</code> | Calculates the discrete difference. By default, the first order. |
| <code>sign</code> | Returns the sign of array elements. |
| <code>eig</code> | Returns the eigenvalues and eigenvectors of an array. |

See also

- ▶ The *Installing Matplotlib* recipe in Chapter 1, *Winding Along with IPython*

Discovering a power law

For the purpose of this recipe, imagine that we are operating a Hedge Fund. Let it sink in; you are part of the one percent now!

Power laws occur in a lot of places, see http://en.wikipedia.org/wiki/Power_law for more information. The **Pareto principle** (http://en.wikipedia.org/wiki/Pareto_principle) for instance, which is a power law, states that wealth is unevenly distributed. This principle tells us that if we group people by their wealth, the size of the groups will vary exponentially. To put it simply, there are not a lot of rich people, and there are even less billionaires; hence the one percent.

Assume that there is a power law in the closing stock prices log returns. This is a big assumption, of course, but power law assumptions seem to pop up all over the place.

We don't want to trade too often, because of involved transaction costs per trade. Let's say that we would prefer to buy and sell once a month based on a significant correction (in other words a big drop). The issue is to determine an appropriate signal given that we want to initiate a transaction every one out of about 20 days.

How to do it...

First, let's get historical end-of-day data for the past year from Yahoo Finance. After that, extract the close prices for this period. These steps are described in the previous recipe.

1. Get positive log returns.

Now calculate the log returns for the close prices. For more information on log returns refer to http://en.wikipedia.org/wiki/Rate_of_return.

First, we will take the log of the close prices, and then compute the first difference of these values with the NumPy `diff` function. Let's select the positive values from the log returns. Why the positive values? It doesn't really matter; I like being positive:

```
logreturns = numpy.diff(numpy.log(close))
pos = logreturns[logreturns > 0]
```

2. Get frequencies of returns.

We need to get the frequencies of the returns with the `histogram` function. Counts and an array of the bins are returned. At the end, we need to take the log of the frequencies in order to get a nice linear relation:

```
counts, rets = numpy.histogram(pos)
rets = rets[:-1] + (rets[1] - rets[0])/2
```

```
freqs = 1.0/counts
freqs = numpy.log(freqs)
```

3. Fit the frequencies and returns to a line.

Use the `polyfit` function to do a linear fit:

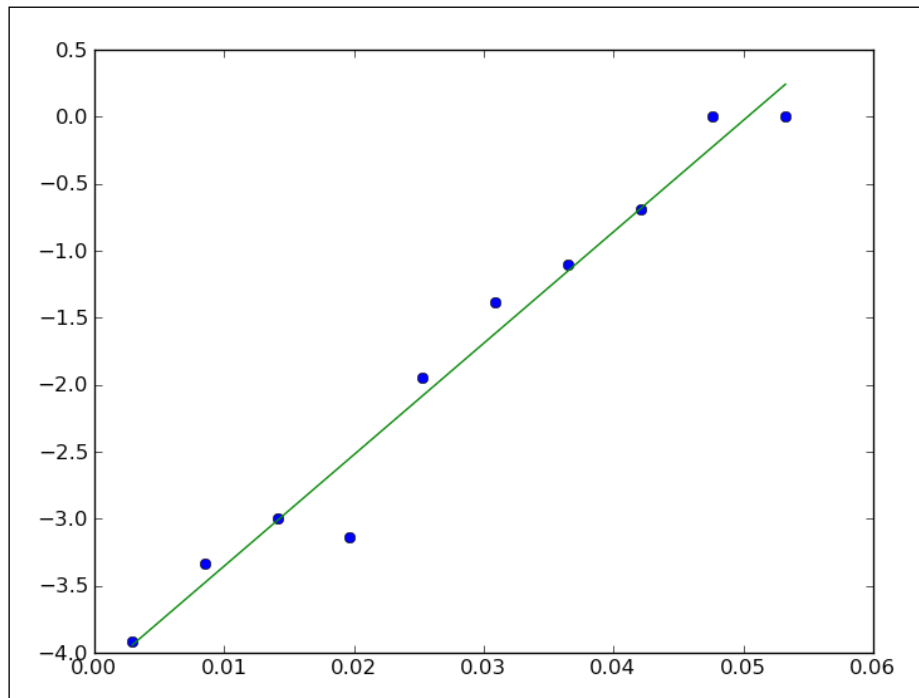
```
p = numpy.polyfit(rets, freqs, 1)
```

4. Plot the results.

Finally, we will plot the data and linear fit with Matplotlib:

```
matplotlib.pyplot.plot(rets, freqs, 'o')
matplotlib.pyplot.plot(rets, p[0] * rets + p[1])
matplotlib.pyplot.show()
```

We get a nice plot of the linear fit, returns, and frequencies:



The following is the complete code:

```
from matplotlib.finance import quotes_historical_yahoo
from datetime import date
import numpy
import sys
import matplotlib.pyplot

#1. Get close prices.
today = date.today()
start = (today.year - 1, today.month, today.day)

quotes = quotes_historical_yahoo(sys.argv[1], start, today)
close = numpy.array([q[4] for q in quotes])
#2. Get positive log returns.
logreturns = numpy.diff(numpy.log(close))
pos = logreturns[logreturns > 0]

#3. Get frequencies of returns.
counts, rets = numpy.histogram(pos)
rets = rets[:-1] + (rets[1] - rets[0])/2
freqs = 1.0/counts
freqs = numpy.log(freqs)

#4. Fit the frequencies and returns to a line.
p = numpy.polyfit(rets, freqs, 1)

#5. Plot the results.
matplotlib.pyplot.plot(rets, freqs, 'o')
matplotlib.pyplot.plot(rets, p[0] * rets + p[1])
matplotlib.pyplot.show()
```

How it works...

The `histogram` function calculates the histogram of a data set. It returns the histogram values and bin edges. The `polyfit` function fits data to a polynomial of given order. In this case, we chose for a linear fit. We "discovered" a power law—you have to be careful making such claims, but the evidence looks promising.

See also

- ▶ The *Installing Matplotlib* recipe in Chapter 1, *Winding Along with IPython*

Trading periodically on dips

Stock prices periodically dip and go up. We will have a look at the probability distribution of the stock price log returns.

Let's start by downloading the historical data for a stock; for instance, AAPL. Next, calculate the daily log returns (http://en.wikipedia.org/wiki/Rate_of_return) of the close prices. We will skip these steps because they were already done in the previous recipe.

Getting ready

If necessary, install Matplotlib and SciPy. Refer to the *See Also* section for the corresponding recipes.

How to do it...

Now comes the interesting part.

1. Calculate breakout and pullback.

Let's say we want to trade five times per year, or roughly every 50 days. One strategy would be to buy when the price drops by a certain percentage—a pullback, and sell when the price increases by another percentage—a breakout.

By setting the percentile appropriate for our trading frequency, we can match the corresponding log returns. SciPy offers the `scoreatpercentile` function, which we will use:

```
freq = 1/float(sys.argv[2])
breakout = scipy.stats.scoreatpercentile
           (logreturns, 100 * (1 - freq) )
pullback = scipy.stats.scoreatpercentile
           (logreturns, 100 * freq)
```

2. Generate buys and sells.

Use the NumPy `compress` function to generate buys and sells for our close price data. This function returns elements based on a condition:

```
buys = numpy.compress
      (logreturns < pullback, close)
sells = numpy.compress
       (logreturns > breakout, close)
print buys
print sells
print len(buys), len(sells)
print sells.sum() - buys.sum()
```

The output for AAPL and a 50-day period is as follows:

```
[ 340.1  377.35  378.   373.17  415.99]
[ 357.   370.8  366.48  395.2   419.55]
5 5
24.42
```

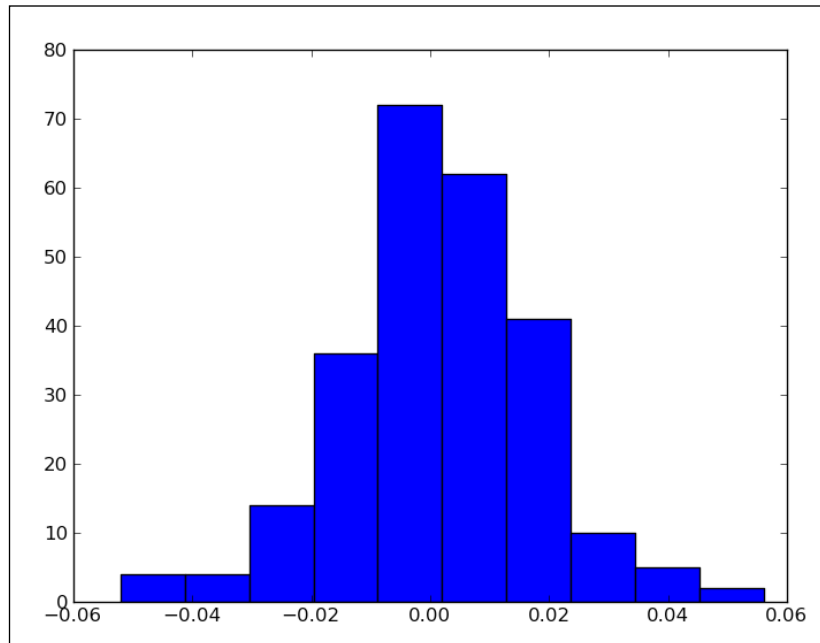
So we have a profit of 24 dollars, if we buy and sell an AAPL share five times.

3. Plot a histogram of the log returns.

Just for fun, let's plot the histogram of the log returns with Matplotlib:

```
matplotlib.pyplot.hist(logreturns)
matplotlib.pyplot.show()
```

This is what the histogram looks like.



The following is the complete code:

```
from matplotlib.finance import quotes_historical_yahoo
from datetime import date
import numpy
import sys
import scipy.stats
import matplotlib.pyplot
```

```

#1. Get close prices.
today = date.today()
start = (today.year - 1, today.month, today.day)

quotes = quotes_historical_yahoo(sys.argv[1], start, today)
close = numpy.array([q[4] for q in quotes])

#2. Get log returns.
logreturns = numpy.diff(numpy.log(close))

#3. Calculate breakout and pullback
freq = 1/float(sys.argv[2])
breakout = scipy.stats.scoreatpercentile(logreturns, 100 * (1 - freq)
)
pullback = scipy.stats.scoreatpercentile(logreturns, 100 * freq)
#4. Generate buys and sells
buys = numpy.compress(logreturns < pullback, close)
sells = numpy.compress(logreturns > breakout, close)
print buys
print sells
print len(buys), len(sells)
print sells.sum() - buys.sum()

#5. Plot a histogram of the log returns
matplotlib.pyplot.hist(logreturns)
matplotlib.pyplot.show()

#AAPL 50
#[ 340.1  377.35  378.    373.17  415.99]
#[ 357.    370.8  366.48  395.2  419.55]
#5 5
#24.42

```

How it works...

We encountered the `compress` function, which returns an array containing the array elements of the input that satisfy a given condition. The input array remains unchanged.

See also

- ▶ The *Installing Matplotlib* recipe in Chapter 1, *Winding Along with IPython*
- ▶ The *Installing SciPy* in Chapter 2, *Advanced Indexing and Array Concepts*
- ▶ The *Discovering a power law* recipe

Simulating trading at random

In the previous recipe, we tried out a trading idea. However, we have no benchmark that can tell us if the result we got was any good. It is common in such cases to trade at random, under the assumption that we should be able to beat a random process. We will simulate trading by taking some random days from a trading year. This should illustrate working with random numbers using NumPy.

Getting ready

If necessary, install Matplotlib. Refer to the *See Also* section for the corresponding recipe.

How to do it...

First, we need an array filled with random integers.

1. Generate random indices.

Generate random integers with the NumPy `randint` function. This will be linked to random days of a trading year:

```
return numpy.random.randint(0, high, size)
```

2. Simulate trades.

Simulate trades with the random indices from the previous step. Use the NumPy `take` function to extract random close prices from the array of step 1:

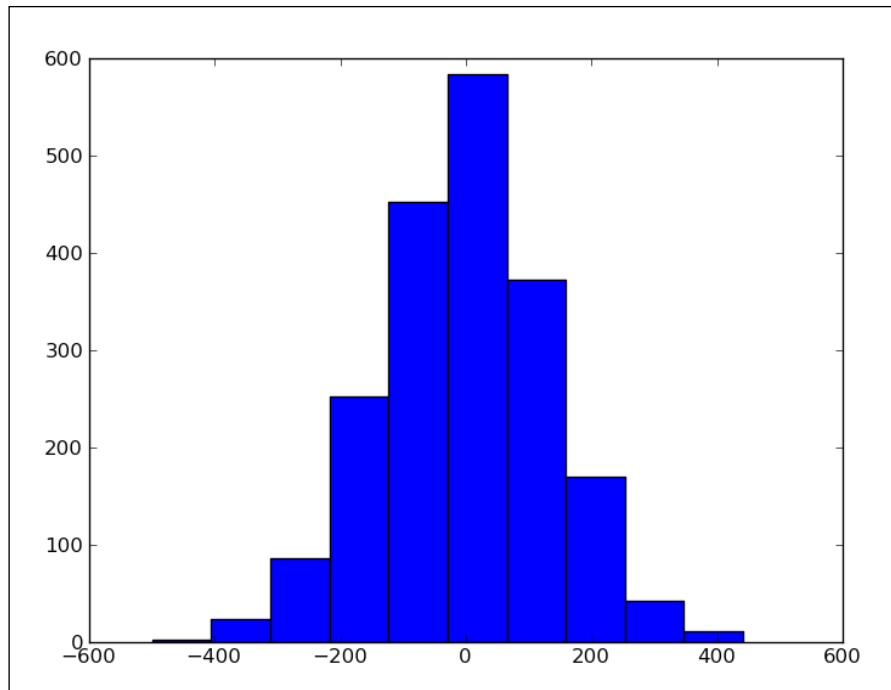
```
buys = numpy.take(close, get_indices(len(close), nbuys))
sells = numpy.take(close, get_indices(len(close), nbuys))
profits[i] = sells.sum() - buys.sum()
```

3. Plot a histogram of the profits.

Let's plot a histogram of the profits for a large number of simulations:

```
matplotlib.pyplot.hist(profits)
matplotlib.pyplot.show()
```

The resulting histogram of 2000 simulations for AAPL with 5 buys and sells in a year:



The following is the complete code:

```
from matplotlib.finance import quotes_historical_yahoo
from datetime import date
import numpy
import sys
import matplotlib.pyplot

def get_indices(high, size):
    #2. Generate random indices
    return numpy.random.randint(0, high, size)

#1. Get close prices.
today = date.today()
start = (today.year - 1, today.month, today.day)

quotes = quotes_historical_yahoo(sys.argv[1], start, today)
close = numpy.array([q[4] for q in quotes])

nbuys = int(sys.argv[2])
N = int(sys.argv[3])
profits = numpy.zeros(N)
```

```
for i in xrange(N):
    #3. Simulate trades
    buys = numpy.take(close, get_indices(len(close), nbuys))
    sells = numpy.take(close, get_indices(len(close), nbuys))
    profits[i] = sells.sum() - buys.sum()

print "Mean", profits.mean()
print "Std", profits.std()

#4. Plot a histogram of the profits
matplotlib.pyplot.hist(profits)
matplotlib.pyplot.show()

#python random_periodic.py AAPL 5 2000
#Mean -2.566465
#Std 133.746039463
```

How it works...

We used the `randint` function, which can be found in the `numpy.random` module. This module contains more convenient random generators, as described in the following table:

| Function | Description |
|----------------------|--|
| <code>rand</code> | Creates an array from a uniform distribution over [0,1] with a shape based on dimension parameters. If no dimensions are specified a single float is returned. |
| <code>randn</code> | Sample values from the normal distribution with mean 0 and variance 1. The dimension parameters function the same way as for <code>rand</code> . |
| <code>randint</code> | Returns an integer array given a low boundary, an optional high bound, and an optional output shape. |

See also

- ▶ The *Installing Matplotlib* recipe in Chapter 1, *Winding Along with IPython*

Sieving integers with the Sieve of Eratosthenes

The **Sieve of Eratosthenes** (http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes) is an algorithm that filters out prime numbers. It iteratively identifies multiples of found primes. This sieve is efficient for primes smaller than 10 million. Let's now try to find the 10001st prime number.

How to do it...

The first mandatory step is to create a list of natural numbers.

1. Create a list of consecutive integers.

NumPy has the `arange` function for that:

```
a = numpy.arange(i, i + LIM, 2)
```

2. Sieve out multiples of `p`.

We are not sure if this is what Eratosthenes wanted us to do, but it works. In the following code, we are passing a NumPy array and getting rid of all the elements that have a zero remainder, when divided by `p`:

```
a = a[a % p != 0]
```

The following is the entire code for this problem:

```
import numpy

LIM = 10 ** 6
N = 10 ** 9
P = 10001
primes = []
p = 2

#By listing the first six prime numbers: 2, 3, 5, 7, 11, and 13,
we can see that the 6th prime is 13.
#What is the 10 001st prime number?

def check_primes(a, p):
    #2. Sieve out multiples of p
    a = a[a % p != 0]

    return a

for i in xrange(3, N, LIM):
    #1. Create a list of consecutive integers
    a = numpy.arange(i, i + LIM, 2)

    while len(primes) < P:
        a = check_primes(a, p)
        primes.append(p)

        p = a[0]

print len(primes), primes[P-1]
```


4

Connecting NumPy with the Rest of the World

In this chapter, we will cover:

- ▶ Using the buffer protocol
- ▶ Using the array interface
- ▶ Exchanging data with MATLAB and Octave
- ▶ Installing RPy2
- ▶ Interfacing with R
- ▶ Installing JPype
- ▶ Sending a NumPy array to JPype
- ▶ Installing Google App Engine
- ▶ Deploying NumPy code in the Google cloud
- ▶ Running NumPy code in a Python Anywhere web console
- ▶ Setting up PiCloud

Introduction

This chapter is about **interoperability**. We have to keep reminding ourselves that NumPy is not alone in the scientific (Python) software ecosystem. Working together with SciPy and Matplotlib is pretty easy. Protocols exist for interoperability with other Python packages. Outside of the Python ecosystem languages such as Java, R, C, and FORTRAN are pretty popular. We will go into the details of exchanging data with these environments.

Also, we will discuss how to get our NumPy code in the cloud. This is a continuously evolving technology, in a fast-moving space. Many options are available to you, of which Google App Engine, PiCloud, and Python Anywhere will be covered.



The danger here is to seem subjective, so please be assured that the author is in no way affiliated with any of these companies.

Using the buffer protocol

C-based Python objects have a so called "buffer interface". Python objects can expose their data for direct access without the need to copy it. The buffer protocol enables us to communicate with other Python software such as the **Python Imaging Library (PIL)**. We will see an example of saving a PIL image from a NumPy array.

Getting ready

Install PIL and SciPy, if necessary. Check the *See Also* section of this recipe for instructions.

How to do it...

First, we need a NumPy array with which to play.

1. Create an array from image data.

In previous chapters, we saw how to load the "Lena" sample image of Lena Soderberg. We will create an array filled with zeroes, and populate the alpha channel with the image data:

```
lena = scipy.misc.lena()
data = numpy.zeros((lena.shape[0], lena.shape[1], 4), dtype=numpy.int8)
data[:, :, 3] = lena.copy()
```

2. Save the data as a PIL image.

Now, we will use the PIL API to save the data as a RGBA image:

```
img = Image.frombuffer("RGBA", lena.shape, data)
img.save('lena_frombuffer.png')
```

3. Modify the data array and save the image.

Modify the data array by getting rid of the image data and making the image *red*.

Save the image with the PIL API:

```
data[:, :, 3] = 255  
data[:, :, 0] = 222  
img.save('lena_modified.png')
```

The following is the *before* image:



The data of the PIL image object has changed by the magic of the "buffer interface", and therefore, we see the following image:



The complete code for this recipe is as follows:

```
import numpy
import Image
import scipy.misc

lena = scipy.misc.lena()
data = numpy.zeros((lena.shape[0], lena.shape[1], 4), dtype=numpy.
int8)
data[:, :, 3] = lena.copy()
img = Image.frombuffer("RGBA", lena.shape, data)
img.save('lena_frombuffer.png')

data[:, :, 3] = 255
data[:, :, 0] = 222
img.save('lena_modified.png')
```

How it works...

We created a PIL image from a buffer—a NumPy array. After changing the buffer, we saw the changes being reflected in the image object. This was done without copying the PIL image object; instead, we directly accessed and modified its data to make a red image out of the picture of the model.

See also

- ▶ The *Installing PIL* recipe in *Chapter 2, Advanced Indexing and Array Concepts*
- ▶ The *Installing SciPy* recipe in *Chapter 2, Advanced Indexing and Array Concepts*

Using the array interface

The array interface is a yet another mechanism to communicate with other Python applications. This protocol, as its name suggests, is only applicable to array-like objects. A demonstration is in order. Let's use PIL again, but without saving files.

Getting ready

We will be reusing part of the code from the previous recipe, so the prerequisites are similar. We will skip the first step of the previous step here, and assume it is already known.

How to do it...

The following steps will let us explore the array interface:

1. The PIL image array interface attribute.

The PIL image object has a `__array_interface__` attribute. Let's inspect its contents. The value of this attribute is a dictionary:

```
array_interface = img.__array_interface__
print "Keys", array_interface.keys()
print "Shape", array_interface['shape']
print "Typestr", array_interface['typestr']
```

This code prints the following information:

```
Keys ['shape', 'data', 'typestr']
Shape (512, 512, 4)
Typestr |u1
```

2. The NumPy array interface attributes.

The NumPy `ndarray` module has a `__array_interface__` attribute as well. We can convert the PIL image to a NumPy array with the `asarray` function:

```
numpy_array = numpy.asarray(img)
print "Shape", numpy_array.shape
print "Data type", numpy_array.dtype
```

The shape and data type of the array:

```
Shape (512, 512, 4)
```

```
Data type uint8
```

As you can see, the shape has not changed. The code for this recipe is as follows:

```
import numpy
import Image
import scipy.misc

lena = scipy.misc.lena()
data = numpy.zeros((lena.shape[0],
                  lena.shape[1], 4), dtype=numpy.int8)
data[:, :, 3] = lena.copy()
img = Image.frombuffer("RGBA", lena.shape, data)
array_interface = img.__array_interface__
print "Keys", array_interface.keys()
print "Shape", array_interface['shape']
print "Typestr", array_interface['typestr']

numpy_array = numpy.asarray(img)
print "Shape", numpy_array.shape
print "Data type", numpy_array.dtype
```

How it works...

The array interface or protocol lets us share data between array-like Python objects. Both NumPy and PIL provide such an interface.

See also

- ▶ *Using the buffer protocol* in this chapter

Exchanging data with MATLAB and Octave

MATLAB and its open source alternative Octave are popular mathematical applications. The `scipy.io` package has the `savemat` function, which allows you to store NumPy arrays in a `.mat` file as a value of a dictionary.

Getting ready

Installing MATLAB or Octave is outside of the scope of this book. The Octave website has some pointers for installing: <http://www.gnu.org/software/octave/download.html>. Check the See Also section of this recipe, for instructions on installing SciPy, if necessary.

How to do it...

Once you have installed MATLAB or Octave, you need to follow the subsequent steps to store NumPy arrays:

1. Call `savemat`.

Create a NumPy array, and call `savemat` to store the array in a `.mat` file. This function has two parameters—a file name and a dictionary containing variable names and values.

```
a = numpy.arange(7)
scipy.io.savemat("a.mat",
    {"array": a})
```

2. Load the `.mat` file.

Navigate to the directory where you created the file. Load the file, and check the array:

```
octave-3.4.0:2> load a.mat
octave-3.4.0:3> array
array =
```

```
0
1
2
3
4
5
6
```

The complete code for this recipe is as follows:

```
import numpy
import scipy.io

a = numpy.arange(7)
scipy.io.savemat("a.mat",
    {"array": a})
```

See also

- ▶ *Installing SciPy in Chapter 2*

Installing RPy2

R is a popular scripting language used for statistics and data analysis. **RPy2** is an interface between R and Python. We will install RPy2 in this recipe.

How to do it...

If you want to install RPy2, choose one of the following options:

- ▶ Installing with `pip` or `easy_install`.
RPy2 is available on PYPI, so we can install it with either of the following two commands:

```
easy_install rpy2
```

or

```
sudo pip install rpy2
```

- ▶ Installing from source.
We can install RPy2 from the source `tar.gz`:

```
tar -xzf <rpy2_package>.tar.gz  
cd <rpy2_package>  
python setup.py build install
```

Interfacing with R

RPy2 can only be used to call R from Python, and not the other way around. We will import some sample R datasets, and plot the data of one of them.

Getting ready

Install RPy2 if necessary. See the previous recipe.

How to do it...

Let's start by loading a sample R dataset.

1. Load a data set into an array.

Load the datasets with the RPy2 `importr` function. This function can import R packages. In this example, we will import the datasets R package. Create a NumPy array from the `mtcars` dataset:

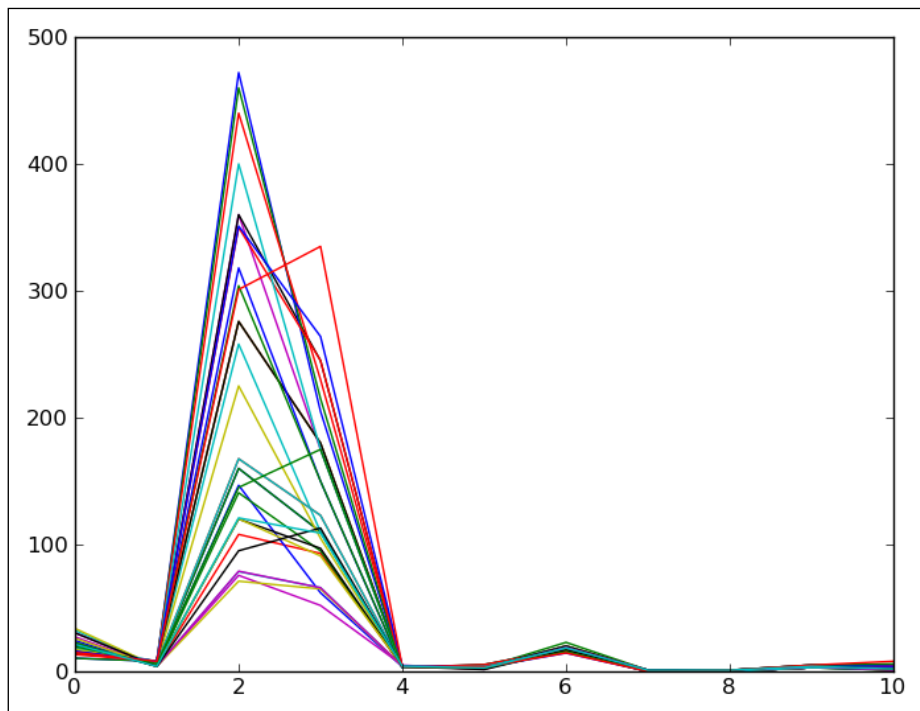
```
datasets = importr('datasets')
mtcars = numpy.array(datasets.mtcars)
```

2. Plot the dataset.

Plot the dataset with Matplotlib:

```
matplotlib.pyplot.plot(mtcars)
matplotlib.pyplot.show()
```

The following image shows the data, which is a two dimensional array:



The complete code for this recipe is as follows:

```
from rpy2.robjects.packages import importr
import numpy
import matplotlib.pyplot

datasets = importr('datasets')
mtcars = numpy.array(datasets.mtcars)

matplotlib.pyplot.plot(mtcars)
matplotlib.pyplot.show()
```

See also

- ▶ *Installing Matplotlib in Chapter 1, Winding Along with IPython*

Installing JPyype

Jython is the default interoperability solution for Python and Java. However, Jython runs on the Java Virtual Machine, and therefore, cannot access NumPy modules because they are mostly written in C. **JPyype** is an open-source project that tries to solve this problem. The interfacing occurs on the native level between the Python and Java virtual machines. Let's install JPyype.

How to do it...

Follow the ensuing steps for installing JPyype:

1. Download JPyype.
Download JPyype from <http://sourceforge.net/projects/jpyype/files/>.
2. Build JPyype.
Unpack JPyype and run the following command:

```
python setup.py install
```

Sending a NumPy array to JPyype

In this recipe, we will start a JVM and send a NumPy array to it. We will print the received array using standard Java calls. Obviously, you will need to have Java installed.

How to do it...

First, we need to start the JVM from JPype.

1. Start the JVM.

JPype is conveniently able to find the default JVM path:

```
jpype.startJVM(jpype.getDefaultJVMPath())
```

2. Print hello world.

Just because of tradition, let's print hello world:

```
jpype.java.lang.System.out.println("hello world")
```

3. Send a NumPy array.

Create a NumPy array, convert it to a Python list, and pass it to JPype. Now, it's trivial to print the array elements:

```
values = numpy.arange(7)
java_array = jpype.JArray
             (jpype.JDouble, 1)(values.tolist())
```

```
for item in java_array:
    jpype.java.lang.System.out.println(item)
```

4. Shutdown the JVM.

After we are done, we will shutdown the JVM:

```
jpype.shutdownJVM()
```

Only one JVM can run at a time in JPype. If we forget to shutdown the JVM, it could lead to unexpected errors. The program output is as follows:

```
hello world
0.0
1.0
2.0
3.0
4.0
5.0
6.0
JVM activity report      :
  classes loaded        : 31
JVM has been shutdown
```


The complete code for this recipe is as follows:

```
import jpyype
import numpy

#1. Start the JVM
jpyype.startJVM(jpyype.getDefaultJVMPath())

#2. Print hello world
jpyype.java.lang.System.out.println("hello world")

#3. Send a NumPy array
values = numpy.arange(7)
java_array = jpyype.JArray(jpyype.JDouble, 1)(values.tolist())

for item in java_array:
    jpyype.java.lang.System.out.println(item)

#4. Shutdown the JVM
jpyype.shutdownJVM()
```

How it works...

JPyype allows us to start up and shut down a Java Virtual Machine. It provides wrappers for standard Java API calls. As we saw in this example, we can pass Python lists to be transformed to Java arrays by the `JArray` wrapper. JPyype uses the **Java Native Interface (JNI)**, which is a bridge between native C code and Java. Unfortunately, using JNI hurts performance, so you have to be mindful of that fact.

See also

- ▶ *Installing JPyype* in this chapter

Installing Google App Engine

Google App Engine (GAE) enables you to build web applications in the Google cloud. Since 2012, there is official support for NumPy; you need to have a Google account to use GAE.

How to do it...

The first step is to download GAE.

1. Download GAE.

Download GAE for your operating system from <https://developers.google.com/appengine/downloads>.

From this page, you can download documentation and the GAE Eclipse plugin as well. If you are developing with Eclipse, you should definitely install it.

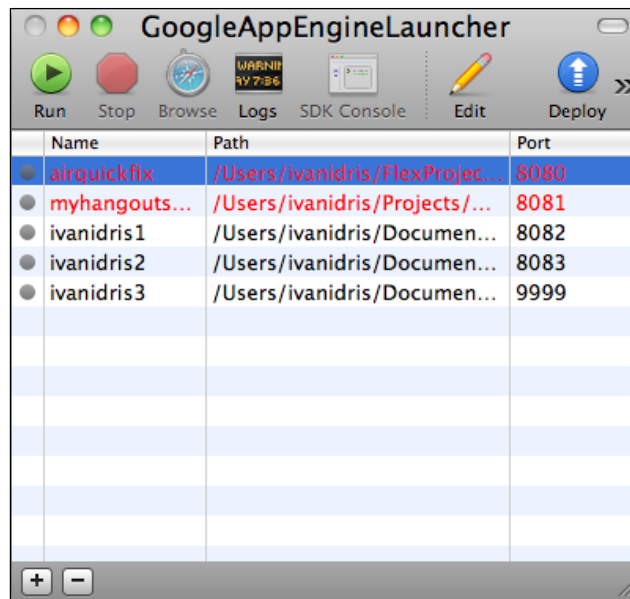
2. The development environment.

GAE comes with a development environment, which simulates the production cloud. GAE, at the time of writing, only supported officially Python 2.5 and 2.7. GAE will try to find Python on your system; however, it may be necessary to set that yourself, for instance, if you have multiple Python versions. You can set this setting in the **Preferences** dialog of the launcher application.

There are two important scripts in the SDK:

- `dev_appserver.py`: Development server
- `appcfg.py`: Deploys to the cloud

On Windows and Mac, there is a GAE launcher application. The launcher has **Run** and **Deploy** buttons that do the same as the aforementioned scripts:



Deploying NumPy code in the Google cloud

Deploying GAE applications is pretty easy. For NumPy an extra configuration step is required, but that will take only minutes.

How to do it...

Let's create a new application.

1. Create a new application.

Create a new application with the launcher (**File | New Application**). Name it `numpycloud`. This will create a folder with the same name containing the following files:

- `app.yaml`: YAML application configuration file
- `favicon.ico`: Icon image
- `index.yaml`: Auto generated file
- `main.py`: Main entry point for the web application

2. Add NumPy to the libraries.

First, we need to let GAE know that we want to use NumPy. Add the following lines to the `app.yaml` configuration file in the libraries section:

```
- name: NumPy
  version: "1.6.1"
```

The configuration file should have the following contents:

```
application: numpycloud
version: 1
runtime: python27
api_version: 1
threadsafe: yes

handlers:
- url: /favicon\.ico
  static_files: favicon.ico
  upload: favicon\.ico

- url: .*
  script: main.app

libraries:
- name: webapp2
  version: "2.5.1"
```

```
- name: numpy
  version: "1.6.1"
```

3. Write NumPy code.

To demonstrate that we can use NumPy code, let's modify the `main.py` file. There is a `MainHandler` class with a handler method for `get` requests. Replace this method with the following code:

```
def get(self):
    self.response.out.write
        ('Hello world!<br/>')
    self.response.out.write
        ('NumPy sum = ' + str
        (numpy.arange(7).sum()))
```

We will have the following code in the end:

```
import webapp2
import numpy

class MainHandler
(webapp2.RequestHandler):
    def get(self):
        self.response.out.write('Hello world!<br/>')
        self.response.out.write('NumPy sum = ' +
            str(numpy.arange(7).sum()))

app = webapp2.WSGIApplication([('/', MainHandler)],
                              debug=True)
```

If you click on the **Browse** button in the GAE launcher, you should see a web page in your default browser, with the following text:

```
Hello world!
NumPy sum = 21
```

How it works...

GAE is free depending on how much of the resources are used. You can create up to ten web applications. GAE takes the sandboxing approach, which means that NumPy was not available for a while, but now it is, as demonstrated in this recipe.

You should also be aware that GAE currently does not support relational databases. There are other features too, which might make portability a concern.

Running NumPy code in a Python Anywhere web console

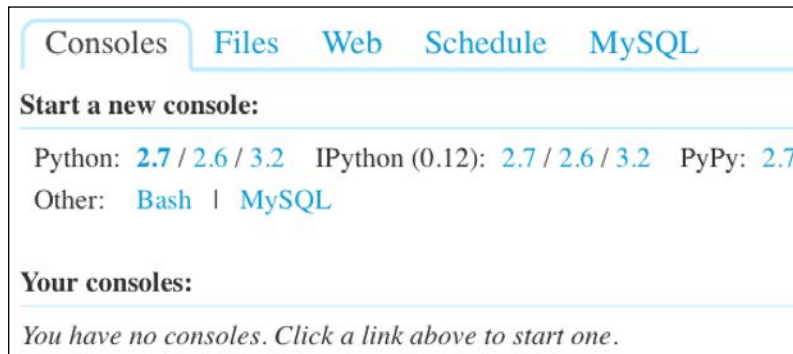
In *Chapter 1*, we already saw a Python Anywhere console in action, without having an account. This recipe will require you to have an account, but don't worry—it's free; at least if you don't need too many resources.

Signing up is a pretty straightforward process and will not be covered here. NumPy is already installed along with a long list of other Python software. For a complete list, see https://www.pythonanywhere.com/batteries_included/.

We will setup a simple script that gets price data from Google Finance every minute, and does simple statistics with the prices using NumPy.

How to do it...

Once we have signed up, we can login and have a look at the Python Anywhere dashboard:



1. Write the code.

The complete code for this example is as follows:

```
import urllib2
import re
import time
import sys
import numpy

prices = numpy.array([])

for i in xrange(3):
```

```
req = urllib2.Request
    ('http://finance.google.com/finance/
    info?client=ig&q=' + sys.argv[1])
req.add_header('User-agent',
    'Mozilla/5.0')
response = urllib2.urlopen(req)
page = response.read()
m = re.search('l_cur" : "(.*)"', page)
prices = numpy.append(prices,
    float(m.group(1)))
avg = prices.mean()
sigma = prices.std()

    evFactor = float(sys.argv[2])
bottom = avg - devFactor * sigma
top = avg + devFactor * sigma
timestr = time.strftime
    ("%H:%M:%S", time.gmtime())

print timestr, "Average", avg,
    "-Std", bottom, "+Std", top
time.sleep(60)
```

Most of it is standard Python, except the bits where we grow a NumPy array containing prices and calculate the mean and standard deviation of the prices. A URL is used to download price data in JSON format from Google Finance given a stock ticker such as AAPL. This URL could change, of course.

Next, we parse the JSON with regular expressions to extract a price. This price is added to a NumPy array. We compute the mean and standard deviation for the prices. The price is printed with a timestamp `bottom` and `top`, based on the standard deviation times some factor to be specified by us.

2. Upload the code.

After we are done with the code on our local machine, we can upload the script to Python Anywhere. Go to the dashboard, and click on the **Files** tab. Upload the script from the widget at the bottom of the page.

3. Run the code.

To run the code, click on the **Consoles** tab, and then click on the **Bash** link. Python Anywhere should create a bash console for us right now.

We can now run our program for AAPL with a one standard deviation band, as shown in the following screenshot:

```
18:32 ~/cookbook $ python avg_price.py AAPL 1
15:31:29 Average 667.24 -Std 667.24 +Std 667.24
15:32:29 Average 667.365 -Std 667.24 +Std 667.49
15:33:29 Average 667.376666667 -Std 667.273279584 +Std 667.480053749
```

How it works...

Python Anywhere is perfect if you want to run NumPy code on a remote server; especially, if you need your program to execute at scheduled times. For the free account, at least, it's not so convenient to do interactive work, since there is a certain lag whenever you enter text in the web console.

However, as we saw, it is possible to create and test a program locally, and upload it to Python Anywhere. This frees resources on your local machine as well. We can do fancy things such as sending emails based on the stock price, for instance, or schedule our scripts to be activated during trading hours. By the way, this is also possible with Google App Engine, but it is done the Google way; so you will need to learn about their API.

Setting up PiCloud

PiCloud is another cloud computing provider, which is actually using the EC2 Amazon infrastructure. However, they do offer environments with preinstalled Python software, including NumPy. These environments are just EC2 instances that we can ssh into. In this recipe, we will be using the Python 2.7—Ubuntu Natty 11.04 environment. For the installed packages in this environment, see http://www.picloud.com/docs/base_environment/2/installed/. PiCloud follows the freemium model, meaning that you can start out for free and pay later on if you require more resources.

How to do it...

After the mandatory sign up, log in to PiCloud.

1. Create an environment.

We start out without any environments. In order to create an environment, first click on the **Environments** tab. Next, click on the "**create a new environment**" button. Select the Python 2.7 base environment. Currently, you can choose between a Python 2.7 and 2.6 environments, both on Ubuntu.

Creating an environment takes a few minutes. When the environment is ready, you will receive an e-mail, and you should see something like the following screenshot in your browser:

| Environments being configured | | | | |
|---|---------------------------------|---------|--|--------------|
| name | description | version | status | action |
| py27UbuntuNatty1104 | Python 2.7 - Ubuntu Natty 11.04 | | ec2-107-20-19-178.compute-1.amazonaws.com In configurable mode for 00:46:04 | connect save |
| <input type="button" value="create new environment"/> | | | | |

2. Connect to the environment.

If we click on the **connect** link in the **action** column, we get a pop up with the following instructions:

```
chmod 400 privatekey.pem
ssh -i privatekey.pem picloud@yourserver.amazonaws.com
```

Download the private key using the link in the pop up. Follow the instructions further to connect to the environment.

3. Check the versions.

In order to prove that we can use NumPy, we will check the available version. We will do that for Matplotlib as well.

First, let's start an IPython shell. Recall from *Chapter 1* that the `pylab` switch allows us to auto-import several packages, including NumPy. Execute the following command:

```
ipython -pylab
```

NumPy and Matplotlib have a `__version__` attribute, which tells us the version. Print the version attributes:

```
In [1]: numpy.__version__
Out[1]: '1.6.1'
```

```
In [4]: matplotlib.__version__
Out[4]: '1.0.1'
```

How it works...

PiCloud gives us preconfigured EC2 Amazon instances with NumPy and other Python packages. This gives us direct access from a terminal. We can customize our environment and save it for later. A customized environment can be used as a template as well.

5

Audio and Image Processing

In this chapter, we will cover basic image and audio (WAV files) processing with NumPy and SciPy. We will use NumPy to do interesting things with sounds and images:

- ▶ Loading images into memory maps
- ▶ Combining images
- ▶ Blurring images
- ▶ Repeating audio fragments
- ▶ Generating sounds
- ▶ Designing an audio filter
- ▶ Edge detection with the Sobel filter

Introduction

This should be a fun chapter. Although all the chapters in this book are fun, in this chapter we are going to really go for it and concentrate on having fun. In *Chapter 10, Fun with Scikits*, you will find a few more image processing recipes that use scikits-image.

Unfortunately, this book does not have a direct support for audio files, so you will really need to run the code examples to appreciate the recipes fully. The source code should be available from the book website <http://www.packtpub.com/>.

Loading images into memory map

It is recommended to load large files into memory maps. Memory-mapped files only load a small part of large files. NumPy memory maps are array-like. In this example, we will generate an image of colored squares and load it into a memory map.

Getting ready

If necessary, install Matplotlib. The *See Also* section of this recipe has a reference to the corresponding recipe.

How to do it...

We will begin by initializing arrays.

1. First, we need to initialize the following arrays:
 - an array that holds the image data
 - an array with random coordinates of the centers of the squares
 - an array with random radii of the squares
 - an array with random colors of the squares

Initialize the arrays as follows:

```
img = numpy.zeros((N, N),
                  numpy.uint8)
centers = numpy.random.random_integers
        (0, N, size=(NSQUARES, 2))
radii = numpy.random.randint
       (0, N/9, size=NSQUARES)
colors = numpy.random.randint
       (100, 255, size=NSQUARES)
```

As you can see, we are initializing the first array to zeroes. The other arrays are initialized with functions from the `numpy.random` package that generate random integers.

2. Generate squares.

The next step is to generate squares. We create the squares using the arrays in the previous step. With the `clip` function, we will make sure that the squares do not wander outside the image area.

The `meshgrid` function gives us the coordinates of the squares. If we give this function two arrays with size N and M , it will give us two arrays of shape N by M . The first array will have its elements repeated along the x axis. The second array will have its elements repeated along the y axis. The following example of an IPython session should make this clearer:

```
In: x = linspace(1, 3, 3)
```

```
In: x
```

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

```
In: y = linspace(1, 2, 2)
```

```
In: y
```

```
Out: array([ 1.,  2.])
```

```
In: meshgrid(x, y)
```

```
Out:
```

```
[array([[ 1.,  2.,  3.],
        [ 1.,  2.,  3.]]) ,
 array([[ 1.,  1.,  1.],
        [ 2.,  2.,  2.]])]
```

Finally, we will set the colors of the squares:

```
for i in xrange(NSQUARES):
    xindices = range(centers[i][0] - radii[i], centers[i][0] +
                    radii[i])
    xindices = numpy.clip(xindices, 0, N - 1)
    yindices = range(centers[i][1] - radii[i], centers[i][1] +
                    radii[i])
    yindices = numpy.clip(yindices, 0, N - 1)

    if len(xindices) == 0 or len(yindices) == 0:
        continue

    coordinates = numpy.meshgrid(xindices, yindices)
    img[coordinates] = colors[i]
```

3. Load into memory map.

Before we load the image data into a memory map, we need to store it into a file with the `tofile` function. Then, we load the image data from this file into a memory map with the `memmap` function:

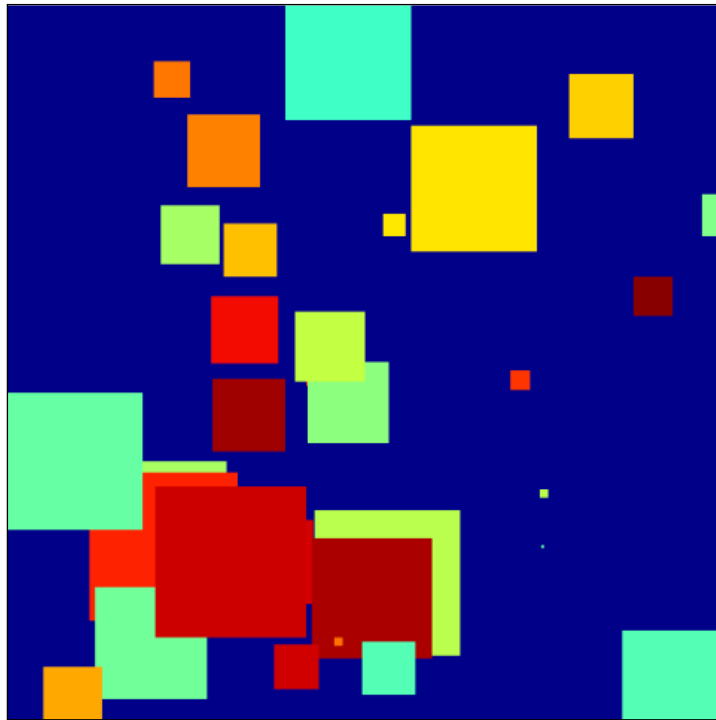
```
img.tofile('random_squares.raw')
img_memmap = numpy.memmap
    ('random_squares.raw', shape=img.shape)
```

4. Display the image.

To demonstrate that everything worked fine, we will display the image with Matplotlib:

```
matplotlib.pyplot.imshow(img_memmap)
matplotlib.pyplot.axis('off')
matplotlib.pyplot.show()
```

Notice that we are not displaying the axes. The following is an example of a generated image:



The following is the complete source code for this recipe:

```
import numpy
import matplotlib.pyplot
import sys

N = 512

if(len(sys.argv) != 2):
    print "Please input the number of squares to generate"
    sys.exit()

NSQUARES = int(sys.argv[1])

# Initialize
img = numpy.zeros((N, N), numpy.uint8)
centers = numpy.random.random_integers(0, N, size=(NSQUARES, 2))
radii = numpy.random.randint(0, N/9, size=NSQUARES)
colors = numpy.random.randint(100, 255, size=NSQUARES)

# Generate squares
for i in xrange(NSQUARES):
    xindices = range(centers[i][0] - radii[i], centers[i][0] +
radii[i])
    xindices = numpy.clip(xindices, 0, N - 1)
    yindices = range(centers[i][1] - radii[i], centers[i][1] +
radii[i])
    yindices = numpy.clip(yindices, 0, N - 1)

    if len(xindices) == 0 or len(yindices) == 0:
        continue

    coordinates = numpy.meshgrid(xindices, yindices)
    img[coordinates] = colors[i]

# Load into memory map
img.tofile('random_squares.raw')
img_memmap = numpy.memmap('random_squares.raw', shape=img.shape)

# Display image
matplotlib.pyplot.imshow(img_memmap)
matplotlib.pyplot.axis('off')
matplotlib.pyplot.show()
```

How it works...

We used the following functions in this recipe:

| Function | Description |
|------------------------------|--|
| <code>zeros</code> | Gives an array filled with zeroes. |
| <code>random_integers</code> | Returns an array with random integer values between a high and low bound. |
| <code>randint</code> | Synonym for <code>random_integers</code> . |
| <code>clip</code> | Clips values of an array, given a minimum and a maximum. |
| <code>meshgrid</code> | Returns coordinate arrays from an array containing x-coordinates, and an array containing y-coordinates. |
| <code>tofile</code> | Writes an array to a file. |
| <code>memmap</code> | Creates a NumPy memory map from a file given the name of a file. Optionally, you can specify the shape of the array. |
| <code>axis</code> | Matplotlib function that configures the plot axes. For instance, we can turn them off. |

See also

- ▶ The *Installing Matplotlib* recipe in *Chapter 1, Winding Along with IPython*

Combining images

In this recipe, we will combine the famous Mandelbrot fractal (for more information on Mandelbrot set visit http://en.wikipedia.org/wiki/Mandelbrot_set) and the image of Lena. These types of fractals are defined by a recursive formula, where you calculate the next complex number in a series by multiplying the current complex number you have, by itself and adding a constant to it.

Getting ready

Install SciPy, if necessary. The *See Also* section of this recipe, has a reference to the related recipe.

How to do it...

We will start by initializing the arrays, followed by generating and plotting the fractal, and finally, combining the fractal with the Lena image.

1. Initialize the arrays.

We will initialize `x`, `y`, and `z` arrays corresponding to the pixels in the image area with the `meshgrid`, `zeros`, and `linspace` functions:

```
x, y = numpy.meshgrid(numpy.linspace
    (x_min, x_max, SIZE),
    numpy.linspace(y_min, y_max, SIZE))
c = x + 1j * y
z = c.copy()
fractal = numpy.zeros
    (z.shape, dtype=numpy.uint8) +
    MAX_COLOR
```

2. Generate the fractal.

If `z` is a complex number, you have the following relation for a Mandelbrot fractal:

$$z_{n+1} = z_n^2 + c$$

In this equation, `c` is a constant complex number. This can be graphed in the complex plane with horizontal real values axis and vertical imaginary values axis. We will use the so-called "escape time algorithm" to draw the fractal.

The algorithm scans the points in a small region around the origin on a distance of about two. Each of these points is used as the `c` value, and is assigned a color based on the number of iterations it takes to escape the region. If it takes more than a predefined number of iterations to escape, the pixel gets the default background color. For more information see the Wikipedia article already mentioned in this recipe:

```
for n in range(ITERATIONS):
    print n
    mask = numpy.abs(z) <= 4
    z[mask] = z[mask] ** 2 + c[mask]
    fractal[(fractal == MAX_COLOR) &
        (-mask)] = (MAX_COLOR - 1) *
        n / ITERATIONS
```


3. Plot the fractal.

Plot the fractal with Matplotlib:

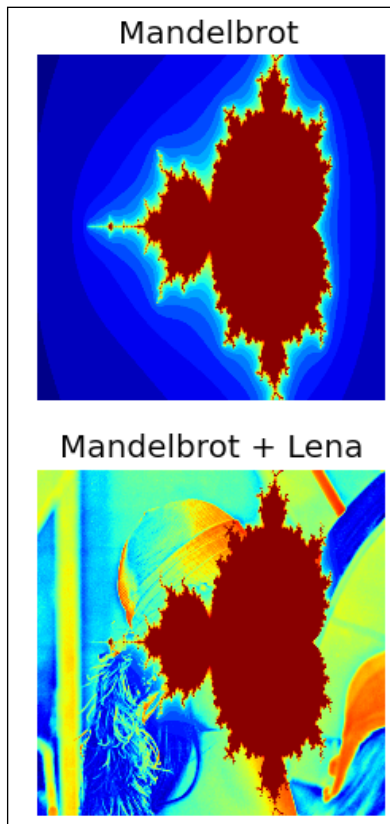
```
matplotlib.pyplot.subplot(211)
matplotlib.pyplot.imshow(fractal)
matplotlib.pyplot.title('Mandelbrot')
matplotlib.pyplot.axis('off')
```

4. Combine the fractal and Lena.

Use the `choose` function to pick value from the fractal or Lena array:

```
matplotlib.pyplot.subplot(212)
matplotlib.pyplot.imshow(numpy.choose
    (fractal < lena, [fractal, lena]))
matplotlib.pyplot.axis('off')
matplotlib.pyplot.title
    ('Mandelbrot + Lena')
```

The following is the resulting image:



The following is the complete code for this recipe:

```
import numpy
import matplotlib.pyplot
import sys
import scipy

if(len(sys.argv) != 2):
    print "Please input the number of
          iterations for the fractal"
    sys.exit()

ITERATIONS = int(sys.argv[1])
lena = scipy.misc.lena()
SIZE = lena.shape[0]
MAX_COLOR = 255.
x_min, x_max = -2.5, 1
y_min, y_max = -1, 1

# Initialize arrays
x, y = numpy.meshgrid(numpy.linspace
    (x_min, x_max, SIZE),
    numpy.linspace(y_min, y_max, SIZE))
c = x + 1j * y
z = c.copy()
fractal = numpy.zeros(z.shape,
    dtype=numpy.uint8) + MAX_COLOR

# Generate fractal
for n in range(ITERATIONS):
    print n
    mask = numpy.abs(z) <= 4
    z[mask] = z[mask] ** 2 + c[mask]
    fractal[(fractal == MAX_COLOR) &
        (-mask)] = (MAX_COLOR - 1) * n / ITERATIONS

# Display the fractal
matplotlib.pyplot.subplot(211)
matplotlib.pyplot.imshow(fractal)
matplotlib.pyplot.title('Mandelbrot')
matplotlib.pyplot.axis('off')

# Combine with lena
matplotlib.pyplot.subplot(212)
```

```
matplotlib.pyplot.imshow(numpy.choose
    (fractal < lena, [fractal, lena]))
matplotlib.pyplot.axis('off')
matplotlib.pyplot.title('Mandelbrot + Lena')

matplotlib.pyplot.show()
```

How it works...

The following functions were used in this example:

| Function | Description |
|-----------------------|--|
| <code>linspace</code> | Returns numbers within a range with a specified interval between them. |
| <code>choose</code> | Creates an array by choosing values from arrays based on a condition. |
| <code>meshgrid</code> | Returns coordinate arrays from an array containing x-coordinates, and an array containing y-coordinates. |

See also

- ▶ The *Installing Matplotlib* recipe in *Chapter 1, Winding Along with IPython*
- ▶ The *Installing SciPy* recipe in *Chapter 2, Advanced Indexing and Array Concepts*

Blurring images

We can blur images with a Gaussian filter (for more information on Gaussian filter visit http://en.wikipedia.org/wiki/Gaussian_filter). This filter is based on the normal distribution. A corresponding SciPy function requires the standard deviation as a parameter.

In this recipe, we will also plot a *polar rose* and a *spiral* (for more information on Polar coordinate system visit http://en.wikipedia.org/wiki/Polar_coordinate_system). These figures are not directly related, but it seemed more fun to combine them here.

How to do it...

We will start by initializing the polar plots, after which we will blur the Lena image and plot in the polar coordinates.

1. Initialization.

Initialize the polar plots as follows:

```
NFIGURES = int(sys.argv[1])
k = numpy.random.random_integers
    (1, 5, NFIGURES)
a = numpy.random.random_integers
    (1, 5, NFIGURES)
```

```
colors = ['b', 'g', 'r', 'c',
          'm', 'y', 'k']
```

2. Blur Lena.

In order to blur Lena, we will apply the Gaussian filter with standard deviation of four:

```
matplotlib.pyplot.subplot(212)
blurred = scipy.ndimage.gaussian_filter
    (lena, sigma=4)
```

```
matplotlib.pyplot.imshow(blurred)
matplotlib.pyplot.axis('off')
```

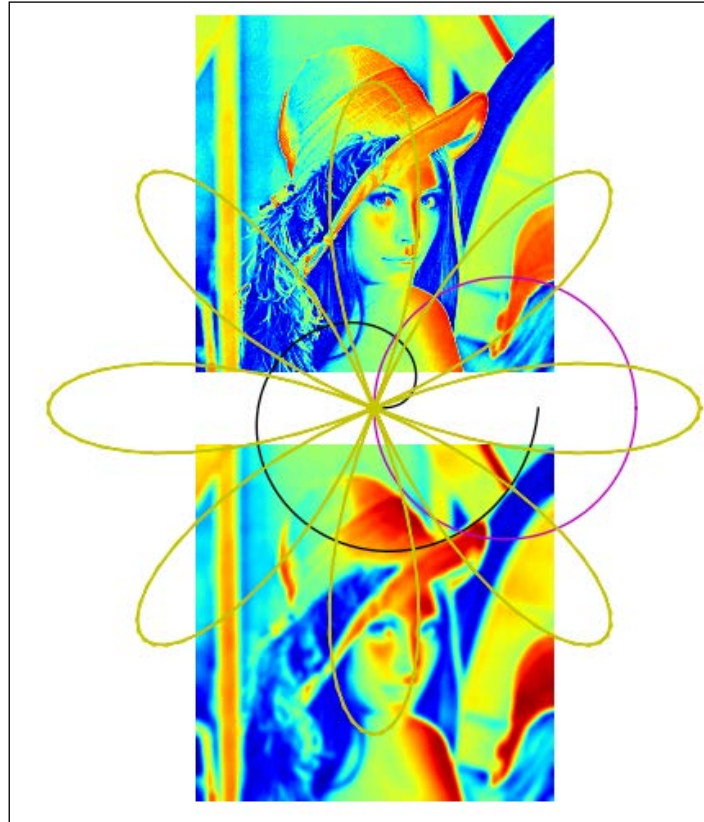
3. Plot in polar coordinates.

Matplotlib has a `polar` function, which plots in polar coordinates:

```
theta = numpy.linspace
    (0, k[0] * numpy.pi, 200)
matplotlib.pyplot.polar
    (theta, numpy.sqrt(theta), choice(colors))
```

```
for i in xrange(1, NFIGURES):
    theta = numpy.linspace
        (0, k[i] * numpy.pi, 200)
    matplotlib.pyplot.polar
        (theta, a[i] * numpy.cos(k[i] *
        theta), choice(colors))
```

The result will look like the following image:



The following is the complete code for this recipe:

```
import numpy
import matplotlib.pyplot
from random import choice
import sys
import scipy
import scipy.ndimage

# Initialization
NFIGURES = int(sys.argv[1])
k = numpy.random.random_integers
(1, 5, NFIGURES)
a = numpy.random.random_integers
(1, 5, NFIGURES)
```

```
colors = ['b', 'g', 'r', 'c',
          'm', 'y', 'k']

lena = scipy.misc.lena()
matplotlib.pyplot.subplot(211)
matplotlib.pyplot.imshow(lena)
matplotlib.pyplot.axis('off')

# Blur Lena
matplotlib.pyplot.subplot(212)
blurred = scipy.ndimage.gaussian_filter
          (lena, sigma=4)

matplotlib.pyplot.imshow(blurred)
matplotlib.pyplot.axis('off')

# Plot in polar coordinates
theta = numpy.linspace(0, k[0] *
                      numpy.pi, 200)
matplotlib.pyplot.polar(theta,
                       numpy.sqrt(theta), choice(colors))

for i in xrange(1, NFIGURES):
    theta = numpy.linspace(0, k[i] *
                          numpy.pi, 200)
    matplotlib.pyplot.polar(theta, a[i] *
                          numpy.cos(k[i] * theta), choice(colors))

matplotlib.pyplot.axis('off')

matplotlib.pyplot.show()
```

How it works...

We made use of the following functions in this tutorial:

| Function | Description |
|------------------------------|---|
| <code>gaussian_filter</code> | Applies a Gaussian filter. |
| <code>random_integers</code> | Returns an array with random integer values between a high and low bound. |
| <code>polar</code> | Plots a figure using polar coordinates. |

Repeating audio fragments

As we saw in *Chapter 2*, we can do neat things with WAV files. It's just a matter of downloading the file and loading it with SciPy. Let's download a WAV file and repeat it three times. We will skip some of the steps that we already saw in *Chapter 2*.

How to do it...

1. Repeating the audio fragment.

Although NumPy has a `repeat` function, in this case, it is more appropriate to use the `tile` function. The `repeat` function would have the effect of enlarging the array by repeating individual elements, and not repeating the contents of it.

The following IPython session should clarify the difference between these functions:

```
In: x = array([1, 2])
```

```
In: x
```

```
Out: array([1, 2])
```

```
In: repeat(x, 3)
```

```
Out: array([1, 1, 1, 2, 2, 2])
```

```
In: tile(x, 3)
```

```
Out: array([1, 2, 1, 2, 1, 2])
```

Now armed with this knowledge apply the `tile` function:

```
repeated = numpy.tile(data, int(sys.argv[1]))
```

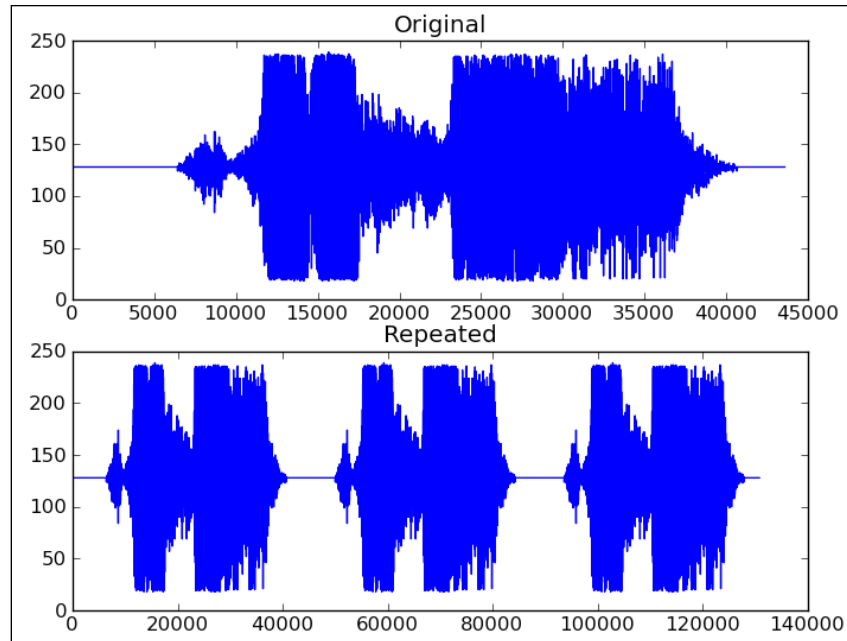
2. Plot the audio data.

We can plot the audio data with Matplotlib:

```
matplotlib.pyplot.title("Repeated")
```

```
matplotlib.pyplot.plot(repeated)
```

The original sound data and the repeated data plots are shown as follows:



The complete code for this recipe is as follows:

```
import scipy.io.wavfile
import matplotlib.pyplot
import urllib2
import numpy
import sys

response = urllib2.urlopen
    ('http://www.thesoundarchive.com/
    austinpowers/smashingbaby.wav')
print response.info()
WAV_FILE = 'smashingbaby.wav'
filehandle = open(WAV_FILE, 'w')
filehandle.write(response.read())
filehandle.close()
sample_rate, data =
    scipy.io.wavfile.read(WAV_FILE)
print "Data type", data.dtype,
    "Shape", data.shape

matplotlib.pyplot.subplot(2, 1, 1)
```



```

matplotlib.pyplot.title("Original")
matplotlib.pyplot.plot(data)

matplotlib.pyplot.subplot(2, 1, 2)

# Repeat the audio fragment
repeated = numpy.tile(data,
    int(sys.argv[1]))

# Plot the audio data
matplotlib.pyplot.title("Repeated")
matplotlib.pyplot.plot(repeated)
scipy.io.wavfile.write("repeated_yababy.wav",
    sample_rate, repeated)

matplotlib.pyplot.show()

```

How it works...

The following are the most important functions in this recipe:

| Function | Description |
|-------------------------------------|---|
| <code>scipy.io.wavfile.read</code> | Reads a WAV file into an array. |
| <code>numpy.tile</code> | Repeats an array a specified number of times. |
| <code>scipy.io.wavfile.write</code> | Creates a WAV file out of a NumPy array with a specified sample rate. |

Generating sounds

A sound can be represented mathematically by a sine wave, with a certain amplitude, frequency, and phase. We can randomly select frequencies from a list specified on Wikipedia at http://en.wikipedia.org/wiki/Piano_key_frequencies that complies with the following formula:

$$440 \cdot 2^{\frac{n-49}{12}}$$

The variable **n** in this formula is the number of the piano key. We will number the keys from 1 to 88. We will also select the amplitude, duration, and phase at random.

How to do it...

We will begin by initializing random values, then generate sine waves, compose a melody, and finally, plot the generated audio data with Matplotlib.

1. Initialization.

Initialize to random values:

- the amplitude between 200 to 2000,
- the duration to 0.01 to 0.2,
- the frequencies using the formula already mentioned
- the phase to values between 0 and 2 pi
- `NTONES = int(sys.argv[1])`

```
amps = 2000. * numpy.random.random
      ((NTONES,)) + 200.
durations = 0.19 * numpy.random.random
           ((NTONES,)) + 0.01
keys = numpy.random.random_integers
      (1, 88, NTONES)
freqs = 440.0 * 2 ** ((keys - 49.)/12.)
phi = 2 * numpy.pi * numpy.random.random
     ((NTONES,))
```

2. Generate sine waves.

Write a generate function to generate sine waves:

```
def generate(freq, amp, duration, phi):
    t = numpy.linspace
      (0, duration, duration * RATE)
    data = numpy.sin(2 * numpy.pi *
                    freq * t + phi) * amp

    return data.astype(DTYPE)
```

3. Compose.

Once we have generated a few tones, we only need to compose a coherent melody. For now, we will just concatenate the sine waves—this does not give a nice melody, but could serve as a starting point for more experimenting:

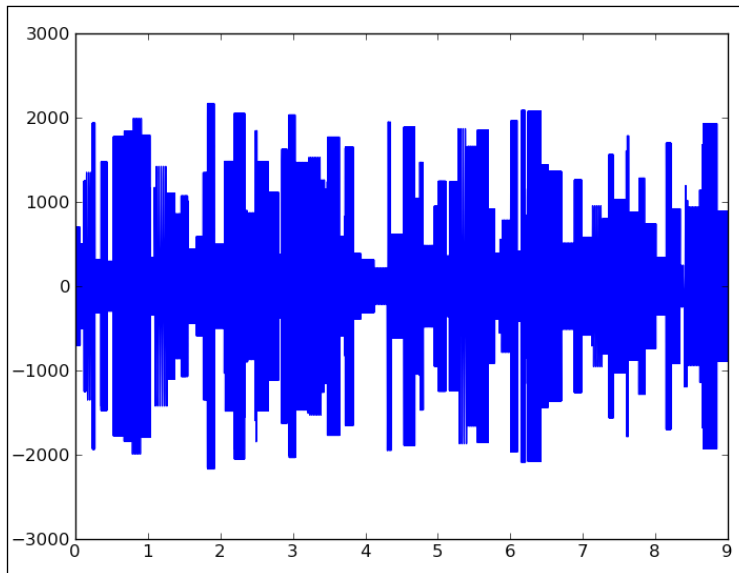
```
for i in xrange(NTONES):
    newtone = generate(freqs[i], amp=amps[i],
                      duration=durations[i], phi=phi[i])
    tone = numpy.concatenate((tone, newtone))
```

4. Plot the data.

Plot the generated audio data with Matplotlib:

```
matplotlib.pyplot.plot(numpy.linspace  
    (0, len(tone)/RATE, len(tone)), tone)  
matplotlib.pyplot.show()
```

The generated audio data plot is shown as follows:



The source code for this example is as follows:

```
import scipy.io.wavfile  
import numpy  
import sys  
import matplotlib.pyplot  
  
RATE = 44100  
DTYPE = numpy.int16  
  
# Generate sine wave  
def generate(freq, amp, duration, phi):  
    t = numpy.linspace  
        (0, duration, duration * RATE)  
    data = numpy.sin(2 * numpy.pi *  
        freq * t + phi) * amp
```

```

    return data.astype(DTYPE)

if len(sys.argv) != 2:
    print "Please input the
        number of tones to generate"
    sys.exit()

# Initialization
NTONES = int(sys.argv[1])
amps = 2000. * numpy.random.random
    ((NTONES,)) + 200.
durations = 0.19 * numpy.random.random
    ((NTONES,)) + 0.01
keys = numpy.random.random_integers
    (1, 88, NTONES)
freqs = 440.0 * 2 ** ((keys - 49.)/12.)
phi = 2 * numpy.pi * numpy.random.random
    ((NTONES,))

tone = numpy.array([], dtype=DTYPE)

# Compose
for i in xrange(NTONES):
    newtone = generate(freqs[i], amp=amps[i],
        duration=durations[i], phi=phi[i])
    tone = numpy.concatenate((tone, newtone))

scipy.io.wavfile.write
    ('generated_tone.wav', RATE, tone)

# Plot audio data
matplotlib.pyplot.plot
    (numpy.linspace(0, len(tone)/RATE,
        len(tone)), tone)
matplotlib.pyplot.show()

```

How it works...

We created a WAV file with randomly generated sounds. The `concatenate` function was used to concatenate sine waves.

Designing an audio filter

I remember learning in the *Analog Electronics* class about all types of filters. Then we actually constructed these filters. As you can imagine, it's much easier to make a filter in software than in hardware.

We will build a filter and apply it to an audio fragment that we will download. We have done some of these steps before in this chapter, so we will leave out those parts.

How to do it...

The `iirdesign` function, as its name suggests, allows us to construct several types of analog and digital filters. It can be found in the `scipy.signal` module. This module contains a comprehensive list of signal processing functions.

1. Design the filter.

Design the filter with `iirdesign` function of the `scipy.signal` module.

IIR stands for **infinite impulse response**; for more information visit Wikipedia at http://en.wikipedia.org/wiki/Infinite_impulse_response. We are not going to go into all the details of the `iirdesign` function. Have a look at the documentation at <http://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.iirdesign.html>, if necessary.

In short, we will set the following parameters:

- Frequencies normalized from 0 to 1
- Maximum loss
- Minimum attenuation
- Filter type

```
b,a = scipy.signal.iirdesign
      (wp=0.2, ws=0.1, gstop=60,
       gpass=1, ftype='butter')
```

The configuration of this filter corresponds to a *Butterworth bandpass filter* (for more information on Butterworth bandpass filter visit http://en.wikipedia.org/wiki/Butterworth_filter).

2. Apply filter.

We can apply the filter with the `scipy.signal.lfilter` function. It accepts as arguments the values from the previous step and of course, the data array to filter:

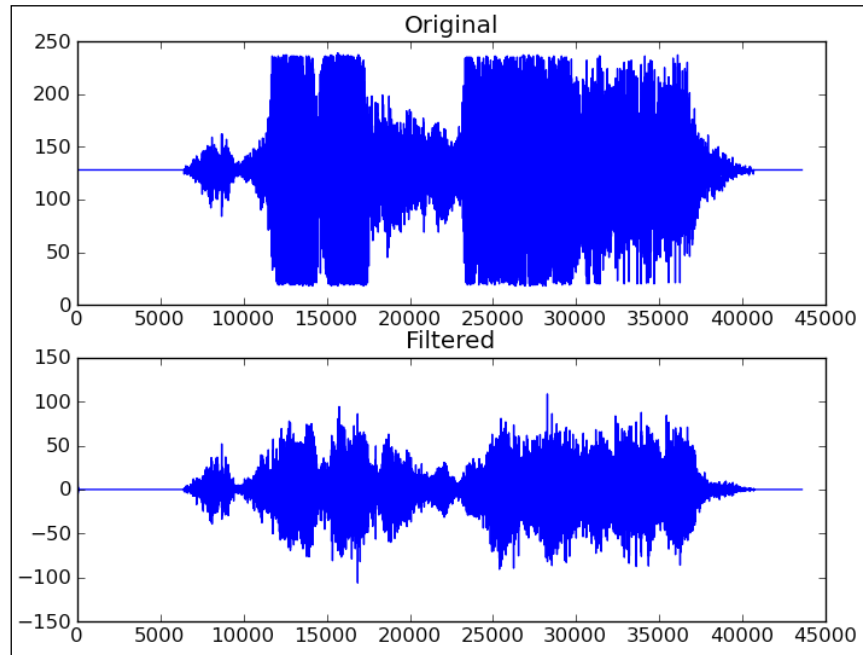
```
filtered = scipy.signal.lfilter(b, a, data)
```

3. Write the new audio file.

When writing the new audio file, we need to make sure that it is of the same data type as the original data array:

```
scipy.io.wavfile.write('filtered.wav',
    sample_rate, filtered.astype(data.dtype))
```

After plotting the original and filtered data, we get the following plot:



The code for the audio filter is as follows:

```
import scipy.io.wavfile
import matplotlib.pyplot
import urllib2
import scipy.signal

response = urllib2.urlopen
('http://www.thesoundarchive.com/austinpowers/
smashingbaby.wav')
print response.info()
WAV_FILE = 'smashingbaby.wav'
filehandle = open(WAV_FILE, 'w')
filehandle.write(response.read())
filehandle.close()
```

```
sample_rate, data = scipy.io.wavfile.read(WAV_FILE)
print "Data type", data.dtype, "Shape", data.shape

matplotlib.pyplot.subplot(2, 1, 1)
matplotlib.pyplot.title("Original")
matplotlib.pyplot.plot(data)

# Design the filter
b,a = scipy.signal.iirdesign
    (wp=0.2, ws=0.1, gstop=60, gpass=1, ftype='butter')

# Filter
filtered = scipy.signal.lfilter(b, a, data)

# Plot filtered data
matplotlib.pyplot.subplot(2, 1, 2)
matplotlib.pyplot.title("Filtered")
matplotlib.pyplot.plot(filtered)

scipy.io.wavfile.write('filtered.wav', sample_rate,
    filtered.astype(data.dtype))

matplotlib.pyplot.show()
```

How it works...

We created and applied a Butterworth bandpass filter. The following functions were introduced to create the filter:

| Function | Description |
|-------------------------------------|--|
| <code>scipy.signal.iirdesign</code> | Creates an IIR digital or analog filter. This function has an extensive parameter list, which is documented at http://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.iirdesign.html . |
| <code>scipy.signal.lfilter</code> | Filters an array given a digital filter. |

Edge detection with the Sobel filter

The *Sobel operator* (for more information on Sobel operator visit http://en.wikipedia.org/wiki/Sobel_operator) can be used for edge detection in images. The edge detection is based on performing a discrete differentiation on the image intensity. Because an image is two-dimensional, the gradient also has two components; unless we limit ourselves to one dimension, of course. We will apply the Sobel filter to the picture of Lena Soderberg.

How to do it...

In this section, we will learn how to apply the Sobel filter to detect edges in the Lena image.

1. Apply the Sobel filter in the x direction.

To apply the Sobel filter in the x direction, we need to set the `axis` parameter to 0:

```
sobelx = scipy.ndimage.sobel  
(lena, axis=0, mode='constant')
```

2. Apply the Sobel filter in the y direction.

To apply the Sobel filter in the y direction, we need to set the `axis` parameter to 1:

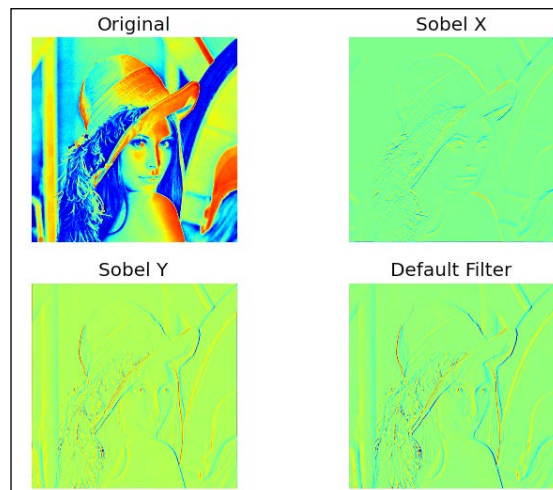
```
sobely = scipy.ndimage.sobel  
(lena, axis=1, mode='constant')
```

3. Apply the default Sobel filter.

The default Sobel filter only requires the input array:

```
default = scipy.ndimage.sobel(lena)
```

The original and resulting image plots showing edge detection with the Sobel filter:



The complete edge detection code is as follows:

```
import scipy
import scipy.ndimage
import matplotlib.pyplot

lena = scipy.misc.lena()

matplotlib.pyplot.subplot(221)
matplotlib.pyplot.imshow(lena)
matplotlib.pyplot.title('Original')
matplotlib.pyplot.axis('off')

# Sobel X filter
sobelx = scipy.ndimage.sobel
        (lena, axis=0, mode='constant')

matplotlib.pyplot.subplot(222)
matplotlib.pyplot.imshow(sobelx)
matplotlib.pyplot.title('Sobel X')
matplotlib.pyplot.axis('off')

# Sobel Y filter
sobely = scipy.ndimage.sobel
        (lena, axis=1, mode='constant')

matplotlib.pyplot.subplot(223)
matplotlib.pyplot.imshow(sobely)
matplotlib.pyplot.title('Sobel Y')
matplotlib.pyplot.axis('off')

# Default Sobel filter
default = scipy.ndimage.sobel(lena)

matplotlib.pyplot.subplot(224)
matplotlib.pyplot.imshow(default)
matplotlib.pyplot.title('Default Filter')
matplotlib.pyplot.axis('off')

matplotlib.pyplot.show()
```

How it works...

We applied the Sobel filter to the picture of the famous Playboy model Lena Soderberg. As we saw in this example, we can specify the axis along which to do the computation. The default setting is axis independent.

6

Special Arrays and Universal Functions

In this chapter, we will cover the following topics:

- ▶ Creating a universal function
- ▶ Finding Pythagorean triples
- ▶ Performing string operations with `chararray`
- ▶ Creating a masked array
- ▶ Ignoring negative and extreme values
- ▶ Creating a scores table with `recarray`

Introduction

This chapter is about special arrays and universal functions. These are topics that you may not encounter every day, but are still important enough to mention here. **Universal functions (Ufuncs)** work on arrays, element-by-element, or on scalars. Ufuncs accept a set of scalars as input, and produce a set of scalars as output. Universal functions can typically be mapped to mathematical counterparts, such as `add`, `subtract`, `divide`, `multiply`, and so on. The special arrays mentioned here, are all subclassed from the basic NumPy array object, and offer additional functionality.

Creating a universal function

We can create a universal function from a Python function with the NumPy `frompyfunc` function.

How to do it...

The following steps let us create a universal function:

1. Define the Python function.

Let's define a simple Python function that just doubles the input:

```
def double(a):  
    return 2 * a
```

2. Create the universal function.

Create the universal function with `frompyfunc`. We need to specify the number of input arguments and the number of objects returned:

```
import numpy  
  
def double(a):  
    return 2 * a  
  
ufunc = numpy.frompyfunc(double, 1, 1)  
print "Result", ufunc(numpy.arange(4))
```

The code prints the following output, when executed:

```
Result [0 2 4 6]
```

How it works...

We defined a Python function, which doubles the numbers it receives. Actually, we could also have strings as input, because that is legal in Python. We created a universal function from this Python function with the NumPy `frompyfunc` function.

Finding Pythagorean triples

For this tutorial you may need to read the Wikipedia page about **Pythagorean triple** (for more information on Pythagorean triple visit http://en.wikipedia.org/wiki/Pythagorean_triple). Pythagorean triples are closely related to the Pythagorean Theorem, which you probably have learned about in high school geometry.

Pythagorean triples represent the three sides of a right triangle, and therefore, obey the Pythagorean Theorem. Let's find the Pythagorean triple that has a components sum of 1000. We will do this using Euclid's formula:

$$a = m^2 - n^2, b = 2mn, c = m^2 + n^2$$

In this example we will see some universal functions in action.

How to do it...

The Euclid's formula defines indices m and n .

1. Create m and n arrays.

We will create arrays to hold these indices:

```
m = numpy.arange(33)
n = numpy.arange(33)
```

2. Calculate a , b , and c of the Pythagorean triple.

The second step is to calculate a , b , and c of the Pythagorean triple using Euclid's formula. Use the `outer` function to get the Cartesian products, difference, and sums we require:

```
a = numpy.subtract.outer(m ** 2, n ** 2)
b = 2 * numpy.multiply.outer(m, n)
c = numpy.add.outer(m ** 2, n ** 2)
```

3. Find the index.

At this point, we have a number of arrays containing a , b and c values. However, we still need to find the values that conform to the problem's condition. Find the index of those values with the NumPy `where` function:

```
idx = numpy.where((a + b + c) == 1000)
```

4. Check the solution.

Check the solution with the `numpy.testing` module:

```
numpy.testing.assert_equal
(a[idx]**2 + b[idx]**2, c[idx]**2)
```

The following is the complete code:

```
import numpy
import numpy.testing

#A Pythagorean triplet is a set of three natural numbers, a < b < c,
for which,
#a ** 2 + b ** 2 = c ** 2
#
#For example, 32 + 42 = 9 + 16 = 25 = 52.
#
```

```
#There exists exactly one Pythagorean triplet for which a + b + c =
1000.
#Find the product abc.

#1. Create m and n arrays
m = numpy.arange(33)
n = numpy.arange(33)

#2. Calculate a, b and c
a = numpy.subtract.outer(m ** 2, n ** 2)
b = 2 * numpy.multiply.outer(m, n)
c = numpy.add.outer(m ** 2, n ** 2)

#3. Find the index
idx = numpy.where((a + b + c) == 1000)

#4. Check solution
numpy.testing.assert_equal
(a[idx]**2 + b[idx]**2, c[idx]**2)
print a[idx] * b[idx] * c[idx]
```

How it works...

Universal functions are not really functions, but objects representing functions. Ufuncs have the `outer` method, which we saw in action just now. Many of the NumPy standard universal functions are implemented in C, and are, therefore, faster than regular Python code. Ufuncs support element-by-element processing and type casting, which in practice means less loops.

Performing string operations with chararray

NumPy has a specialized `chararray` object, which can hold strings. It is a subclass of `ndarray`, and has special string methods. We will download a text from the Python website and use those methods. The advantages of `chararray` over a normal array of strings are as follows:

- ▶ Whitespace of array elements is automatically trimmed on indexing
- ▶ Whitespace at the ends of strings is also trimmed by comparison operators
- ▶ Vectorized string operations are available, so loops are not needed

How to do it...

Let's create the character array.

1. Create the character array.

We can create the character array as a view:

```
carray = numpy.array(html).view(numpy.chararray)
```

2. Expand tabs to spaces.

Expand tabs to spaces with the `expandtabs` function. This function accepts the tab size as argument. The value is 8, if not specified:

```
carray = carray.expandtabs(1)
```

3. Split lines.

The `splitlines` function can split a string into separate lines:

```
carray = carray.splitlines()
```

The following is the complete code for this example:

```
import urllib2
import numpy
import re

response = urllib2.urlopen('http://python.org/')
html = response.read()
html = re.sub(r'<.*?>', '', html)
carray = numpy.array(html).view(numpy.chararray)
carray = carray.expandtabs(1)
carray = carray.splitlines()
print carray
```

How it works...

In this example, we saw the specialized `chararray` class in action. It offers several vectorized string operations and convenient behavior regarding whitespace.

Creating a masked array

Masked arrays can be used to ignore missing or invalid data items. A `MaskedArray` function from the `numpy.ma` module is a subclass of `ndarray`, with a mask. In this recipe, we will use the Lena Soderberg image as data source, and pretend that some of this data is corrupt. At the end, we will plot the original image, log values of the original image, the masked array, and log values thereof.

How to do it...

Let's create the masked array.

1. Create the masked array.

In order to create a masked array, we need to specify a mask. Let's create a random mask. This mask has values, which are either zero or one:

```
random_mask = numpy.random.randint  
(0, 2, size=lena.shape)
```

2. Create a masked array.

Using the mask in the previous step, create a masked array:

```
masked_array = numpy.ma.array  
(lena, mask=random_mask)
```

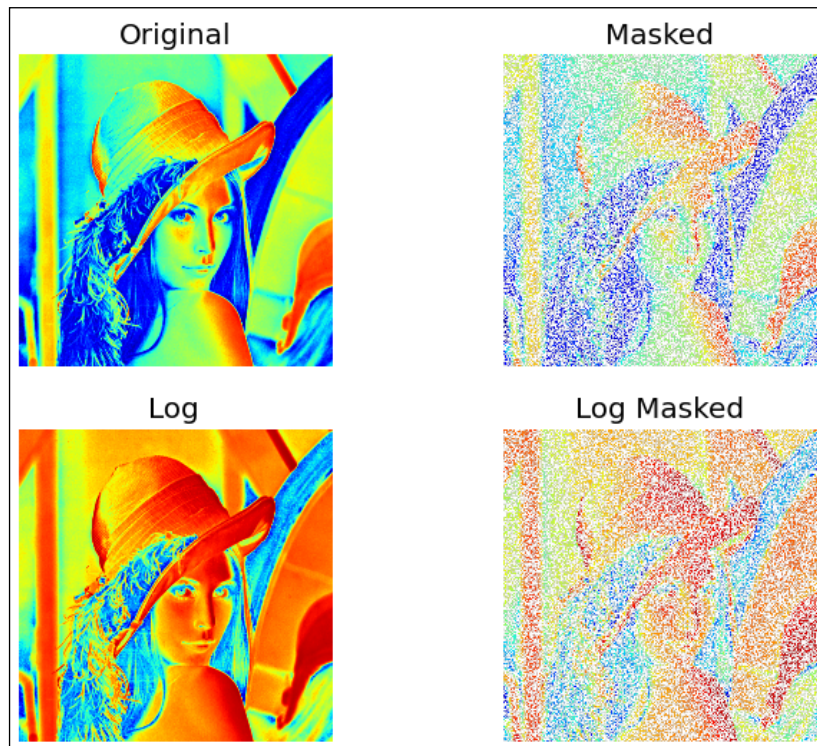
The following is the complete code for this masked array tutorial:

```
import numpy  
import scipy  
import matplotlib.pyplot  
  
lena = scipy.misc.lena()  
random_mask = numpy.random.randint  
(0, 2, size=lena.shape)  
  
matplotlib.pyplot.subplot(221)  
matplotlib.pyplot.title("Original")  
matplotlib.pyplot.imshow(lena)  
matplotlib.pyplot.axis('off')  
  
masked_array = numpy.ma.array  
(lena, mask=random_mask)  
print masked_array  
  
matplotlib.pyplot.subplot(222)  
matplotlib.pyplot.title("Masked")  
matplotlib.pyplot.imshow(masked_array)  
matplotlib.pyplot.axis('off')  
  
matplotlib.pyplot.subplot(223)  
matplotlib.pyplot.title("Log")  
matplotlib.pyplot.imshow(numpy.log(lena))  
matplotlib.pyplot.axis('off')
```

```
matplotlib.pyplot.subplot(224)
matplotlib.pyplot.title("Log Masked")
matplotlib.pyplot.imshow(numpy.log(masked_array))
matplotlib.pyplot.axis('off')

matplotlib.pyplot.show()
```

The resulting images are shown in the following screenshot:



How it works...

We applied a random mask to NumPy arrays. This had the effect of ignoring the data corresponding to the mask. There is a whole range of masked array operations to be found in the `numpy.ma` module. In this tutorial, we only demonstrated how to create a masked array.

Ignoring negative and extreme values

Masked arrays are useful when we want to ignore negative values, for instance, when taking the logarithm of array values. Another use case for masked arrays is excluding extreme values. This works based on an upper and lower bound for extreme values.

In this tutorial, we will apply these techniques to stock price data. We will skip the steps for downloading data, as they are repeated in previous chapters.

How to do it...

We will take the logarithm of an array that contains negative numbers.

1. Take the logarithm of negative numbers.

First, let's create an array containing numbers divisible by three:

```
triples = numpy.arange(0, len(close), 3)
print "Triples", triples[:10], "..."
```

Next, we will create an array with the ones that have the same size as the price data array:

```
signs = numpy.ones(len(close))
print "Signs", signs[:10], "..."
```

We will set each third number to be negative, with the help of indexing tricks we learned about in *Chapter 2, Advanced Indexing and Array Concepts*.

```
signs[triples] = -1
print "Signs", signs[:10], "..."
```

Finally, we will take the logarithm of this array:

```
ma_log = numpy.ma.log(close * signs)
print "Masked logs", ma_log[:10], "..."
```

This should print the following output for AAPL:

```

Triples [ 0  3  6  9 12 15 18 21 24 27] ...
Signs [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.] ...
Signs [-1.  1.  1. -1.  1.  1. -1.  1.  1. -1.] ...
Masked logs [-- 5.93655586575 5.95094223368 -- 5.97468290742
5.97510711452 --
6.01674381162 5.97889061623 --] ...
```

2. Ignoring extreme values.

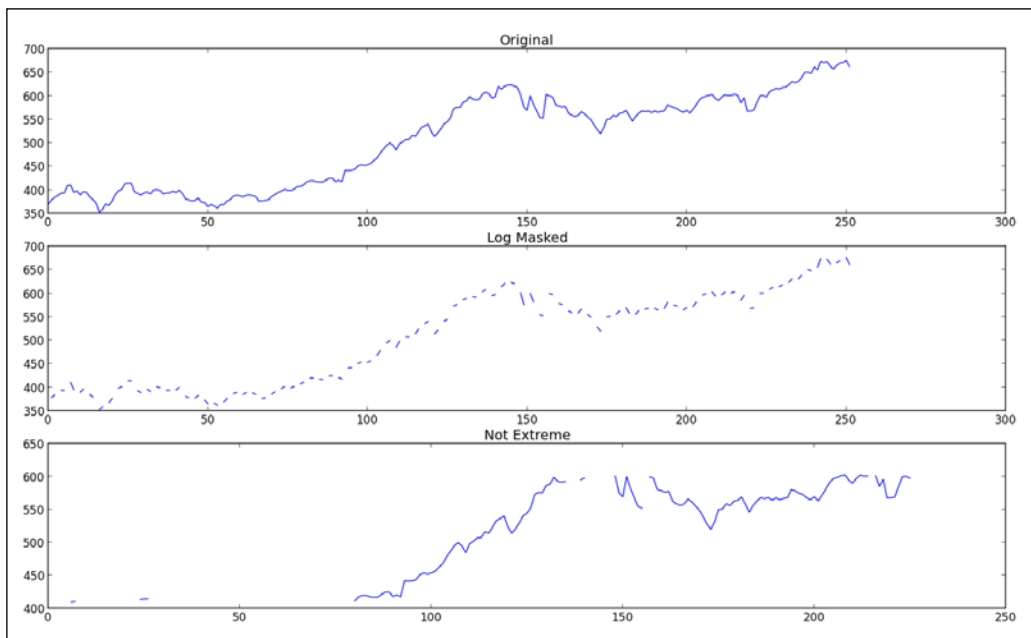
Let's define extreme values as being below one standard deviation of the mean, or one standard deviation above the mean. This definition leads us to write the following code, which will mask extreme values:

```
dev = close.std()
avg = close.mean()
inside = numpy.ma.masked_outside
    (close, avg - dev, avg + dev)
print "Inside", inside[:10], "..."
```

This code prints for the first ten elements:

```
Inside [-- -- -- -- -- 409.429675172
        410.240597855 -- --] ...
```

Let's plot the original price data, the data after taking the logarithm, and the exponent back again, and finally the data after applying the standard deviation based mask. The result will be as shown in the following screenshot:



The complete program for this tutorial is as follows:

```
import numpy
from matplotlib.finance
import quotes_historical_yahoo
from datetime import date
```

```
import sys
import matplotlib.pyplot

def get_close(ticker):
    today = date.today()
    start = (today.year - 1,
            today.month, today.day)

    quotes = quotes_historical_yahoo
            (ticker, start, today)

    return numpy.array(
            [q[4] for q in quotes])

close = get_close(sys.argv[1])

triples = numpy.arange(0, len(close), 3)
print "Triples", triples[:10], "...

signs = numpy.ones(len(close))
print "Signs", signs[:10], "...

signs[triples] = -1
print "Signs", signs[:10], "...

ma_log = numpy.ma.log(close * signs)
print "Masked logs", ma_log[:10], "...

dev = close.std()
avg = close.mean()
inside = numpy.ma.masked_outside
            (close, avg - dev, avg + dev)
print "Inside", inside[:10], "...

matplotlib.pyplot.subplot(311)
matplotlib.pyplot.title("Original")
matplotlib.pyplot.plot(close)

matplotlib.pyplot.subplot(312)
matplotlib.pyplot.title("Log Masked")
matplotlib.pyplot.plot(numpy.exp(ma_log))

matplotlib.pyplot.subplot(313)
matplotlib.pyplot.title("Not Extreme")
matplotlib.pyplot.plot(inside)

matplotlib.pyplot.show()
```

How it works...

Functions in the `numpy.ma` module mask array elements, which we regard as illegal. For instance, negative values are not allowed for the `log` and `sqrt` functions. A masked value is like a `NULL` value in databases and programming. All operations with a masked value result in a masked value.

Creating a scores table with recarray

The `recarray` class is a subclass of `ndarray`. They can hold records like in a database, with different data types. For instance, we can store records about employees, containing numerical data such as salary, and strings such as the employee name.

Modern economic theory tells us that an investing boils down to optimizing risk and reward. Risk is represented by the *standard deviation of log returns* (for more information on Arithmetic and logarithmic return visit http://en.wikipedia.org/wiki/Rate_of_return#Arithmetic_and_logarithmic_return). Reward on the other hand, is represented by the *average of log returns*. We can come up with a relative score, where a high score means low risk and high reward. We will calculate the scores for several stocks and store them together with the stock symbol using a table format in a NumPy `recarray` function.

How to do it...

We will start by creating the record array.

1. Create the record array.

Create a record array with for each record a symbol, standard deviation score, mean score, and overall score:

```
weights = numpy.recarray((len(tickers),),
                        dtype=[('symbol', numpy.str_, 16),
                              ('stdscore', float), ('mean', float),
                              ('score', float)])
```

2. Initialize the scores.

To keep things simple, we will initialize the scores in a loop based on the log returns:

```
for i in xrange(len(tickers)):
    close = get_close(tickers[i])
    logrets = numpy.diff
        (numpy.log(close))
    weights[i]['symbol'] =
        tickers[i]
    weights[i]['mean'] =
        logrets.mean()
```

```
weights[i]['stdscore'] =
    1/logrets.std()
weights[i]['score'] = 0
```

As you can see, we can access elements using the field names we defined in the previous step.

3. Normalize the scores.

We now have some numbers, but they are not easy to compare with each other. Normalize the scores, so that we can combine them later. Here, normalizing means making sure that the scores add up to one:

```
for key in ['mean', 'stdscore']:
    wsum = weights[key].sum()
    weights[key] = weights[key]/wsum
```

4. Calculate the overall score and sort it.

The overall score will just be the average of the intermediate scores. Sort the records on the overall score to produce a ranking:

```
weights['score'] = (weights
    ['stdscore'] + weights['mean'])/2
weights['score'].sort()
```

The following is the complete code for this example:

```
import numpy
from matplotlib.finance
import quotes_historical_yahoo
from datetime import date

# DJIA stock with div yield > 4 %
tickers = ['MRK', 'T', 'VZ']

def get_close(ticker):
    today = date.today()
    start = (today.year - 1,
        today.month, today.day)

    quotes = quotes_historical_yahoo
        (ticker, start, today)

    return numpy.array
        ([q[4] for q in quotes])
```

```

weights = numpy.recarray((len(tickers)),
    dtype=[('symbol', numpy.str_, 16),
    ('stdscore', float), ('mean', float),
    ('score', float)])

for i in xrange(len(tickers)):
    close = get_close(tickers[i])
    logrets = numpy.diff(numpy.log(close))
    weights[i]['symbol'] = tickers[i]
    weights[i]['mean'] = logrets.mean()
    weights[i]['stdscore'] = 1/logrets.std()
    weights[i]['score'] = 0

for key in ['mean', 'stdscore']:
    wsum = weights[key].sum()
    weights[key] = weights[key]/wsum

weights['score'] = (weights['stdscore'] +
    weights['mean'])/2
weights['score'].sort()

for record in weights:
    print "%s,mean=%.4f,stdscore=%.4f,
    score=%.4f" % (record['symbol'],
    record['mean'], record['stdscore'],
    record['score'])

```

This program produces the following output:

```

MRK,mean=0.1862,stdscore=0.2886,score=0.2374
T,mean=0.3570,stdscore=0.3556,score=0.3563
VZ,mean=0.4569,stdscore=0.3557,score=0.4063

```

How it works...

We computed scores for several stocks, and stored them in a NumPy `recarray` object. This array enables us to mix data of different data types, in this case, stock symbols and numerical scores. Record arrays allow us to access fields as array members, for example, `arr.field`. This tutorial covered the creation of a record array. More record array related functions can be found in the `numpy.recarray` module.

7

Profiling and Debugging

In this chapter, we will cover:

- ▶ Profiling with `timeit`
- ▶ Profiling with IPython
- ▶ Installing `line_profiler`
- ▶ Profiling code with `line_profiler`
- ▶ Profiling code with the `cProfile` extension
- ▶ Debugging with IPython
- ▶ Debugging with `puddb`

Introduction

Debugging is the act of finding and removing bugs in software. Profiling is about building a profile of a software program in order to collect information about memory usage or time complexity. Profiling and debugging are activities that are an integral part of the life of a developer. This is especially true for non-trivial software. The good news is that there are lots of tools to help you. We will review a number of techniques that are popular amongst NumPy users.

Profiling with `timeit`

`timeit` is a module that allows you to time pieces of code. It is part of the standard Python library. We will time the NumPy `sort` function with several different array sizes. The classic *quicksort* and *merge sort* algorithms have an average running time of **$O(n \log n)$** ; so we will try to fit our result to such a model.

How to do it...

We will require arrays to sort.

1. Create arrays to sort.

We will create arrays of varying sizes containing random integer values:

```
times = numpy.array([])

for size in sizes:
    integers = numpy.random.random_integers
        (1, 10 ** 6, size)
```

2. Measure time.

In order to measure time, we need to create a timer and give it a function to execute and specify the relevant imports. After that, sort 100 times to get some data about the sorting times:

```
def measure():
    timer = timeit.Timer('dosort()',
        'from __main__ import dosort')

    return timer.timeit(10 ** 2)
```

3. Build measurement time arrays.

Build the measurement time arrays by appending times one by one:

```
times = numpy.append
    (times, measure())
```

4. Fit to $n \log n$.

Fit the times to the theoretical model of $n \log n$. Because we are varying the array size as powers of two, this is easy:

```
fit = numpy.polyfit(sizes * powersOf2,
    times, 1)
```

The following is the complete timing code:

```
import numpy
import timeit
import matplotlib.pyplot

# This program measures the performance of the NumPy sort function
# and plots time vs array size.
integers = []
def dosort():
    integers.sort()

def measure():
    timer = timeit.Timer('dosort()', '
        from __main__ import dosort')

    return timer.timeit(10 ** 2)

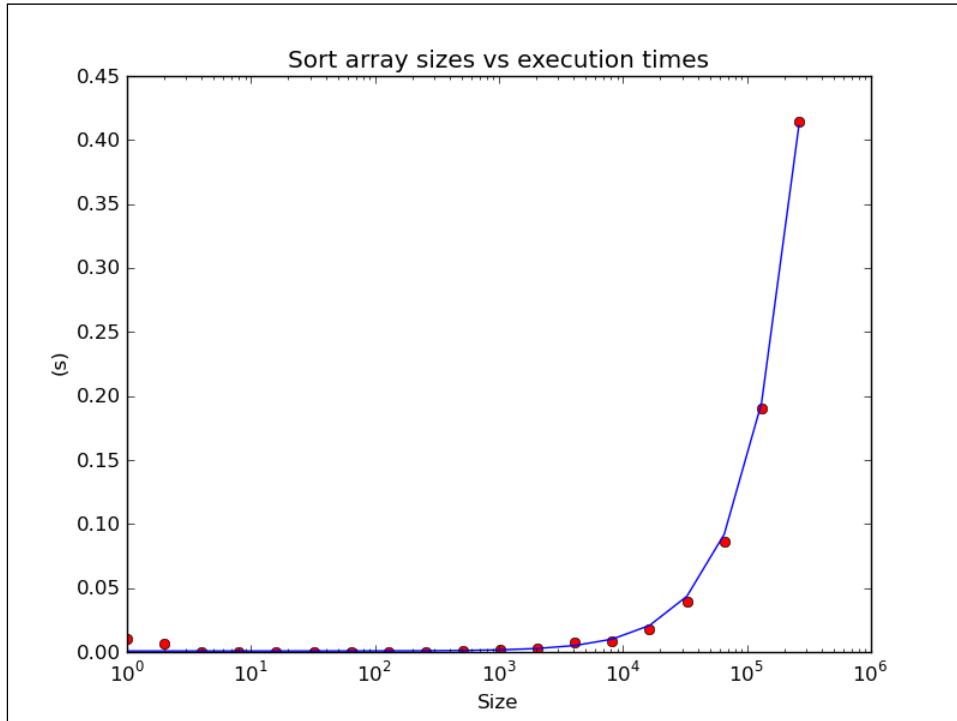
powersOf2 = numpy.arange(0, 19)
sizes = 2 ** powersOf2

times = numpy.array([])

for size in sizes:
    integers = numpy.random.random_integers
        (1, 10 ** 6, size)
    times = numpy.append(times, measure())

fit = numpy.polyfit(sizes * powersOf2, times, 1)
print fit
matplotlib.pyplot.title
    ("Sort array sizes vs execution times")
matplotlib.pyplot.xlabel("Size")
matplotlib.pyplot.ylabel("(s)")
matplotlib.pyplot.semilogx(sizes, times, 'ro')
matplotlib.pyplot.semilogx(sizes,
    numpy.polyval(fit, sizes * powersOf2))
matplotlib.pyplot.show()
```

The resulting plot for the running time versus array size is shown in the following screenshot:



How it works...

We measured the average running time of the NumPy `sort` function. The following functions were used in this recipe:

| Function | Description |
|------------------------------|---|
| <code>random_integers</code> | Creates an array of random integers given a range for the values and array size. |
| <code>append</code> | Appends a value to a NumPy array. |
| <code>polyfit</code> | Fits data to a polynomial of a given degree. |
| <code>polyval</code> | Evaluates a polynomial and returns the corresponding value for a certain "x" value. |
| <code>semilogx</code> | Plots data using logarithmic scale on the X-axis. |

Profiling with IPython

In IPython, we can profile small snippets of code using `timeit`. We can also profile a larger script. We will show both approaches.

How to do it...

First, we will time a small snippet.

1. Timing a snippet.

Start IPython in pylab mode:

```
ipython -pylab
```

Create an array containing 1000 integer values between 0 and 1000:

```
In [1]: a = arange(1000)
```

Measure the time taken for searching "the answer to everything"—42, in the array. Yes, the answer to everything is 42. If you don't believe me please read http://en.wikipedia.org/wiki/42_%28number%29.

```
In [2]: %timeit searchsorted(a, 42)
100000 loops, best of 3: 7.58 us per loop
```

2. Profile a script.

We will profile this small script that inverts a matrix of varying size containing random values. The `.I` attribute (that's uppercase I) of a NumPy array represents the inverse of a matrix:

```
import numpy

def invert(n):
    a = numpy.matrix(numpy.random.
        rand(n, n))
    return a.I

sizes = 2 ** numpy.arange(0, 12)

for n in sizes:
    invert(n)
```

We can time this as follows:

```
In [1]: %run -t invert_matrix.py
```

```
IPython CPU timings (estimated):
```

```
User      :      6.08 s.
System    :      0.52 s.
Wall time:      19.26 s.
```

Then we can profile the script with the `p` option:

```
In [2]: %run -p invert_matrix.py
```

```
852 function calls in 6.597 CPU seconds
```

```
Ordered by: internal time
```

| | ncalls | tottime | percall | cumtime | percall | |
|---------------------------|--------|---------|---------|---------|---|--|
| filename:lineno(function) | | | | | | |
| 12 | 3.228 | 0.269 | 3.228 | 0.269 | {numpy.linalg.lapack_lite.dgesv} | |
| 24 | 2.967 | 0.124 | 2.967 | 0.124 | {numpy.core.multiarray._fastCopyAndTranspose} | |
| 12 | 0.156 | 0.013 | 0.156 | 0.013 | {method 'rand' of 'mtrand.RandomState' objects} | |
| 12 | 0.087 | 0.007 | 0.087 | 0.007 | {method 'copy' of 'numpy.ndarray' objects} | |
| 12 | 0.069 | 0.006 | 0.069 | 0.006 | {method 'astype' of 'numpy.ndarray' objects} | |
| 12 | 0.025 | 0.002 | 6.304 | 0.525 | linalg.py:404(inv) | |
| 12 | 0.024 | 0.002 | 6.328 | 0.527 | defmatrix.py:808(getI) | |
| 1 | 0.017 | 0.017 | 6.596 | 6.596 | invert_matrix.py:1(<module>) | |
| 24 | 0.014 | 0.001 | 0.014 | 0.001 | {numpy.core.multiarray.zeros} | |
| 12 | 0.009 | 0.001 | 6.580 | 0.548 | invert_matrix.py:3(inv) | |

```

12  0.000  0.000  6.264  0.522 linalg.py:244(solve)
12  0.000  0.000  0.014  0.001 numeric.
py:1875(identity)
1  0.000  0.000  6.597  6.597 {execfile}
36  0.000  0.000  0.000  0.000 defmatrix.py:279(__
array_finalize__)
12  0.000  0.000  2.967  0.247 linalg.py:139(__
fastCopyAndTranspose)
24  0.000  0.000  0.087  0.004 defmatrix.py:233(__
new__)
12  0.000  0.000  0.000  0.000 linalg.py:99(__
commonType)
24  0.000  0.000  0.000  0.000 {method '__array_
prepare__' of 'numpy.ndarray' objects}
36  0.000  0.000  0.000  0.000 linalg.py:66(__
makearray)
36  0.000  0.000  0.000  0.000 {numpy.core.
multiarray.array}
12  0.000  0.000  0.000  0.000 {method 'view' of
'numpy.ndarray' objects}
12  0.000  0.000  0.000  0.000 linalg.py:127(__to_
native_byte_order)
1  0.000  0.000  6.597  6.597 interactiveshell.
py:2270(safe_execfile)

```

How it works...

We run the aforementioned NumPy code through a profiler. The following table summarizes the profiler output:

| Column | Description |
|---------|--|
| ncalls | Number of calls. |
| tottime | Total time spent in a function. |
| percall | Time per call, calculate by dividing the total time by the calls count. |
| cumtime | Cumulative time spent in function and functions called by the function, including recursive calls. |

Installing line_profiler

`line_profiler` was created by one of the NumPy developers. This module does line-by-line profiling of Python code. We will describe the necessary installation steps in this recipe.

Getting ready

You might need to install `setuptools`. This is covered in a previous recipe; refer to the *See Also* section if necessary. In order to install the development version, you will need Mercurial. Installing Mercurial is outside the scope of this book. Steps to install Mercurial can be found at <http://mercurial.selenic.com/wiki/Download>.

How to do it...

Choose the install option appropriate for you:

- ▶ Install with `easy_install`.

You can install `line_profiler` with `easy_install` by using any one of the following commands:

```
easy_install line_profiler
pip install line_profiler
```

- ▶ Install development version.

We can check out the source with Mercurial:

```
$ hg clone https://bitbucket.org/robertkern/line_profiler
```

After checking out the source, we can build it as follows:

```
$ python setup.py install
```

See also

- ▶ The *Installing IPython* recipe in *Chapter 1, Winding Along with IPython*

Profiling code with line_profiler

Now that we installed `line_profiler`, we can start profiling.

How to do it...

Obviously, we will need some code to profile.

1. Write code to profile.

We will write code to multiply a random matrix of varying size with itself. Also, the thread will sleep for a few seconds. The function to profile will be annotated with `@profile`:

```
import numpy
import time

@profile
def multiply(n):
    A = numpy.random.rand(n, n)
    time.sleep(numpy.random.randint(0, 2))
    return numpy.matrix(A) ** 2

for n in 2 ** numpy.arange(0, 10):
    multiply(n)
```

2. Profile the code.

Run the profiler with the following command:

```
$ kernprof.py -l -v mat_mult.py
Wrote profile results to mat_mult.py.lprof
Timer unit: 1e-06 s
```

```
File: mat_mult.py
Function: multiply at line 4
Total time: 3.19654 s
```

```

Line #      Hits      Time  Per Hit  % Time  Line Contents
=====
      4                                @profile
      5                                def multiply(n) :
      6      10      13461   1346.1    0.4    A = numpy.
random.rand(n, n)
      7      10     3000689 300068.9   93.9    time.
sleep(numpy.random.randint(0, 2))
      8      10     182386  18238.6    5.7    return numpy.
matrix(A) ** 2

```

How it works...

The `@profile` decorator tells `line_profiler` which functions to profile. The following table explains the output of the profiler:

| Column | Description |
|---------------|---|
| Line # | The line number in the file. |
| Hits | The number of times the line was executed. |
| Time | Time spent executing the line. |
| Per Hit | Average time spent executing the line. |
| % Time | Percentage of time spent executing the line relative to the time spent executing all the lines. |
| Line Contents | The contents of the line. |

Profiling code with the cProfile extension

`cProfile` is a C extension introduced in Python 2.5. It can be used for deterministic profiling. Deterministic profiling means that the time measurements are precise, and no sampling is used. Contrast this with statistical profiling, where measurements come from random samples. We will profile a small NumPy program, using `cProfile` that transposes an array with random values.

How to do it...

Again we require code to profile.

1. Write the code to profile.

We will write the `transpose` function that creates the array with random values and transposes it:

```
def transpose(n):
    random_values = numpy.random.random((n, n))
    return random_values.T
```

2. Run the profiler.

Run the profiler and give it the function to profile:

```
cProfile.run('transpose(%d)' % (int(sys.argv[1])))
```

The complete code for this tutorial can be found in the following snippet:

```
import numpy
import cProfile
import sys

def transpose(n):
    random_values = numpy.random.random((n, n))
    return random_values.T

cProfile.run('transpose(%d)' % (int(sys.argv[1])))
```

For a 1000-by-1000 array, we get the following output:

```
4 function calls in 0.029 CPU seconds
  Ordered by: standard name
  ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
          1    0.001    0.001    0.029    0.029 <string>:1(<module>)
          1    0.000    0.000    0.028    0.028 cprofile_transpose.
py:5(transpose)
          1    0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}
          1    0.028    0.028    0.028    0.028 {method 'random_
sample' of 'mtrand.RandomState' objects}
```

The columns in the output are the same as in the IPython profiling recipe.

Debugging with IPython

Debugging is one of those things nobody really likes, but is very important to master. It can take hours, and because of Murphy's law, you most likely, don't have that time. Therefore, it is important to be systematic and know your tools well. After you are done finding the bug and implementing a fix, you should have a test in place. This way at least you will not have to go through the hell of debugging again. Unit testing is covered in the next chapter. We will debug the following buggy code, which tries to access an array element that is not present:

```
import numpy

a = numpy.arange(7)
print a[8]
```

The IPython debugger works as the normal Python `pdb` debugger; it adds features such as tab completion and syntax highlighting.

How to do it...

The following steps illustrate a typical debugging session:

1. Run the buggy script in IPython.

Start the IPython shell. Run the buggy script in IPython by issuing the following command:

```
In [1]: %run buggy.py
-----
-----
IndexError                                Traceback (most recent
call last)
.../site-packages/IPython/utils/py3compat.py in execfile(fname,
*where)
    173         else:
    174             filename = fname
--> 175         __builtin__.execfile(filename, *where)

.../buggy.py in <module>()
      2
      3 a = numpy.arange(7)
----> 4 print a[8]

IndexError: index out of bounds
```

2. Start the debugger.

Now that our program has crashed, we can start the debugger. This will set a breakpoint on the line where the error occurred:

```
In [2]: %debug
> .../buggy.py(4)<module>()
      2
      3 a = numpy.arange(7)
----> 4 print a[8]
```

3. List code.

We can list code with the `list` command, or use the shorthand `l`:

```
ipdb> list
      1 import numpy
      2
      3 a = numpy.arange(7)
----> 4 print a[8]
```

4. Evaluate the code at the current line.

We can now evaluate arbitrary code at the current line, the line to which the debugger is currently pointing:

```
ipdb> len(a)
7

ipdb> print a
[0 1 2 3 4 5 6]
```

5. View the call stack.

The call stack is a stack containing information about active functions of a running program. We can view the call stack with the `bt` command:

```
ipdb> bt
.../py3compat.py(175)execfile()
      171         if isinstance(fname, unicode):
      172             filename = fname.encode(sys.
getfilesystemencoding())
      173         else:
      174             filename = fname
--> 175             __builtin__.execfile(filename, *where)
> .../buggy.py(4)<module>()
      0 print a[8]
```

Move up the call stack:

```
ipdb> u
> .../site-packages/IPython/utils/py3compat.py(175)execfile()
    173         else:
    174             filename = fname
--> 175         __builtin__.execfile(filename, *where)
```

Move down the call stack:

```
ipdb> d
> .../buggy.py(4)<module>()
    2
    3 a = numpy.arange(7)
----> 4 print a[8]
```

How to do it...

In this tutorial, we saw how to debug a NumPy program using IPython. We set a breakpoint and navigated the call stack. The following debugger commands were used:

| Command | Description |
|-----------|--------------------------|
| list or l | Lists source code. |
| bt | Shows call stack. |
| u | Moves up a call stack. |
| d | Moves down a call stack. |

Debugging with pudb

Pudb is a visual full-screen, console-based Python debugger that is easy to install. Pudb supports cursor keys and vi commands. The debugger can also be integrated with IPython, if required.

How to do it...

We'll start with the installation of pudb.

1. Installing pudb.

In order to install pudb, we only need to execute the following command:

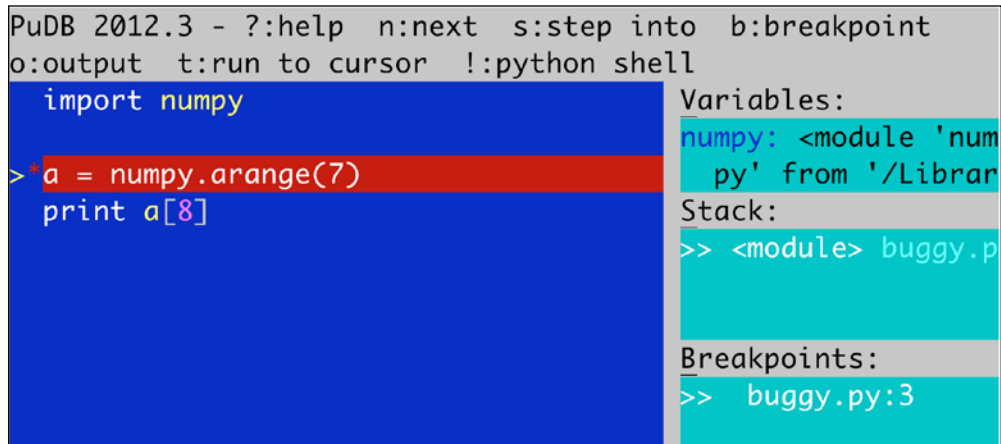
```
sudo easy_install pudb
```

2. Starting the debugger.

Let's debug the buggy program from the previous example. We can start the debugger as follows:

```
python -m pdb buggy.py
```

The user interface of the debugger is shown in the following screenshot:



```
PuDB 2012.3 - ?:help n:next s:step into b:breakpoint
o:output t:run to cursor !:python shell
import numpy
> a = numpy.arange(7)
print a[8]
```

Variables:
numpy: <module 'numpy' from '/Library/Python/2.7/...>
Stack:
>> <module> buggy.p
Breakpoints:
>> buggy.py:3

The screenshot shows the most important debugging commands at the top. We can also see the code being debugged, variables, the stack, and the defined breakpoints. Typing `q` exits most menus. Typing `n` moves the debugger to the next line. We can also move with the cursor keys or `vi` `j` and `k` keys to, for instance, set a breakpoint by typing `b`.

8

Quality Assurance

In this chapter, we will cover the following topics:

- ▶ Installing Pyflakes
- ▶ Performing static analysis with Pyflakes
- ▶ Analyzing code with Pylint
- ▶ Performing static analysis with Pychecker
- ▶ Testing code with docstrings
- ▶ Writing unit tests
- ▶ Testing code with mocks
- ▶ Testing the BDD way

Introduction

Quality assurance, contrary to popular belief, is not so much about finding bugs as it is about preventing them. We will discuss two ways to improve the code quality, thereby preventing issues. First, we will do static analysis of already existing code. Second, we will cover unit testing; this includes mocking and **Behavior Driven Development (BDD)**.

Installing Pyflakes

Pyflakes is a Python code analysis package. It can analyze your code, and spot potential problems such as:

- ▶ Unused imports
- ▶ Unused variables

Getting ready

Install `pip` or `easy_install`, if necessary.

How to do it...

Choose one of the following listed options to install `pyflakes`:

- ▶ Installing with `pip`.

We can install `pyflakes` with the `pip` command:

```
sudo pip install pyflakes
```

- ▶ Installing with `easy_install`.

We can install `pyflakes` with the `easy_install` command:

```
sudo easy_install pyflakes
```

- ▶ Installing on Linux.

The Linux package name is `pyflakes` as well. For instance, on Red Hat do the following:

```
sudo yum install pyflakes
```

On Debian/Ubuntu, the command is:

```
sudo apt-get install pyflakes
```

Performing static analysis with Pyflakes

We will perform static analysis of a part of the NumPy codebase. In order to do this, we will check out the code using Git. We will then run static analysis on part of the code using `pyflakes`.

How to do it...

1. Check out the code.

To check out the NumPy code, we will need Git. Installing Git is outside the scope of this book. The Git command to retrieve the code is as follows:

```
git clone git://github.com/numpy/numpy.git numpy
```

Alternatively, we can download a zip archive from

<https://github.com/numpy/numpy>.

2. Analyze the code.

The previous step should have created a `numpy` directory with all the NumPy code. Go to this directory, and within it run the following command:

```
$ pyflakes *.py
pavement.py:71: redefinition of unused 'md5' from line 69
pavement.py:88: redefinition of unused 'GIT_REVISION' from line 86
pavement.py:314: 'virtualenv' imported but unused
pavement.py:315: local variable 'e' is assigned to but never used
pavement.py:380: local variable 'sdir' is assigned to but never used
pavement.py:381: local variable 'bdir' is assigned to but never used
pavement.py:536: local variable 'st' is assigned to but never used
setup.py:21: 're' imported but unused
setup.py:27: redefinition of unused 'builtins' from line 25
setup.py:124: redefinition of unused 'GIT_REVISION' from line 118
setuptools.py:17: 'setup' imported but unused
setuptools.py:61: 'numpy' imported but unused
setuptools.py:64: 'numscons' imported but unused
setuptools.py:6: 'setup' imported but unused
```

This will run analysis on the code style, and check for PEP-8 violations in all the Python scripts within the current directory. You can also analyze a single file if you prefer.

How it works...

As you can see, it is pretty simple to analyze code style and look for PEP-8 violations with Pyflakes. The other advantage is speed; however, the number of error types that Pyflakes reports is pretty limited.

Analyzing code with Pylint

Pylint is another open source static analyzer originally created by Logilab. Pylint is more complex than Pyflakes; it allows more customization. However, it is slower than Pyflakes. For more information check out http://www.logilab.org/card/pylint_manual.

In this recipe, we will again download the NumPy code from the Git repository—this step is omitted for brevity.

Getting ready

You can install Pylint from the source distribution. However, there are many dependencies, so you are better off installing with either `easy_install`, or `pip`. The installation commands are as follows:

```
easy_install pylint
sudo pip install pylint
```

How to do it...

We will again analyze from the top directory of the NumPy codebase. Please notice that we are getting much more output. In fact, Pylint prints so much text that most of it had to be cut out here:

```
pylint *.py
$ pylint *.py
No config file found, using default configuration
***** Module pavement
C: 60: Line too long (81/80)
C:139: Line too long (81/80)
...
W: 50: TODO
W:168: XXX: find out which env variable is necessary to avoid the
pb with python
W: 71: Reimport 'md5' (imported line 143)
F: 73: Unable to import 'paver'
F: 74: Unable to import 'paver.easy'
C: 79: Invalid name "setup_py" (should match (([A-Z_][A-Z0-
9_]*)|(_.*_))$)
F: 86: Unable to import 'numpy.version'
E: 86: No name 'version' in module 'numpy'
C:149: Operator not followed by a space
if sys.platform == "darwin":
    ^^
C:202:prepare_nsis_script: Missing docstring
W:228:bdist_superpack: Redefining name 'options' from outer scope
(line 74)
C:231:bdist_superpack.copy_bdist: Missing docstring
W:275:bdist_wininst_nosse: Redefining name 'options' from outer
scope (line 74)
...
```

How it works...

Pylint outputs raw text, by default; but we could have requested HTML output, if desired. The messages have the following format:

```
MESSAGE_TYPE: LINE_NUM: [OBJECT:] MESSAGE
```

The *message type* can be one of the following:

- ▶ [R] meaning that refactoring is recommended
- ▶ [C] means that there was a code style violation
- ▶ [W] for warning about a minor issue
- ▶ [E] for error or potential bug
- ▶ [F] indicating that a fatal error occurred, blocking further analysis

See also

- ▶ The *Performing static analysis with Pyflakes* recipe

Performing static analysis with Pychecker

Pychecker is an old, static analysis tool, which is not very actively developed, but it's good enough to be mentioned here. The last version at the time of writing was 0.8.19, and was last updated in 2011. Pychecker tries to import each module and process it. The code is then analyzed to find issues such as passing incorrect number of parameters, incorrect format strings using non-existing methods, and other problems. In this recipe, we will again analyze code, but this time with Pychecker.

How to do it...

1. Install from tarball.

Download the `tar.gz` from Sourceforge (<http://pychecker.sourceforge.net/>). Unpack the tarball and run the following command:

```
python setup.py install
```

2. Install using `pip`.

We can, alternatively, install Pychecker using `pip`:

```
sudo pip install http://sourceforge.net/projects/pychecker/files/pychecker/0.8.19/pychecker-0.8.19.tar.gz/download
```

3. Analyze the code.

Let's analyze the code, just like in the previous recipes. The command we need is:

```
pychecker *.py
...
Warnings...

...

setup.py:21: Imported module (re) not used
setup.py:27: Module (builtins) re-imported

...
```

Testing code with docstrings

Docstrings are strings embedded in Python code that resemble interactive sessions. These strings can be used to test certain assumptions, or just provide examples. We need to use the `doctest` module to run these tests.

Let's write a simple example that is supposed to calculate the factorial, but doesn't cover all the possible boundary conditions. In other words, some tests will fail.

How to do it...

1. Write the docstring.

Write the docstring with a test that will pass, and a test that will fail. This should look like what you would normally see in a Python shell:

```
"""
Test for the factorial of 3 that should pass.
>>> factorial(3)
6

Test for the factorial of 0 that should fail.
>>> factorial(0)
1
"""
```

2. Write the NumPy code.

Write the following NumPy code:

```
return numpy.arange(1, n+1).cumprod() [-1]
```

We want this code to fail on purpose, sometimes. It will create an array of sequential numbers, calculate the cumulative product of the array, and return the last element.

3. Run the test.

As previously stated, we need to use the `doctest` module to run the tests:

```
doctest.testmod()
```

The following is the complete factorial and docstring test example code:

```
import numpy
import doctest

def factorial(n):
    """
    Test for the factorial of 3 that should pass.
    >>> factorial(3)
    6

    Test for the factorial of 0 that should fail.
    >>> factorial(0)
    1
    """
    return numpy.arange(1, n+1).cumprod() [-1]

doctest.testmod()
```

We can get verbose output with the `-v` option, as shown here:

```
python docstringtest.py -v
```

Trying:

```
factorial(3)
```

Expecting:

```
6
```

ok

Trying:

```
factorial(0)
```

Expecting:

```
1
```

```
*****
```



```
File "docstringtest.py", line 11, in __main__.factorial
Failed example:
    factorial(0)
Exception raised:
Traceback (most recent call last):
  File ".../doctest.py", line 1253, in __run
    compileflags, 1) in test.globs
  File "<doctest __main__.factorial[1]>", line 1, in <module>
    factorial(0)
  File "docstringtest.py", line 14, in factorial
    return numpy.arange(1, n+1).cumprod() [-1]
IndexError: index out of bounds
1 items had no tests:
    __main__
*****
1 items had failures:
    1 of 2 in __main__.factorial
2 tests in 2 items.
1 passed and 1 failed.
***Test Failed*** 1 failures.
```

How it works...

As you can see, we didn't take into account zero and negative numbers. Actually, we got an `index out of bounds` error due to an empty array. This is easy to fix of course, which we will do in the next tutorial.

Writing unit tests

Test-driven development (TDD) is the best thing that happened to software development this century. One of the most important aspects of TDD is the almost manic focus on unit testing.

Unit tests are automated tests that test a small piece of code, usually a function or method. Python has the PyUnit API for unit testing. As NumPy users, we can make use of the convenience functions in the `numpy.testing` module, as well. This module, as its name suggests, is dedicated to testing.

How to do it...

Let's write some code to be tested.

1. Write the factorial function.

We start by writing the following factorial function:

```
def factorial(n):
    if n == 0:
        return 1

    if n < 0:
        raise ValueError, "Don't be so negative"

    return numpy.arange(1, n+1).cumprod()
```

The code is the same as in the previous recipe, but we added a few checks for boundary conditions.

2. Write the unit test.

Now we will write the unit test. Maybe you have noticed that we don't have that many classes in this book; somehow, it didn't seem that necessary.

Let's write a class for a change. This class will contain the unit tests. It extends the `TestCase` class from the `unittest` module, which is a part of standard Python. We test for calling the factorial function with:

- a positive number, the happy path
- boundary condition 0
- negative numbers, which should result in an error

```
class FactorialTest(unittest.TestCase):
    def test_factorial(self):
        #Test for the factorial of 3 that should pass.
        self.assertEqual(6, factorial(3)[-1])
        numpy.testing.assert_equal(numpy.array([1, 2, 6]),
        factorial(3))

    def test_zero(self):
        #Test for the factorial of 0 that should pass.
        self.assertEqual(1, factorial(0))

    def test_negative(self):
        #Test for the factorial of negative numbers that should
        fail.
        # It should throw a ValueError, but we expect IndexError
        self.assertRaises(IndexError, factorial(-10))
```

The code for the factorial and the unit test in its entirety is as follows:

```
import numpy
import unittest

def factorial(n):
    if n == 0:
        return 1

    if n < 0:
        raise ValueError, "Don't be so negative"

    return numpy.arange(1, n+1).cumprod()

class FactorialTest(unittest.TestCase):
    def test_factorial(self):
        #Test for the factorial of 3 that should pass.
        self.assertEqual(6, factorial(3)[-1])
        numpy.testing.assert_equal(numpy.array([1, 2, 6]), factorial(3))

    def test_zero(self):
        #Test for the factorial of 0 that should pass.
        self.assertEqual(1, factorial(0))

    def test_negative(self):
        #Test for the factorial of negative numbers that should fail.
        # It should throw a ValueError, but we expect IndexError
        self.assertRaises(IndexError, factorial(-10))

if __name__ == '__main__':
    unittest.main()
```

The negative numbers test failed, as you can see in the following output:

```
.E.
=====
ERROR: test_negative (__main__.FactorialTest)
-----
Traceback (most recent call last):
  File "unit_test.py", line 26, in test_negative
    self.assertRaises(IndexError, factorial(-10))
  File "unit_test.py", line 9, in factorial
    raise ValueError, "Don't be so negative"
```

```
ValueError: Don't be so negative
```

```
-----
Ran 3 tests in 0.001s
```

```
FAILED (errors=1)
```

How it works...

We saw how to implement simple unit tests using the standard `unittest` Python module. We wrote a test class, which extends the `TestCase` class from the `unittest` module. The following functions were used to perform various tests:

| Function | Description |
|---|--|
| <code>numpy.testing.assert_equal</code> | Tests whether two NumPy arrays are equal |
| <code>unittest.assertEqual</code> | Tests whether two values are equal |
| <code>unittest.assertRaises</code> | Tests whether an exception is thrown |

The NumPy `testing` package has a number of test functions that we should know about:

| Function | Description |
|--|--|
| <code>assert_almost_equal</code> | Raises an exception if two numbers are not equal up to a specified precision. |
| <code>assert_approx_equal</code> | Raises an exception if two numbers are not equal up to a certain significance. |
| <code>assert_array_almost_equal</code> | Raises an exception if two arrays are not equal up to a specified precision. |
| <code>assert_array_equal</code> | Raises an exception if two arrays are not equal. |
| <code>assert_array_less</code> | Raises an exception if two arrays do not have the same shape, and the elements of the first array are strictly less than the elements of the second array. |
| <code>assert_raises</code> | Fails if a specified exception is not raised by a callable invoked with defined arguments. |
| <code>assert_warns</code> | Fails if a specified warning is not thrown. |
| <code>assert_string_equal</code> | Asserts that two strings are equal. |

Testing code with mocks

Mocks are objects created as substitutes for real objects, with the purpose of testing a part of the behavior of the real objects. If you have seen the *Body Snatchers* movie, you already have an understanding of the basic idea. Generally speaking, mocking is only useful when the real objects under test are expensive to create, such as a database connection, or when testing could have undesired side effects; for instance, we might not want to write to the file system or database.

In this recipe, we will test a nuclear reactor—not a real one, of course. This nuclear reactor class performs a factorial calculation that could, in theory, cause a chain reaction with a nuclear disaster as consequence. We will mock the factorial computation with a mock, using the `mock` package.

How to do it...

First, we will install the mock package; after which, we will create a mock and test a piece of code.

1. Install mock.

In order to install the `mock` package, execute the following command:

```
sudo easy_install mock
```

2. Create a mock.

The nuclear reactor class has a `do_work` method, which calls a dangerous `factorial` method, which we want to mock. Create a mock as follows:

```
reactor.factorial = MagicMock(return_value=6)
```

This ensures that the mock returns a value of 6.

3. Assert behavior.

We can check the behavior of a mock and from that, the behavior of the real object under test, in several ways. For instance, we can assert that the potentially explosive `factorial` method is being called with the correct arguments as follows:

```
reactor.factorial.assert_called_with(3, "mocked")
```

The complete test code with mocks is as follows:

```
from mock import MagicMock
import numpy
import unittest

class NuclearReactor():
    def __init__(self, n):
```

```
        self.n = n

    def do_work(self, msg):
        print "Working"

        return self.factorial(self.n, msg)

    def factorial(self, n, msg):
        print msg

        if n == 0:
            return 1

        if n < 0:
            raise ValueError, "Core meltdown"

        return numpy.arange(1, n+1).cumprod()

class NuclearReactorTest(unittest.TestCase):
    def test_called(self):
        reactor = NuclearReactor(3)
        reactor.factorial = MagicMock(return_value=6)
        result = reactor.do_work("mocked")
        self.assertEqual(6, result)
        reactor.factorial.assert_called_with(3, "mocked")

    def test_unmocked(self):
        reactor = NuclearReactor(3)
        reactor.factorial(3, "unmocked")
        numpy.testing.assert_raises(ValueError)

if __name__ == '__main__':
    unittest.main()
```

We pass a string to the `factorial` method to show that the code with mock does not exercise the real code. The unit test works in the same way as the unit test in the previous recipe. The second test here does not test anything. The purpose of the second test is just to demonstrate what happens if we exercise the real code without mocks.

The output of the tests is as follows:

```
Working
.unmocked
.
```

```
-----
Ran 2 tests in 0.000s
```

```
OK
```

How it works...

Mocks do not have any behavior. They are like alien clones pretending to be real people, only dumber than aliens. An alien clone wouldn't be able to tell you the birthday of the real person it is replacing. We need to set them up to respond in an appropriate manner. For instance, the mock returned `6` in this example. We can record what is happening to the mock—how many times it is being called and with which arguments.

Testing the BDD way

BDD (Behavior Driven Development) is another hot acronym that you might have come across. In BDD, we start by defining (in English) the expected behavior of the system under test, according to certain conventions and rules. In this recipe, we will see an example of those conventions.

The idea behind this approach is that we can have people who may not be able to program, write a major part of the tests. A feature written by these people takes the form of a sentence consisting of several steps. Each step is more or less a unit test that we can write, for instance, using NumPy. There are many Python BDD frameworks. In this recipe, we will be using *Lettuce* to test the factorial function.

How to do it...

In this section, we will see how to install Lettuce, set up the tests, and write the specifications for the test.

1. Installing Lettuce.

In order to install Lettuce, run either of the following commands:

```
pip install lettuce
sudo easy_install lettuce
```

2. Setting up the tests.

Lettuce requires a special directory structure for the tests. In the `tests` directory, we will have a directory named `features` containing the `factorial.feature` file, along with the functional descriptions and test code in the `steps.py` file:

```
./tests:
features

./tests/features:
factorial.feature  steps.py
```

3. Writing the specifications.

Coming up with the business requirements is a hard job. Writing it all down in such a way that it is easy to test is even harder. Luckily, the requirements for these recipes are pretty trivial—we will just write down different input values and the expected outputs.

We have different scenarios with `Given`, `When`, and `Then` sections, which correspond to different test steps. We will define the following three scenarios for the factorial feature:

Feature: Compute factorial

```
Scenario: Factorial of 0
  Given I have the number 0
  When I compute its factorial
  Then I see the number 1
```

```
Scenario: Factorial of 1
  Given I have the number 1
  When I compute its factorial
  Then I see the number 1
```

```
Scenario: Factorial of 3
  Given I have the number 3
  When I compute its factorial
  Then I see the number 1, 2, 6
```


4. Defining the steps.

We will define methods that correspond to the steps of our scenario. We should pay extra attention to the text used to annotate the methods. It matches the text in the business scenarios file, and we are using regular expressions to get input parameters.

In the first two scenarios, we are matching numbers, and in the last we match any text. The NumPy `fromstring` function is used to create a string from a NumPy array, with an integer data type and comma separator in the string. The following code tests our scenarios:

```
from lettuce import *
import numpy

@step('I have the number (\d+)')
def have_the_number(step, number):
    world.number = int(number)

@step('I compute its factorial')
def compute_its_factorial(step):
    world.number = factorial(world.number)

@step('I see the number (.*)')
def check_number(step, expected):
    expected = numpy.fromstring(expected, dtype=int, sep=',')
    numpy.testing.assert_equal(world.number, expected, \
        "Got %s" % world.number)

def factorial(n):
    if n == 0:
        return 1

    if n < 0:
        raise ValueError, "Core meltdown"

    return numpy.arange(1, n+1).cumprod()
```

5. Run the tests.

In order to run the tests, go to the `tests` directory and type the following command:

```
$ lettuce
```

```
Feature: Compute factorial          # features/factorial.feature:1

Scenario: Factorial of 0           # features/factorial.feature:3
  Given I have the number 0        # features/steps.py:5
```

```
When I compute its factorial # features/steps.py:9
Then I see the number 1     # features/steps.py:13

Scenario: Factorial of 1    # features/factorial.feature:8
  Given I have the number 1 # features/steps.py:5
  When I compute its factorial # features/steps.py:9
  Then I see the number 1   # features/steps.py:13

Scenario: Factorial of 3    # features/factorial.feature:13
  Given I have the number 3 # features/steps.py:5
  When I compute its factorial # features/steps.py:9
  Then I see the number 1, 2, 6 # features/steps.py:13

1 feature (1 passed)
3 scenarios (3 passed)
9 steps (9 passed)
```

How it works...

We defined a feature with three scenarios and corresponding steps. We used the NumPy testing functions to test the different steps, and the `fromstring` function to create a NumPy array from the specifications text.

9

Speed Up Code with Cython

In this chapter, we will cover:

- ▶ Installing Cython
- ▶ Building a Hello World program
- ▶ Using Cython with NumPy
- ▶ Calling C functions
- ▶ Profiling Cython code
- ▶ Approximating factorials with Cython

Introduction

Cython is a relatively young programming language based on Python. The difference with Python is that we can optionally declare static types. Many programming languages, such as C, have static typing, which means that we have to tell C the type of variables, function parameters, and return types. Another difference is that C is a compiled language, while Python is an interpreted language. As a rule of thumb, we can say that C is faster but less flexible than Python. From Cython code, we can generate C or C++ code. After that, we can compile the generated code into Python extension modules.

In this chapter we will be learning about Cython. We will get some simple Cython programs running together with NumPy. Also we will profile Cython code.

Installing Cython

In order to use Cython, we need to install Cython. Enthought and Sage have Cython included. For more information, see <http://www.enthought.com/products/epd.php> and <http://sagemath.org/>. We will not discuss here how to install these distributions. Obviously, we need a C compiler to compile the generated C code. On some operating systems, such as Linux, the compiler will already be present. In this recipe, we will assume that you already have the compiler installed.

How to do it...

Cython can be installed using any of the following methods:

- ▶ Installing from tarball (.tar archive): Cython can be installed from tarball by performing the following steps:
 1. Download a tarball from <http://cython.org/#download>.
 2. Unpack it.
 3. Browse to the directory using the `cd` command.
 4. Run the following command:

```
python setup.py install
```

- ▶ Installing with setup tools or pip
We can install Cython from the PyPI repository with `easy_install cython` or `sudo pip install cython`.
- ▶ Installing with Windows installers
We can install Cython on Windows using the unofficial Windows installers from <http://www.lfd.uci.edu/~gohlke/pythonlibs/#cython>.

Building a Hello World program

As is the tradition with programming languages, we will start with a Hello World example. Unlike with Python, we need to compile Cython code. We start with a `.pyx` file, from which we will generate C code. This `.c` file can be compiled and then imported into a Python program.

How to do it...

This section describes how to build a Cython Hello World program.

1. Write the `hello.pyx` code.

First, we will write some pretty trivial code that prints "Hello World". This is just normal Python code, but the file has the `pyx` extension.

```
def say_hello():
    print "Hello World!"
```

2. Write a `distutils setup.py` script.

We need to create a file named `setup.py` to help us build the Cython code.

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

ext_modules = [Extension("hello", ["hello.pyx"])]

setup(
    name = 'Hello world app',
    cmdclass = {'build_ext': build_ext},
    ext_modules = ext_modules
)
```

As you can see, we specified the file from the previous step and gave our application a name.

3. Build using the following command:

```
python setup.py build_ext --inplace
```

This will generate C code, compile it for your platform, and will produce the following output

```
running build_ext
cythoning hello.pyx to hello.c
building 'hello' extension
creating build
```

Now, we can import our module with the following statement:

```
from hello import say_hello
```

How it works...

In this recipe we did a traditional Hello World example. Cython is a compiled language, so we needed to compile our code. We wrote a `.pyx` file containing the Hello World code and a `setup.py` file that was used to generate and build the C code.

Using Cython with NumPy

We can integrate Cython and NumPy code in the same way that we can integrate Cython and Python code. Let's go through an example that analyzes the ratio of up days (days on which a stock closes higher than the previous day) for a stock. We will apply the formula for binomial proportion confidence. You can refer to http://en.wikipedia.org/wiki/Binomial_proportion_confidence_interval for more information. This indicates how significant the ratio is.

How to do it...

This section describes how we can use Cython with NumPy. To demonstrate this, perform the following steps:

1. Write the `.pyx` file.

Let's write a `.pyx` file that contains a function to calculate the ratio of up days and associated confidence. First, this function computes the differences of the prices. Then, we count the number of positive differences, giving us a ratio for the proportion of up days. Finally, we apply the formula for the confidence from the Wikipedia page in the introduction.

```
import numpy

def pos_confidence(numbers):
    diffs = numpy.diff(numbers)
    n = float(len(diffs))
    p = len(diffs[diffs > 0])/n
    confidence = numpy.sqrt(p * (1 - p) / n)

    return (p, confidence)
```

2. Write the `setup.py` file.

We will use the `setup.py` file from the previous example, as the template. There are some obvious things that need to be changed, such as the name of the `.pyx` file.

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext
```

```

ext_modules = [Extension("binomial_proportion", ["binomial_
proportion.pyx"])]

setup(
    name = 'Binomial proportion app',
    cmdclass = {'build_ext': build_ext},
    ext_modules = ext_modules
)

```

We can now build—see the previous recipe for more details.

3. Use the Cython module.

After building, we can use the Cython module from the previous step by importing. We will write a Python program that downloads stock price data with `matplotlib`. Then, we will apply the `confidence` function to the close prices.

```

from matplotlib.finance import quotes_historical_yahoo
from datetime import date
import numpy
import sys
from binomial_proportion import pos_confidence

#1. Get close prices.
today = date.today()
start = (today.year - 1, today.month, today.day)

quotes = quotes_historical_yahoo(sys.argv[1], start, today)
close = numpy.array([q[4] for q in quotes])
print pos_confidence(close)

```

The output of the program for AAPL is shown as follows:

```
(0.56746031746031744, 0.031209043355655924)
```

How it works...

We computed the probability of an up day for AAPL shares and the corresponding confidence. We put NumPy code in a `.pyx` file and built it just like in the previous tutorial, creating a Cython module. At the end, we imported and used the Cython module.

Calling C functions

We can call C functions from Cython. For instance, in this example, we will call the C `log` function. This function works on a single number only. Remember that the NumPy `log` function can also work with arrays. We will compute the so-called log returns of stock prices.

How to do it...

We will start by writing some Cython code:

1. Write the `.pyx` file.

First, we need to import the `C log` function from the `libc` namespace. Second, we will apply this function to numbers in a `for` loop. Finally, we will use the NumPy `diff` function to get the first order difference between the log values in the second step.

```
from libc.math cimport log
import numpy

def logrets(numbers):
    logs = [log(x) for x in numbers]
    return numpy.diff(logs)
```

Building has been covered in the previous recipes already. We only need to change some values in the `setup.py` file.

2. Plot the log returns.

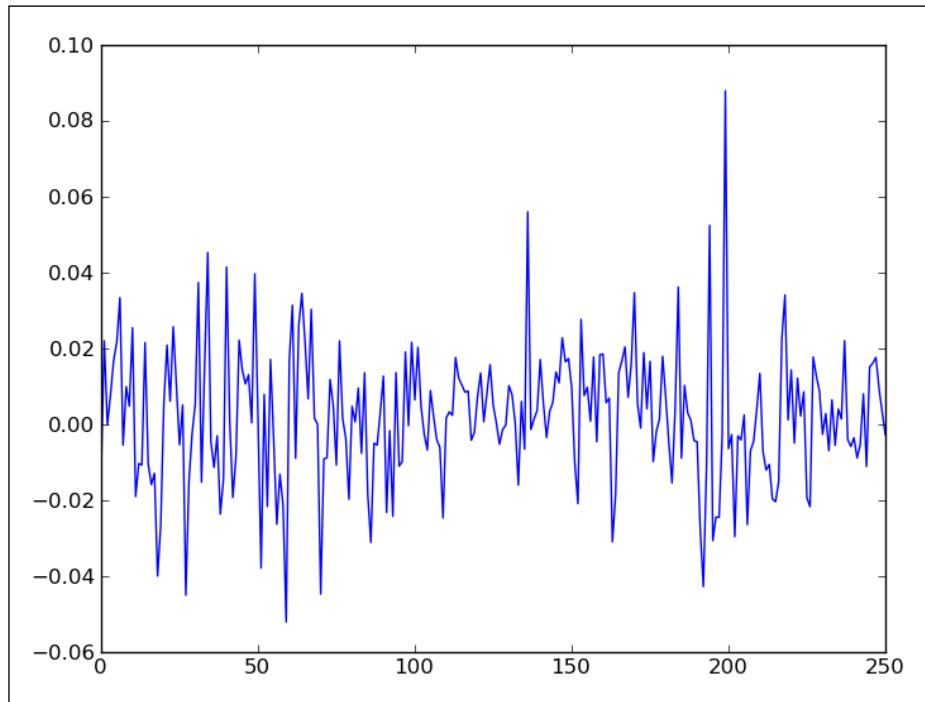
Let's download stock price data with `matplotlib`, again. Apply the Cython `logrets` function that we just created on the prices and plot the result.

```
from matplotlib.finance import quotes_historical_yahoo
from datetime import date
import numpy
import sys
from log_returns import logrets
import matplotlib.pyplot

today = date.today()
start = (today.year - 1, today.month, today.day)

quotes = quotes_historical_yahoo(sys.argv[1], start, today)
close = numpy.array([q[4] for q in quotes])
matplotlib.pyplot.plot(logrets(close))
matplotlib.pyplot.show()
```

The resulting plot of the log returns for AAPL is shown in the following screenshot:



How it works...

We called the C `log` function from Cython code. The function together with NumPy functions was used to calculate log returns of stocks. This way, we can create our own specialized API containing convenience functions. The nice thing is that our code should perform at or near the speed of C code, while looking more or less like Python code.

Profiling Cython code

We will profile Cython and NumPy code that tries to approximate the Euler constant. You can refer to http://en.wikipedia.org/wiki/E_%28mathematical_constant%29 for the required formula.

How to do it...

This section demonstrates how to profile Cython code. To do this, go through the following steps:

- ▶ **NumPy approximation of e:** For the NumPy approximation of e perform the following steps:
 1. First, we will create an array of 1 to n (n is 40 in our example).
 2. Then, we will compute the cumulative product of this array, which happens to be the factorial.
 3. After that, we take the reciprocal of the factorials.
 4. Finally, we apply the formula from the Wikipedia page. At the end, we put standard profiling code, giving us the following program:

```
import numpy
import cProfile
import pstats

def approx_e(n=40):
    # array of [1, 2, ... n-1]
    arr = numpy.arange(1, n)

    # calculate the factorials and convert to floats
    arr = arr.cumprod().astype(float)

    # reciprocal 1/n
    arr = numpy.reciprocal(arr)

    print 1 + arr.sum()

cProfile.runctx("approx_e()", globals(), locals(), "Profile.prof")

s = pstats.Stats("Profile.prof")
s.strip_dirs().sort_stats("time").print_stats()
```

The profiling output and the result for the e approximation is shown in the following snippet. Please refer to *Chapter 7, Profiling and Debugging*, for more information about the profiling output.

```
2.71828182846
```

```
7 function calls in 0.000 CPU seconds
```

Ordered by: internal time

```

ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
      1  0.000   0.000   0.000   0.000 numpy_approx_e.
py:5(approx_e)
      1  0.000   0.000   0.000   0.000 {method 'cumprod' of
'numpy.ndarray' objects}
      1  0.000   0.000   0.000   0.000 {numpy.core.
multiarray.arange}
      1  0.000   0.000   0.000   0.000 {method 'sum' of
'numpy.ndarray' objects}
      1  0.000   0.000   0.000   0.000 {method 'astype' of
'numpy.ndarray' objects}
      1  0.000   0.000   0.000   0.000 <string>:1(<module>)
      1  0.000   0.000   0.000   0.000 {method 'disable' of
'_lsprof.Profiler' objects}

```

- **Cython approximation of e:** The Cython code uses the same algorithm as in the previous step, but the implementation is different. There are less convenience functions, and we actually need a `for` loop now. Also, we need to specify types for some of the variables. The code for the `.pyx` file is shown as follows:

```

def approx_e(int n=40):
    cdef double sum = 0.
    cdef double factorial = 1.
    cdef int k
    for k in xrange(1,n+1):
        factorial *= k
        sum += 1/factorial
    print 1 + sum

```

The following Python program imports the Cython module and does some profiling.

```

import pstats
import cProfile
import pyximport
pyximport.install()

import approx_e
cProfile.runctx("approx_e()", globals(), locals(),
"Profile.prof")

s = pstats.Stats("Profile.prof")
s.strip_dirs().sort_stats("time").print_stats()

```

This is the profiling output of the Cython code:

2.71828182846

3 function calls in 0.000 CPU seconds

Ordered by: internal time

| ncalls | tottime | percall | cumtime | percall | |
|---------------------------|---------|---------|---------|---------|--|
| filename:lineno(function) | | | | | |
| 1 | 0.000 | 0.000 | 0.000 | 0.000 | {approx_e.approx_e} |
| 1 | 0.000 | 0.000 | 0.000 | 0.000 | <string>:1(<module>) |
| 1 | 0.000 | 0.000 | 0.000 | 0.000 | {method 'disable' of '_lsprof.Profiler' objects} |

How it works...

We profiled NumPy and Cython code. NumPy is heavily optimized for speed, so we should not be surprised that both NumPy and Cython programs are high-performing programs.

Approximating factorials with Cython

The last example is about approximating factorials with Cython. We will use two approximation methods. First, we will use the Stirling approximation method (see http://en.wikipedia.org/wiki/Stirling%27s_approximation for more information). The formula for the Stirling approximation is:

$$\sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$

Secondly, we will be using the approximation due to Ramanujan, with the following formula:

$$\sqrt{\pi} \left(\frac{n}{e}\right)^n \sqrt[6]{8n^3 + 4n^2 + n + \frac{1}{30}}$$

How to do it...

This section describes how to approximate factorials using Cython. In this recipe, we will be using types, which as you may remember, is optional in Cython. In theory, declaring static types should speed things up. Static typing offers interesting challenges that you may not encounter when writing Python code, but don't worry, we will try to keep it simple.

1. Write the Cython code.

The Cython code that we will write looks like regular Python code, except that we declare function parameters and a local variable to be an `ndarray` array. In order to get the static types to work, we need to import NumPy using the `cimport` statement. Also we have to use the `cdef` keyword to declare the type of the local variable.

```
import numpy
cimport numpy

def ramanujan_factorial(numpy.ndarray n):
    sqrt_pi = numpy.sqrt(numpy.pi, dtype=numpy.float64)
    cdef numpy.ndarray root = (8 * n + 4) * n + 1
    root = root * n + 1/30.
    root = root ** (1/6.)
    return sqrt_pi * calc_eton(n) * root

def stirling_factorial(numpy.ndarray n):
    return numpy.sqrt(2 * numpy.pi * n) * calc_eton(n)

def calc_eton(numpy.ndarray n):
    return (n/numpy.e) ** n
```

2. Build the code.

Building requires us to create a `setup.py` file, as in the previous tutorials, but we now need to include NumPy-related directories by calling the `get_include` function. With this amendment, the `setup.py` file will have the following content:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext
import numpy

ext_modules = [Extension("factorial", ["factorial.pyx"], include_
dirs = [numpy.get_include()])]

setup(
    name = 'Factorial app',
    cmdclass = {'build_ext': build_ext},
    ext_modules = ext_modules
)
```

3. Plot the relative error.

Let's plot the relative error for both approximation methods. We will calculate the error relative to the factorial values that we will compute with the NumPy `cumprod` function, as we have done throughout the book.

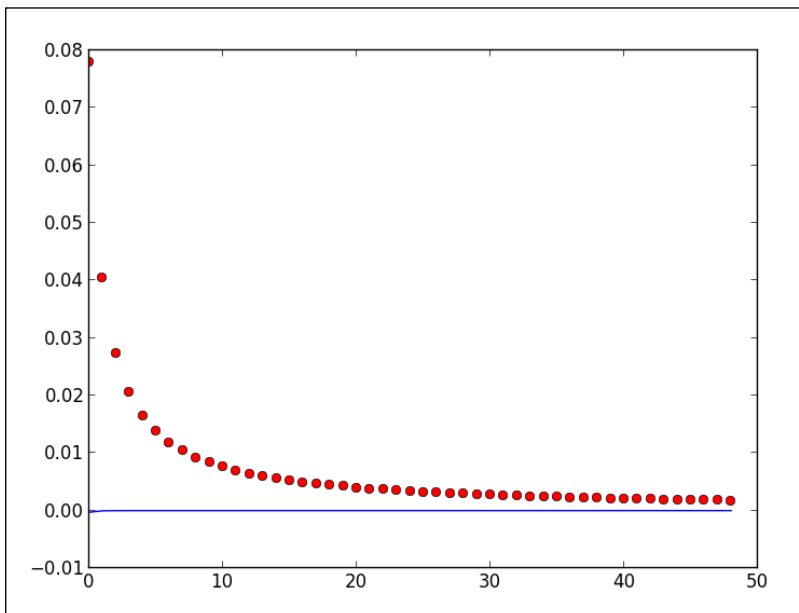
```
from factorial import ramanujan_factorial
from factorial import stirling_factorial
import numpy
import matplotlib.pyplot

N = 50
numbers = numpy.arange(1, N)
factorials = numpy.cumprod(numbers, dtype=float)

def error(approximations):
    return (factorials - approximations)/factorials

matplotlib.pyplot.plot(error(ramanujan_factorial(numbers)), 'b-')
matplotlib.pyplot.plot(error(stirling_factorial(numbers)), 'ro')
matplotlib.pyplot.show()
```

The following plot shows the relative error for the Ramanujan approximation (dots) and the Stirling approximation (line). As you can see, the Ramanujan method is more accurate:



How it works...

In this example, we saw a demonstration of Cython's static types. The main ingredients of this recipe were:

- ▶ `cimport`, which imports C declarations
- ▶ Including directories with the `get_include()` function
- ▶ The `cdef` keyword to define the type of local variables

10

Fun with Scikits

In this chapter, we will cover the following topics:

- ▶ Installing scikits-learn
- ▶ Loading an example dataset
- ▶ Clustering Dow Jones stocks with scikits-learn
- ▶ Installing scikits-statsmodels
- ▶ Performing a normality test with scikits-statsmodels
- ▶ Installing scikits-image
- ▶ Detecting corners
- ▶ Detecting edges
- ▶ Installing pandas
- ▶ Estimating stock returns correlation with Pandas
- ▶ Loading data as `pandas` objects from statsmodels
- ▶ Resampling time series data

Introduction

Scikits are small, independent projects that are related to SciPy in some way, but are not a part of SciPy. These projects are not entirely independent, but operate under an umbrella, as a consortium of sorts. In this chapter, we will discuss several Scikits projects, such as the following:

- ▶ `scikits-learn`, a machine learning package
- ▶ `scikits-statsmodels`, a statistics package
- ▶ `scikits-image`, an image processing package
- ▶ `pandas`, a data analysis package

Installing scikits-learn

The **scikits-learn** project aims to provide an API for *machine learning*. What I like most about it is the amazing documentation. We can install scikits-learn with the package manager of our operating system. This option may or may not be available, depending on the operating system, but should be the most convenient route.

Windows users can just download an installer from the project website. On Debian and Ubuntu, the project is named *python-sklearn*. On MacPorts, the ports are named *py26-scikits-learn* and *py27-scikits-learn*. We can also install from source, or using *easy_install*. There are third-party distributions from Python(x, y), Enthought, and NetBSD.

Getting ready

You need to have SciPy and NumPy installed. Go back to *Chapter 1, Winding Along with lpython*, for instructions, if necessary.

How to do it...

Let us now see how we can install the scikits-learn project.

- ▶ **Installing with easy_install:** We can install by typing any one of the following commands, at the command line:

```
pip install -U scikit-learn
easy_install -U scikit-learn
```

This might not work because of permissions, so you might either need to put `sudo` in front of the commands, or log in as admin.

- ▶ **Installing from source:** Download the source from <http://pypi.python.org/pypi/scikit-learn/>, unpack and `cd` into the downloaded folder. Issue the following command:

```
python setup.py install
```

Loading an example dataset

The scikits-learn project comes with a number of datasets and sample images with which we can experiment. In this recipe, we will load an example dataset, that is included with the scikits-learn distribution. The datasets hold data as a NumPy, two-dimensional array and metadata linked to the data.

How to do it...

We will load a sample data set of the Boston house prices. It is a tiny dataset, so if you are looking for a house in Boston, don't get too excited. There are more datasets as described in <http://scikit-learn.org/dev/modules/classes.html#module-sklearn.datasets>.

We will look at the shape of the raw data, and its maximum and minimum value. The shape is a *tuple*, representing the dimensions of the NumPy array. We will do the same for the target array, which contains values that are the learning objectives. The following code accomplishes our goals:

```
from sklearn import datasets

boston_prices = datasets.load_boston()
print "Data shape", boston_prices.data.shape
print "Data max=%s min=%s" %
      (boston_prices.data.max(),
       boston_prices.data.min())
print "Target shape",
      boston_prices.target.shape
print "Target max=%s min=%s" %
      (boston_prices.target.max(),
       boston_prices.target.min())
```

And the outcome of our program is as follows:

```
Data shape (506, 13)
Data max=711.0 min=0.0
Target shape (506,)
Target max=50.0 min=5.0
```

Clustering Dow Jones stocks with scikits-learn

Clustering is a type of machine learning algorithm, which aims to group items based on similarities. In this example, we will use the log returns of stocks in the Dow Jones Industrial Index to cluster. Most of the steps of this recipe have already passed the review in previous chapters.

How to do it...

First, we will download the EOD price data for those stocks from Yahoo Finance. Second, we will calculate a square affinity matrix. Finally, we will cluster the stocks with the `AffinityPropagation` class.

1. Downloading the price data.

We will download price data for 2011 using the stock symbols of the DJI Index. In this example, we are only interested in the close price:

```
# 2011 to 2012
start = datetime.datetime(2011, 01, 01)
end = datetime.datetime(2012, 01, 01)

#Dow Jones symbols
symbols = ["AA", "AXP", "BA",
           "BAC", "CAT", "CSCO", "CVX",
           "DD", "DIS", "GE", "HD", "HPQ",
           "IBM", "INTC", "JNJ", "JPM",
           "KFT", "KO", "MCD", "MMM", "MRK",
           "MSFT", "PFE", "PG", "T", "TRV",
           "UTX", "VZ", "WMT", "XOM"]

quotes = [finance.quotes_historical_yahoo
          (symbol, start, end, asobject=True)
          for symbol in symbols]

close = numpy.array(
    [q.close for q in quotes])
    .astype(numpy.float)
```

2. Calculating the affinity matrix.

Calculate the similarities between different stocks using the log returns as metric. What we are trying to do is calculate the Euclidean distances for the data points:

```
logreturns = numpy.diff(numpy.log(close))
print logreturns.shape

logreturns_norms = numpy.sum(logreturns ** 2, axis=1)
S = - logreturns_norms[:, numpy.newaxis] -
    logreturns_norms[numpy.newaxis, :] + 2 *
    numpy.dot(logreturns, logreturns.T)
```

3. Clustering the stocks.

Give the `AffinityPropagation` class the result from the previous step. This class labels the data points, or in our case, stocks with the appropriate cluster number:

```
aff_pro = sklearn.cluster.AffinityPropagation().fit(S)
labels = aff_pro.labels_

for i in xrange(len(labels)):
    print '%s in Cluster %d' %
        (symbols[i], labels[i])
```

The complete clustering program is as follows:

```
import datetime
import numpy
import sklearn.cluster
from matplotlib import finance

#1. Download price data

# 2011 to 2012
start = datetime.datetime(2011, 01, 01)
end = datetime.datetime(2012, 01, 01)

#Dow Jones symbols
symbols = ["AA", "AXP", "BA", "BAC", "CAT",
          "CSCO", "CVX", "DD", "DIS", "GE", "HD",
          "HPQ", "IBM", "INTC", "JNJ", "JPM", "KFT",
          "KO", "MCD", "MMM", "MRK", "MSFT", "PFE",
          "PG", "T", "TRV", "UTX", "VZ", "WMT", "XOM"]

quotes = [finance.quotes_historical_yahoo
          (symbol, start, end, asobject=True)
          for symbol in symbols]

close = numpy.array([q.close for q in quotes]).astype(numpy.float)
print close.shape

#2. Calculate affinity matrix
logreturns = numpy.diff(numpy.log(close))
print logreturns.shape

logreturns_norms = numpy.sum
    (logreturns ** 2, axis=1)
```

```
S = - logreturns_norms[:, numpy.newaxis] -  
    logreturns_norms[numpy.newaxis, :] + 2 *  
    numpy.dot(logreturns, logreturns.T)  
  
#3. Cluster using affinity propagation  
aff_pro = sklearn.cluster  
    .AffinityPropagation().fit(S)  
labels = aff_pro.labels_  
  
for i in xrange(len(labels)):  
    print '%s in Cluster %d' % (symbols[i], labels[i])
```

The output with the cluster numbers for each stock is as follows:

```
(30, 252)  
(30, 251)  
AA in Cluster 0  
AXP in Cluster 6  
BA in Cluster 6  
BAC in Cluster 1  
CAT in Cluster 6  
CSCO in Cluster 2  
CVX in Cluster 7  
DD in Cluster 6  
DIS in Cluster 6  
GE in Cluster 6  
HD in Cluster 5  
HPQ in Cluster 3  
IBM in Cluster 5  
INTC in Cluster 6  
JNJ in Cluster 5  
JPM in Cluster 4  
KFT in Cluster 5  
KO in Cluster 5  
MCD in Cluster 5  
MMM in Cluster 6  
MRK in Cluster 5  
MSFT in Cluster 5  
PFE in Cluster 7
```

PG in Cluster 5
T in Cluster 5
TRV in Cluster 5
UTX in Cluster 6
VZ in Cluster 5
WMT in Cluster 5
XOM in Cluster 7

How it works...

The following table is an overview of the functions we used in this recipe:

| Function | Description |
|--|---|
| <code>sklearn.cluster.AffinityPropagation()</code> | Creates an <code>AffinityPropagation</code> object. |
| <code>sklearn.cluster.AffinityPropagation.fit</code> | Computes an affinity matrix from Euclidian distances and applies affinity propagation clustering. |
| <code>diff</code> | Calculates differences of numbers within a NumPy array. If not specified, first-order differences are computed. |
| <code>log</code> | Calculates the natural log of elements in a NumPy array. |
| <code>sum</code> | Sums the elements of a NumPy array. |
| <code>dot</code> | Does matrix multiplication for 2D arrays. Calculates the inner product for 1D arrays. |

Installing scikits-statsmodels

The **scikits-statsmodels** package focuses on statistical modeling. It can be integrated with NumPy and Pandas (more about Pandas later in this chapter).

How to do it...

Source and binaries can be downloaded from <http://statsmodels.sourceforge.net/install.html>. If you are installing from source, you need to run the following command:

```
python setup.py install
```

If you are using `setuptools`, the command is:

```
easy_install statsmodels
```


Performing a normality test with scikits-statsmodels

The scikits-statsmodels package has lots of statistical tests. We will see an example of such a test—the *Anderson-Darling test* for normality (http://en.wikipedia.org/wiki/Anderson%E2%80%93Darling_test).

How to do it...

We will download price data as in the previous recipe; but this time for a single stock. Again, we will calculate the log returns of the close price of this stock, and use that as an input for the normality test function.

This function returns a tuple containing a second element—a *p-value* between zero and one. The complete code for this tutorial is as follows:

```
import datetime
import numpy
from matplotlib import finance
from statsmodels.stats.adnorm import normal_ad
import sys

#1. Download price data

# 2011 to 2012
start = datetime.datetime(2011, 01, 01)
end = datetime.datetime(2012, 01, 01)

print "Retrieving data for", sys.argv[1]
quotes = finance.quotes_historical_yahoo
    (sys.argv[1], start, end, asobject=True)

close = numpy.array(quotes.close).astype(numpy.float)
print close.shape

print normal_ad(numpy.diff(numpy.log(close)))
```

The following shows the output of the script with p-value of 0.13:

```
Retrieving data for AAPL
(252,)
(0.57103805516803163, 0.13725944999430437)
```

How it works...

This recipe demonstrated the Anderson Darling statistical test for normality, as found in `scikits-statsmodels`. We used the stock price data, which does not have a normal distribution, as input. For the data, we got a p-value of 0.13. Since probabilities range between zero and one, this confirms our hypothesis.

Installing `scikits-image`

`scikits image` is a toolkit for image processing, which requires PIL, SciPy, Cython, and NumPy. There are Windows installers available for it. It is part of Enthought Python Distribution, as well as the Python(x, y) distribution.

How to do it...

As usual, we can install using either of the following two commands:

```
pip install -U scikits-image
easy_install -U scikits-image
```

Again, you might need to run these commands as root.

Another option is to obtain the latest development version by cloning the Git repository, or downloading the repository as a zip file from Github. Then, you will need to run the following command:

```
python setup.py install
```

Detecting corners

Corner detection (http://en.wikipedia.org/wiki/Corner_detection) is a standard technique in Computer Vision. `scikits-image` offers a Harris Corner Detector, which is great, because corner detection is pretty complicated. Obviously, we could do it ourselves from scratch, but that would violate the cardinal rule of not reinventing the wheel.

Getting ready

You might need to install `jpeglib` on your system to be able to load the scikits-learn image, which is a JPEG file. If you are on Windows, use the installer; otherwise, download the distribution, unpack it, and build from the top folder with the following command:

```
./configure
make
sudo make install
```

How to do it...

We will load a sample image from scikits-learn. This is not absolutely necessary for this example; you can use any other image instead.

1. Load the sample image.

scikits-learn currently has two sample JPEG images in a dataset structure. We will look at the first image only:

```
dataset = load_sample_images()
img = dataset.images[0]
```

2. Detect corners.

Call the `harris` function to get the coordinates of corners:

```
harris_coords = harris(img)
print "Harris coords shape", harris_coords.shape
y, x = numpy.transpose(harris_coords)
```

The code for the corner detection is as follows:

```
from sklearn.datasets import load_sample_images
from matplotlib.pyplot import imshow, show, axis, plot
import numpy
from skimage.feature import harris

dataset = load_sample_images()
img = dataset.images[0]
harris_coords = harris(img)
print "Harris coords shape", harris_coords.shape
y, x = numpy.transpose(harris_coords)
axis('off')
imshow(img)
plot(x, y, 'ro')
show()
```

We get an image with red dots, where corners are detected, as shown in the following screenshot:



How it works...

We applied the Harris corner detection on a sample image from scikits-image. The result is pretty good, as you can see. We could have done this with NumPy only, since it is just a straightforward, linear-algebra type computation, still it could have become pretty messy. The scikits-image toolkit has a lot more similar functions, so check the scikits-image documentation if you are in need of an image processing routine.

Detecting edges

Edge detection is another popular image processing technique (http://en.wikipedia.org/wiki/Edge_detection). scikits-image has a *Canny filter implementation*, based on the standard deviation of the Gaussian distribution, which can perform edge detection out of the box. In addition to the image data as a 2D array, this filter accepts the following parameters:

- ▶ Standard deviation of the Gaussian distribution
- ▶ Lower bound threshold
- ▶ Upper bound threshold

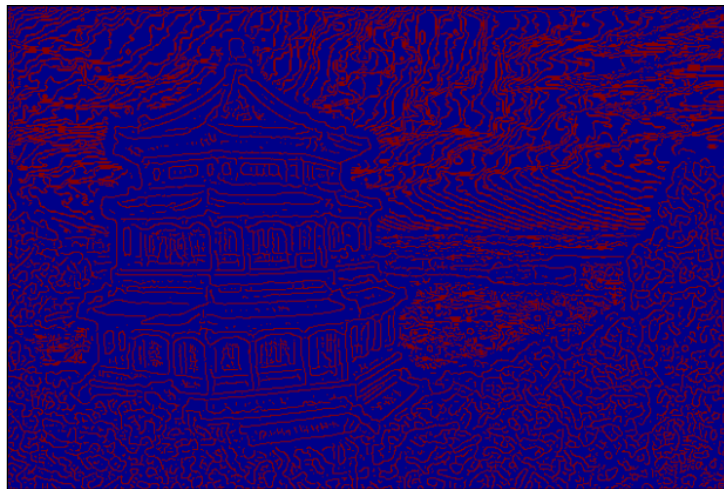
How to do it...

We will use the same image as in the previous recipe. The code is almost the same. You should pay extra attention to the one line where we call the Canny filter function:

```
from sklearn.datasets import load_sample_images
from matplotlib.pyplot import imshow, show, axis
import numpy
import skimage.filter

dataset = load_sample_images()
img = dataset.images[0]
edges = skimage.filter.canny(img[...], 2, 0.3, 0.2)
axis('off')
imshow(edges)
show()
```

The code produces an image of the edges within the original picture, as shown in the following screenshot:



Installing Pandas

Pandas is a Python library for data analysis. It has some similarities with the R programming language, which are not coincidental. R is a specialized programming language popular with data scientists. For instance, the core `DataFrame` object is inspired by R.

How to do it...

On *PyPi*, the project is called `pandas`. So, for instance, run either of the following two command:

```
sudo easy_install -U pandas
pip install pandas
```

If you are using a Linux package manager, you will need to install the `python-pandas` project. On Ubuntu, you would do the following:

```
sudo apt-get install python-pandas
```

You can also install from source (requires Git):

```
git clone git://github.com/pydata/pandas.git
cd pandas
python setup.py install
```

Estimating stock returns correlation with Pandas

A Pandas `DataFrame` is a matrix and dictionary-like data structure similar to the functionality available in R. In fact, it is the central data structure in Pandas and you can apply all kinds of operations on it. It is quite common to have a look, for instance, at the correlation matrix of a portfolio. So let's do that.

How to do it...

First, we will create the `DataFrame` with Pandas for each symbol's daily log returns. Then we will join these on the date. At the end, the correlation will be printed, and plot will be shown.

1. Creating the data frame.

To create the data frame, we will create a dictionary containing stock symbols as keys, and the corresponding log returns as values. The data frame itself has the date as index and the stock symbols as column labels:

```
data = {}

for i in xrange(len(symbols)):
    data[symbols[i]] = numpy.diff(numpy.log(close[i]))

df = pandas.DataFrame(data,
    index=dates[0][:-1], columns=symbols)
```

2. Operating on the data frame.

We can now perform operations, such as calculating a correlation matrix or plotting on the data frame:

```
print df.corr()
df.plot()
```

The complete source code that also downloads the price data is as follows:

```
import pandas
from matplotlib.pyplot import show, legend
from datetime import datetime
from matplotlib import finance
import numpy

# 2011 to 2012
start = datetime(2011, 01, 01)
end = datetime(2012, 01, 01)

symbols = ["AA", "AXP", "BA", "BAC", "CAT"]

quotes = [finance.quotes_historical_yahoo
          (symbol, start, end, asobject=True)
          for symbol in symbols]

close = numpy.array([q.close for q in quotes])
        .astype(numpy.float)
dates = numpy.array([q.date for q in quotes])

data = {}

for i in xrange(len(symbols)):
    data[symbols[i]] = numpy.diff(numpy.log(close[i]))

df = pandas.DataFrame(data,
                      index=dates[0][:-1], columns=symbols)

print df.corr()
df.plot()
legend(symbols)
show()
```

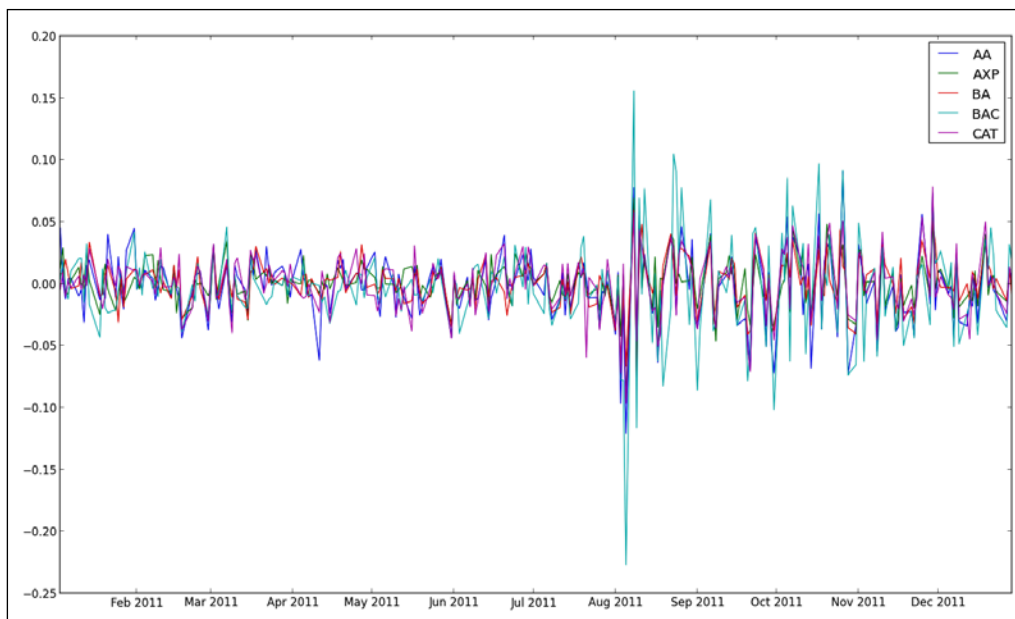
Output for the correlation matrix:

```

      AA      AXP      BA      BAC      CAT
AA  1.000000  0.768484  0.758264  0.737625  0.837643
AXP 0.768484  1.000000  0.746898  0.760043  0.736337
BA  0.758264  0.746898  1.000000  0.657075  0.770696
BAC 0.737625  0.760043  0.657075  1.000000  0.657113
CAT 0.837643  0.736337  0.770696  0.657113  1.000000

```

The following image shows the plot for the log returns of the five stocks:



How it works...

We used the following `DataFrame` methods:

| Method | Description |
|------------------------------------|--|
| <code>pandas.DataFrame</code> | Constructs <code>DataFrame</code> with specified data, index (row), and column labels. |
| <code>pandas.DataFrame.corr</code> | Computes pair-wise correlation of columns, ignoring the missing values. By default, Pearson correlation is used. |
| <code>pandas.DataFrame.plot</code> | Plots the data frame with Matplotlib. |

Loading data as pandas objects from statsmodels

Statsmodels has quite a lot of sample datasets in its distributions. The complete list can be found at <https://github.com/statsmodels/statsmodels/tree/master/statsmodels/datasets>.

In this tutorial, we will concentrate on the copper dataset, which contains information about copper prices, world consumption, and other parameters.

Getting ready

Before we start, we might need to install *patsy*. It is easy enough to see if this is necessary just run the code. If you get errors related to *patsy*, you will need to execute any one of the following two commands:

```
sudo easy_install patsy
pip install --upgrade patsy
```

How to do it...

In this section, we will see how we can load a dataset from statsmodels as a Pandas DataFrame or Series object.

1. Loading the data.

The function we need to call is `load_pandas`. Load the data as follows:

```
data = statsmodels.api.datasets
      .copper.load_pandas()
```

This loads the data in a `DataSet` object, which contains pandas objects.

2. Fitting the data.

The `DataSet` object has an attribute `exog`, which when loaded as a pandas object, becomes a `DataFrame` object with multiple columns. It also has an `endog` attribute containing values for the world consumption of copper in our case.

Perform an ordinary least squares calculation by creating an `OLS` object, and calling its `fit` method as follows:

```
x, y = data.exog, data.endog

fit = statsmodels.api.OLS(y, x).fit()
print "Fit params", fit.params
```

This should print the result of the fitting procedure, as follows:

```
Fit params COPPERPRICE      14.222028
INCOMEINDEX      1693.166242
ALUMPRICE        -60.638117
INVENTORYINDEX   2515.374903
TIME             183.193035
```

3. Summarize.

The results of the OLS fit can be summarized by the `summary` method as follows:

```
print fit.summary()
```

This will give us the following output for the regression results:

| OLS Regression Results | | | | | | |
|--|------------------|---------------------|----------|-------|--------------------|----------|
| Dep. Variable: | WORLDCONSUMPTION | R-squared: | 0.978 | | | |
| Model: | OLS | Adj. R-squared: | 0.974 | | | |
| Method: | Least Squares | F-statistic: | 224.9 | | | |
| Date: | Fri, 28 Sep 2012 | Prob (F-statistic): | 2.55e-16 | | | |
| Time: | 20:25:26 | Log-Likelihood: | -172.62 | | | |
| No. Observations: | 25 | AIC: | 355.2 | | | |
| Df Residuals: | 20 | BIC: | 361.3 | | | |
| Df Model: | 4 | | | | | |
| | coef | std err | t | P> t | [95.0% Conf. Int.] | |
| COPPERPRICE | 14.2220 | 12.090 | 1.176 | 0.253 | -10.998 | 39.442 |
| INCOMEINDEX | 1693.1662 | 1970.555 | 0.859 | 0.400 | -2417.339 | 5803.671 |
| ALUMPRICE | -60.6381 | 32.023 | -1.894 | 0.073 | -127.437 | 6.161 |
| INVENTORYINDEX | 2515.3749 | 1670.948 | 1.505 | 0.148 | -970.162 | 6000.912 |
| TIME | 183.1930 | 36.879 | 4.967 | 0.000 | 106.264 | 260.122 |
| Omnibus: | 8.007 | Durbin-Watson: | 1.316 | | | |
| Prob(Omnibus): | 0.018 | Jarque-Bera (JB): | 6.014 | | | |
| Skew: | -0.936 | Prob(JB): | 0.0494 | | | |
| Kurtosis: | 4.506 | Cond. No. | 2.20e+03 | | | |
| The condition number is large, 2.2e+03. This might indicate that there are strong multicollinearity or other numerical problems. | | | | | | |

The code to load the copper data set is as follows:

```
import statsmodels.api

# See https://github.com/statsmodels
# /statsmodels/tree/master/statsmodels/datasets
data = statsmodels.api.datasets.copper.load_pandas()

x, y = data.exog, data.endog

fit = statsmodels.api.OLS(y, x).fit()
print "Fit params", fit.params
print
print "Summary"
print
print fit.summary()
```

How it works...

The data in the `Dataset` class of `statsmodels` follows a special format. Among others, this class has the `endog` and `exog` attributes. `Statsmodels` has a `load` function, which loads data as NumPy arrays. Instead, we used the `load_pandas` method, which loads data as Pandas objects. We did an OLS fit, basically giving us a statistical model for copper price and consumption.

Resampling time series data

In this tutorial, we will learn how to resample time series with Pandas.

How to do it...

We will download the daily price time series data for AAPL, and resample it to monthly data by computing the mean. We will accomplish this by creating a Pandas `DataFrame`, and calling its `resample` method.

1. Creating a date-time index.

Before we can create a Pandas `DataFrame`, we need to create a `DatetimeIndex` method to pass to the `DataFrame` constructor. Create the index from the downloaded quotes data as follows:

```
dt_idx = pandas.DatetimeIndex
         (quotes.date)
```

2. Creating the data frame.

Once we have the date-time index, we can use it together with the close prices to create a data frame:

```
df = pandas.DataFrame
    (quotes.close, index=dt_idx,
     columns=[symbol])
```

3. Resample.

Resample the time series to monthly frequency, by computing the mean:

```
resampled = df.resample
            ('M', how=np.mean)
print resampled
```

The resampled time series, as shown in the following, has one value for each month:

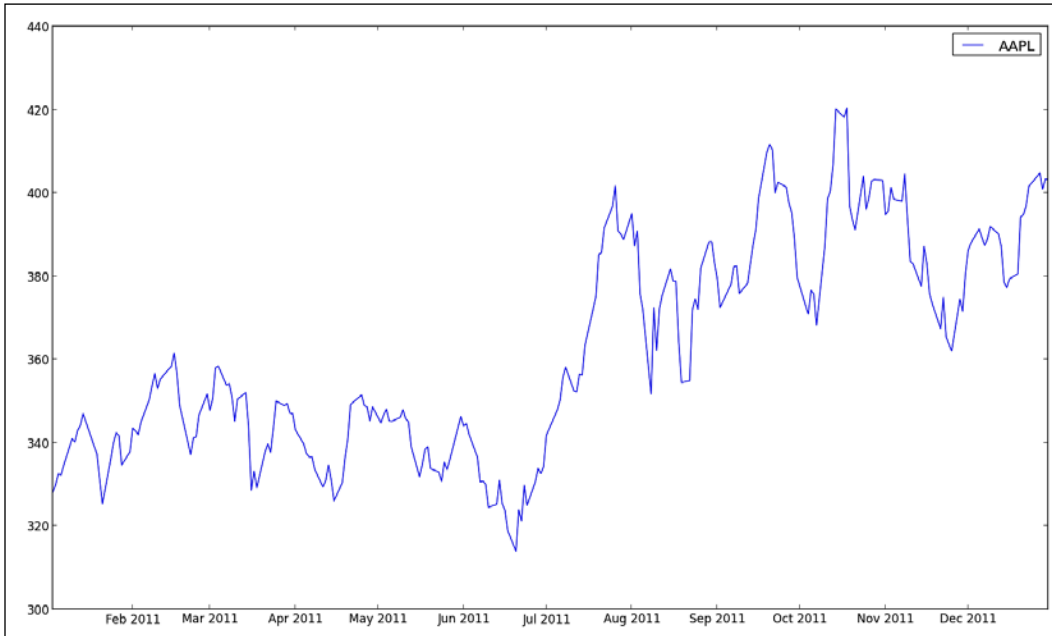
```
                AAPL
2011-01-31  336.932500
2011-02-28  349.680526
2011-03-31  346.005652
2011-04-30  338.960000
2011-05-31  340.324286
2011-06-30  329.664545
2011-07-31  370.647000
2011-08-31  375.151304
2011-09-30  390.816190
2011-10-31  395.532381
2011-11-30  383.170476
2011-12-31  391.251429
```

4. Plot.

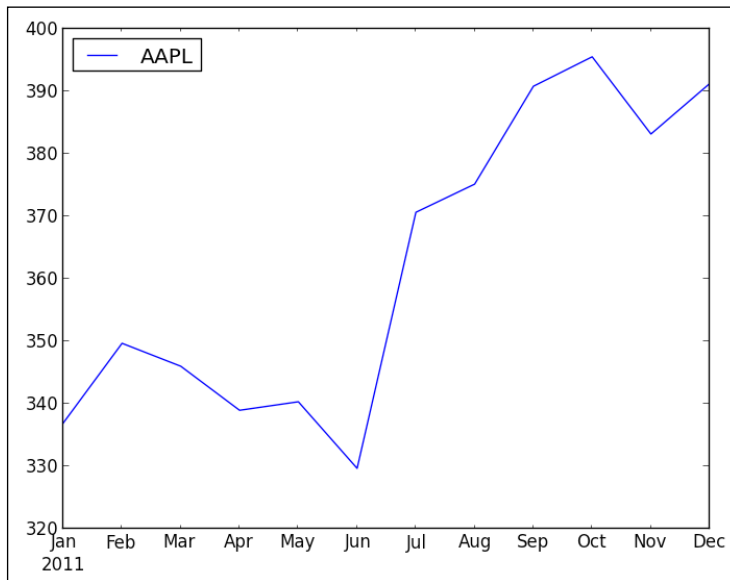
Use the DataFrame `plot` method to plot the data:

```
df.plot()
resampled.plot()
show()
```

The plot for the original time series is as follows:



The resampled data has less data points, and therefore, the resulting plot, as shown in the following image, is choppy:



The complete resampling code is as follows:

```
import pandas
from matplotlib.pyplot import show, legend
from datetime import datetime
from matplotlib import finance
import numpy

# Download AAPL data for 2011 to 2012
start = datetime(2011, 01, 01)
end = datetime(2012, 01, 01)

symbol = "AAPL"
quotes = finance.quotes_historical_yahoo(
    symbol, start, end, asobject=True)

# Create date time index
dt_idx = pandas.DatetimeIndex(quotes.date)

# Create data frame
df = pandas.DataFrame(quotes.close,
    index=dt_idx, columns=[symbol])

# Resample with monthly frequency
resampled = df.resample('M', how=numpy.mean)
print resampled

# Plot
df.plot()
resampled.plot()
show()
```

How it works...

We created a date-time index from a list of date and times. This index was then used to create a Pandas data frame. We then resampled our time series data. The resampling frequency is given by a single character:

- ▶ D for daily
- ▶ M for monthly
- ▶ A for annual

The `how` parameter of the `resample` method indicates how the data is sampled. This defaults to calculating the mean.

Index

`__array_interface__` attribute 79

[C], message type 155

[E], message type 155

[F], message type 155

%hist command 9

@profile decorator 144

.pyx file

about 170

writing 172-174

[R], message type 155

% Time column 144

[W], message type 155

A

AAPL (Apple Inc.) 61

AffinityPropagation class 186, 187

Anderson-Darling test

URL 190

append function 138

Apple Developer Tools (Xcode)

installing 6

arange function 13, 50, 51, 53

array interface

using 79, 80

asarray function 79

assert_almost_equal function 161

assert_approx_equal function 161

assert_array_almost_equal function 161

assert_array_equal function 161

assert_array_less function 161

assert_raises function 161

assert_string_equal function 161

assert_warns function 161

astype function 50-53

audio filter

designing 114, 115

audio fragments

repeating 108-110

axis function 100

B

BDD

about 151

steps, defining 166

tests, setting up 165

used, for testing 164

way, used for testing 164-167

Behavior Driven Development. *See* BDD

binomial proportion confidence

wikipedia, URL 172

boolean indexing 40-42

broadcasting arrays 45, 46

Browse button 89

bt command 148

buffer interface 76

buffer protocol 76-78

Butterworth bandpass filter

wikipedia, URL 114

C

Canny filter function 194

cdef keyword 181

ceil function 54, 56

C functions

.pyx file, writing 174

calling 173, 175

log returns, plotting 174

chararray object

about 124

string operations. performing 124, 125

Choose function 104

cimport 181
clip function 100
clustering 185
code
 profiling, with cProfile extension 144, 145
 testing, docstrings used 156-158
 testing, mocks used 162, 163
Command Line Interface (CLI) 10
compress function 67
concatenate function 113
corner detection
 about 191, 192
 wikipedia, URL 191
cProfile extension
 code, profiling with 144, 145
create a new environment button 92
cumprod function 180
cumtime column 141
Cython
 about 169
 factorials, approximating 178-181
 installing 170
 installing, from tarball 170
 installing, ways for 170
 installing, with pip 170
 installing, with setup tools 170
 installing, with Windows installers 170
 using, with NumPy 172, 173
Cython approximation
 of e 177
Cython code
 profiling 175-178

D

daily log returns
 wikipedia, URL 67
data
 exchanging, with MATLAB 80, 81
 exchanging, with Octave 80, 81
 loading, as pandas objects, from statsmodels 198-200
DataFrame 195
DataFrame object 198
DataFrame plot method 201
Dataset class 200
Dataset object 198

DatetimeIndex method 200
d command 148
Debian
 PIL, installing 28
debugging 135
diff function 63, 174, 189
dips
 periodically, training on 67-69
distutils setup.py script 171
docstrings
 code, testing with 156-158
dot function 189
Dow Jones stocks
 clustering, with scikits-learn project 185-189
do_work method 162

E

easy_install
 IPython, installing with 7
 used, for installing PIL 28
 used, for installing Pyflakes 152
 used, for installing RPy2 82
 used, for installing scikits-learn project 184
 used, for installing SciPy 27
edge detection
 about 193, 194
 wikipedia, URL 193
eigenvector
 URL 58
eig function 61, 63
endog attribute 198, 200
Euclid's formula 123
Euler constant
 URL 175
exog attribute 200
extreme values
 ignoring 128-131

F

factorial method 163
factorials approximation
 code, building 179
 Cython code, writing 179
 error, plotting 180
 with Cython 178, 179

fancy indexing 36-38

Fermat's factorization method

Wikipedia, URL 54

Fibonacci series

about 50

wikipedia, URL 50

fit method 198

G

GAE

about 86

development environment 87

downloading 87

installing, steps for 87

Gaussian filter

wikipedia, URL 104

gaussian_filter function 107

generate function 111

get_include function 181

golden ratio

wikipedia, URL 50

Google App Engine. See GAE

Google cloud

NumPy code, deploying 88, 89

H

harris function 192

hello.pyx code

writing 171

Hello World program

building 170

building, steps for 171

distutils setup.py script 171

hello.pyx code, writing 171

help command 10

histogram function 66

Hits column 144

how parameter 203

I

iirdesign function 114

images

blurring 104-107

loading, into memory map 96-100

merging 100-104

resizing 29-32

importr function 83

indexing

fancy indexing 36-38

with booleans 40-42

with list of locations 38, 39

Infinite Impulse Response (IIR) 114

inline figures

web notebook, running with 13

installing

Apple Developer Tools (Xcode) 6

cikits-learn project installing, easy_install used 184

Cython 170

Cython, pip used 170

Cython, setup tools used 170

Cython, tarball (.tar archive) used 170

Cython, ways for 170

Cython, with Windows installers 170

GAE 87

IPython, from source 7

IPython, on Linux 6

IPython, on Mac OS X 6

IPython, on Windows 6

IPython, pip used 7

IPython, steps for 6

IPython, with easy_install 7

JPyPe 84

line_profiler, with easy_install 142

Matplotlib 11

Matplotlib, from source 11

Matplotlib, on Linux 11

Matplotlib, on Mac 11

Matplotlib, on Windows 11

Mocks 162

Octave, pointers for 81

Pandas 195

Patsy 198

PIL 28

PIL, on Debian 28

PIL, on Ubuntu 28

PIL, on Windows 28

PIL, pip used 28

PIL, with easy_install 28

PIL, with rip 28

Pudb 148

Pyflakes, easy_install used 152

- Pyflakes, in Linux 152
- Pyflakes, pip used 152
- Pylint, from source distribution 154
- RPy2 82
- RPy2, from source 82
- RPy2, steps for 82
- RPy2, easy_install used 82
- RPy2, pip used 82
- scikits-learn project, from source 184
- scikits-learn project, with easy_install 184
- SciPy, easy_install used 27
- SciPy, from source 26
- SciPy, on Linux 27
- SciPy, on Mac OS X 27
- SciPy, on Windows 27
- SciPy, pip used 27
- SciPy, rip used 27
- setup tools, on Windows 6
- SymPy profile, easy_install used 23, 24
- SymPy profile, pip used 23, 24

interoperability 75

IPython

- about 5
- components 5
- debugging with 146-148
- in cloud URL 6
- installing, from source 7
- installing, on Linux 6
- installing, on Mac OS X 6
- installing, on Windows 6
- installing, steps for 6
- installing, with easy_install 7
- installing, with pip 7
- profiling with 139-141
- URL 5
- using, as shell 8, 10

IPython, installing

- from source 7
- on Linux 6
- on Mac OS X 6
- on Windows 6
- with easy_install 7
- with pip 7

IPython shell

- history, displaying 9
- pylab switch 8
- session, saving 9

- system shell command, executing 9
- using, ways for 8

J

Java Native Interface (JNI) 86

JPy

- about 84
- building 84
- installing, steps for 84
- NumPy array, sending to 84-86
- URL, for downloading 84

Jython 84

L

l command 148

lena function 29

lena image

- flipping 34, 35

linalg module 61

Line # column 144

Line Contents column 144

line_profiler

- about 142
- code, profiling with 143, 144
- development version, installing 142
- installing, with easy_install 142

linspace function 104

Linux

- IPython, installing 6
- Matplotlib, installing 11
- Pyflakes, installing 152
- Scipy, installing 27

list command 147, 148

load function 200

load_pandas method 200

locations list

- used, for indexing 38, 39

log function 50, 53, 173, 189

M

Mac

- Matplotlib, installing 11

Mac OS X

- IPython, installing 6
- Scipy, installing 27

MainHandler class 89

Mandelbrot fractal

wikipedia, URL 100

manual pages

reading 10, 11

Markov chain 58

masked array

creating 125

creating, steps for 126, 127

MaskedArray function 125

MATLAB

data, exchanging with 80, 81

Matplotlib, installing

from source 11

on Linux 11

on Mac 11

on Windows 11

memmap function 98, 100

memory map

images, loading into 96-100

loading into 98

merge sort algorithm 135

meshgrid function 97, 100, 104

message type

[C] 155

[E] 155

[F] 155

[R] 155

[W] 155

mocks

asset behavior 162

creating 162

installing 162

used, for testing code 162, 163

modf function 54, 56

mtcars dataset 83

N

ncalls column 141

ndarray class 42

ndarray module 79

negative values

ignoring 128-131

notebook server

about 12

configuring, steps for 20-22

password, generating 20

profile configuration file, editing 21

server profile, creating 21

SSL certificate, creating 20, 21

starting 21

NumPy

.pyx file, writing 172

about 25

Cython module, using 173

Cython, using with 172, 173

diff function 174

log function 173

setup.py file, writing 172

NumPy approximation

of e 176

NumPy array

sending, to Jype 84-86

NumPy code

deploying, in Google cloud 88, 89

running, in Python Anywhere web console

90-92

numpy.ma module 125

numpy.random module 72

numpy.recarray module 133

numpy.testing.assert_equal function 161

numpy.tile function 110

O

Octave

data, exchanging with 80, 81

pointers, for installing 81

O(nlogn) 135

outer function 57, 58, 123

outer method 124

P

palindromic numbers 56, 58

Pandas

about 194

installing 195

stock returns correlation, estimating 195

time series data, resampling 200-203

pandas.DataFrame.corr method 197

pandas.DataFrame method 197

pandas.DataFrame.plot method 197

Pareto principle

URL 64

patsy

installing 198

percall column 141**Per Hit column 144****PiCloud**

about 92, 93

URL 92

working 93

PIL

about 76

installing, on Debian 28

installing, on Ubuntu 28

installing, on Windows 28

installing, with easy_install 28

installing, with rip 28

pip

IPython, installing with 7

used, for installing Cython 170

used, for installing PIL 28

used, for installing Pyflakes 152

used, for installing RPy2 82

used, for installing SciPy 27

polar function 107**polar rose**

wikipedia, URL 104

polyfit function 66, 138**polyval function 138****power law**

discovering 64-66

prime factors

finding 54-56

Wikipedia, URL 54

profiler output

cumtime column 141

ncalls column 141

percall column 141

tottime column 141

profiling

script, profiling 139

snippet, timing 139

with IPython 139

pdb

debugging with 148, 149

installing 148

py26-scikits-learn 184**py27-scikits-learn 184****Pychecker**

static analysis, performing 155

Pyflakes

about 151

installing, on Linux 152

installing, with easy_install 152

installing, with pip 152

static analysis, performing 152, 153

pylab mode

web notebook, running 13

pylab switch 8**PyLint**

about 153

installing, from source distribution 154

working 155

Pythagorean triples

finding 122, 124

wikipedia, URL 122

Python Anywhere web console

NumPy code, running 90-92

Python Image Library (PIL) 26**python-pandas project 195****python-sklearn 184****Q**

quicksort algorithm 135

R**R**

interfacing with 82-84

rand function 72**randint function 72, 100****randn function 72****random**

trading, simulating at 70, 72

random_integers function 100, 107, 138**ravel function 55, 56****recarray**

score tables, creating with 131-133

recarray class 131**repeat function 32****resample method 200****RPy2**

installing 82

installing, from source 82

installing, steps for 82
installing, with easy_install 82
installing, with pip 82

S

savemat function 80

Scikits 183

scikits-image 191

scikits-learn project

about 184

Dow Jones stocks, clustering 185-189

example dataset, loading 184

installing, from source 184

installing, with easy_install 184

Scikits, projects

pandas 183

scikits-image 183

scikits-learn 183

scikits-statsmodels 183

scikits-statsmodels package

about 189

binaries downloading, URL 189

normality test, performing 190, 191

source downloading, URL 189

SciPy

about 26

installation, checking 27

installing, easy_install used 27

installing, from source 26

installing, on Linux 27

installing, on Mac OS X 27

installing, on Windows 27

installing, rip used 27

scipy.io.wavfile.read function 110

scipy.io.wavfile.write function 110

scipy.signal.iirdesign function 116

Scipy.signal.lfilter function 116

score tables

creating, with recarray 131-133

semilogx function 138

server profile

creating 21

setup.py file

writing 172

setup tools

installing, on Windows 6

used, for installing Cython 170

shell

IPython, using as 8

Sieve of Erasthothenes

integers, sieving with 72

URL 72

sign function 63

sinc function

plotting 14

sklearn.cluster.AffinityPropagation.fit function

189

sklearn.cluster.AffinityPropagation() function

189

Sobel filter

edge detection with 117-119

Sobel operator

URL 117

sort method 57

sounds

generating 110, 112

splitlines function 125

sqrt function 50, 53

SSL certificate

creating 20, 21

standard deviation of log returns

wikipedia, URL 131

static analysis

performing, with Pyflakes 152-155

statsmodels

data, loading as pandas objects 198-200

steady state

about 59, 60

URL 58

stirling approximation method

URL 178

stochastic matrix

URL 58

stride tricks

for Sudoku 42-44

string operations

performing, with chararray 124

performing, with chararray object 124, 125

subplot function 30

Sudoku

stride tricks for 42-44

wikipedia, URL 42

sum function 50, 53, 189

SymPy profile

- exploring 23
- installing, easy_install used 23, 24
- installing, pip used 23, 24

T

take function 55, 56

tarball (.tar archive)

- used, for installing Cython 170

TDD 158

TestCase class 159

Test-driven development. See TDD

Time column 144

timeit

- about 135
- append function 138
- arrays, creating to sort 136
- measurement time arrays, building 136
- measure time 136
- nlogn, fitting to 136
- polyfit function 138
- polyval function 138
- random_integers function 138
- semilogx function 138

time series data

- resampling, with Pandas 200-203

tofile function 98, 100

tottime column 141

trading

- simulating, at random 70, 72

tuple 185

U

Ubuntu

- PIL, installing 28

u command 148

Ufuncs. See universal function

unittest.assertEqual function 161

unittest.assertRaises function 161

unit tests

- about 158
- implementing, standard unittest Python module used 161

- writing 159, 160

universal function

- about 121
- creating 121, 122
- creating, steps for 122

U state 59

V

views

- creating 32-34

Vi (m) editor 6

W

web notebook

- array, creating 13
- creating 13
- downloading 15
- exporting 14
- exporting, print option used 14
- features 12
- importing, steps for 16-19
- running 12
- running, in pylab mode 13
- running, with inline figures 13
- saving 16
- sinc function, plotting 14

web notebook, exporting

- via download button 15
- via print option 14
- via save notebook option 16

where function 54, 56

Windows

- Matplotlib, installing 11
- PIL, installing 28
- SciPy, installing 27
- setup tools, installing 6

Windows installers

- used, for installing Cython 170

Z

zeros function 60, 100



Thank you for buying NumPy Cookbook

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

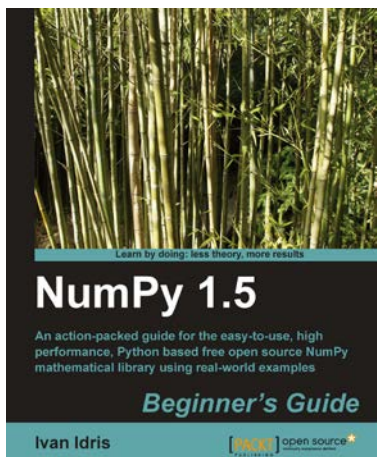
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



NumPy 1.5 Beginner's Guide

ISBN: 978-1-84951-530-6 Paperback: 234 pages

An action-packed guide for the easy-to-use, high performance, Python based free open source NumPy mathematical library using real-world examples

1. The first and only book that truly explores NumPy practically
2. Perform high performance calculations with clean and efficient NumPy code
3. Analyze large data sets with statistical functions
4. Execute complex linear algebra and mathematical computations



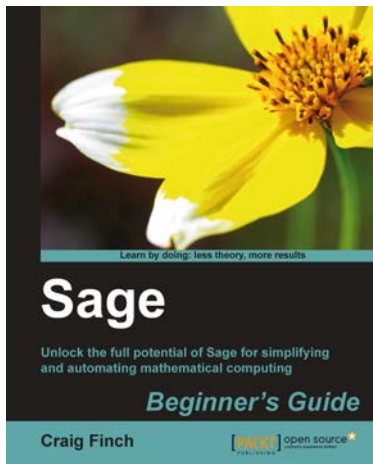
Matplotlib for Python Developers

ISBN: 978-1-84719-790-0 Paperback: 308 pages

Build remarkable publication-quality plots the easy way

1. Create high quality 2D plots by using Matplotlib productively
2. Incremental introduction to Matplotlib, from the ground up to advanced levels
3. Embed Matplotlib in GTK+, Qt, and wxWidgets applications as well as web sites to utilize them in Python applications

Please check www.PacktPub.com for information on our titles

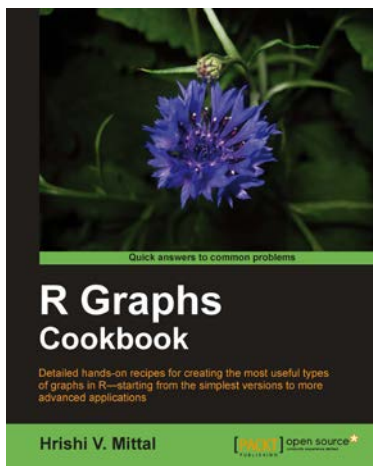


Sage Beginner's Guide

ISBN: 978-1-84951-446-0 Paperback: 364 pages

Unlock the full potential of Sage for simplifying and automating mathematical computing

1. The best way to learn Sage which is an open source alternative to Magma, Maple, Mathematica, and Matlab
2. Learn to use symbolic and numerical computation to simplify your work and produce publication-quality graphics
3. Numerically solve systems of equations, find roots, and analyze data from experiments or simulations



R Graph Cookbook

ISBN: 978-1-84951-306-7 Paperback: 272 pages

Detailed hands-on recipes for creating the most useful types of graphs in R—starting from the simplest versions to more advanced applications

1. Learn to draw any type of graph or visual data representation in R
2. Filled with practical tips and techniques for creating any type of graph you need; not just theoretical explanations
3. All examples are accompanied with the corresponding graph images, so you know what the results look like
4. Each recipe is independent and contains the complete explanation and code to perform the task as efficiently as possible

Please check www.PacktPub.com for information on our titles