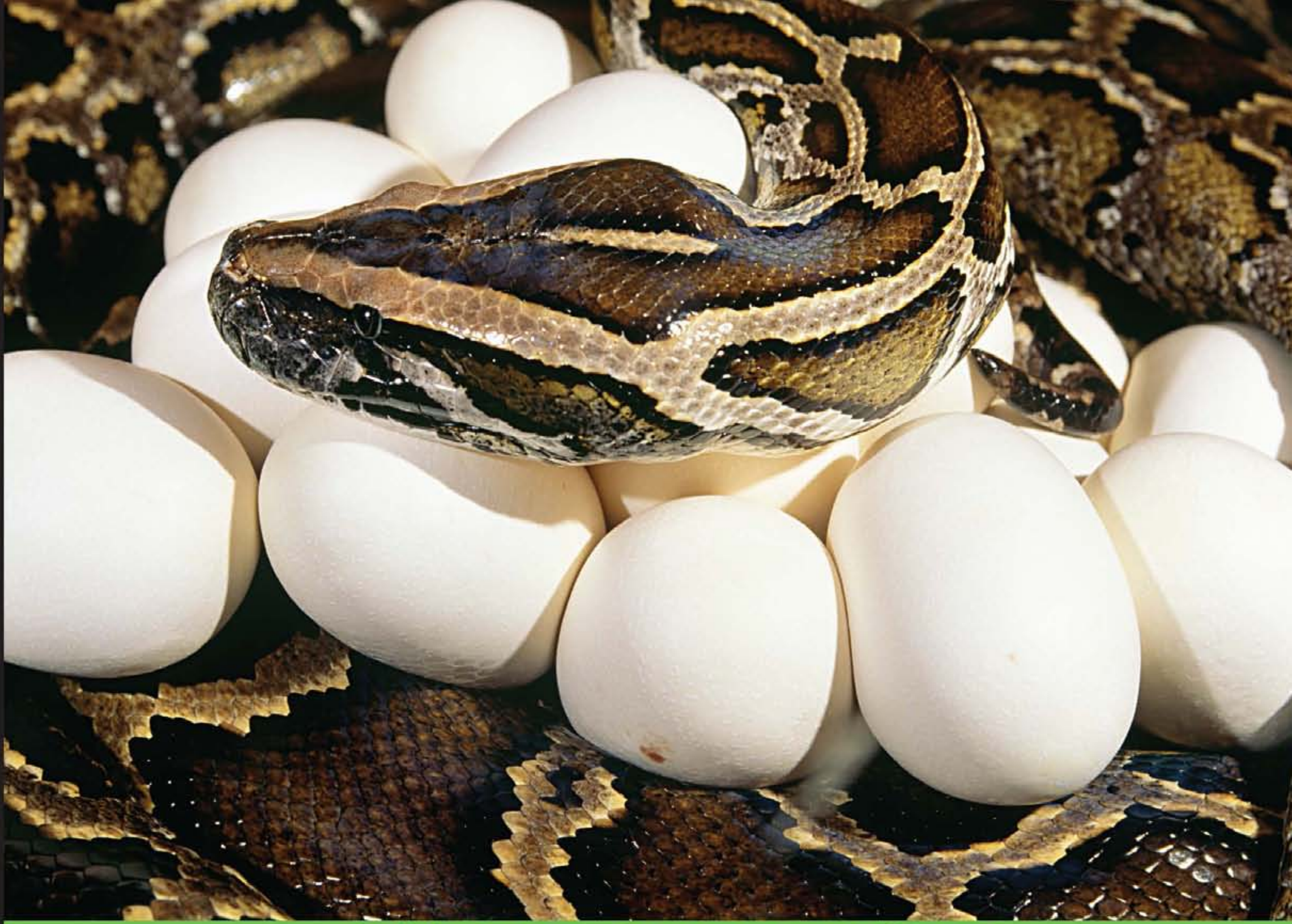


Quick answers to common problems

NumPy Cookbook

Second Edition

Over 90 fascinating recipes to learn and perform mathematical, scientific, and engineering Python computations with NumPy

Ivan Idris

[PACKT] PUBLISHING open source*
community experience distilled

NumPy Cookbook
Second Edition

Over 90 fascinating recipes to learn and perform mathematical, scientific, and engineering Python computations with NumPy

Ivan Idris

[PACKT] open source
PUBLISHING
community experience distilled

BIRMINGHAM - MUMBAI

NumPy Cookbook
Second Edition

First published: October 2012

Second edition: April 2015

Production reference: 1270415

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

ISBN 978-1-78439-094-5

www.packtpub.com

Credits

Author
Ivan Idris

Reviewers
Lev E. Givon
Mark Livingstone
Lijun Xue

Commissioning Editor
Kartikey Pandey

Acquisition Editors
Nadeem N. Bagban
Owen Roberts

Content Development Editor
Parita Khedekar

Technical Editors
Utkarsha S. Kadam
Shiny Poojary

Copy Editor
Vikrant Phadke

Project Coordinator
Rashi Khivansara

Proofreaders
Maria Gould
Clyde Jenkins

Indexer
Monica Ajmera Mehta

Graphics
Abhinash Sahu

Production Coordinator
Shantanu N. Zagade

Cover Work
Shantanu N. Zagade

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. Ivan enjoys writing clean, testable code and interesting technical articles. He is the author of *NumPy Beginner's Guide*, *NumPy Cookbook*, *Python Data Analysis*, and *Learning NumPy*, all by Packt Publishing. You can find more information about him and a few NumPy examples at `http://ivanidris.net/wordpress/`.

I would like to take this opportunity to thank the reviewers and the team at Packt Publishing for making this book possible. Also, thanks 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

Lev E. Givon is a doctoral candidate and neurocomputing researcher at the department of electrical engineering in Columbia University, New York. His research focuses on developing computational tools and techniques to study information processing and representation by neural circuits in the brain of the fruit fly. He is one of the developers of Neurokernel (`http://neurokernel.github.io`), an open software framework written in Python for the emulation of the fruit fly brain on multiple graphics processing units.

Mark Livingstone started his career by working for many years in three international computer companies (which no longer exist) in engineering, support, programming, and training roles. He got tired of being made redundant. He then graduated from Griffith University, Gold Coast, Australia, in 2011 with a bachelor's in information technology. In 2013, Mark received a B.InfoTech (Hons) degree. He is currently a PhD candidate, with his confirmation rapidly approaching. All of his research software is written in Python on a Mac system.

Mark enjoys mentoring students with special needs. He was the chairman of IEEE in Griffith University's Gold Coast Student Branch. He volunteers as a qualified justice of peace at the local district courthouse. He is also a credit union director, and has completed 105 blood donations.

Lijun Xue is a developer of Theano, which is a Python library that allows you to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently. He was a research assistant at Carnegie Mellon University doing research projects related to machine learning and data mining. He is a Pythonista and has passion towards machine learning and data mining. He is currently working on some deep learning research projects, which aims to solve image classification problems in university. You can know more about him at `http://royxue.me/`.

Table of Contents

Preface

This second edition adds two new chapters on the new NumPy functionality and data analysis. We NumPy users live in exciting times. New NumPy-related developments seem to come to our attention every week, or maybe even daily. At the time of the first edition, the NumFocus, short for NumPy Foundation for Open Code for Usable Science, was created. The Numba project—a NumPy-aware dynamic Python compiler using LLVM—was also announced. Further, Google added support to their cloud product called 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 the Python ecosystem, languages such as R, C, and Fortran are pretty popular. We will cover the details of exchanging data with these environments.

What this book covers

Chapter 1, *Winding Along with IPython*, introduces IPython, a toolkit mostly known for its shell. The web-based notebook is an exciting feature covered in detail here. Think of MATLAB and Mathematica, but in your browser, it's open source and free.

Chapter 2, *Advanced Indexing and Array Concepts*, shows that NumPy has very efficient arrays that are easy to use due to the powerful indexing mechanism. This chapter describes some of the more advanced and tricky indexing techniques.

Chapter 3, *Getting 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*, the number of programming languages, libraries, and tools one encounters in the real world is mind-boggling. Some of the software runs on the cloud, while some of it lives on your local machine or a remote server. Being able to fit and connect NumPy with such an environment is just as important as being able to write standalone NumPy code.

Chapter 5, *Audio and Image Processing*, assumes that when you think of NumPy, you probably don't think of sounds or images. This will change after reading this chapter.

Chapter 6, *Special Arrays and Universal Functions*, introduces pretty technical topics. This chapter explains how to perform string operations, ignore illegal values, and store heterogeneous data.

Chapter 7, *Profiling and Debugging*, shows the skills necessary to produce good software. We demonstrate several convenient profiling and debugging tools.

Chapter 8, *Quality Assurance*, deserves a lot of attention because it's about quality. We discuss common methods and techniques, such as unit testing, mocking, and BDD, using the NumPy testing utilities.

Chapter 9, *Speeding Up Code with Cython*, introduces Cython, which tries to combine the speed of C and the strengths of Python. We show you how Cython works from the NumPy perspective.

Chapter 10, *Fun with Scikits*, covers Scikits, which are a yet another part of the fascinating scientific Python ecosystem. A quick tour guides you through some of the most useful Scikits projects.

Chapter 11, *Latest and Greatest NumPy*, showcases new functionality not covered in the first edition.

Chapter 12, *Exploratory and Predictive Data Analysis with NumPy*, presents real-world analysis of meteorological data. I've added this chapter in the second edition.

What you need for this book

To try 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 for installing other relevant software packages are provided throughout this book.

Who this book is for

This book is for scientists, engineers, programmers, or analysts with basic knowledge of Python and NumPy, who want to go to the next level. Also, some affinity—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, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
from __future__ import print_function
from matplotlib.finance import quotes_historical_yahoo
from datetime import date
import numpy as np
import matplotlib.pyplot as plt

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

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

```
from sklearn.datasets import load_sample_images
import matplotlib.pyplot as plt
import skimage.feature

dataset = load_sample_images()
img = dataset.images[0]
edges = skimage.feature.canny(img[..., 0])
plt.axis('off')
plt.imshow(edges)
plt.show()
```

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

```
$ sudo easy_install patsy
```

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: "The **Print** button doesn't actually print the notebook."

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or 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 via 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 from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/0945OS.pdf`.

Errata

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

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

Piracy

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

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

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

Questions

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

1
Winding Along with IPython

In this chapter, we will cover the following recipes:

- Installing IPython
- Using IPython as a shell
- Reading manual pages
- Installing matplotlib
- Running an IPython notebook
- Exporting an IPython 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. IPython provides an architecture for interactive computing. The most notable part of this project is the IPython shell. IPython 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, 3.2, 3.3, and 3.4. The compatibility depends on the IPython version. For instance, IPython 2.3.0 requires Python 2.7 or 3.3+.

You can try IPython in the cloud without installing it on your system by going to `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. PythonAnywhere also has a Vi (m) editor, which if you like vi, is obviously great. You can save and edit files from your IPython sessions.

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 a popular `package` manager for Python. `pip` can be installed once you have `easy_install`. 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 setuptools 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`, like this:

  ```
  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 at `https://developer.apple.com/xcode/`. Follow the `easy_install/pip` instructions or the instructions for installation from source provided later in this section.
- **Installing IPython on Linux**: Since 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:

  ```
  $ sudo easy_install ipython pyzmq tornado readline
  ```

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

  ```
  $ sudo easy_install pip
  ```

  After that, install IPython using `pip`:

  ```
  $ 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 source archive from `https://github.com/ipython/ipython/archive/master.zip`.
  2. Unpack the source code from the archive:

     ```
     $ tar xzf ipython-<version>.tar.gz
     ```

  3. Instead, if you have Git installed, you can clone the Git repository:

     ```
     $ git clone https://github.com/ipython/ipython.git
     ```

  4. Go to the root directory within the downloaded source:

     ```
     $ cd ipython
     ```

  5. Run the setup script. This may require you to run the command with `sudo`, as follows:

     ```
     $ sudo python 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.

See also

- Instructions from the official IPython website at `http://ipython.org/install.html`

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, Maple, and R.

The following is a list of features of the IPython shell:

- Tab completion
- History mechanism
- Inline editing
- The ability to call external Python scripts with `%run`
- The ability to call magic functions that interact with the operating system shell
- 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:

- `pylab`: 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 2.4.1 -- 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. This is done 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 using this command:

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

- **Executing a system shell command**: You can execute a system shell command 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. We can also 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, like this:

```
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 look up 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 Publishing 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 get 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 a magic function is used in a line by itself, the `%` prefix is optional.

See also

- *IPython as a system shell* from the official IPython website at `http://ipython.org/ipython-doc/dev/interactive/shell.html`

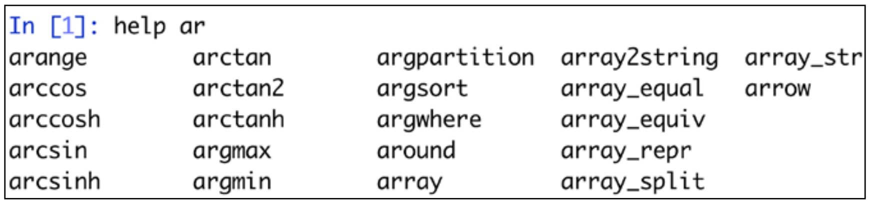
Reading manual pages

We can open the documentation 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. For instance, let's browse the available information for the `arange()` function.

How to do it...

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

- **Calling the help function**: Call the `help` command. Type a few characters of the function and then press the *Tab* key (see the following screenshot):



```
In [1]: help ar
arange      arctan      argpartition  array2string  array_str
arccos      arctan2     argsort       array_equal   arrow
arccosh     arctanh     argwhere      array_equiv
arcsin      argmax      around        array_repr
arcsinh     argmin      array         array_split
```

- **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 the `help` command:

```
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 (all lowercase by convention) is a very useful Python plotting library, and we will need it for the following recipes as well as more later on. 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 on Windows, Linux, and Mac OS X, and also how to install it from source:

- **Installing matplotlib on Windows**: You can install this with the Enthought distribution, also known as Canopy (`http://www.enthought.com/products/epd.php`).

  It might be necessary to put the `msvcp71.dll` file in your `C:\Windows\system32` directory. You can get it from `http://www.dll-files.com/dllindex/dll-files.shtml?msvcp71`.

- **Installing matplotlib on Linux**: Let's see how matplotlib can be installed in the various distributions of Linux:

  Here is the install command on Debian and Ubuntu:

  ```
  $ sudo apt-get install python-matplotlib
  ```

  - The install command on Fedora/Redhat is as follows:

    ```
    $ su - yum install python-matplotlib
    ```

- **Installing from source**: You can 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 matplotlib as usual with the following commands:

  ```
  $ cd matplotlib
  $ sudo python setup.py install
  ```

- **Installing matplotlib on Mac OS X**: Get the latest DMG file from `http://sourceforge.net/projects/matplotlib/files/matplotlib/` and install it. You can also use the Mac Ports, Fink, or Homebrew package managers.

See also

- Instructions from the official matplotlib documentation are given at `http://matplotlib.org/users/installing.html`
- Installing the SciPy stack is explained at `http://www.scipy.org/install.html`

Running an IPython notebook

IPython has an exciting feature—the web notebook. A so-called **notebook server** can serve notebooks over the Web. We can now start a notebook server and get a web-based IPython environment. This environment has most of the features that the regular IPython environment has. The IPython notebook's features include the following:

- Displaying images and inline plots
- Using HTML and **Markdown** (this is a simplified HTML-like language see `https://en.wikipedia.org/wiki/Markdown`) in text cells
- Importing and exporting of notebooks

Getting ready

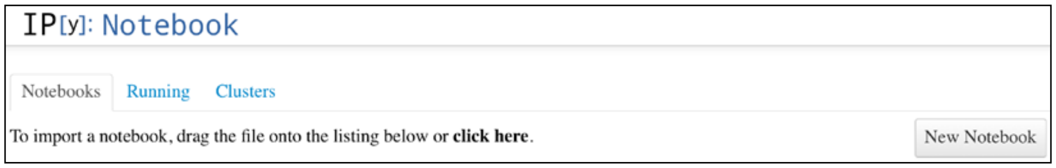
Before we start, we should make sure that all of the required software is installed. There is a dependency on `tornado` and `zmq`. See the *Installing IPython* recipe in this chapter for more information.

How to do it...

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

  ```
  $ 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. You 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 (see the following screenshot):



To import a notebook, drag the file onto the listing below or **click here**.

  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 by pressing *Ctrl* + *C*.

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

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

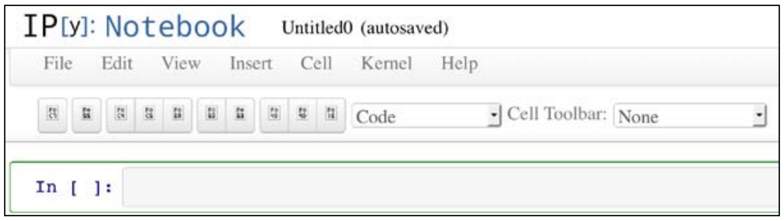
  This loads the `SciPy`, `NumPy`, and `matplotlib` modules.

- **Running a notebook with inline figures**: We can display inline matplotlib plots with the `inline` directive using the following command:

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

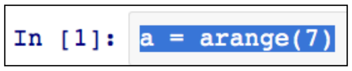
The following steps demonstrate the IPython notebook functionality:

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



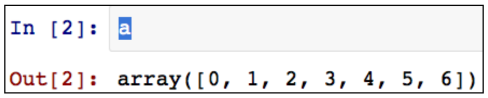
```
In [ ]:
```

2. Create an array with the `arange()` function. Type the command shown in the following screenshot and click on **Cell**/**Run**:



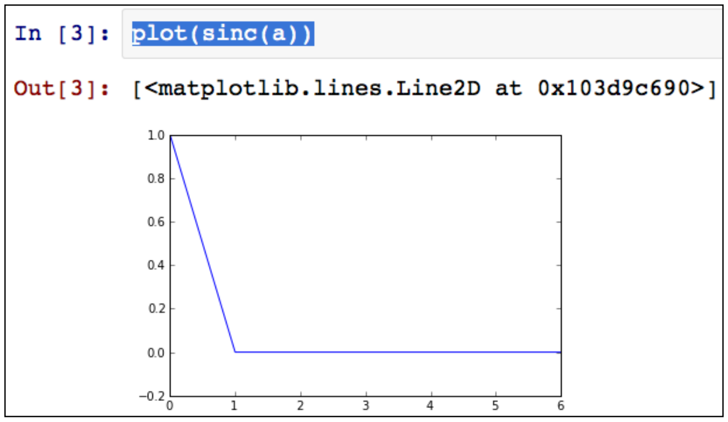
```
In [1]: a = arange(7)
```

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



```
In [2]: a

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

4. Apply the `sinc()` function to the array and plot the result, as shown in this screenshot:



```
In [3]: plot(sinc(a))

Out[3]: [<matplotlib.lines.Line2D at 0x103d9c690>]
```

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 found in this chapter
- Example notebooks at `http://nbviewer.ipython.org/github/ipython/ipython/blob/2.x/examples/Notebook/Index.ipynb`
- Documentation for the `sinc()` function at `http://docs.scipy.org/doc/numpy/reference/generated/numpy.sinc.html`
- Documentation for the `plot()` function at `http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.plot`

Exporting an IPython notebook

Sometimes, you would 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 a 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 a `.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 even reading, but it is pretty readable if you ignore the image representation part. For more information about JSON, 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 format, `.ipynb`. 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.

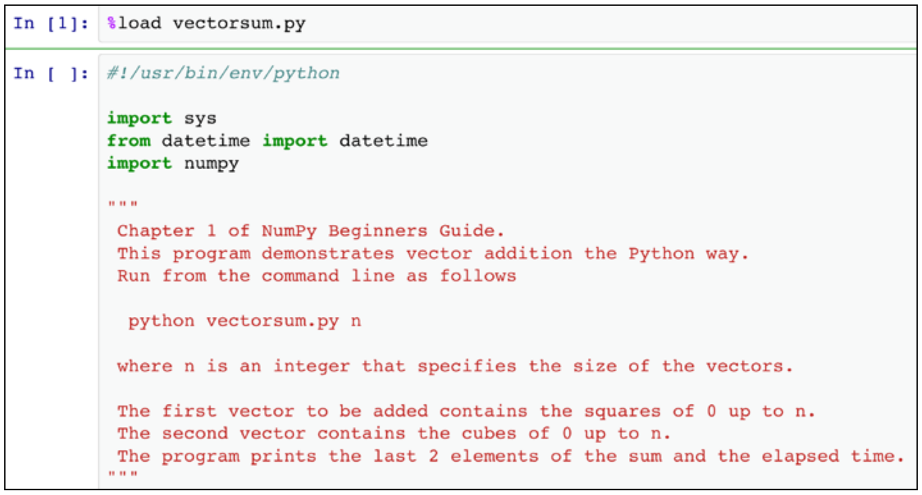
How to do it...

This recipe shows you how a Python script can be imported as a web notebook.

Load a Python script with this command:

```
% load vectorsum.py
```

The following screenshot shows an example of what we see after loading `vectorsum.py` from *NumPy Beginner's Guide* into the notebook page:



```
In [1]: %load vectorsum.py

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.
        """
```

Configuring a notebook server

A public notebook server needs to be secure. You should set a password and use an 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`). HTTPS adds a secure layer on top of the standard HTTP protocol widely used on the Internet. HTTPS also encrypts data sent from the client to the server and back. A **certificate authority** is often a commercial organization that issues certificates for websites. Web browsers have knowledge of certificate authorities and can recognize certificates. A website administrator needs to create a certificate and get it signed by a certificate authority.

How to do it...

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

1. 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 you will need it later on.

2. To create a SSL certificate, you will need the `openssl` command in your path.

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

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

```
$ 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 out the relevant man page (short for manual page) as follows:

```
$ man openssl
```

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

   ```
   $ ipython profile create nbserver
   ```

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

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

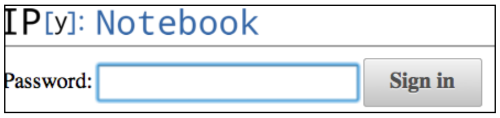
   ```
   c.NotebookApp.certfile = u'/absolute/path/to/your/certificate'
   c.NotebookApp.password = u'sha1:b...your password'
   c.NotebookApp.port = 9999
   ```

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

5. 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, you should see a login page. Also, you will probably need to accept a security exception in your browser.



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 a configuration file, among others. 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
```

See also

- IPython documentation for the `passwd()` function at `http://ipython.org/ipython-doc/2/api/generated/IPython.lib.security.html`
- OpenSSL documentation at `https://www.openssl.org/docs/apps/openssl.html`

Exploring the SymPy profile

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

Getting ready

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

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

How to do it...

The following steps will help you explore the SymPy profile:

1. Look at the configuration file, which can be found at `~/.ipython/profile_sympy/ipython_config.py`. The content is 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']
```

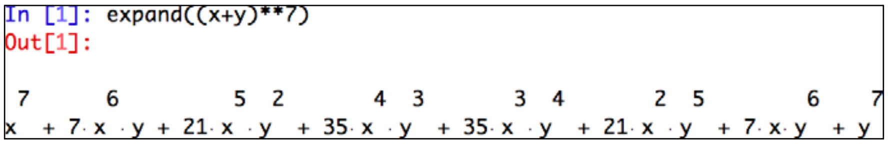
This code accomplishes the following:

- Loads the default profile
- Imports the SymPy packages
- Defines symbols

2. Start IPython with the SymPy profile using this 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]:

 7      6         5  2       4  3       3  4       2  5         6    7
x  + 7⋅x ⋅y + 21⋅x ⋅y  + 35⋅x ⋅y  + 35⋅x ⋅y  + 21⋅x ⋅y  + 7⋅x⋅y  + y
```

See also

- The SymPy homepage at `http://sympy.org/en/index.html`

2
Advanced Indexing and Array Concepts

In this chapter, we will cover the following recipes:

- 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. If you feel it is required, review the *Installing matplotlib* recipe in *Chapter 1*, *Winding Along with IPython*.

In this chapter and in other chapters, we will use the following imports:

```
import numpy as np
import matplotlib.pyplot as plt
import scipy
```

We will also use the newest syntax for the `print()` Python function as much as possible.

Python 2 is a still popular major Python version, but it is not compatible with Python 3. Python 2 is officially supported until 2020. One of the main differences is the syntax for the `print()` function. This book uses code that is as compatible with Python 2 and Python 3 as possible.

Some of the examples in this chapter 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 the same project many years ago. SciPy, just like NumPy, is an open source project available under the BSD license. In this recipe, we will install SciPy. SciPy provides advanced functionality, including statistics, signal processing, linear algebra, optimization, FFT, ODE solvers, interpolation, special functions, and integration. There is some overlap with NumPy, but NumPy primarily provides array functionality.

Getting ready

In *Chapter 1*, *Winding Along with IPython*, we discussed how to install `setuptools` 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 SciPy to your home directory. It requires Python 2.6 or later versions.

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

  - The `BLAS` and `LAPACK` libraries
  - The C and Fortran compilers

There is a chance that you have already installed this software as 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 (you may need to log in as root or have sudo privileges):
  - 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 this 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 this instruction:

    ```
    $ 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 the Apple Developer's connection website at `https://developer.apple.com/xcode/`. Make sure that everything, including all the optional packages, is installed.

  You probably 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`**: You can install SciPy with either of these two commands (the need for `sudo` depends on privileges):

  ```
  $ [sudo] pip install scipy
  $ [sudo] easy_install scipy
  ```

- **Installing on Windows**: If you already have Python installed, the preferred method is to download and use the binary distribution. Alternatively, you can install the Anaconda or 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 take care of dependencies (if there are any) for you. However, in some cases, you need to install them manually. This is beyond the scope of this book.

See also

If you run into problems, you can ask for help at:

- The `#scipy` IRC channel of `freenode`
- The SciPy mailing lists at `http://www.scipy.org/scipylib/mailing-lists.html`

Installing PIL

PIL, the Python imaging library, is a prerequisite for the image processing recipes in this chapter. If you prefer, you can install Pillow, which is a fork of PIL. Some people prefer the Pillow API; however, we are not going to cover its installation in this book.

How to do it...

Let's see how to install PIL:

- **Installing PIL on Windows**: Install PIL using the Windows executable from the PIL website at `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 appears that the package managers of Red Hat, Fedora, and CentOS do not have direct support for PIL. Therefore, 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
  ```

See also

- Instructions for Pillow (a fork of PIL) can be found at `http://pillow.readthedocs.org/en/latest/installation.html`

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 image in question is completely safe for work.

We will resize the image using the `repeat()` function. This function repeats an array, which means resizing the image by a certain factor in our use case.

Getting ready

A prerequisite for this recipe is to have SciPy, matplotlib, and PIL installed. Take a look at the corresponding recipes in this chapter and *Chapter 1*, *Winding Along with IPython*.

How to do it...

Resize the image with the following steps:

1. First, import `SciPy`. SciPy has a `lena()` function. It is used to 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 as follows:

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

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

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

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

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

4. We will plot the Lena image and the resized image in two subplots that are parts of the same grid. Plot the Lena array in a subplot using this code:

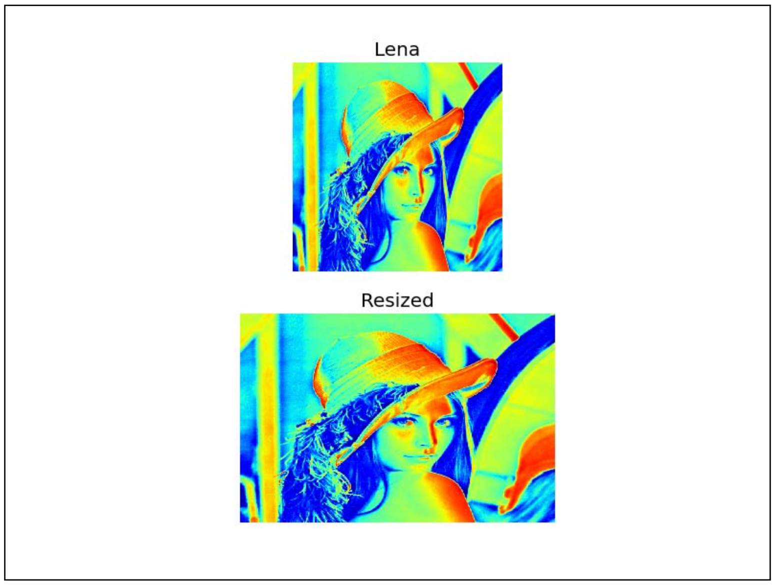
```
plt.subplot(211)
plt.title("Lena")
plt.axis("off")
plt.imshow(lena)
```

The matplotlib `subplot()` function creates a subplot. This function accepts a three-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:

```
plt.subplot(212)
plt.title("Resized")
plt.axis("off")
plt.imshow(resized)
plt.show()
```

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



The following is the complete code for this recipe from the `resize_lena.py` file in this book's code bundle:

```
import scipy.misc
import matplotlib.pyplot as plt
import numpy as np

# This script resizes the Lena image from Scipy.

# 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
np.testing.assert_equal((LENA_X, LENA_Y), lena.shape)

# Set the resize factors
yfactor = 2
xfactor = 3

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

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

# Plot the Lena array
plt.subplot(211)
plt.title("Lena")
plt.axis("off")
plt.imshow(lena)

#Plot the resized array
plt.subplot(212)
plt.title("Resized")
plt.axis("off")
plt.imshow(resized)
plt.show()
```

How it works...

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

See also

- *Installing matplotlib* in *Chapter 1*, *Winding Along with IPython*
- *Installing SciPy* in this chapter
- *Installing PIL* in this chapter
- The `repeat()` function is described at `http://docs.scipy.org/doc/numpy/reference/generated/numpy.repeat.html`

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

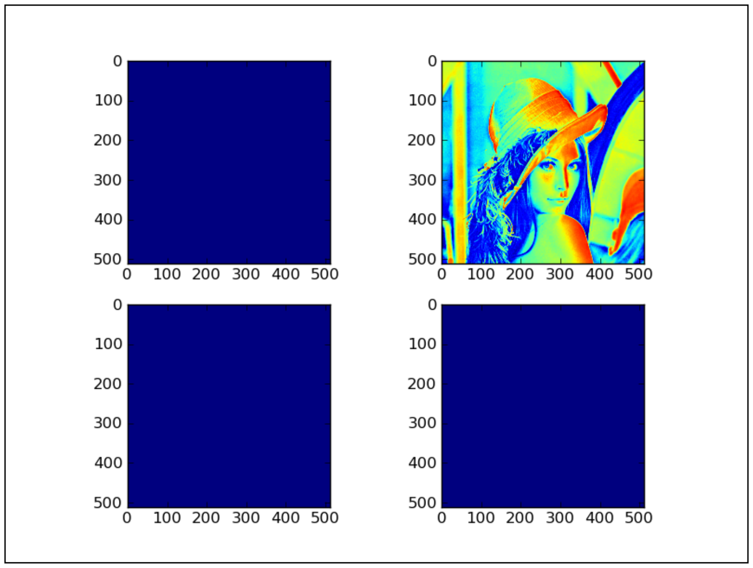
The prerequisites are the same as those for the previous recipe.

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 (the image that is related to the copy) shows the Playboy model. The other images disappear completely:



The following is the code of this tutorial, showing the behavior of array views and copies from the `copy_view.py` file in this book's code bundle:

```
import scipy.misc
import matplotlib.pyplot as plt

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

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

#Plot the copy
plt.subplot(222)
plt.imshow(acopy)

#Plot the view
plt.subplot(223)
plt.imshow(aview)

# Plot the view after changes
```

```
aview.flat = 0
plt.subplot(224)
plt.imshow(aview)

plt.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 three blue (or blank 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.

See also

- The documentation of the NumPy `view()` function is at `http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.view.html`

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 are as follows:

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

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

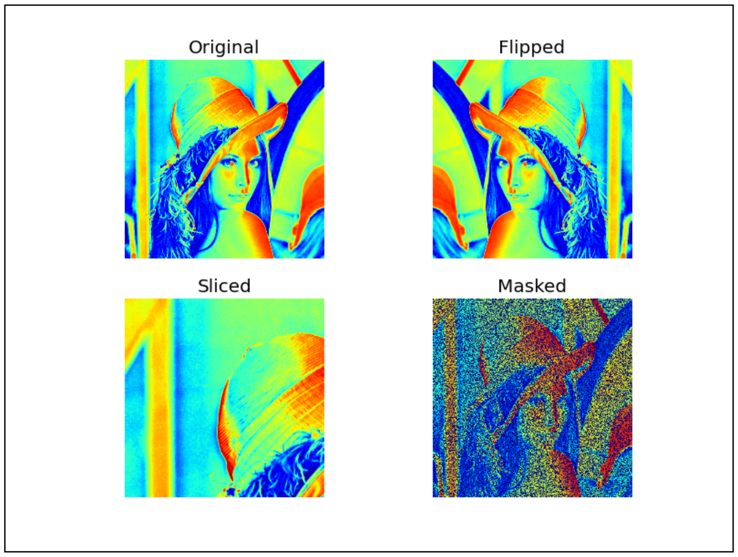
2. Take a slice out of the image and plot it. In this step, we will take a look at the shape of the Lena array. The shape is a tuple representing the dimensions of the array. The following code effectively selects the top-left quadrant of the Playboy picture:

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

3. 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 on the image (dark spots if you are looking at a black-and-white image):

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

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



Here is the complete code for this recipe from the `flip_lena.py` file in this book's code bundle:

```
import scipy.misc
import matplotlib.pyplot as plt

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

# Plot the Lena array
plt.subplot(221)
plt.title('Original')
plt.axis('off')
plt.imshow(lena)

#Plot the flipped array
plt.subplot(222)
plt.title('Flipped')
plt.axis('off')
plt.imshow(lena[:,::-1])

#Plot a slice array
plt.subplot(223)
plt.title('Sliced')
plt.axis('off')
plt.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
plt.subplot(224)
plt.title('Masked')
plt.axis('off')
plt.imshow(masked_lena)

plt.show()
```

See also

- *Installing matplotlib* in *Chapter 1*, *Winding Along with IPython*
- *Installing SciPy* in this chapter
- *Installing PIL* in this chapter

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, 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; it 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:

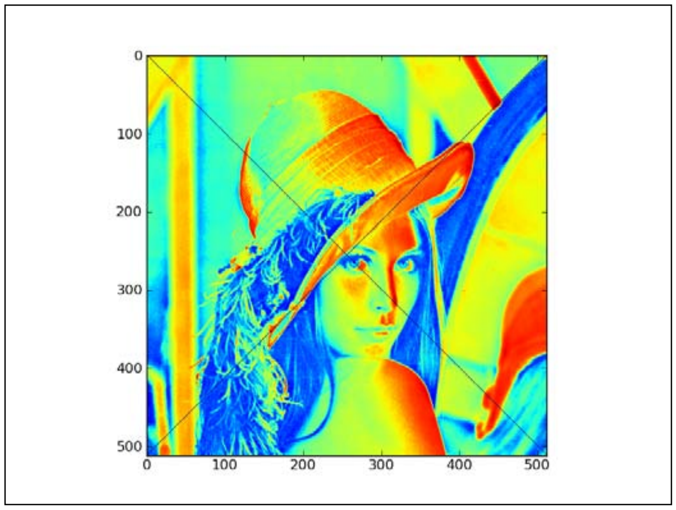
   ```
   lena[range(xmax), range(ymax)] = 0
   ```

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 marked, as shown in the following screenshot:



The following is the complete code for this recipe from the `fancy.py` file in this book's code bundle:

```
import scipy.misc
import matplotlib.pyplot as plt

# 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
plt.imshow(lena)
plt.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 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.

See also

- The fancy indexing implementation is documented at `http://docs.scipy.org/doc/numpy-dev/reference/internals.code-explanations.html#fancy-indexing-check`

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:

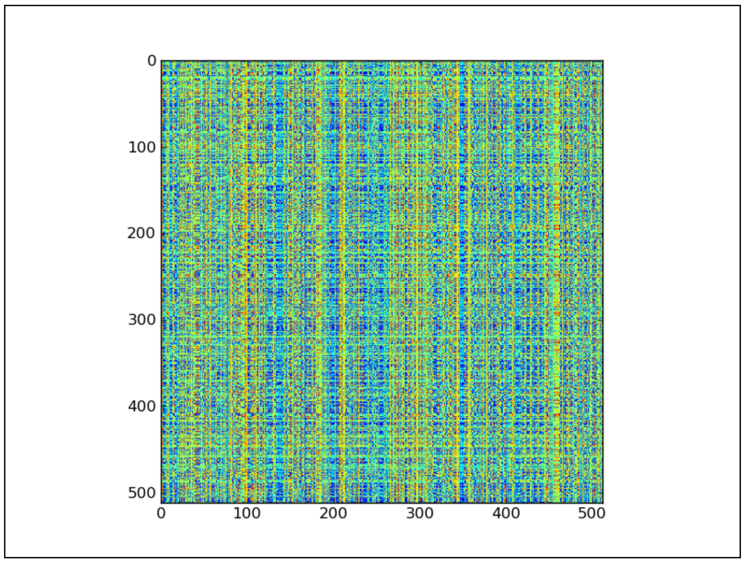
1. Create a random indices array with the `shuffle()` function of the `numpy.random` module:

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

      return arr
   ```

2. Plot the shuffled indices:

   `plt.imshow(lena[np.ix_(xindices, yindices)])`

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



Here is the complete code for the recipe from the `ix.py` file in this book's code bundle:

```
import scipy.misc
import matplotlib.pyplot as plt
import numpy as np

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

def shuffle_indices(size):
   '''
   Shuffles an array with values 0 - size
   '''
```

```
      arr = np.arange(size)
      np.random.shuffle(arr)

      return arr

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

   # Plot Lena
   plt.imshow(lena[np.ix_(xindices, yindices)])
   plt.show()
```

See also

- The `ix_()` function's documentation page at `http://docs.scipy.org/doc/numpy/reference/generated/numpy.ix_.html`

Indexing with Booleans

Boolean indexing is indexing based on a boolean array and falls under the category of 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 = np.arange(size)
      return arr % 4 == 0
   ```

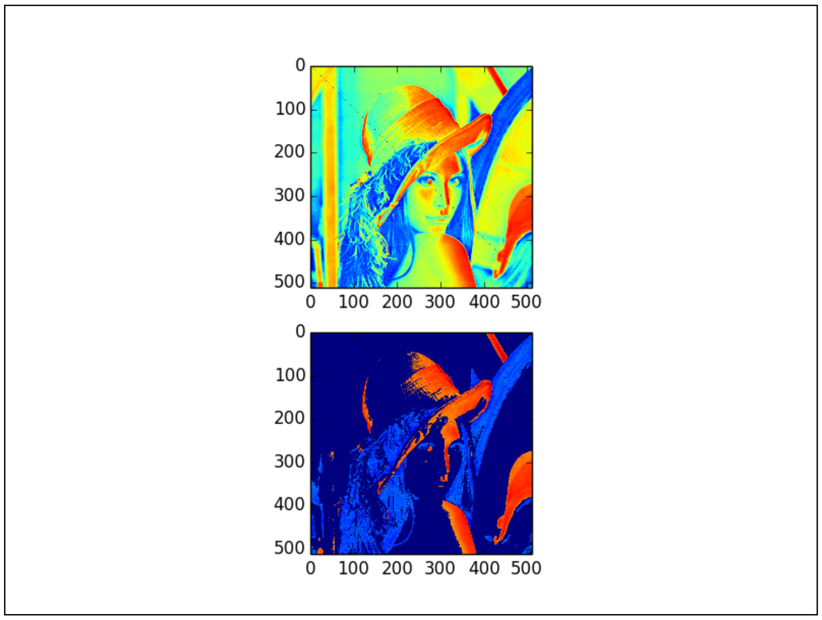
Then we just apply this selection and plot the points:

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

2. Select the 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 what is shown in the following screenshot:



Here is the complete code for this recipe from the `boolean_indexing.py` file in this book's code bundle:

```
import scipy.misc
import matplotlib.pyplot as plt
import numpy as np

# Load the Lena array
```

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

def get_indices(size):
   arr = np.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
plt.subplot(211)
plt.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
plt.subplot(212)
plt.imshow(lena2)


plt.show()
```

How it works...

Since indexing with Booleans 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

- *Fancy Indexing*

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 into the 3 x 3 squares it is composed of.

> Explaining the rules of Sudoku is outside the scope of this book. In a nutshell, a Sudoku puzzle consists of 3 x 3 squares. Each of these squares contains nine numbers. For more information see `http://en.wikipedia.org/wiki/Sudoku`.

How to do it...

Apply the stride tricks as follows:

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

   ```
   sudoku = np.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. The `itemsize` field of `ndarray` gives us the number of bytes in an array. Given the `itemsize` calculate the strides:

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

3. Now we can split the puzzle into squares with the `as_strided()` function of the `np.lib.stride_tricks` module:

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

   The code prints separate Sudoku squares, as follows:

   ```
   [[[[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 from the `strides.py` file in this book's code bundle:

```
import numpy as np

sudoku = np.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 * np.array([27, 3, 9, 1])

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

How it works...

We applied stride tricks to split a Sudoku puzzle into its constituent 3 x 3 squares. The strides tell us the number of bytes we need to skip at each step when going through the Sudoku array.

See also

- The `strides` property is documented at `http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.strides.html`

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 of it that is quieter.

How to do it...

Let's start by reading a WAV file:

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

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

2. Plot the original WAV data with matplotlib. Name the subplot `Original`:

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

3. 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. In the end, we need to make sure that we have the same data type as that in the original array, because of the WAV format:

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

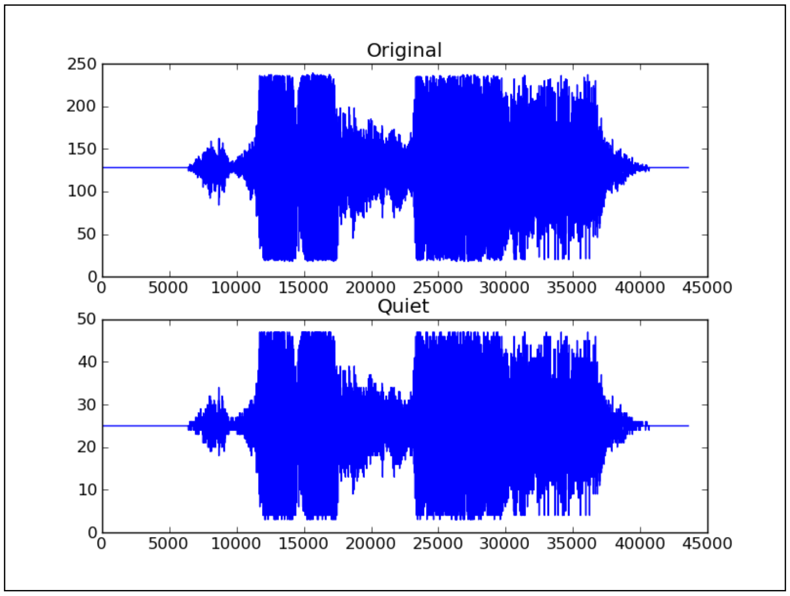
4. This new array can be written to a new WAV file, as follows:

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

5. Plot the new data array with matplotlib:

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

plt.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:



Here is the complete code for this recipe from the `broadcasting.py` file in this book's code bundle:

```
import scipy.io.wavfile
import matplotlib.pyplot as plt
import urllib2
import numpy as np

# Download audio file
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)

# Plot values original audio
plt.subplot(2, 1, 1)
plt.title("Original")
plt.plot(data)

# Create quieter audio
newdata = data * 0.2
newdata = newdata.astype(np.uint8)
print("Data type", newdata.dtype, "Shape", newdata.shape)

# Save quieter audio file
scipy.io.wavfile.write("quiet.wav",
    sample_rate, newdata)

# Plot values quieter file
plt.subplot(2, 1, 2)
plt.title("Quiet")
plt.plot(newdata)

plt.show()
```

See also

The following links give more background information:

- The `scipy.io.read()` function page at `http://docs.scipy.org/doc/scipy/reference/generated/scipy.io.wavfile.read.html`
- The `scipy.io.write()` function page at `http://docs.scipy.org/doc/scipy/reference/generated/scipy.io.wavfile.write.html`
- The broadcasting concept is explained at `http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html`

3
Getting 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()`, and `eig()`
- `histogram()` and `polyfit()`
- `compress()` and `randint()`

We will be discussing these functions in the following recipes:

- Summing Fibonacci numbers
- Finding prime factors
- Finding palindromic numbers
- The steady state vector
- 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 NumPy 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 are the bare minimum with which we should be familiar.

Summing Fibonacci numbers

In this recipe, we will sum the even-valued terms in the Fibonacci sequence whose values do not exceed 4 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 (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 ...).

The sequence was published by Fibonacci in 1202 and originally did not include zero. Actually, it was already known to Indian mathematicians in earlier centuries. Fibonacci numbers have applications in mathematics, computer science, and biology.

[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. The golden ratio is given by the following formula:

$$\varphi = \frac{1+\sqrt{5}}{2}$$

We will use the `sqrt()`, `log()`, `arange()`, `astype()`, and `sum()` functions. The Fibonacci sequence's recurrence relation has the following solution, which involves the golden ratio:

$$F_n = \frac{\varphi^n - \left(-\varphi\right)^{-n}}{\sqrt{5}}$$

How to do it...

The following is the complete code for this recipe from the `sum_fibonacci.py` file in this book's code bundle:

```
import numpy as np
```

```
#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 + np.sqrt(5))/2
print("Phi", phi)

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

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

#4. Compute Fibonacci numbers
fib = (phi**n - (-1/phi)**n)/np.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())
```

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

1. Use the `sqrt()` function to calculate the square root of `5`:

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

This prints the golden mean:

```
Phi 1.61803398875
```

2. Next, in the recipe, we need to find the index of the Fibonacci number below 4 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 = np.log(4 * 10 ** 6 * np.sqrt(5)
    + 0.5)/np.log(phi)
print(n)
```

The value of `n` is as follows:

```
33.2629480359
```

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

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

4. 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)/np.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 that tests a small unit of code, such as a function. This variation of the recipe is left as an exercise for you.]

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

We are not starting with the number 0 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. Okay, 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 the even-valued terms.

   This recipe demands that we select the even-valued terms now. This should be easy for you if you followed the *Indexing with Booleans* recipe in *Chapter 2*, *Advanced Indexing and Array Concepts*:

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

   There we go:

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

How it works...

In this recipe, we used the `sqrt()`, `log()`, `arange()`, `astype()`, and `sum()` functions. Their description is as follows:

Function	**Description**
`sqrt()`	This function calculates the square root of array elements (see `http://docs.scipy.org/doc/numpy/reference/generated/numpy.sqrt.html`)
`log()`	This function calculates the natural logarithm of array elements (see `http://docs.scipy.org/doc/numpy/reference/generated/numpy.log.html#numpy.log`)
`arange()`	This function creates an array with the specified range (see `http://docs.scipy.org/doc/numpy/reference/generated/numpy.arange.html`)
`astype()`	This function converts array elements to a specified data type (see `http://docs.scipy.org/doc/numpy/reference/generated/numpy.chararray.astype.html`)
`sum()`	This function calculates the sum of array elements (see `http://docs.scipy.org/doc/numpy/reference/generated/numpy.sum.html`)

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 exactly divide an integer without leaving a remainder. Finding prime factors seems almost impossible for big numbers. Therefore, prime factors have an application in cryptography. However, using the right algorithm—Fermat's factorization method (`http://en.wikipedia.org/wiki/Fermat%27s_factorization_method`) and NumPy—factoring becomes relatively easy for small numbers. 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 following is the entire code needed to solve the problem of finding the largest prime factor of 600851475143 (see the `fermatfactor.py` file in this book's code bundle):

```
from __future__ import print_function
import numpy as np

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

   #2. Check whether b is a square
```

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

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

        #4. Find the first occurrence of a 0 fraction
        a = np.ravel(np.take(a, indices))[0]
                   # Or a = a[indices][0]

        a = int(a)
        b = np.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 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 not to 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 = np.ceil(np.sqrt(n))
   lim = min(n, LIM)
   a = np.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 = np.modf(np.sqrt(b2))[0]
   ```

3. Find 0 fractions.

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

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

4. Find the first occurrence of a zero fraction.

   First, call the `take()` NumPy function with the `indices` array from the previous step to get the values of zero fractions. Now latten this array with the `ravel()` NumPy function:

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

   This line is a bit convoluted, but it does demonstrate two useful functions. It would have been simpler to write `a = a[indices][0]`.

   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 `ceil()`, `modf()`, `where()`, `ravel()`, and `take()` NumPy functions. The description of these functions is as follows:

Function	Description
`ceil()`	Calculates the ceiling of array elements (see `http://docs.scipy.org/doc/numpy/reference/generated/numpy.ceil.html`)
`modf()`	Returns the fractional and integral part of floating-point numbers (see `http://docs.scipy.org/doc/numpy/reference/generated/numpy.modf.html`)
`where()`	Returns array indices based on condition (see `http://docs.scipy.org/doc/numpy/reference/generated/numpy.where.html`)
`ravel()`	Returns a flattened array (see `http://docs.scipy.org/doc/numpy/reference/generated/numpy.ravel.html`)
`take()`	Takes an element from an array (see `http://docs.scipy.org/doc/numpy/reference/generated/numpy.take.html`)

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 x 99. Let's try to find the largest palindrome made from the product of two 3-digit numbers.

How to do it...

The following is the complete program from the `palindromic.py` file in this book's code bundle:

```
import numpy as np

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

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

#3. Find largest palindromic number
for number in numbers[::-1]:
   s = str(numbers[i])

   if s == s[::-1]:
      print(s)
      break
```

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

1. Create an array of three-digit numbers.

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

   ```
   a = np.arange(100, 1000)
   np.testing.assert_equal(100, a[0])
   np.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 three-digit 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 that the array is properly sorted. After that, we can do some sanity checks:

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

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 outer product of two vectors (one-dimensional lists of numbers) creates a matrix. This is the opposite of an inner product, which returns a scalar number for two vectors. The outer product is used in physics, signal processing, and statistics. 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 the NumPy way.

The steady state vector

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. The chain doesn't have any memory about the states. Markov chains are often used to model phenomena in physics, chemistry, finance, and computer science. For instance, Google's PageRank algorithm uses Markov chains to rank web pages.

I would like to define a Markov chain for a stock. Let's say that we have the states **flat**, **up**, and **down**. We can determine the steady state based on the end-of-the-day close prices.

Far into the distant future or theoretically after an infinite amount of time, the state of our Markov chain system will not change anymore. This is called a steady state (see http://en.wikipedia.org/wiki/Steady_state). A dynamic equilibrium is a type of steady state. For a stock, achieving a steady state may mean that the related company has become stable. The **stochastic matrix** (see http://en.wikipedia.org/wiki/Stochastic_matrix) *A* contains the state transition probabilities, and when applied to the steady state, it yields the same state x. The mathematical notation for this is as follows:

$$Ax = x$$

Another way to look at this is as the eigenvector (see http://en.wikipedia.org/wiki/Eigenvalues_and_eigenvectors) for eigenvalue 1. Eigenvalues and eigenvectors are fundamental concepts of linear algebra with applications in quantum mechanics, machine learning, and other sciences.

How to do it...

The following is the complete code for the steady state vector example from the steady_state_vector.py file in this book's code bundle:

```
from __future__ import print_function
from matplotlib.finance import quotes_historical_yahoo
from datetime import date
import numpy as np


today = date.today()
```

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

quotes = quotes_historical_yahoo('AAPL', start, today)
close =  [q[4] for q in quotes]

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

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

signs = [-1, 0, 1]
k = 1

for i, signi in enumerate(signs):
   #we start the transition from the state with the specified sign
   start_indices = np.where(states[:-1] == signi)[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, signj in enumerate(signs):
      # number of occurrences of this transition
      occurrences = len(end_values[end_values == signj])
      SM[i][j] = (occurrences + k)/float(N)

print(SM)
eig_out = np.linalg.eig(SM)
print(eig_out)

idx_vec = np.where(np.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", np.dot(SM, x))
```

Now we need to obtain the data:

1. Obtain 1 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 of the last 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.

   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 `diff()` NumPy function. The state is then given by the sign of the difference. The `sign()` NumPy function returns `-1` for a negative number, `1` for a positive number, and `0` otherwise:

   ```
   states = np.sign(np.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 an up state, we could switch to:

   - Up
   - Flat
   - Down

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

   ```
   NDIM = 3
   SM = np.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 `where()` NumPy function. Here, `k` is a smoothing constant, which we will discuss later on:

   ```
   signs = [-1, 0, 1]
   k = 1

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

6. Smoothing and the stochastic matrix.

   We can now count the number of occurrences of each transition. Dividing it 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 overfitting.

   In real life, we could have a day when the close price does not change, although this is 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, signj in enumerate(signs):
       # number of occurrences of this transition
       occurrences = len
         (end_values[end_values == signj])
       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. On one of the test runs, I got the following matrix:

```
[[ 0.5047619   0.00952381  0.48571429]
 [ 0.33333333  0.33333333  0.33333333]
 [ 0.33774834  0.00662252  0.65562914]]
```

7. Eigenvalues and eigenvectors.

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

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

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

```
(array([ 1.        ,  0.16709381,  0.32663057]), array([[
5.77350269e-01,   7.31108409e-01,   7.90138877e-04],
       [  5.77350269e-01,  -4.65117036e-01,  -9.99813147e-01],
       [  5.77350269e-01,  -4.99145907e-01,   1.93144030e-02]]))
```

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 `1`, so we should build a margin for error. We can find the index for eigenvalue between `0.9` and `1.1`, as follows:

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

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

   The rest of the output for this code is as follows:

```
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. Since we are 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
`diff()`	Calculates the discrete difference. By default, the first order (see `http://docs.scipy.org/doc/numpy/reference/generated/numpy.diff.html`).
`sign()`	Returns the sign of array elements (see `http://docs.scipy.org/doc/numpy/reference/generated/numpy.sign.html`).
`eig()`	Returns the eigenvalues and eigenvectors of an array (see `http://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.eig.html`).

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. In such a law, one variable is equal to the power of another:

$$y = cx^k$$

The Pareto principle (see `http://en.wikipedia.org/wiki/Pareto_principle`) for instance, is a power law. It 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 the involved transaction costs per trade. Let's say that we would prefer to buy and sell once a month based on a significant correction (with other words a big drop). The issue is to determine an appropriate signal given that we want to initiate a transaction for every 1 out of about 20 days.

How to do it...

The following is the complete code from the `powerlaw.py` file in this book's code bundle:

```
from matplotlib.finance import quotes_historical_yahoo
from datetime import date
import numpy as np
import matplotlib.pyplot as plt

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

quotes = quotes_historical_yahoo('IBM', start, today)
close =  np.array([q[4] for q in quotes])

#2. Get positive log returns.
logreturns = np.diff(np.log(close))
pos = logreturns[logreturns > 0]

#3. Get frequencies of returns.
counts, rets = np.histogram(pos)
# 0 counts indices
indices0 = np.where(counts != 0)
rets = rets[:-1] + (rets[1] - rets[0])/2
# Could generate divide by 0 warning
freqs = 1.0/counts
freqs = np.take(freqs, indices0)[0]
rets = np.take(rets, indices0)[0]
freqs =  np.log(freqs)

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

#5. Plot the results.
plt.title('Power Law')
plt.plot(rets, freqs, 'o', label='Data')
plt.plot(rets, p[0] * rets + p[1], label='Fit')
plt.xlabel('Log Returns')
plt.ylabel('Log Frequencies')
plt.legend()
plt.grid()
plt.show()
```

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

1. Get the 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 `diff()` NumPy function. Let's select the positive values from the log returns:

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

2. Get the frequencies of the 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 = np.histogram(pos)
   # 0 counts indices
   indices0 = np.where(counts != 0)
   rets = rets[:-1] + (rets[1] - rets[0])/2
   # Could generate divide by 0 warning
   freqs = 1.0/counts
   freqs = np.take(freqs, indices0)[0]
   rets = np.take(rets, indices0)[0]
   freqs =  np.log(freqs)
   ```

3. Fit the frequencies and returns into a line.

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

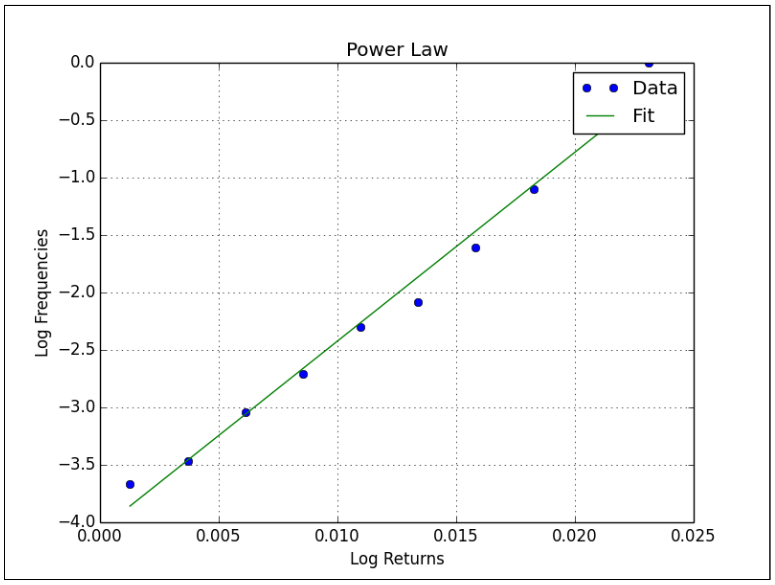
   ```
   p = np.polyfit(rets,freqs, 1)
   ```

4. Plot the results.

   Finally, we will plot the data and linearly fit it with matplotlib:

   ```
   plt.title('Power Law')
   plt.plot(rets, freqs, 'o', label='Data')
   plt.plot(rets, p[0] * rets + p[1], label='Fit')
   plt.xlabel('Log Returns')
   plt.ylabel('Log Frequencies')
   plt.legend()
   plt.grid()
   plt.show()
   ```

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



How it works...

The `histogram()` function calculates the histogram of a dataset. It returns the histogram values and bin edges. The `polyfit()` function fits data to a polynomial of a given order. In this case, we chose 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*
- The documentation page for the `histogram()` function at `http://docs.scipy.org/doc/numpy/reference/generated/numpy.histogram.html`
- The documentation page for the `polyfit()` function at `http://docs.scipy.org/doc/numpy/reference/generated/numpy.polyfit.html`

Trading periodically on dips

Stock prices periodically dip and go up. We will take a look at the probability distribution of the stock price log returns and try a very simple strategy. This strategy is based on regression towards the mean. This is a concept originally discovered in genetics by Sir Francis Galton. It was discovered that children of tall parents tend to be shorter than their parents. Children of short parents tend to be taller than their parents. Of course, this is a statistical phenomenon and doesn't take into account fundamental factors and trends such as improvement in nutrition. Regression towards the mean is also relevant to the stock market. However, it gives no guarantees. If a company starts making bad products or makes bad investments, regression towards the mean will not save the stock.

Let's start by downloading the historical data for a stock, for instance, AAPL. Next, we calculate the daily log returns (`http://en.wikipedia.org/wiki/Rate_of_return`) of the close prices. We will skip these steps since 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...

The following is the complete code from the `periodic.py` file in this book's code bundle:

```
from __future__ import print_function
from matplotlib.finance import quotes_historical_yahoo
from datetime import date
import numpy as np
import scipy.stats
import matplotlib.pyplot as plt

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

quotes = quotes_historical_yahoo('AAPL', start, today)
close =  np.array([q[4] for q in quotes])

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

#3. Calculate breakout and pullback
```

```
freq = 0.02
breakout = scipy.stats.scoreatpercentile(logreturns, 100 * (1 - freq)
)
pullback = scipy.stats.scoreatpercentile(logreturns, 100 * freq)

#4. Generate buys and sells
buys = np.compress(logreturns < pullback, close)
sells = np.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
plt.title('Periodic Trading')
plt.hist(logreturns)
plt.grid()
plt.xlabel('Log Returns')
plt.ylabel('Counts')
plt.show()
```

Now comes the interesting part:

1. Calculate the breakout and pullback.

   Let's say we want to trade five times a 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 = 0.02
   breakout = scipy.stats.scoreatpercentile
     (logreturns, 100 * (1 - freq) )
   pullback = scipy.stats.scoreatpercentile
     (logreturns, 100 * freq)
   ```

2. Generate buys and sells.

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

   ```
   buys = np.compress(logreturns < pullback, close)
   sells = np.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:

```
[  77.76375466   76.69249773  102.72         101.2          98.57
]
[ 74.95502967  76.55980292  74.13759123  80.93512599  98.22      ]
5 5
-52.1387025726
```

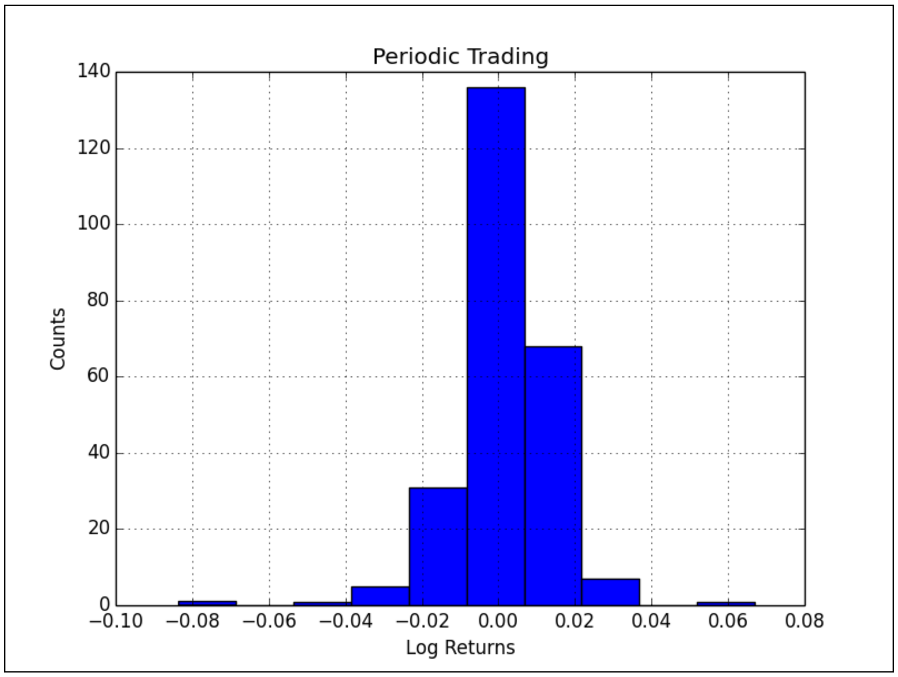
Thus, we have a loss of 52 dollars if we buy and sell an AAPL share five times. When I ran the script, the entire market was in recovery mode after a correction. You may want to look at not just the AAPL stock price but maybe the ratio of AAPL and SPY. SPY can be used as a proxy for the U.S. stock market.

3. Plot a histogram of the log returns.

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

```
plt.title('Periodic Trading')
plt.hist(logreturns)
plt.grid()
plt.xlabel('Log Returns')
plt.ylabel('Counts')
plt.show()
```

   This is what the histogram looks like:



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 recipe in *Chapter 2*, *Advanced Indexing and Array Concepts*
- The Discovering a power law recipe in this chapter
- The documentation page for the `compress()` function at `http://docs.scipy.org/doc/numpy/reference/generated/numpy.compress.html`

Simulating trading at random

In the previous recipe, we tried out a trading idea. However, we have no benchmark that can tell us whether 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 of the corresponding recipe.

How to do it...

The following is the complete code from the `random_periodic.py` file in this book's code bundle:

```
from __future__ import print_function
from matplotlib.finance import quotes_historical_yahoo
from datetime import date
import numpy as np
import matplotlib.pyplot as plt

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

#1. Get close prices.
```

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

quotes = quotes_historical_yahoo('AAPL', start, today)
close =  np.array([q[4] for q in quotes])

nbuys = 5
N = 2000
profits = np.zeros(N)

for i in xrange(N):
   #3. Simulate trades
   buys = np.take(close, get_indices(len(close), nbuys))
   sells = np.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
plt.title('Simulation')
plt.hist(profits)
plt.xlabel('Profits')
plt.ylabel('Counts')
plt.grid()
plt.show()
```

First we need an array filled with random integers:

1. Generate random indices.

   You can generate random integers with the `randint()` NumPy function. This will be linked to random days of a trading year:

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

2. Simulate trades.

   You can simulate trades with the random indices from the previous step. Use the `take()` NumPy function to extract random close prices from the array of step 1:

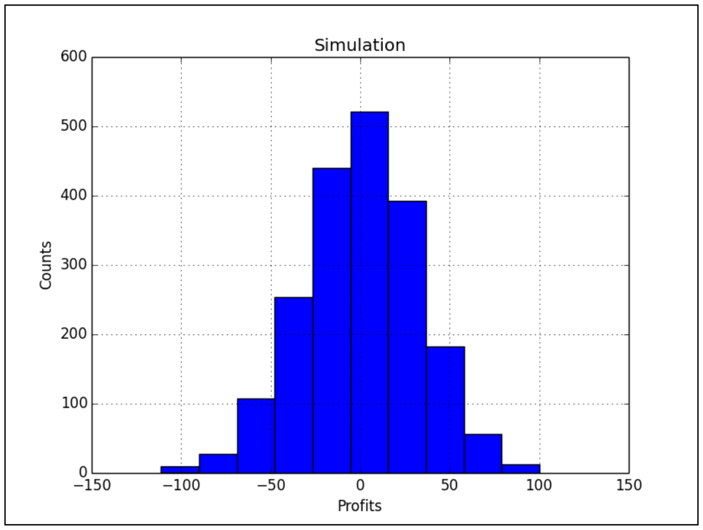
   ```
   buys = np.take(close, get_indices(len(close), nbuys))
   sells = np.take(close, get_indices(len(close), nbuys))
   profits[i] = sells.sum() - buys.sum()
   ```

3. Plot a histogram of the profits for a large number of simulations:

   ```
   plt.title('Simulation')
   ```

```
plt.hist(profits)
plt.xlabel('Profits')
plt.ylabel('Counts')
plt.grid()
plt.show()
```

Here is a screenshot of the resulting histogram of 2,000 simulations for AAPL, with five buys and sells in a year:



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
`rand()`	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 (see `http://docs.scipy.org/doc/numpy/reference/generated/numpy.random.rand.html`).
`randn()`	Sample values from the normal distribution with mean `0` and variance `1`. The dimension parameters function the same as for `rand()` (see `http://docs.scipy.org/doc/numpy/reference/generated/numpy.random.randn.html`).
`randint()`	Returns an integer array given a low boundary, an optional high bound, and an optional output shape (see `http://docs.scipy.org/doc/numpy/reference/generated/numpy.random.randint.html`).

See also

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

Sieving integers with the Sieve of Eratosthenes

The **Sieve of Eratosthenes** (see `http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes`) is an algorithm that filters prime numbers. It iteratively identifies multiples of found primes. The multiples are, by definition, not primes and can be eliminated. This sieve is efficient for primes less 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 = np.arange(i, i + LIM, 2)
   ```

2. Sieve out the 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:

   ```
   from __future__ import print_function
   import numpy as np

   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 sieve_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 = np.arange(i, i + LIM, 2)

   while len(primes) < P:
      a = sieve_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 the following recipes:

- 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 the NumPy code on the Google Cloud
- Running the NumPy code in a PythonAnywhere web console

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 also 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 on the cloud. This is a continuously evolving technology in a fast-moving space. Many options are available for you, of which Google App Engine and PythonAnywhere will be covered.

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 pieces of 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 out the *See also* section of this recipe for instructions.

How to do it...

The complete code for this recipe is in the `buffer.py` file in this book's code bundle:

```
import numpy as np
import Image #from PIL import Image (Python 3)
import scipy.misc

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

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

First we need a NumPy array to play with:

1. In previous chapters, we saw how to load the sample image of Lena Söderberg. Create an array filled with zeros and populate the alpha channel with the image data:

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

2. Use the PIL API to save the data as an RGBA image:

   ```
   img = Image.frombuffer("RGBA", lena.shape, data, 'raw', "RGBA", 0,
   1)
   img.save('lena_frombuffer.png')
   ```

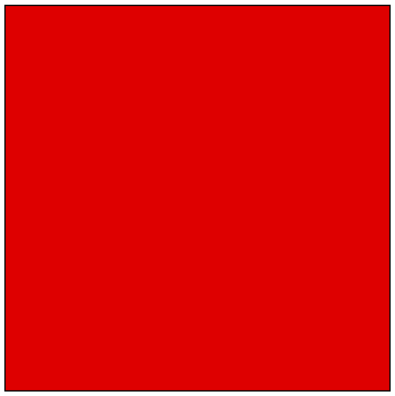
3. Modify the data array by getting rid of the image data and making the image red. Save the image with the PIL API:

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

The following is the before image:

In computer graphics, the position of the origin is different than in the usual Cartesian coordinate system you know from high-school mathematics. The origin is in the top-left corner of the screen, canvas, or image, and the *y* axis goes down (see `http://en.wikipedia.org/wiki/2D_computer_graphics#Non-standard_orientation_of_the_coordinate_system`).

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



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. We did this 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. With a few simple changes, the code should work with other PIL-based libraries, such as Pillow.

See also

- *Installing PIL* in *Chapter 2, Advanced Indexing and Array Concepts*
- *Installing SciPy* in *Chapter 2, Advanced Indexing and Array Concepts*
- The Python buffer protocol is described at `http://docs.python.org/2/c-api/buffer.html`

Using the array interface

The array interface is a yet another mechanism used 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'll reuse part of the code from the previous recipe, so the prerequisites are similar. We will skip the first step of the previous recipe here, and assume that it is already known.

How to do it...

The code for this recipe is in the `array_interface.py` file in this book's code bundle:

```
from __future__ import print_function
import numpy as np
import Image
import scipy.misc

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

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

The following steps will allow us to explore the array interface:

1. The PIL Image object has an `__array_interface__` attribute. Let's inspect its content. The value of this attribute is a Python 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 `ndarray` NumPy class has an `__array_interface__` attribute as well. We can convert the PIL image into a NumPy array with the `asarray()` function:

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

   The shape and data type of the array are as follows:

   ```
   Shape (512, 512, 4)
   Data type uint8
   ```

As you can see, the shape has not changed.

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
- The array interface is described in detail at `http://docs.scipy.org/doc/numpy/reference/arrays.interface.html`

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 Python dictionary.

Getting ready

Installing MATLAB or Octave is beyond the scope of this book. The Octave website has some pointers for installing at `http://www.gnu.org/software/octave/download.html`. Check out the *See also* section of this recipe, for instructions on installing SciPy, if necessary.

How to do it...

The complete code for this recipe is in the `octave.py` file in this book's code bundle:

```
import numpy as np
import scipy.io

a = np.arange(7)

scipy.io.savemat("a.mat", {"array": a})
```

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

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

2. 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
```

See also

- *Installing SciPy* in *Chapter 2*, *Advanced Indexing and Array Concepts*
- The SciPy documentation for the `savemat()` function at `http://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.io.savemat.html`

Installing RPy2

R is a popular scripting language used in 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 this command:

  ```
  $ easy_install rpy2
  ```

  Alternatively, we can use the following command:

  ```
  $ sudo pip install rpy2
  $ pip freeze|grep rpy2
  rpy2==2.4.2
  ```

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

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

See also

- The R programming language homepage at `http://www.r-project.org/`
- The RPy2 project page is at `http://rpy.sourceforge.net/`

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...

The complete code for this recipe is in the `rdatasets.py` file in this book's code bundle:

```
from rpy2.robjects.packages import importr
import numpy as np
import matplotlib.pyplot as plt

datasets = importr('datasets')
mtcars = datasets.__rdata__.fetch('mtcars')['mtcars']

plt.title('R mtcars dataset')
plt.xlabel('wt')
plt.ylabel('mpg')
plt.plot(mtcars)
plt.grid(True)
plt.show()
```

The `motorcars` dataset is described at `https://stat.ethz.ch/R-manual/R-devel/library/datasets/html/mtcars.html`. Let's start by loading this sample R dataset:

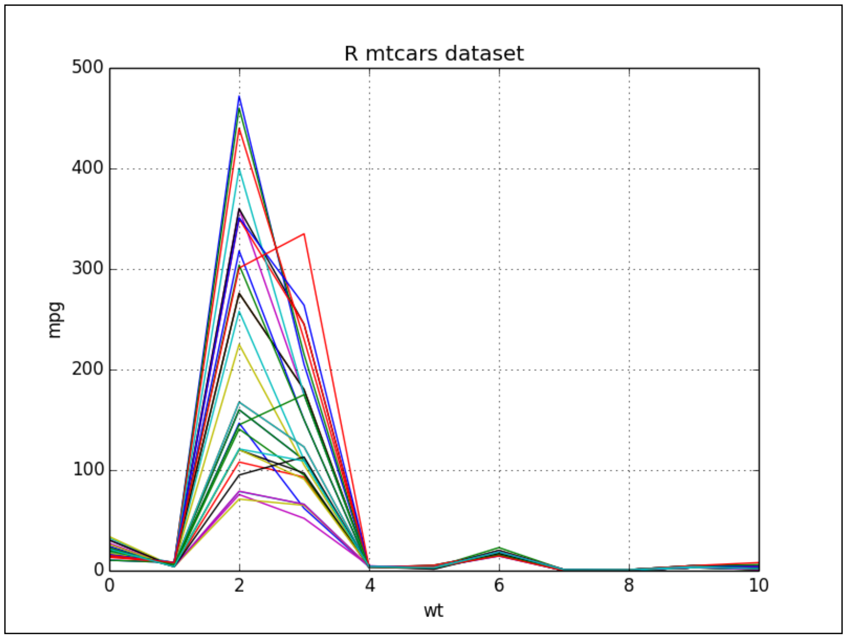
1. Load a dataset into an array with the RPy2 `importr()` function. This function can import `R` packages. In this example, we will import the datasets R package. Create a NumPy array from the `mtcars` dataset:

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

2. Plot the dataset with matplotlib:

   ```
   plt.plot(mtcars)
   plt.show()
   ```

The data contains **miles per gallon** (**mpg**) and **weight** (**wt**) values in lb/1000. The following screenshot shows the data, which is a two-dimensional array:



See also

- *Installing matplotlib* in *Chapter 1*, *Winding Along with IPython*

Installing JPype

Jython is the default interoperability solution for Python and Java. However, Jython runs on the **Java virtual machine** (**JVM**). Therefore, it cannot access NumPy modules, which are mostly written in C. **JPype** is an open source project that tries to solve this problem. The interfacing occurs on the native level between Python and JVM. Let's install JPype.

How to do it...

1. Download JPype from `http://sourceforge.net/projects/jpype/files/`.
2. Unpack it and run the following command:

```
$ python setup.py install
```

Sending a NumPy array to JPype

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...

The complete code for this recipe is in the `hellojpype.py` file in this book's code bundle:

```
import jpype
import numpy as np

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

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

#3. Send a NumPy array
values = np.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
jpype.shutdownJVM()
```

First, we need to start the JVM from JPype:

1. Start the JVM from JPype; JPype is conveniently able to find the default JVM path:

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

2. Just because of tradition, let's print `"hello world"`:

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

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

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

   for item in java_array:
   ```

```
    jpype.java.lang.System.out.
      println(item)
```

4. After we are done, let's shut down the JVM:

   ```
   jpype.shutdownJVM()
   ```

   Only one JVM can run at a time in JPype. If we forget to shut down 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
   ```

How it works...

JPype allows us to start up and shut down a JVM. It provides wrappers for standard Java API calls. As we saw in this example, we can pass Python lists to be transformed into Java arrays by the JArray wrapper. JPype 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 JPype* in this chapter
- The JPype homepage at `http://jpype.sourceforge.net/`

Installing Google App Engine

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

How to do it...

The first step is to download GAE:

1. 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.

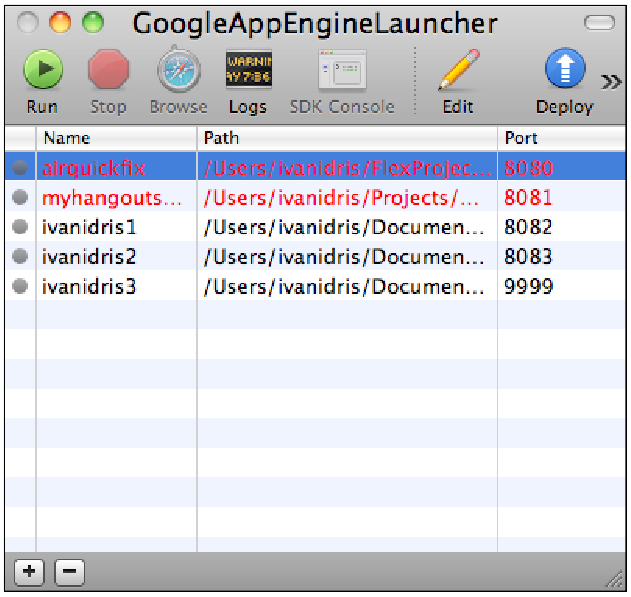
2. The development environment.

   GAE comes with a development environment that simulates the production cloud. At the time of writing this book, GAE officially supported only 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`: The development server
   - `appcfg.py`: Deploys on the cloud

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



Name	Path	Port
airquickfix	/Users/ivanidris/FlexProjec...	8080
myhangouts...	/Users/ivanidris/Projects/...	8081
ivanidris1	/Users/ivanidris/Documen...	8082
ivanidris2	/Users/ivanidris/Documen...	8083
ivanidris3	/Users/ivanidris/Documen...	9999

Deploying the NumPy code on 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 with the launcher (**File** | **New Application**). Name it `numpycloud`. This will create a folder with the same name containing the following files:
   - `app.yaml`: A YAML application configuration file
   - `favicon.ico`: An icon
   - `index.yaml`: An autogenerated file
   - `main.py`: The 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"
   ```

   This is not the latest NumPy version, but it is the latest version currently supported by GAE. 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. 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 (on Linux, run `dev_appserver.py` with the project root as its argument), 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 10 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.

Running the NumPy code in a PythonAnywhere web console

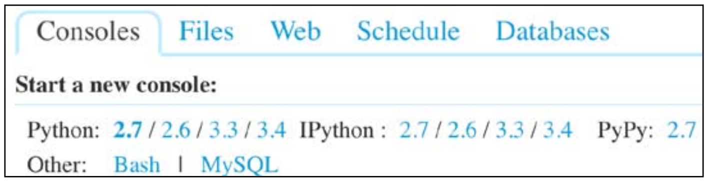
In *Chapter 1*, *Winding Along with IPython*, we already saw a PythonAnywhere console in action, without having an account. This recipe will require you to have an account, but don't worry—it's free, at least if you don't need too many resources.

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

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

How to do it...

Once we have signed up, we can log in and take a look at the **PythonAnywhere** dashboard.


Consoles Files Web Schedule Databases
Start a new console:
Python: 2.7 / 2.6 / 3.3 / 3.4 IPython : 2.7 / 2.6 / 3.3 / 3.4 PyPy: 2.7
Other: Bash | MySQL

1. Write the code. The complete code for this example is as follows:

```
from __future__ import print_function
import urllib2
import re
import time
import numpy as np

prices = np.array([])

for i in xrange(3):
   req = urllib2.Request('http://finance.google.com/finance/
info?client=ig&q=AAPL')
   req.add_header('User-agent', 'Mozilla/5.0')
   response = urllib2.urlopen(req)
   page = response.read()
   m = re.search('l_cur" : "(.*)"', page)
   prices = np.append(prices, float(m.group(1)))
   avg = prices.mean()
```

```
        stddev = prices.std()

        devFactor = 1
        bottom = avg - devFactor * stddev
        top = avg + devFactor * stddev
        timestr = time.strftime("%H:%M:%S", time.gmtime())

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

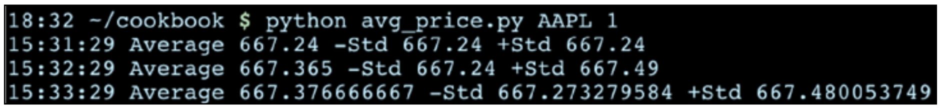
Most of this is standard Python, except the bits where we grow a NumPy array containing prices and calculate the mean and standard deviation of the prices. A URL is used to download price data in JSON format from Google Finance given a stock ticker such as AAPL. This URL could change, of course.

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 multiplied by 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 PythonAnywhere. Go to the dashboard and click on the **Files** tab. Upload the script from the widget at the bottom of the page.

3. To run the code, click on the **Consoles** tab and then on the **Bash** link. PythonAnywhere should create a bash console for us right away. 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...

PythonAnywhere 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 PythonAnywhere. This frees resources on your local machine as well. We can do fancy things such as sending e-mails based on the stock price or scheduling 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.

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 in the following recipes:

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

Introduction

Although all the chapters in this book are fun, in this chapter, we are really going to 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 direct support for audio files, so you really need to run the code examples to fully appreciate the recipes.

Loading images into memory maps

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 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 (plural of radius) of the squares
    - An array with random colors of the squares

    Initialize the arrays:

    ```
    img = np.zeros((N, N), np.uint8)
    NSQUARES = 30
    centers = np.random.random_integers(0, N, size=(NSQUARES, 2))
    radii = np.random.randint(0, N/9, size=NSQUARES)
    colors = np.random.randint(100, 255, size=NSQUARES)
    ```

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

2. The next step is to generate the 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 sizes `N` and `M`, it will give us two arrays of shape N x 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 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.]])]
```

3. 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 = np.clip(xindices, 0, N - 1)
   yindices = range(centers[i][1] - radii[i], centers[i][1]
+ radii[i])
   yindices = np.clip(yindices, 0, N - 1)

   if len(xindices) == 0 or len(yindices) == 0:
      continue
   coordinates = np.meshgrid(xindices, yindices)
   img[coordinates] = colors[i]
```

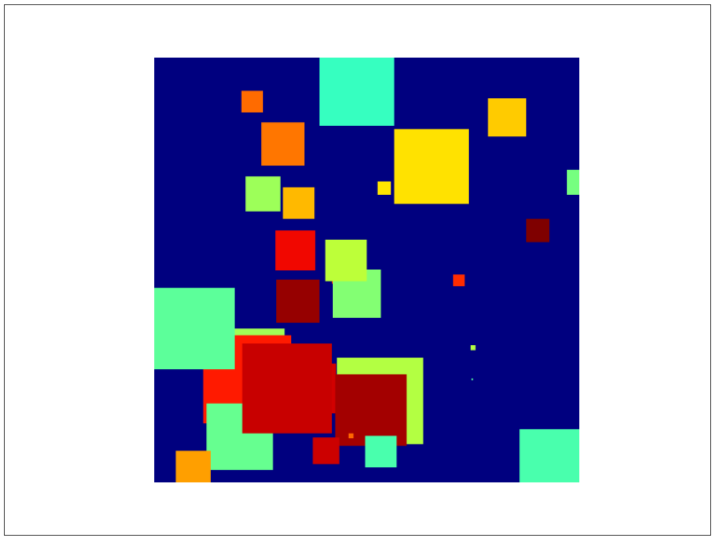
4. Before we load the image data into a memory map, we need to store it in a file with the `tofile()` function. Then load the image data from this file into a memory map with the `memmap()` function:

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

5. To confirm that everything worked fine, we display the image with matplotlib:

```
plt.imshow(img_memmap)
plt.axis('off')
plt.show()
```

Notice that we are not displaying the axes. An example of a generated image is shown here:



Here is the complete source code for this recipe from the `memmap.py` file in this book's code bundle:

```
import numpy as np
import matplotlib.pyplot as plt

N = 512
NSQUARES = 30

# Initialize
img = np.zeros((N, N), np.uint8)
centers = np.random.random_integers(0, N, size=(NSQUARES, 2))
radii = np.random.randint(0, N/9, size=NSQUARES)
colors = np.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 = np.clip(xindices, 0, N - 1)
   yindices = range(centers[i][1] - radii[i], centers[i][1]
+ radii[i])
   yindices = np.clip(yindices, 0, N - 1)

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

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

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

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

How it works...

We used the following functions in this recipe:

Function	Description
`zeros()`	This function gives an array filled with zeros.
`random_integers()`	This function returns an array with random integer values between a high and low bound.
`randint()`	This function has the same functionality as `random_integers()`, except that it uses a half-open interval instead of a closed interval.
`clip()`	This function clips values of an array, given a minimum and a maximum.
`meshgrid()`	This function returns coordinate arrays from an array containing *x* coordinates and an array containing *y* coordinates.
`tofile()`	This function writes an array to a file.
`memmap()`	This function creates a NumPy memory map from a file, given the name of a file. Optionally, you can specify the shape of the array.
`axis()`	This function is the matplotlib function that configures the plot axes. For instance, we can turn them off.

See also

- *Installing matplotlib* in *Chapter 1, Winding Along with IPython*
- The NumPy memory map documentation at `http://docs.scipy.org/doc/numpy/reference/generated/numpy.memmap.html`

Combining images

In this recipe, we will combine the famous **Mandelbrot fractal** (see `http://en.wikipedia.org/wiki/Mandelbrot_set`) and the image of Lena. The Mandelbrot set was invented by the mathematician Benoit Mandelbrot. 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. More details will be covered in this recipe.

Getting ready

Install SciPy if necessary. The *See also* section has a reference to the related recipe.

How to do it...

Start by initializing the arrays, followed by generating and plotting the fractal, and finally combining the fractal with the Lena image:

1. Initialize the `x`, `y`, and `z` arrays, corresponding to the pixels in the image area with the `meshgrid()`, `zeros()`, and `linspace()` functions:

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

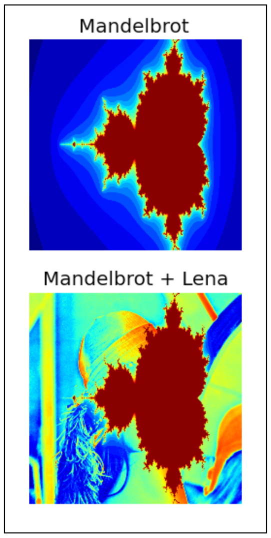
2. If `z` is a complex number, we have this relation for a Mandelbrot fractal:

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

Here, `c` is a constant complex number. This can be graphed in the complex plane with the horizontal axis showing real values and the vertical axis showing imaginary values. We will use the so-called **escape time algorithm** to draw the fractal. This algorithm scans the points in a small region around the origin at a distance of about 2 units. 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
Plot the fractal with matplotlib:
plt.subplot(211)
plt.imshow(fractal)
plt.title('Mandelbrot')
plt.axis('off')
Use the choose() function to pick a value from the fractal or Lena
array:
plt.subplot(212)
plt.imshow(numpy.choose(fractal < lena, [fractal, lena]))
plt.axis('off')
plt.title('Mandelbrot + Lena')
```

The resulting image is shown here:



The following is the complete code for this recipe from the `mandelbrot.py` file in this book's code bundle:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.misc import lena

ITERATIONS = 10
lena = 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 = np.meshgrid(np.linspace(x_min, x_max, SIZE),
                   np.linspace(y_min, y_max, SIZE))
c = x + 1j * y
z = c.copy()
fractal = np.zeros(z.shape, dtype=np.uint8) + MAX_COLOR
# Generate fractal
for n in range(ITERATIONS):
    mask = np.abs(z) <= 4
    z[mask] = z[mask] ** 2 +  c[mask]
    fractal[(fractal == MAX_COLOR) & (-mask)] = (MAX_COLOR - 1) *
n / ITERATIONS

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

# Combine with lena
plt.subplot(212)
plt.imshow(np.choose(fractal < lena, [fractal, lena]))
plt.axis('off')
plt.title('Mandelbrot + Lena')

plt.show()
```

How it works...

The following functions were used in this example:

Function	Description
`linspace()`	This function returns numbers within a range with a specified interval between them
`choose()`	This function creates an array by choosing values from arrays based on a condition
`meshgrid()`	This function 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 Arrays*

Blurring images

We can blur images with a Gaussian filter (`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 `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 start by initializing the polar plots, after which we will blur the Lena image and plot using polar coordinates:

1. Initialize the polar plots:

```
NFIGURES = 5
k = np.random.random_integers(1, 5, NFIGURES)
a = np.random.random_integers(1, 5, NFIGURES)
colors = ['b', 'g', 'r', 'c', 'm', 'y', 'k']
```

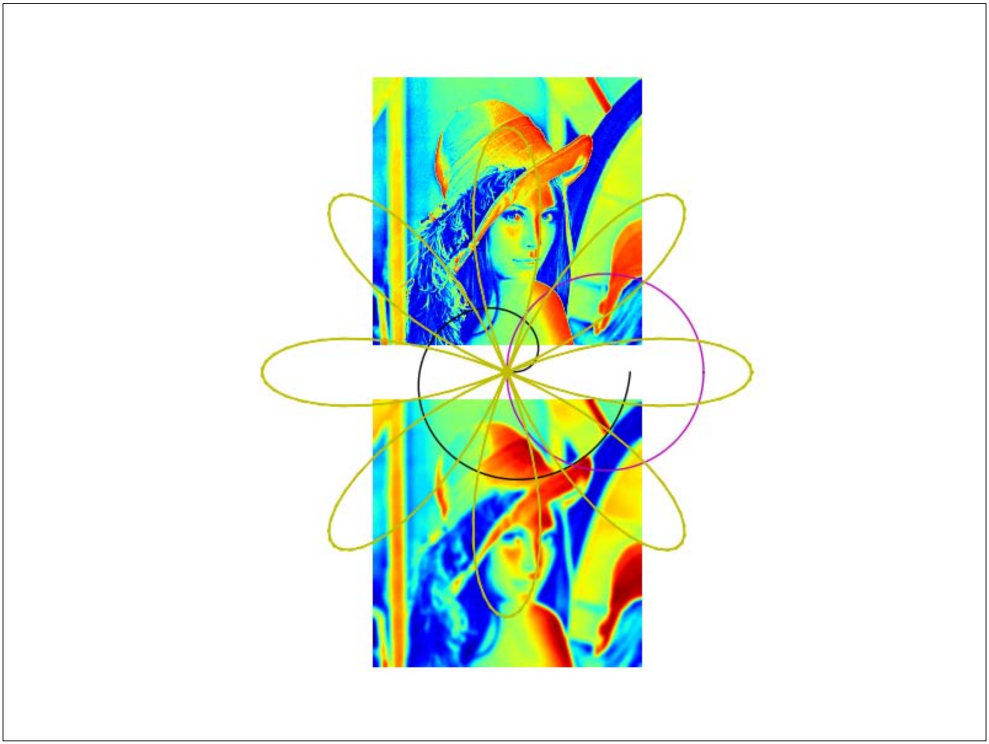
2. To blur Lena, apply the Gaussian filter with a standard deviation of `4`:

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

```
plt.imshow(blurred)
plt.axis('off')
```

3. matplotlib has a `polar()` function, which plots in polar coordinates:

```
theta = np.linspace(0, k[0] * np.pi, 200)
plt.polar(theta, np.sqrt(theta), choice(colors))

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



Here is the complete code for this recipe from the `spiral.py` file in this book's code bundle:

```
import numpy as np
import matplotlib.pyplot as plt
from random import choice
import scipy
import scipy.ndimage
```

```
# Initialization
NFIGURES = 5
k = np.random.random_integers(1, 5, NFIGURES)
a = np.random.random_integers(1, 5, NFIGURES)

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

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

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

plt.imshow(blurred)
plt.axis('off')

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

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

plt.axis('off')

plt.show()
```

How it works...

We made use of the following functions in this tutorial:

Function	**Description**
`gaussian_filter()`	This function applies a Gaussian filter
`random_integers()`	This function returns an array with random integer values between a high and low bound
`polar()`	This function plots a figure using polar coordinates

See also

- The `scipy.ndimage` documentation can be found at `http://docs.scipy.org/doc/scipy-0.14.0/reference/ndimage.html`

Repeating audio fragments

As we saw in *Chapter 2*, *Advanced Indexing and Array Concepts*, we can do neat things with WAV files. It's just a matter of downloading the file with the `urllib2` standard Python module 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've already seen in *Chapter 2*, *Advanced Indexing and Array Concepts*.

How to do it...

1. 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])
   ```

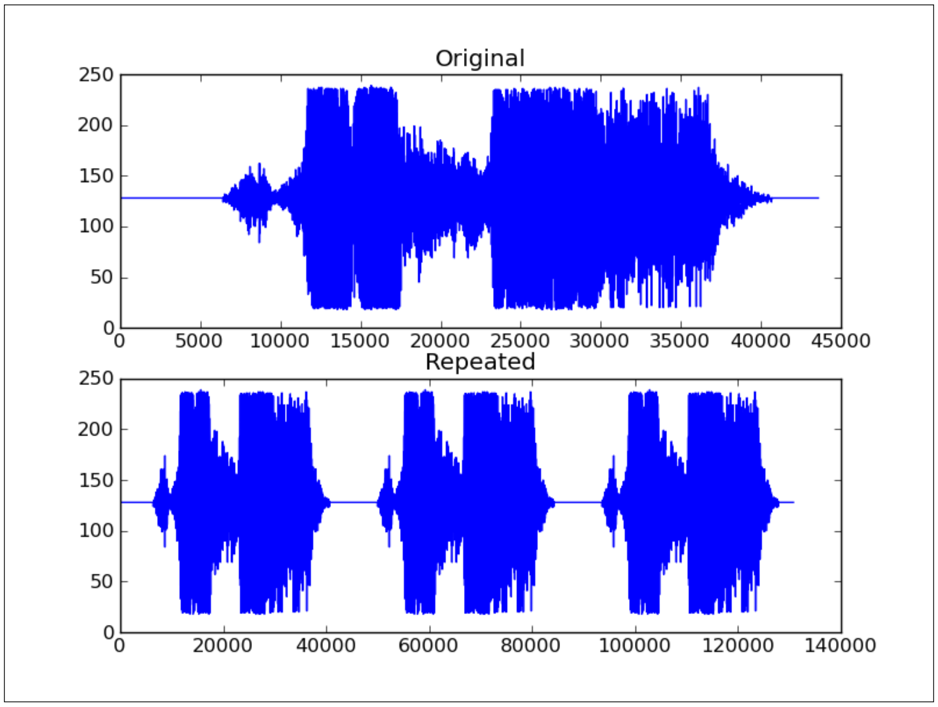
   Now, armed with this knowledge, apply the `tile()` function:

   ```
   repeated = np.tile(data, 3)
   ```

2. Plot the audio data with matplotlib:

   ```
   plt.title("Repeated")
   plt.plot(repeated)
   ```

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



Here is the complete code for this recipe from the `repeat_audio.py` file in this book's code bundle:

```
import scipy.io.wavfile
import matplotlib.pyplot as plt
import urllib2
import numpy as np

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)
```

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

plt.subplot(2, 1, 2)

# Repeat the audio fragment
repeated = np.tile(data, 3)

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

plt.show()
```

How it works...

The following are the most important functions in this recipe:

Function	**Description**
`scipy.io.wavfile.read()`	Reads a WAV file into an array
`numpy.tile()`	Repeats an array a specified number of times
`scipy.io.wavfile.write()`	Creates a WAV file out of a NumPy array with a specified sample rate

See also

- The `scipy.io` documentation can be found at `http://docs.scipy.org/doc/scipy-0.14.0/reference/io.html`

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 in `http://en.wikipedia.org/wiki/Piano_key_frequencies` that comply with the following formula:

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

Here, `n` is the number of the piano key. We will number the keys from 1 to 88. We will select the amplitude, duration, and phase at random.

How to do it...

Begin by initializing random values, then generate sine waves, compose a melody, and finally, plot the generated audio data with matplotlib:

1. Initialize to random values:
   - The amplitude between `200` - `2000`
   - The duration to `0.01` - `0.2`
   - The frequencies using the formula already mentioned
   - The phase to values between `0` and `2 pi`:

   ```
   NTONES = 89
   amps = 2000. * np.random.random((NTONES,)) + 200.
   durations = 0.19 * np.random.random((NTONES,)) + 0.01
   keys = np.random.random_integers(1, 88, NTONES)
   freqs = 440.0 * 2 ** ((keys - 49.)/12.)
   phi = 2 * np.pi * np.random.random((NTONES,))
   ```

2. Write a `generate()` function to generate sine waves:

   ```
   def generate(freq, amp, duration, phi):
    t = np.linspace(0, duration, duration * RATE)
    data = np.sin(2 * np.pi * freq * t + phi) * amp

    return data.astype(DTYPE)
   ```

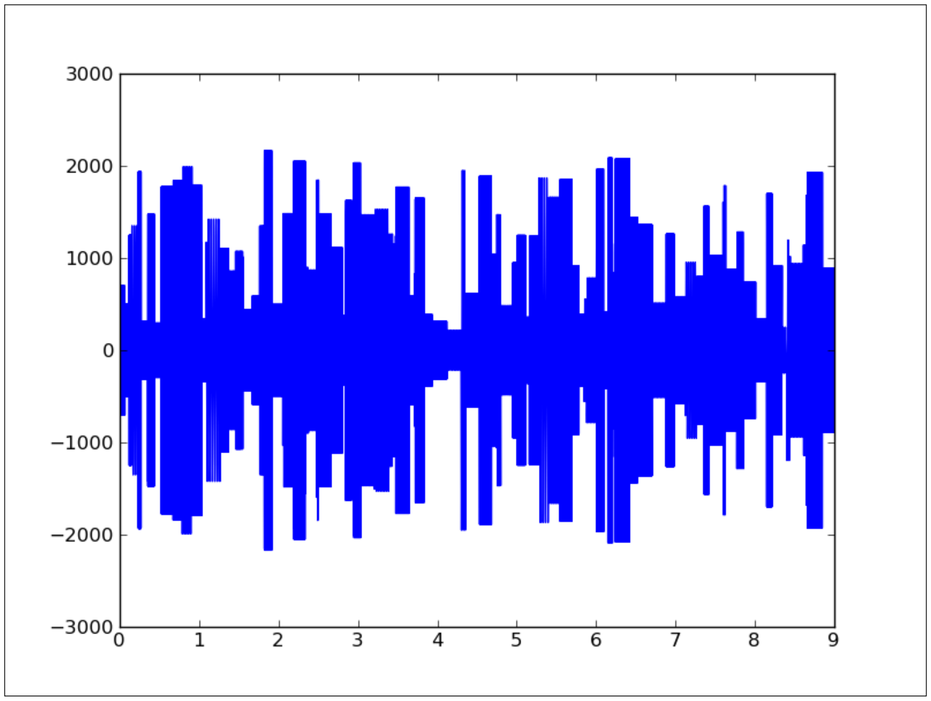
3. Once we have generated a few tones, we only need to compose a coherent melody. For now, we will just concatenate the sine waves. This does not give a nice melody, but can serve as a starting point for more experimenting:

```
for i in xrange(NTONES):
   newtone = generate(freqs[i], amp=amps[i], 
duration=durations[i], phi=phi[i])
   tone = np.concatenate((tone, newtone))
```

4. Plot the generated audio data with matplotlib:

```
plt.plot(np.linspace(0, len(tone)/RATE, len(tone)), tone)
plt.show()
```

The generated audio data plot is as follows:



The source code for this example can be found here, and it is from the `tone_generation.py` file in this book's code bundle:

```
import scipy.io.wavfile
import numpy as np
import matplotlib.pyplot as plt

RATE = 44100
DTYPE = np.int16

# Generate sine wave
def generate(freq, amp, duration, phi):
 t = np.linspace(0, duration, duration * RATE)
 data = np.sin(2 * np.pi * freq * t + phi) * amp

 return data.astype(DTYPE)

# Initialization
NTONES = 89
amps = 2000. * np.random.random((NTONES,)) + 200.
durations = 0.19 * np.random.random((NTONES,)) + 0.01
keys = np.random.random_integers(1, 88, NTONES)
freqs = 440.0 * 2 ** ((keys - 49.)/12.)
phi = 2 * np.pi * np.random.random((NTONES,))

tone = np.array([], dtype=DTYPE)

# Compose
for i in xrange(NTONES):
   newtone = generate(freqs[i], amp=amps[i], 
duration=durations[i], phi=phi[i])
   tone = np.concatenate((tone, newtone))

scipy.io.wavfile.write('generated_tone.wav', RATE, tone)

# Plot audio data
plt.plot(np.linspace(0, len(tone)/RATE, len(tone)), tone)
plt.show()
```

How it works...

We created a WAV file with randomly generated sounds. The `concatenate()` function was used to concatenate sine waves.

See also

- The `concatenate()` function is documented at `http://docs.scipy.org/doc/numpy/reference/generated/numpy.concatenate.html`

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 with the `iirdesign()` function of the `scipy.signal` module. IIR stands for infinite impulse response; for more information, see `http://en.wikipedia.org/wiki/Infinite_impulse_response`.
   We are not going to go into all the details of the `iirdesign()` function. Take a look at the documentation at `http://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.iirdesign.html` if necessary. In short, these are the parameters we will set:
   - 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 fillter (`http://en.wikipedia.org/wiki/Butterworth_filter`). The **Butterworth filter** was first described by the physicist Stephen Butterworth in 1930.

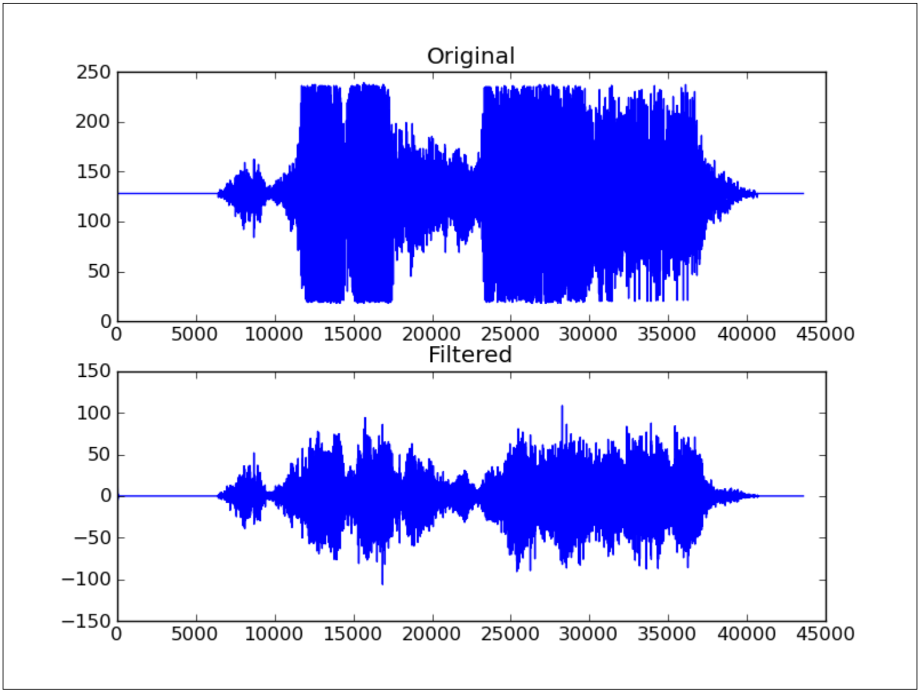
2. Apply the filter with the `scipy.signal.lfilter()` function. It accepts as arguments the values from the previous step and, of course, the data array to filter:

   ```
   filtered = scipy.signal.lfilter(b, a, data)
   ```

   When writing the new audio file, 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 listed as follows:

```
import scipy.io.wavfile
import matplotlib.pyplot as plt
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)

plt.subplot(2, 1, 1)
plt.title("Original")
plt.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
plt.subplot(2, 1, 2)
plt.title("Filtered")
plt.plot(filtered)

scipy.io.wavfile.write('filtered.wav', sample_rate, filtered.
astype(data.dtype))

plt.show()
```

How it works...

We created and applied a Butterworth bandpass filter. The following functions were introduced to create the filter:

Function	**Description**
`scipy.signal.iirdesign()`	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`.
`scipy.signal.lfilter()`	Filters an array, given a digital filter.

Edge detection with the Sobel filter

The **Sobel operator** (`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. Since 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 Söderberg.

How to do it...

In this section, you will learn how to apply the Sobel filter to detect edges in the Lena image:

1. To apply the Sobel filter in the *x* direction, set the axis parameter to `0`:

   ```
   sobelx = scipy.ndimage.sobel(lena, axis=0, mode='constant')
   ```

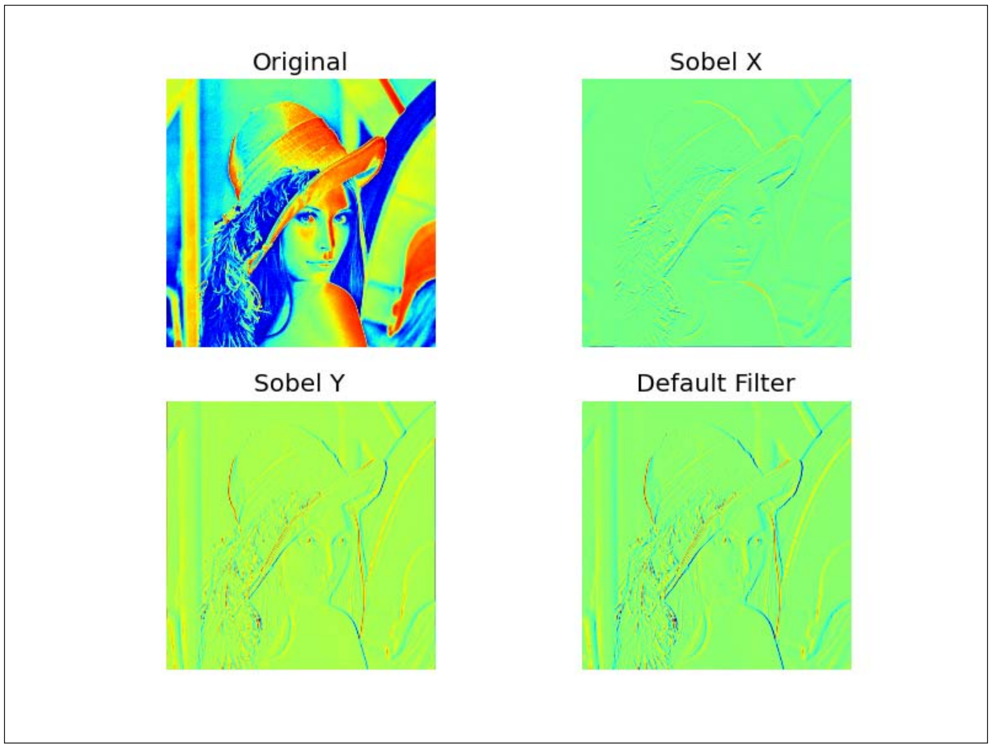
2. To apply the Sobel filter in the *y* direction, set the axis parameter to `1`:

   ```
   sobely = scipy.ndimage.sobel(lena, axis=1, mode='constant')
   ```

3. The default Sobel filter only requires the input array:

   ```
   default = scipy.ndimage.sobel(lena)
   ```

   Here are 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 as plt

lena = scipy.misc.lena()

plt.subplot(221)
plt.imshow(lena)
plt.title('Original')
plt.axis('off')

# Sobel X filter
sobelx = scipy.ndimage.sobel(lena, axis=0, mode='constant')

plt.subplot(222)
plt.imshow(sobelx)
plt.title('Sobel X')
plt.axis('off')

# Sobel Y filter
sobely = scipy.ndimage.sobel(lena, axis=1, mode='constant')

plt.subplot(223)
plt.imshow(sobely)
plt.title('Sobel Y')
plt.axis('off')

# Default Sobel filter
default = scipy.ndimage.sobel(lena)

plt.subplot(224)
plt.imshow(default)
plt.title('Default Filter')
plt.axis('off')

plt.show()
```

How it works...

We applied the Sobel filter to the picture of the famous model Lena Söderberg. As we saw in this example, we can specify along which axis to do the computation. The default setting is axis independent.

6
Special Arrays and Universal Functions

In this chapter, we will cover the following recipes:

- 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 a `recarray` function

Introduction

This chapter is about special arrays and universal functions. These are topics that you may not encounter every day, but they 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 the input and produce a set of scalars as the output. Universal functions can typically be mapped onto their mathematical counterparts such as addition, subtraction, division, multiplication, and so on. The special arrays mentioned here are all subclasses of 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 `frompyfunc()` NumPy function.

How to do it...

The following steps help us create a universal function:

1. Define a simple Python function that doubles the input:

   ```
   def double(a):
       return 2 * a
   ```

2. Create the universal function with `frompyfunc()`. Specify the number of input arguments and the number of objects (both are equal to `1`) returned:

   ```
   from __future__ import print_function
   import numpy as np

   def double(a):
      return 2 * a

   ufunc = np.frompyfunc(double, 1, 1)
   print("Result", ufunc(np.arange(4)))
   ```

   The code prints the following output when executed:

   ```
   Result [0 2 4 6]
   ```

How it works...

We defined a Python function that doubles the numbers it receives. Actually, we can also have strings as the input, as that is legal in Python. We created a universal function from this Python function with the `frompyfunc()` NumPy function. A universal function is a NumPy class with special features such as broadcasting and element-by-element processing as applicable to NumPy arrays. Many NumPy functions are, in fact, universal functions, but were written in C.

See also

- The documentation of the `frompyfunc()` NumPy function is at `http://docs.scipy.org/doc/numpy/reference/generated/numpy.frompyfunc.html`

Finding Pythagorean triples

For this tutorial, you may need to read the Wikipedia page about **Pythagorean triple** (`http://en.wikipedia.org/wiki/Pythagorean_triple`). A Pythagorean triple is a set of three natural numbers, a < b < c, for which $a^2 + b^2 = c^2$.

Here is an example of Pythagorean triples: $3^2 + 4^2 = 5^2$.

Pythagorean triples are closely related to the **Pythagorean Theorem**, which you have probably learned 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 1,000. 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 universal functions in action.

How to do it...

Euclid's formula defines the `m` and `n` indices.

1. Create arrays to hold these indices:

   ```
   m = np.arange(33)
   n = np.arange(33)
   ```

2. The second step is to calculate the numbers `a`, `b`, and `c` of the Pythagorean triples using Euclid's formula. Use the `outer()` function to get the Cartesian products, difference, and sums:

   ```
   a = np.subtract.outer(m ** 2, n ** 2)
   b = 2 * np.multiply.outer(m, n)
   c = np.add.outer(m ** 2, n ** 2)
   ```

3. Now, 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 `where()` NumPy function:

   ```
   idx =  np.where((a + b + c) == 1000)
   ```

4. Check the solution with the `numpy.testing` module:

   ```
   np.testing.assert_equal
     (a[idx]**2 + b[idx]**2, c[idx]**2)
   ```

The following code is from the `triplets.py` file in this book's code bundle:

```
from __future__ import print_function
import numpy as np

#A Pythagorean triplet is a set of three natural numbers, a < b < c,
for which,
```

```
#a ** 2 + b ** 2 = c ** 2
#
#For example, 3 ** 2 + 4 ** 2 = 9 + 16 = 25 = 5 ** 2.
#
#There exists exactly one Pythagorean triplet for which a + b + c =
1000.
#Find the product abc.

#1. Create m and n arrays
m = np.arange(33)
n = np.arange(33)

#2. Calculate a, b and c
a = np.subtract.outer(m ** 2, n ** 2)
b = 2 * np.multiply.outer(m, n)
c = np.add.outer(m ** 2, n ** 2)

#3. Find the index
idx =  np.where((a + b + c) == 1000)

#4. Check solution
np.testing.assert_equal(a[idx]**2 + b[idx]**2, c[idx]**2)
print(a[idx], b[idx], c[idx])
      # [375] [200] [425]
```

How it works...

Universal functions are not real functions, but objects representing functions. Ufuncs have the `outer()` method, which we saw in action. Many of NumPy's 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 means fewer loops.

See also

- The documentation for the `outer()` universal function at `http://docs.scipy.org/doc/numpy/reference/generated/numpy.ufunc.outer.html`

Performing string operations with chararray

NumPy has a specialized `chararray` object that holds strings. It is a subclass of `ndarray` and has special string methods. We will download 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 as a view:

   ```
   carray = np.array(html).view(np.chararray)
   ```

2. Expand tabs to spaces with the `expandtabs()` function. This function accepts the tab size as an argument. The value is `8` if not specified:

   ```
   carray = carray.expandtabs(1)
   ```

3. Split lines with the `splitlines()` function into separate lines:

   ```
   carray = carray.splitlines()
   ```

   The following is the complete code for this example:

   ```
   import urllib2
   import numpy as np
   import re

   response = urllib2.urlopen('http://python.org/')
   html = response.read()
   html = re.sub(r'<.*?>', '', html)
   carray = np.array(html).view(np.chararray)
   carray = carray.expandtabs(1)
   carray = carray.splitlines()
   print(carray)
   ```

How it works...

We saw the specialized `chararray` class in action. It offers several vectorized string operations and convenient behavior regarding whitespace.

See also

- The documentation for the specialized `chararray` class is at `http://docs.scipy.org/doc/numpy/reference/generated/numpy.chararray.html`

Creating a masked array

Masked arrays can be used to ignore missing or invalid data items. A `MaskedArray` class from the `numpy.ma` module is a subclass of `ndarray`, with a mask. We will use the Lena Söderberg image as the data source and pretend that some of this data is corrupt. Finally, 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. To create a masked array, we need to specify a mask. Create a random mask with values that are either `0` or `1`:

   ```
   random_mask = np.random.randint
     (0, 2, size=lena.shape)
   ```

2. Using the mask from the previous step, create a masked array:

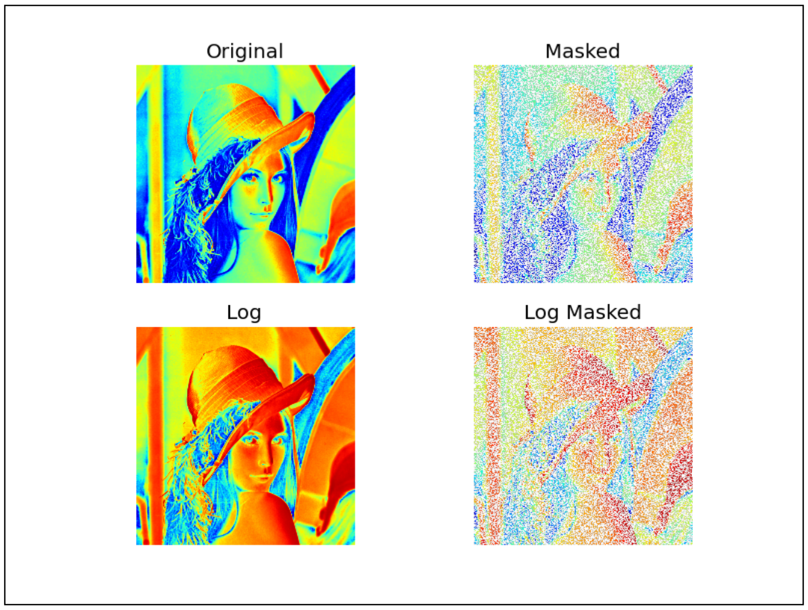
   ```
   masked_array = np.ma.array
     (lena, mask=random_mask)
   ```

   The following is the complete code for this masked array tutorial:

   ```
   from __future__ import print_function
   import numpy as np
   from scipy.misc import lena
   import matplotlib.pyplot as plt


   lena = lena()
   random_mask = np.random.randint(0, 2, size=lena.shape)

   plt.subplot(221)
   plt.title("Original")
   plt.imshow(lena)
   plt.axis('off')

   masked_array = np.ma.array(lena, mask=random_mask)
   print(masked_array)
   plt.subplot(222)
   plt.title("Masked")
   plt.imshow(masked_array)
   plt.axis('off')
   ```

```
plt.subplot(223)
plt.title("Log")
plt.imshow(np.log(lena))
plt.axis('off')

plt.subplot(224)
plt.title("Log Masked")
plt.imshow(np.log(masked_array))
plt.axis('off')

plt.show()
```

Here is a screenshot that shows the resulting images:



How it works...

We applied a random mask to NumPy arrays. This had the effect of ignoring the data that corresponds to the mask. You can find a range of masked array operations in the `numpy.ma` module. In this tutorial, we only demonstrated how to create a masked array.

See also

- The documentation for the `numpy.ma` module is at `http://docs.scipy.org/doc/numpy/reference/maskedarray.html`

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 upper and lower bounds for extreme values.

We will apply these techniques to stock price data. We will skip the steps for downloading data, as they were already covered in the previous chapters.

How to do it...

We will take the logarithm of an array that contains negative numbers:

1. Create an array containing numbers divisible by three:

   ```
   triples = np.arange(0, len(close), 3)
   print("Triples", triples[:10], "...")
   ```

   Next, create an array with the ones that have the same size as the price data array:

   ```
   signs = np.ones(len(close))
   print("Signs", signs[:10], "...")
   ```

   Set every third number to be negative, with the help of the indexing tricks you learned in *Chapter 2*, *Advanced Indexing and Array Concepts*.

   ```
   signs[triples] = -1
   print("Signs", signs[:10], "...")
   ```

   Finally, take the logarithm of this array:

   ```
   ma_log = np.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. Let's define extreme values as being one standard deviation below the mean, or one standard deviation above the mean (this is just for demonstration purposes). Write the following code to mask extreme values:

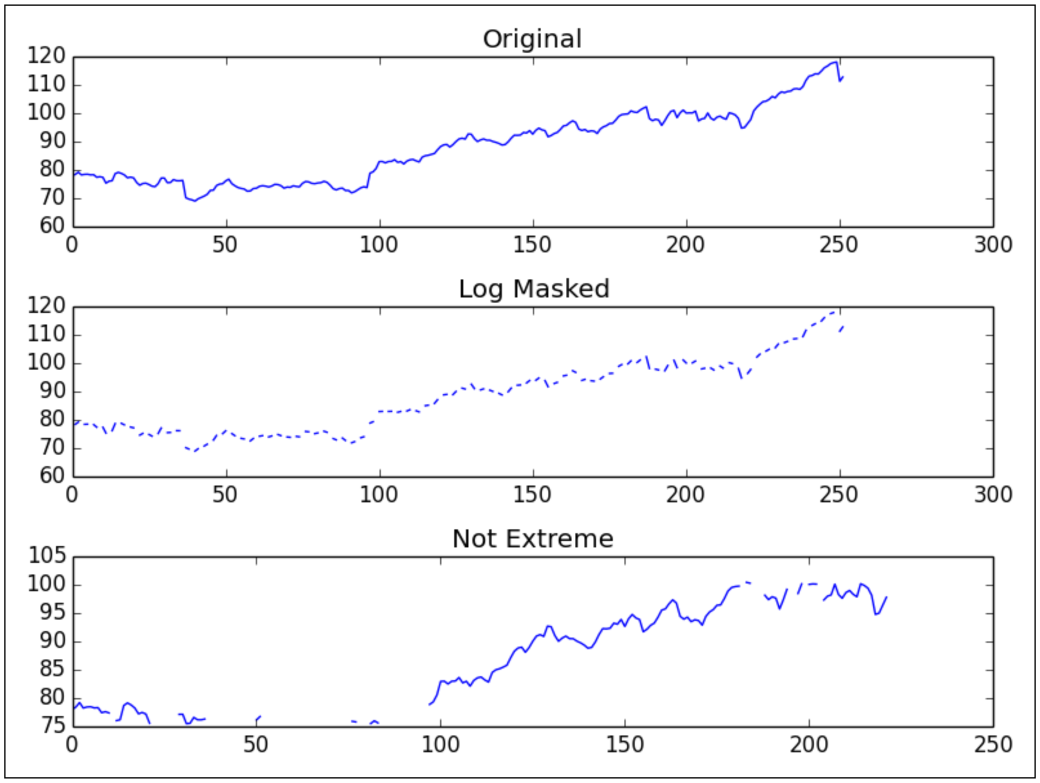
   ```
   dev = close.std()
   avg = close.mean()
   ```

```
inside = numpy.ma.masked_outside
  (close, avg - dev, avg + dev)
print("Inside", inside[:10], "...")
```

This code prints the first ten elements:

```
Inside [-- -- -- -- -- -- 409.429675172
  410.240597855 -- --] ...
```

Plot the original price data, the data after taking the logarithm, the exponent back again, and finally the data after applying the standard-deviation-based mask. The following screenshot shows the result (for this run):



The complete program for this tutorial is as follows:

```
from __future__ import print_function
import numpy as np
from matplotlib.finance import quotes_historical_yahoo
from datetime import date
import matplotlib.pyplot as plt
```

```
def get_close(ticker):
   today = date.today()
   start = (today.year - 1, today.month, today.day)

   quotes = quotes_historical_yahoo(ticker, start, today)

   return np.array([q[4] for q in quotes])


close = get_close('AAPL')

triples = np.arange(0, len(close), 3)
print("Triples", triples[:10], "...")

signs = np.ones(len(close))
print("Signs", signs[:10], "...")

signs[triples] = -1
print("Signs", signs[:10], "...")

ma_log = np.ma.log(close * signs)
print("Masked logs", ma_log[:10], "...")

dev = close.std()
avg = close.mean()
inside = np.ma.masked_outside(close, avg - dev, avg + dev)
print("Inside", inside[:10], "...")

plt.subplot(311)
plt.title("Original")
plt.plot(close)

plt.subplot(312)
plt.title("Log Masked")
plt.plot(np.exp(ma_log))

plt.subplot(313)
plt.title("Not Extreme")
plt.plot(inside)

plt.tight_layout()
plt.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` or `None` value in databases and programming. All operations with a masked value result in a masked value.

See also

- The documentation for the `numpy.ma` module is at `http://docs.scipy.org/doc/numpy/reference/maskedarray.html`

Creating a scores table with a recarray function

The `recarray` class is a subclass of `ndarray`. These arrays can hold records as 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 investing boils down to optimizing risk and reward. Risk is represented by the **standard deviation of log returns** (see `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. This is just theoretical and untested, so do not take it too seriously. 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...

Let's start by creating the record array:

1. Create a record array with a symbol, standard deviation score, mean score, and overall score for each record:

```
weights = np.recarray((len(tickers),),
  dtype=[('symbol', np.str_, 16),
    ('stdscore', float), ('mean', float),
      ('score', float)])
```

2. To keep things simple, initialize the scores in a loop based on the log returns:

```
for i, ticker in enumerate(tickers):
   close = get_close(ticker)
   logrets = np.diff(np.log(close))
   weights[i]['symbol'] = ticker
   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. We now have some numbers, but they are difficult 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. 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:

```
from __future__ import print_function
import numpy as np
from matplotlib.finance import quotes_historical_yahoo
from datetime import date

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 np.array([q[4] for q in quotes])


weights = np.recarray((len(tickers),), dtype=[('symbol', np.str_,
16),
   ('stdscore', float), ('mean', float), ('score', float)])

for i, ticker in enumerate(tickers):
   close = get_close(ticker)
   logrets = np.diff(np.log(close))
   weights[i]['symbol'] = ticker
   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=0.8185,stdscore=0.2938,score=0.2177
T,mean=0.0927,stdscore=0.3427,score=0.2262
VZ,mean=0.0888,stdscore=0.3636,score=0.5561
```

The score is normalized, so the values are between `0` and `1`, and we try to get the optimal return and risk combinations using the definitions from the start of the recipe. According to the output, `VZ` has the highest score, and therefore is the best investment. Of course, this is just a NumPy demo with little data, so don't consider this a recommendation.

How it works...

We computed scores for several stocks, and stored them in a `recarray` NumPy 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. You can find more record-array-related functions in the `numpy.recarray` module.

See also

- The documentation for the `numpy.recarray` module is at `http://docs.scipy.org/doc/numpy/reference/generated/numpy.recarray.html`

7
Profiling and Debugging

In this chapter, we will cover the following recipes:

- 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 `PuDB`

Introduction

Debugging is the act of finding and removing bugs from software. Profiling means building a profile of a program in order to collect information about memory usage or time complexity. Profiling and debugging are activities that are integral to the life of a developer. This is especially true for complex software. The good news is that many tools can help you. We will review techniques popular among 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 `sort()` NumPy function with several 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 varying sizes containing random integer values:

```
times = np.array([])

for size in sizes:
    integers = np.random.random_integers
      (1, 10 ** 6, size)
```

2. To measure time, create a timer, give it a function to execute, and specify the relevant imports. Then, sort 100 times to get data about the sorting times:

```
def measure():
    timer = timeit.Timer('dosort()',
      'from __main__ import dosort')
    return timer.timeit(10 ** 2)
```

3. Build the measurement time arrays by appending the times one by one:

```
times = np.append(times, measure())
```

4. Fit the times into the theoretical model of `n log n`. Since we are varying the array size as powers of two, this is easy:

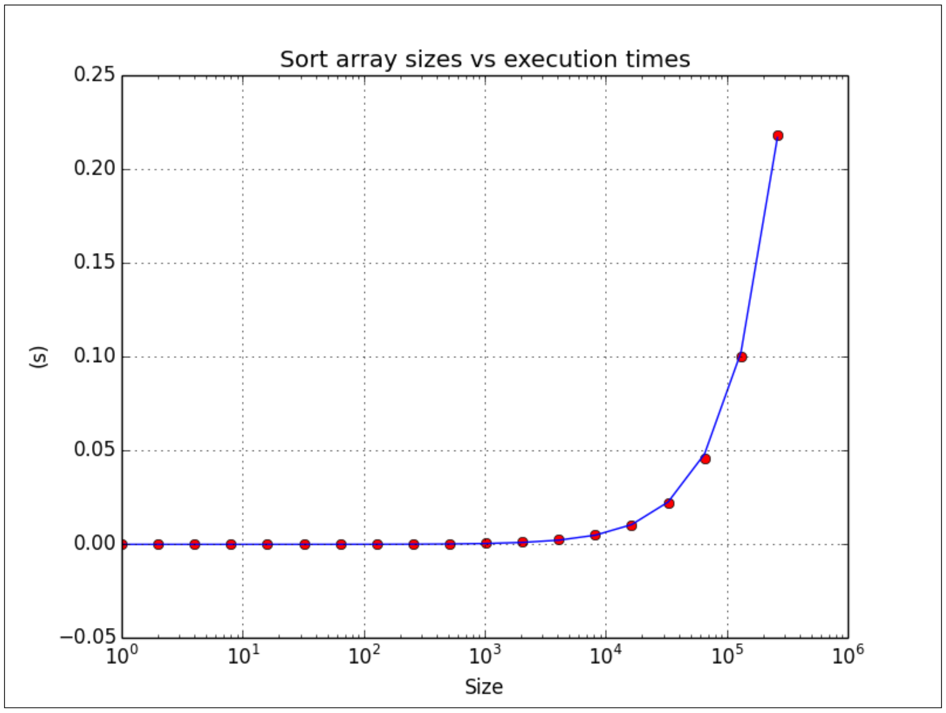
```
fit = np.polyfit(sizes * powersOf2, times, 1)
```

The following is the complete timing code:

```
import numpy as np
import timeit
import matplotlib.pyplot as plt


# 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 = np.arange(0, 19)
```

```
sizes = 2 ** powersOf2

times = np.array([])

for size in sizes:
   integers = np.random.random_integers(1, 10 ** 6, size)
   times = np.append(times, measure())

fit = np.polyfit(sizes * powersOf2, times, 1)
print(fit)
plt.title("Sort array sizes vs execution times")
plt.xlabel("Size")
plt.ylabel("(s)")
plt.semilogx(sizes, times, 'ro')
plt.semilogx(sizes, np.polyval(fit, sizes * powersOf2))
plt.grid()
plt.show()
```

The following screenshot shows the resulting plot for the running time versus array size:



How it works...

We measured the average running time of the `sort()` NumPy function. The following functions were used in this recipe:

Function	Description
`random_integers()`	This function creates an array of random integers when a range is given for the values and array size
`append()`	This function appends a value to a NumPy array
`polyfit()`	This function fits data into a polynomial of a given degree
`polyval()`	This function evaluates a polynomial and returns the corresponding value for a given value of `x`
`semilogx()`	This function plots data using a logarithmic scale on the *X* axis

See also

- The documentation for `timeit` is at `http://docs.python.org/2/library/timeit.html`

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. 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, 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 the following small script that inverts a matrix of varying size containing random values. The `.I` attribute (that's an uppercase I) of a NumPy matrix represents the inverse of that matrix:

```
import numpy as np

def invert(n):
  a = np.matrix(np.random.rand(n, n))

  return a.I

sizes = 2 ** np.arange(0, 12)

for n in sizes:
  invert(n)
```

Time this code 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 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(invert)
       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 ran the aforementioned NumPy code through a profiler. The following table summarizes the profiler output:

Column	Description
`ncalls`	This is the number of calls
`tottime`	This is the total time spent in a function
`percall`	This is the time spent per call, calculated by dividing the total time by the count of calls
`cumtime`	This is the cumulative time spent in function and functions called by the function, including recursive calls

See also

- The IPython magics documentation at `http://ipython.org/ipython-doc/dev/interactive/magics.html`

Installing line_profiler

`line_profiler` was created by one of the developers of NumPy. 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 was covered in a previous recipe; refer to the *See also* section if necessary. In order to install the development version, you will need Git. Installing Git is beyond the scope of this book.

How to do it...

Choose the install option appropriate for you:

- Install `line_profiler` with `easy_install`, using any one of the following commands:

  ```
  $ easy_install line_profiler
  $ pip install line_profiler
  ```

- Install the development version.

  Check out the source with Git:

  ```
  $ git clone https://github.com/rkern/line_profiler
  ```

  After checking out the source, build it as follows:

  ```
  $ python setup.py install
  ```

See also

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

Profiling code with line_profiler

Now that we've installed `line_profiler`, we can start profiling.

How to do it...

Obviously, we will need code to profile:

1. Write the following code to multiply a random matrix of varying size by itself. Also, the thread will sleep for a few seconds. Annotate the function to profile with `@profile`:

```
import numpy as np
import time

@profile
def multiply(n):
  A = np.random.rand(n, n)
  time.sleep(np.random.randint(0, 2))
  return np.matrix(A) ** 2

for n in 2 ** np.arange(0, 10):
  multiply(n)
```

2. 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`	The time spent executing the line
`Per Hit`	The average time spent executing the line
`% Time`	The percentage of time spent executing the line relative to the time spent executing all the lines
`Line Contents`	The content of the line

See also

- The Github `line_profiler` project page is at `https://github.com/rkern/line_profiler`

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 obtained are precise and no sampling is used. This contrasts with statistical profiling, where measurements come from random samples. We will profile a small NumPy program using `cProfile`, which transposes an array with random values.

How to do it...

Again, we require code to profile:

1. Write the following `transpose()` function to create an array with random values and transpose it:

```
def transpose(n):
  random_values = np.random.random((n, n))
  return random_values.T
```

2. Run the profiler and give it the function to profile:

   ```
   cProfile.run('transpose (1000)')
   ```

   The complete code for this tutorial can be found in the following snippet:

   ```
   import numpy as np
   import cProfile

   def transpose(n):
      random_values = np.random.random((n, n))
      return random_values.T

   cProfile.run('transpose (1000)')
   ```

   For a 1000 x 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 those seen in the IPython profiling recipe.

See also

- The Python profilers documentation at `http://docs.python.org/2/library/profile.html`
- The working with `pstats` tutorial at `http://pymotw.com/2/profile/#module-pstats`

Debugging with IPython

"If debugging is the process of removing software bugs, then programming must be the process of putting them in." Edsger Dijkstra, Dutch computer scientist, winner of the 1972 Turing Award

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've found the bug and implemented a fix, you should have a unit test in place (if the bug has a related ID from an issue tracker, I usually name the test by appending the ID at the end). In this way, you will at least 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. It tries to access an array element that is not present:

```
import numpy as np

a = np.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. 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.pyc 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. Now that your program crashed, start the debugger. 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 the code with the `list` command, or use the shorthand `l`:

```
ipdb> list
      1 import numpy as np
      2
      3 a = np.arange(7)
----> 4 print(a[8])
```

4. We can now evaluate arbitrary code at the line the debugger is currently at:

```
ipdb> len(a)
7

ipdb> print(a)
[0 1 2 3 4 5 6]
```

5. The **call stack** is a stack containing information about active functions of a running program. 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 = np.arange(7)
----> 4 print(a[8])
```

How it works...

In this tutorial, you learned 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 the source code
`bt`	Shows the call stack
`u`	Moves up the call stack
`d`	Moves down the call stack

See also

- The Python debugger documentation at `http://docs.python.org/2/library/pdb.html`
- The `ipdb` package's homepage at `https://pypi.python.org/pypi/ipdb`

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. We can also integrate this debugger with IPython if required.

How to do it...

We'll start with the installation of `pudb`:

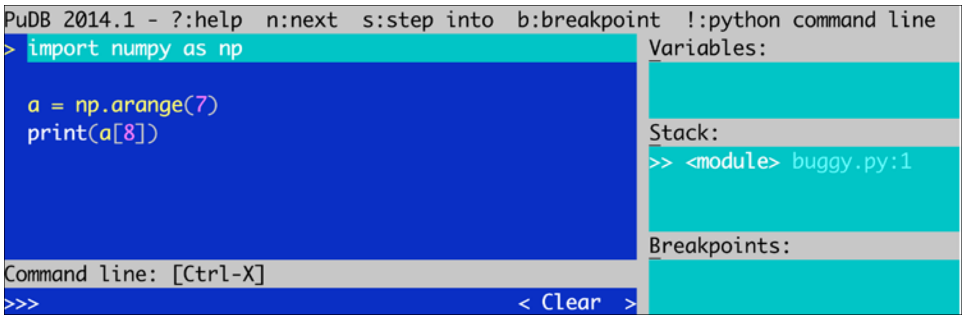
1. To install `pudb`, we only need to execute the following command (or the equivalent `pip` command):

   ```
   $ sudo easy_install pudb
   $ pip install pudb
   $ pip freeze|grep pudb
   pudb==2014.1
   ```

2. Let's debug the buggy program from the previous example. Start the debugger as follows:

   ```
   $ python -m pudb buggy.py
   ```

   The following screenshot shows the user interface of the debugger:



```
PuDB 2014.1 - ?:help  n:next  s:step into  b:breakpoint  !:python command line
> import numpy as np

  a = np.arange(7)
  print(a[8])
```

The screenshot shows the most important debugging commands at the top. We can also see the code being debugged, the 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`.

See also

- The PyPi PuDB page is at `https://pypi.python.org/pypi/pudb`

8
Quality Assurance

"If you lie to the computer, it will get you."

- Perry Farrar, Communications of the ACM, Volume 28

In this chapter, we'll cover the following recipes:

- 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 code quality, thereby preventing issues. First, we will carry out static analysis of already existing code. Then, 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 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 options to install `pyflakes`:

- Install pyflakes with the `pip` command:

  ```
  $ sudo pip install pyflakes
  ```

- Install pyflakes with the `easy_install` command:

  ```
  $ sudo easy_install pyflakes
  ```

- Here are two ways of installing this package 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 as follows:

  ```
  $ sudo apt-get install pyflakes
  ```

See also

- The Pyflakes homepage at `https://launchpad.net/pyflakes`

Performing static analysis with Pyflakes

We will perform static analysis of part of the NumPy codebase. 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...

To check out the NumPy code, we need Git. Installing Git is beyond the scope of this book (see `http://git-scm.com/book/en/v2/Getting-Started-Installing-Git`):

1. The Git command to retrieve the code is as follows:

   ```
   $ git clone git://github.com/numpy/numpy.git numpy
   ```

   Alternatively, download a source archive from `https://github.com/numpy/numpy`.

2. The previous step creates a `numpy` directory with the entire 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
   setupegg.py:17: 'setup' imported but unused
   setupscons.py:61: 'numpy' imported but unused
   setupscons.py:64: 'numscons' imported but unused
   setupsconsegg.py:6: 'setup' imported but unused
   ```

   This runs analysis on the code style and checks 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 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 and code checks. However, it is slower than Pyflakes. For more information, check out the manual at `http://www.logilab.org/card/pylint_manual`.

In this recipe, we again download the NumPy code from the Git repository—this step has been 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. Notice that we get more output. In fact, Pylint prints so much text that most of it had to be omitted here:

```
$ 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 can request HTML output if we want. The messages have the following format:

```
MESSAGE_TYPE: LINE_NUM:[OBJECT:] MESSAGE
```

The message type can be one of the following:

- `[R]`: This means that refactoring is recommended
- `[C]`: This means that there was a code style violation
- `[W]`: This is used for warnings about minor issues
- `[E]`: This is used for errors or potential bugs
- `[F]`: This indicates that a fatal error occurred, blocking further analysis

See also

- Performing static analysis with Pyflakes

Performing static analysis with Pychecker

Pychecker is an old, static analysis tool. It is not very actively developed, but it's fast and good enough to mention here. The last version at the time of writing this book was 0.8.19, and it was last updated in 2011. Pychecker tries to import each module and process it. It then searches for issues such as passing an 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. Download the `tar.gz` from `Sourceforge` (`http://pychecker.sourceforge.net/`). Unpack the source archive and run the following command:

   ```
   $ python setup.py install
   ```

   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
   ```

2. Analyze the code, just as in the previous recipes. The command we need is as follows:

   ```
   $ pychecker *.py
   ...
   Warnings...


   ...


   setup.py:21: Imported module (re) not used
   setup.py:27: Module (builtins) re-imported


   ...
   ```

Testing code with docstrings

Doctests are comment 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 possible boundary conditions. In other words, some tests will fail.

How to do it...

1. Write the `docstring` with a test that will pass and another test that will fail. The `docstring` text 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 following NumPy code:

```
return np.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. Use the `doctest` module to run the tests:

```
doctest.testmod()
```

   The following is the complete test example code from the `docstringtest.py` file in this book's code bundle:

```
import numpy as np
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 np.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.

See also

- The official `doctest` documentation at `http://docs.python.org/2/library/doctest.html`

Writing unit tests

Test-driven development (**TDD**) is the best thing that has happened to software development this century. One of the most important aspects of TDD is the almost manic focus on unit testing.

The TDD methodology uses the so-called **test-first approach**, where we first write a test that fails and then write the corresponding code to pass the test. The tests should document the developer's intent, but on a lower level than functional design. A suite of tests increases confidence by decreasing the probability of regression and facilitates refactoring.

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 test:

1. 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 np.arange(1, n+1).cumprod()
   ```

   The code is the same as what we covered in the previous recipe, but we've added a few checks for boundary conditions.

2. Let's write a class; this class will contain the unit tests. It extends the `TestCase` class from the `unittest` module, which is part of standard Python. We run tests by calling the `factorial()` function with the following:

   - a positive number—the happy path!
   - boundary condition equal to `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])
           np.testing.assert_equal(np.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()` function and the unit test in its entirety is as follows:

```
import numpy as np
import unittest

def factorial(n):
   if n == 0:
      return 1

   if n < 0:
      raise ValueError, "Don't be so negative"

   return np.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])
      np.testing.assert_equal(np.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 fails, 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 that extends the `TestCase` class from the `unittest` module. The following functions were used to perform various tests:

Function	Description
`numpy.testing.assert_equal()`	Tests whether two NumPy arrays are equal
`unittest.assertEqual()`	Tests whether two values are equal
`unittest.assertRaises()`	Tests whether an exception is thrown

The `testing` NumPy package has a number of test functions we should know about, as follows:

Function	Description
`assert_almost_equal()`	This function raises an exception if two numbers are not equal up to a specified precision
`assert_approx_equal()`	This function raises an exception if two numbers are not equal up to a certain significance
`assert_array_almost_equal()`	This function raises an exception if two arrays are not equal up to a specified amount of precision
`assert_array_equal()`	This function raises an exception if two arrays are not equal
`assert_array_less()`	This function 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

Function	Description
`assert_raises()`	This function fails if a specified exception is not raised by a callable invoked with the defined arguments
`assert_warns()`	This function fails if a specified warning is not thrown
`assert_string_equal()`	This function asserts that two strings are equal

Testing code with mocks

Mocks are objects created as substitutes for real objects with the purpose of testing part of the behavior of the real objects. If you have seen the movie Body Snatchers, you might already have an understanding of the basic idea. Generally speaking, mocking is useful only when the real objects under test are expensive to create, such as a database connection, or when testing could have undesirable side effects; for instance, we might not want to write to the file system or a 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 that, we will create a mock and test a piece of code:

1. To install the `mock` package, execute the following command:

   ```
   $ sudo easy_install mock
   ```

2. 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. We can check the behavior of a mock and, from that, the behavior of the real object under test, in several ways. For instance, assert that the potentially explosive `factorial()` method was called with the correct arguments, as follows:

   ```
   reactor.factorial.assert_called_with(3, "mocked")
   ```

   The complete test code with mocks is as follows:

   ```
   from __future__ import print_function
   from mock import MagicMock
   ```

```
import numpy as np
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 np.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")
      np.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. This 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 won'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.

See also

- The mock package homepage at `http://pypi.python.org/pypi/mock`

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, or 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 use **Lettuce** to test the factorial function.

How to do it...

In this section, you will learn how to install Lettuce, set up the tests, and write the specifications for the tests:

1. To install Lettuce, run either of the following commands:

   ```
   $ pip install lettuce
   $ sudo easy_install lettuce
   ```

2. Lettuce requires a special directory structure for the tests. In the `tests` directory, we will have a directory called `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. Coming up with 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 simple—we just write down different input values and the expected output. We have different scenarios with the `Given`, `When`, and `Then` sections, which correspond to different test steps. 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. We will define methods that correspond to the steps of our scenario. Pay extra attention to the text used to annotate the methods. It matches the text in the business scenarios file, and we use regular expressions to get the input parameters. In the first two scenarios, we match numbers, and in the last, we match any text. The `fromstring()` NumPy 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 as np

@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 = np.fromstring(expected, dtype=int, sep=',')
    np.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 np.arange(1, n+1).cumprod()
```

5. 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 NumPy's testing functions to test the different steps and the `fromstring()` function to create a NumPy array from the specifications text.

See also

- The Lettuce documentation at `http://lettuce.it/`

9
Speeding Up Code with Cython

In this chapter, we will cover the following recipes:

- Installing Cython
- Building a Hello World program
- Using Cython with NumPy
- Calling C functions
- Profiling the Cython code
- Approximating factorials with Cython

Introduction

Cython is a relatively young programming language based on Python. It allows coders to mix the speed of C with the power of 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 value 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, you will learn 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 it. The Enthought Canopy, Anaconda, and Sage distributions include Cython. For more information, see `https://www.enthought.com/products/canopy/`, `https://store.continuum.io/cshop/anaconda/`, 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...

We can install Cython using any of the following methods:

- Install Cython from a source archive by performing the following steps:
  - Download a source archive from `http://cython.org/#download`.
  - Unpack it.
  - Browse to the directory using the `cd` command.
  - Run the following command:

    ```
    $ python setup.py install
    ```

- Install Cython from the PyPI repository with any one of these commands:

  ```
  $ easy_install cython
  $ sudo pip install cython
  ```

- Install Cython on Windows using the unofficial Windows installers from `http://www.lfd.uci.edu/~gohlke/pythonlibs/#cython`.

See also

- The relevant Cython online documentation is at `http://docs.cython.org/src/quickstart/install.html`

Building a Hello World program

As is the tradition with programming languages, we will start with a Hello World example. Unlike 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. First, write some pretty simple 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. Create a file named `setup.py` to help 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 generates C code, compiles it for your platform, and produces 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 created a traditional Hello World example. Cython is a compiled language, so we need to compile our code. We wrote a `.pyx` file containing the `Hello World` code and a `setup.py` file used to generate and build the C code.

See also

- The relevant Cython online documentation is at `http://docs.cython.org/src/quickstart/build.html`

Using Cython with NumPy

We can integrate Cython and NumPy code in the same way 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. The following formula indicates how significant the ratio is:

$$\sqrt{\frac{p(1-p)}{n}}$$

In the formula, **p** is the probability and **n** is the number of observations.

How to do it...

This section describes how to use Cython with NumPy, with the following steps:

1. Write a `.pyx` file that contains a function that calculates the ratio of up days and the associated confidence. First, this function computes the differences between the prices. Then, it counts the number of positive differences, giving us a ratio for the proportion of up days. Finally, apply the formula for confidence from the Wikipedia page in the introduction:

```
import numpy as np

def pos_confidence(numbers):
   diffs = np.diff(numbers)
   n = float(len(diffs))
   p = len(diffs[diffs > 0])/n
   confidence = np.sqrt(p * (1 - p)/ n)

   return (p, confidence)
```

2. Use the `setup.py` file from the previous example as a template. Change the obvious things, 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. After building, 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'll 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 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 as in the previous tutorial—by creating a Cython module. At the end, we imported and used the Cython module.

See also

- The relevant Cython online documentation is at `http://docs.cython.org/src/tutorial/numpy.html`

Calling C functions

We can call C functions from Cython. In this example, we 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 start by writing some Cython code:

1. First, import the C `log()` function from the `libc` namespace. Then, apply this function to numbers in a `for` loop. Finally, use the NumPy `diff()` function to get the first-order difference between the log values in the second step:

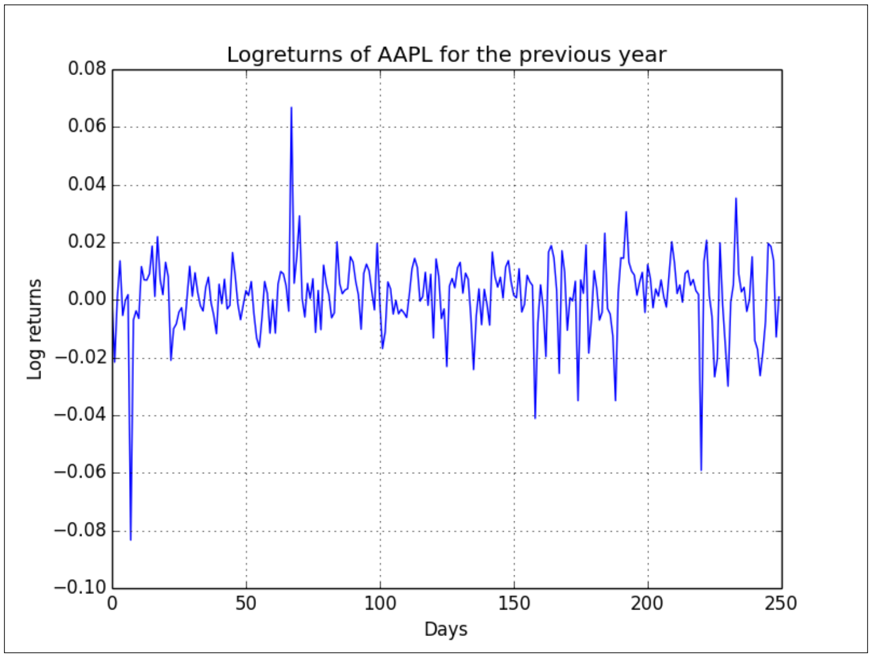
```
from libc.math cimport log
import numpy as np

def logrets(numbers):
   logs = [log(x) for x in numbers]
   return np.diff(logs)
```

   Building has been covered in the previous recipes. We only need to change some values in the `setup.py` file.

2. Again, download the stock price data with `matplotlib`. Apply the Cython `logrets()` function you just created on the prices and plot the result:

```
from matplotlib.finance import quotes_historical_yahoo
from datetime import date
import numpy as np
from log_returns import logrets
import matplotlib.pyplot as plt

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

quotes = quotes_historical_yahoo('AAPL', start, today)
close =  np.array([q[4] for q in quotes])
plt.plot(logrets(close))
plt.title('Logreturns of AAPL for the previous year')
plt.xlabel('Days')
plt.ylabel('Log returns')
plt.grid()
plt.show()
```

The resulting plot of the log returns for AAPL looks like what 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. In 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.

See also

- The relevant Cython online documentation is at `http://docs.cython.org/src/tutorial/external.html`

Profiling the Cython code

We will profile Cython and NumPy code that tries to approximate the Euler constant with the following formula:

$$\sum_{n=0}^{\infty} \frac{1}{n!}$$

See `http://en.wikipedia.org/wiki/E_%28mathematical_constant%29` for more background information.

How to do it...

This section demonstrates how to profile Cython code with the following steps:

1. For the NumPy approximation of e, follow these steps:
   - First, we will create an array of `1` to `n` (`n` is `40` in our example).
   - Then we will compute the cumulative product of this array, which happens to be the factorial. After that, we take the reciprocal of the factorials. Finally, we apply the formula from the Wikipedia page. At the end, we put the standard profiling code, giving us the following program:

```
from __future__ import print_function
import numpy as np
import cProfile
import pstats

def approx_e(n=40, display=False):
   # array of [1, 2, ... n-1]
   arr = np.arange(1, n)

   # calculate the factorials and convert to floats
   arr = arr.cumprod().astype(float)

   # reciprocal 1/n
   arr = np.reciprocal(arr)

   if display:
    print(1 + arr.sum())

# Repeat multiple times because NumPy is so fast
def run(repeat=2000):
```

```
            for i in range(repeat):
                approx_e()


        cProfile.runctx("run()", globals(), locals(),
        "Profile.prof")

        s = pstats.Stats("Profile.prof")
        s.strip_dirs().sort_stats("time").print_stats()

        approx_e(display=True)
```

The profiling output and the result for the *e* approximation are shown in the following snippet. Refer to *Chapter 7*, *Profiling and Debugging*, for more information about the profiling output:

```
         8004 function calls in 0.016 seconds


   Ordered by: internal time


   ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
     2000    0.007    0.000    0.015    0.000 numpy_approxe.
py:6(approx_e)
     2000    0.004    0.000    0.004    0.000 {method 'cumprod' of
'numpy.ndarray' objects}
     2000    0.002    0.000    0.002    0.000 {numpy.core.
multiarray.arange}
     2000    0.002    0.000    0.002    0.000 {method 'astype' of
'numpy.ndarray' objects}
        1    0.001    0.001    0.016    0.016 numpy_approxe.
py:20(run)
        1    0.000    0.000    0.000    0.000 {range}
        1    0.000    0.000    0.016    0.016 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}


2.71828182846
```

2. The Cython code uses the same algorithm as shown 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, display=False):
    cdef double sum = 0.
    cdef double factorial = 1.
    cdef int k

    for k in xrange(1,n+1):
        factorial *= k
        sum += 1/factorial

    if display:
        print(1 + sum)
```

The following Python program imports the Cython module and does some profiling:

```
import pstats
import cProfile
import pyximport
pyximport.install()

import approxe

# Repeat multiple times because Cython is so fast
def run(repeat=2000):
    for i in range(repeat):
        approxe.approx_e()

cProfile.runctx("run()", globals(), locals(), "Profile.prof")

s = pstats.Stats("Profile.prof")
s.strip_dirs().sort_stats("time").print_stats()

approxe.approx_e(display=True)
```

This is the profiling output of the Cython code:

```
         2004 function calls in 0.001 seconds


   Ordered by: internal time
   ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
     2000    0.001    0.000    0.001    0.000 {approxe.approx_e}
        1    0.000    0.000    0.001    0.001 cython_profile.
py:9(run)
        1    0.000    0.000    0.000    0.000 {range}
        1    0.000    0.000    0.001    0.001 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}



2.71828182846
```

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. However, when comparing the total time for 2,000 runs of the approximation code, we realize that NumPy needs 0.016 seconds while Cython only takes 0.001 seconds. Obviously, the actual times depend on your hardware, operating system, and other factors, such as other processes running on your machine. Also, the speedup depends on the type of code, but I hope you agree that as a rule of thumb, Cython code is faster.

See also

- The relevant Cython online documentation is at `http://docs.cython.org/src/tutorial/profiling_tutorial.html`

Approximating factorials with Cython

The last example is about approximating factorials with Cython. We will use two approximation methods. First, we will apply the Stirling approximation method (see `http://en.wikipedia.org/wiki/Stirling%27s_approximation` for more information). The formula for the Stirling approximation is as follows:

$$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Secondly, we will use the approximation due to Ramanujan, with the following formula:

$$\sqrt{\pi}(\frac{n}{e})^n \sqrt[6]{8n^3 + 4n^2 + n + \frac{1}{30}}$$

How to do it...

This section describes how to approximate factorials using Cython. In this recipe, we use types, which are optional in Cython, as you may remember. 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. 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 `cimport` NumPy. 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. Building requires us to create a `setup.py` file, as was shown in the previous tutorials, but we now include NumPy-related directories by calling the `get_include()` function. With this amendment, the `setup.py` file has the following content:

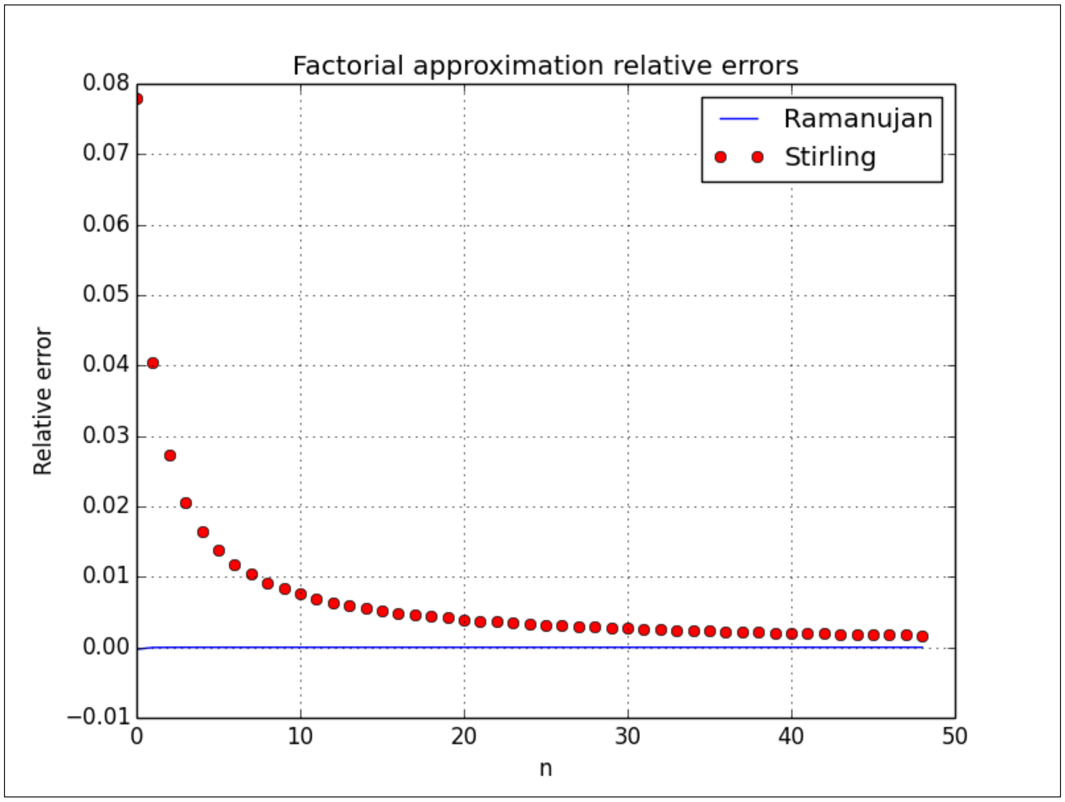
```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext
```

```
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 for both the approximation methods. Calculate the error relative to the factorial values that we will compute with the NumPy `cumprod()` function, as we did throughout the book:

```
from factorial import ramanujan_factorial
from factorial import stirling_factorial
import numpy as np
import matplotlib.pyplot as plt

N = 50
numbers = np.arange(1, N)
factorials = np.cumprod(numbers, dtype=float)

def error(approximations):
   return (factorials - approximations)/factorials

plt.plot(error(ramanujan_factorial(numbers)), 'b-',
label='Ramanujan')
plt.plot(error(stirling_factorial(numbers)), 'ro',
label='Stirling')
plt.title('Factorial approximation relative errors')
plt.xlabel('n')
plt.ylabel('Relative error')
plt.grid()
plt.legend(loc='best')
plt.show()
```

The following plot shows the relative error for the Ramanujan approximation (dots) and the Stirling approximation (line):



How it works...

In this example, we saw a demonstration of Cython's static types. The main ingredients of this recipe were the following:

- `cimport`, which imports C declarations
- Including directories with the `get_include()` function
- The `cdef` keyword, used to define the type of local variables

See also

- The relevant Cython online documentation is at `http://docs.cython.org/src/quickstart/cythonize.html`

10
Fun with Scikits

In this chapter, we will cover the following recipes:

- Installing scikit-learn
- Loading an example dataset
- Clustering Dow Jones stocks with scikit-learn
- Installing statsmodels
- Performing a normality test with statsmodels
- Installing scikit-image
- Detecting corners
- Detecting edges
- Installing pandas
- Estimating correlation of stock returns 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 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:

- scikit-learn, a machine learning package
- `statsmodels`, a statistics package
- scikit-image, an image processing package
- pandas, a data analysis package

Installing scikit-learn

The scikit-learn project aims to provide an API for machine learning. What I like the most about it is the amazing documentation. We can install scikit-learn with the package manager of our operating system. This option may or may not be available, depending on the operating system, but it should be the most convenient route.

Windows users can simply download an installer from the project website. On Debian and Ubuntu, the project is called python-sklearn. On MacPorts, the ports are called py26-scikits-learn and py27-scikits-learn. We can also install from source or using `easy_install`. There are third-party distributions available 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 IPython*, for instructions if necessary.

How to do it...

Let's now see how we can install the scikit-learn project:

- **Installing with** `easy_install`: Type 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 write `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. Then issue the following command:

  ```
  $ python setup.py install
  ```

Loading an example dataset

The scikit-learn project comes with a number of datasets and sample images that we can experiment with. In this recipe, we will load an example dataset included in the scikit-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 dataset of house prices in Boston. It is a tiny dataset, so if you are looking for a house in Boston, don't get too excited! Other datasets are described at `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 values. 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 (determining house price). The following code from `sample_data.py` accomplishes our goals:

```
from __future__ import print_function
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()))
```

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 that aims to group items based on similarities. In this example, we will use the log returns of stocks in the **Dow Jones Industrial Average** (**DJI** or **DJIA**) 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. Then, we will calculate a square affinity matrix. Finally, we will cluster the stocks with the `AffinityPropagation` class:

1. 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",
   "KO", "MCD", "MMM", "MRK", "MSFT", "PFE",
   "PG", "T", "TRV", "UTX", "VZ", "WMT", "XOM"]

quotes = []

for symbol in symbols:
    try:
        quotes.append(finance.quotes_historical_yahoo(symbol,
start, end, asobject=True))
    except urllib2.HTTPError as e:
        print(symbol, e)

close = np.array([q.close for q in quotes]).astype(np.float)
print(close.shape)
```

2. Calculate the similarities between different stocks using the log returns as the metric. What we are trying to do is calculate the Euclidean distances for the data points:

```
logreturns = np.diff(np.log(close))
print(logreturns.shape)

logreturns_norms = np.sum(logreturns ** 2, axis=1)
S = - logreturns_norms[:, np.newaxis] - logreturns_norms[np.
newaxis, :] + 2 * np.dot(logreturns, logreturns.T)
```

3. 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 symbol, label in zip(symbols, labels):
    print('%s in Cluster %d' % (symbol, label))
```

   The complete clustering program is as follows:

```
from __future__ import print_function
import datetime
import numpy as np
import sklearn.cluster
from matplotlib import finance
import urllib2

#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",
   "KO", "MCD", "MMM", "MRK", "MSFT", "PFE",
   "PG", "T", "TRV", "UTX", "VZ", "WMT", "XOM"]

quotes = []

for symbol in symbols:
    try:
        quotes.append(finance.quotes_historical_yahoo(symbol,
start, end, asobject=True))
    except urllib2.HTTPError as e:
        print(symbol, e)

close = np.array([q.close for q in quotes]).astype(np.float)
print(close.shape)
```

```
#2. Calculate affinity matrix
logreturns = np.diff(np.log(close))
print(logreturns.shape)

logreturns_norms = np.sum(logreturns ** 2, axis=1)
S = - logreturns_norms[:, np.newaxis] - logreturns_norms[np.
newaxis, :] + 2 * np.dot(logreturns, logreturns.T)

#3. Cluster using affinity propagation
aff_pro = sklearn.cluster.AffinityPropagation().fit(S)
labels = aff_pro.labels_

for symbol, label in zip(symbols, labels):
    print('%s in Cluster %d' % (symbol, label))
```

The output with the cluster numbers for each stock is as follows:

```
(29, 252)
(29, 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
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
`sklearn.cluster.AffinityPropagation()`	Creates an `AffinityPropagation` object.
`sklearn.cluster.AffinityPropagation.fit()`	Computes an affinity matrix from Euclidian distances and applies affinity propagation clustering.
`diff()`	Calculates differences of numbers within a NumPy array. If this is not specified, first-order differences are computed.
`log()`	Calculates the natural log of elements in a NumPy array.
`sum()`	Sums the elements of a NumPy array.
`dot()`	This performs matrix multiplication for two-dimensional arrays. It also calculates the inner product for one-dimensional arrays.

See also

- The relevant documentation is at `http://scikit-learn.org/stable/modules/generated/sklearn.cluster.AffinityPropagation.html`

Installing statsmodels

The `statsmodels` package focuses on statistical modeling. We can integrate it 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, run the following command:

```
$ python setup.py install
```

If you are using `setuptools`, the command is as follows:

```
$ easy_install statsmodels
```

Performing a normality test with statsmodels

The `statsmodels` package has many 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 0 and 1. The complete code for this tutorial is as follows:

```
from __future__ import print_function
import datetime
import numpy as np
from matplotlib import finance
from statsmodels.stats.adnorm import normal_ad

#1. Download price data

# 2011 to 2012
start = datetime.datetime(2011, 01, 01)
end = datetime.datetime(2012, 01, 01)
```

```
quotes = finance.quotes_historical_yahoo('AAPL', start, end,
asobject=True)

close = np.array(quotes.close).astype(np.float)
print(close.shape)

print(normal_ad(np.diff(np.log(close))))

#Retrieving data for AAPL
#(252,)
#(0.57103805516803163, 0.13725944999430437)
```

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 `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 0 and 1, this confirms our hypothesis.

Installing scikit-image

scikit-image is a toolkit used for image processing that requires PIL, SciPy, Cython, and NumPy. Windows installers are available too. The toolkit is part of the Enthought Python Distribution, as well as the Python(*x*, *y*) distribution.

How to do it...

As usual, install scikit-image using any one of the following two commands:

```
$ pip install -U scikit-image
$ easy_install -U scikit-image
```

Again, you may 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 source archive from Github. Then 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. scikit-image offers a **Harris corner detector**, which is great, since 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 scikit-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 commands:

```
$ ./configure
$  make
$  sudo make install
```

How to do it...

We will load a sample image from scikit-learn. This is not absolutely necessary for this example; you can use any other image instead:

1. scikit-learn currently has two sample JPEG images in a dataset structure. Look at the first image only:

   ```
   dataset = load_sample_images()
   img = dataset.images[0]
   ```

2. Since the first edition of this book, the API has changed. For instance, with scikit-image 0.11.2, we need to first convert values of a color images to grayscale values. Gray scale the image as follows:

   ```
   gray_img = rgb2gray(img)
   ```

3. Call the `corner_harris()` function to get the coordinates of the corners:

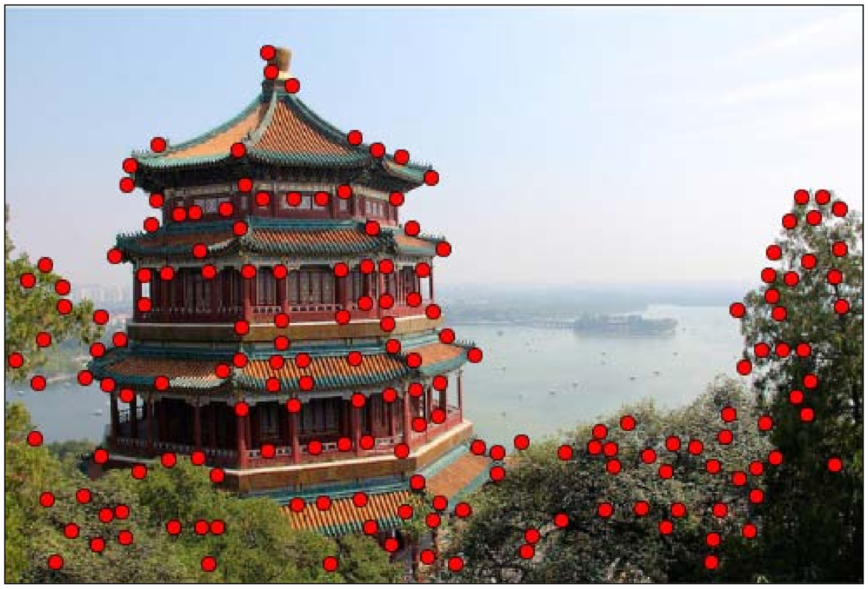
   ```
   harris_coords = corner_peaks(corner_harris(gray_img))
   y, x = np.transpose(harris_coords)
   ```

   The code for the corner detection is as follows:

   ```
   from sklearn.datasets import load_sample_images
   import matplotlib.pyplot as plt
   import numpy as np
   from skimage.feature import corner_harris
   ```

```
from skimage.feature import corner_peaks
from skimage.color import rgb2gray

dataset = load_sample_images()
img = dataset.images[0]
gray_img = rgb2gray(img)
harris_coords = corner_peaks(corner_harris(gray_img))
y, x = np.transpose(harris_coords)
plt.axis('off')
plt.imshow(img)
plt.plot(x, y, 'ro')
plt.show()
```

We get an image with dots, where the script detects corners, as shown in the following screenshot:



How it works...

We applied Harris corner detection on a sample image from scikit-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 messy. The scikit-image toolkit has a lot more similar functions, so check out the scikit-image documentation if you are in need of an image processing routine. Also keep in mind that the API can undergo rapid changes.

See also

- The related scikit-image documentation is at `http://scikit-image.org/docs/dev/auto_examples/plot_corner.html`

Detecting edges

Edge detection is another popular image processing technique (`http://en.wikipedia.org/wiki/Edge_detection`). scikit-image has a **Canny filter** implementation based on the standard deviation of the Gaussian distribution, which can perform edge detection out of the box. Besides the image data as a 2D array, this filter accepts the following parameters:

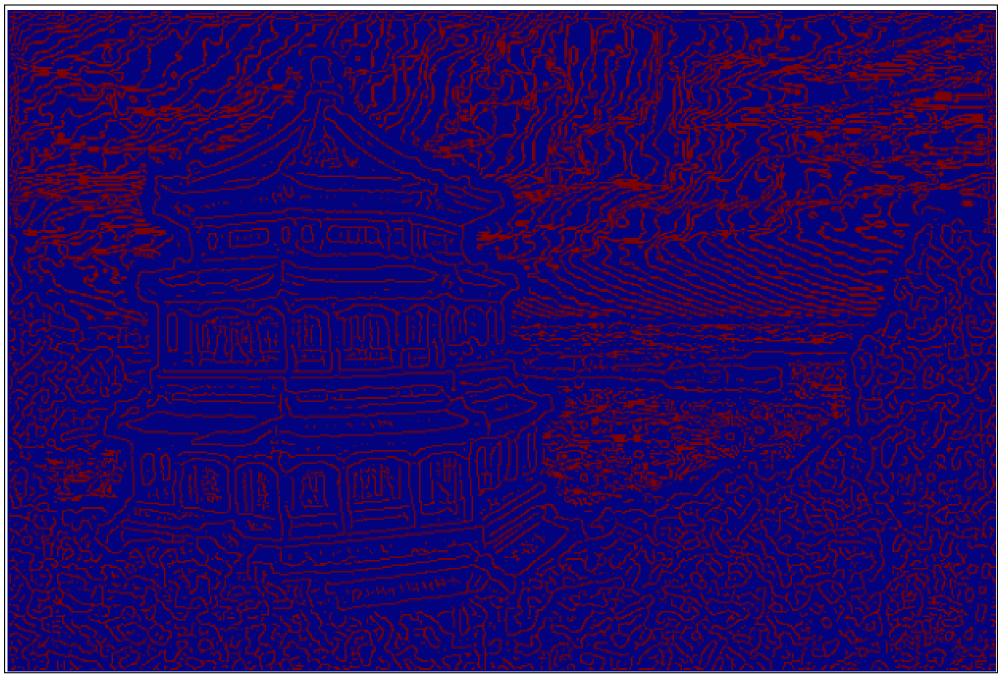
- Standard deviation of the Gaussian distribution
- Lower bound threshold
- Upper bound threshold

How to do it...

We will use the same image as in the previous recipe. The code is almost the same (see `edge_detection.py`). Pay extra attention to the line where we call the Canny filter function:

```
from sklearn.datasets import load_sample_images
import matplotlib.pyplot as plt
import skimage.feature

dataset = load_sample_images()
img = dataset.images[0]
edges = skimage.feature.canny(img[..., 0])
plt.axis('off')
plt.imshow(edges)
plt.show()
```

The code produces an image of the edges within the original image, as shown in the following screenshot:



See also

- The related documentation is at `http://scikit-image.org/docs/dev/auto_examples/plot_canny.html`

Installing pandas

pandas is a Python library used 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, R inspired the core `DataFrame` object in pandas.

How to do it...

On PyPi, the project is called `pandas`. So, you can run either of the following commands:

```
$ 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, do the following:

```
$ sudo apt-get install python-pandas
```

You can also install from source (this requires Git unless you download a source archive):

```
$ git clone git://github.com/pydata/pandas.git
$ cd pandas
$ python setup.py install
```

See also

- The related documentation is at `http://pandas.pydata.org/pandas-docs/stable/install.html`

Estimating correlation of stock returns 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 take 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 a plot will appear:

1. To create the data frame, create a dictionary containing stock symbols as keys and the corresponding log returns as values. The data frame itself has the date as the index and the stock symbols as column labels:

   ```
   data = {}

   for i, symbol in enumerate(symbols):
   ```

```
   data[symbol] = np.diff(np.log(close[i]))

# Convention: import pandas as pd
df = pd.DataFrame(data,
  index=dates[0][:-1], columns=symbols)
```

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, which also downloads the price data, is as follows:

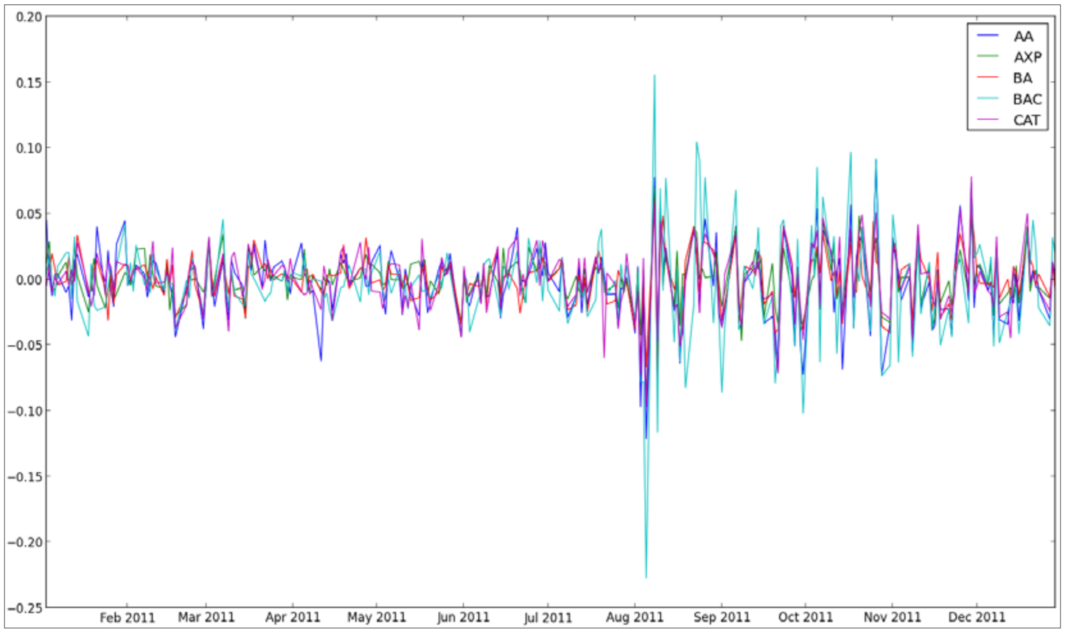
```
from __future__ import print_function
import pandas as pd
import matplotlib.pyplot as plt
from datetime import datetime
from matplotlib import finance
import numpy as np

# 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 = np.array([q.close for q in quotes]).astype(np.float)
dates = np.array([q.date for q in quotes])

data = {}

for i, symbol in enumerate(symbols):
   data[symbol] = np.diff(np.log(close[i]))

df = pd.DataFrame(data, index=dates[0][:-1], columns=symbols)


print(df.corr())
df.plot()
plt.legend(symbols)
plt.show()
```

```
#           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
```

Here is the 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
`pandas.DataFrame()`	This function constructs `DataFrame` with specified data, index (row), and column labels.
`pandas.DataFrame.corr()`	This function computes pair-wise correlation of columns, ignoring the missing values. By default, Pearson correlation is used.
`pandas.DataFrame.plot()`	This function plots the data frame with `matplotlib`.

See also

- The related documentation is at `http://pandas.pydata.org/pandas-docs/dev/generated/pandas.DataFrame.html`
- *Chapter 4*, *pandas Primer*, from Ivan Idris' book *Python Data Analysis*, *Packt Publishing*

Loading data as pandas objects from statsmodels

statsmodels has quite a lot of sample datasets in its distribution. 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**. patsy is a library that describes statistical models. It is easy enough to see whether this library is necessary; just run the code. If you get errors related to patsy, execute any one of the following commands:

```
$ sudo easy_install patsy
$ pip install --upgrade patsy
```

How to do it...

In this section, we will load a dataset from statsmodels as a pandas `DataFrame` or `Series` object.

1. 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. The `Dataset` object has an attribute called `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)
   ```

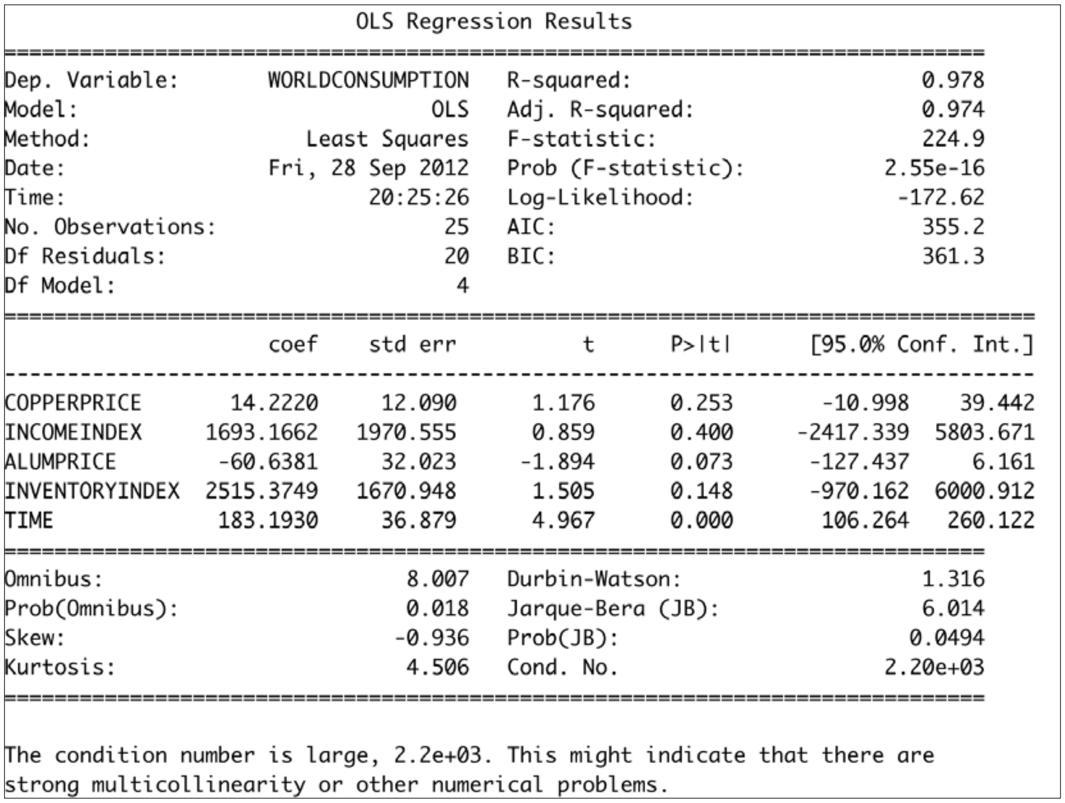
   This should print the result of the fitting procedure:

   ```
   Fit params COPPERPRICE         14.222028
   INCOMEINDEX       1693.166242
   ALUMPRICE          -60.638117
   INVENTORYINDEX    2515.374903
   TIME               183.193035
   ```

3. 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 required to load the copper dataset is as follows:

```
from __future__ import print_function
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 the `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.

See also

- The related documentation is at `http://statsmodels.sourceforge.net/stable/datasets/index.html`

Resampling time series data

In this tutorial, you 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 into monthly data by computing the mean. We will do this by creating a pandas `DataFrame` and calling its `resample()` method:

1. Before we can create a pandas `DataFrame`, we need to create a `DatetimeIndex` object to pass to the `DataFrame` constructor. Create the index from the downloaded quotes data, as follows:

   ```
   dt_idx = pandas.DatetimeIndex(quotes.date)
   ```

2. Once we have the date-time index, we use it together with the close prices to create a data frame:

   ```
   df = pandas.DataFrame (quotes.close, index=dt_idx,
   columns=[symbol])
   ```

3. Resample the time series to monthly frequency by computing the mean:

   ```
   resampled = df.resample('M', how=numpy.mean)
   print(resampled)
   ```

   The resampled time series, as shown in the following lines, has one value for each month:

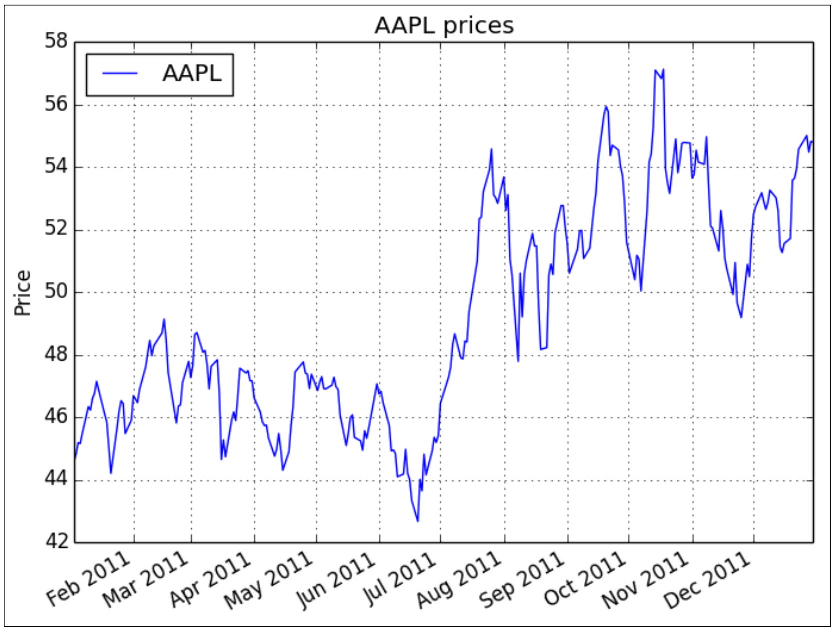
   ```
                     AAPL
   2011-01-31  336.932500
   2011-02-28  349.680526
   2011-03-31  346.005652
   ```

```
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
```

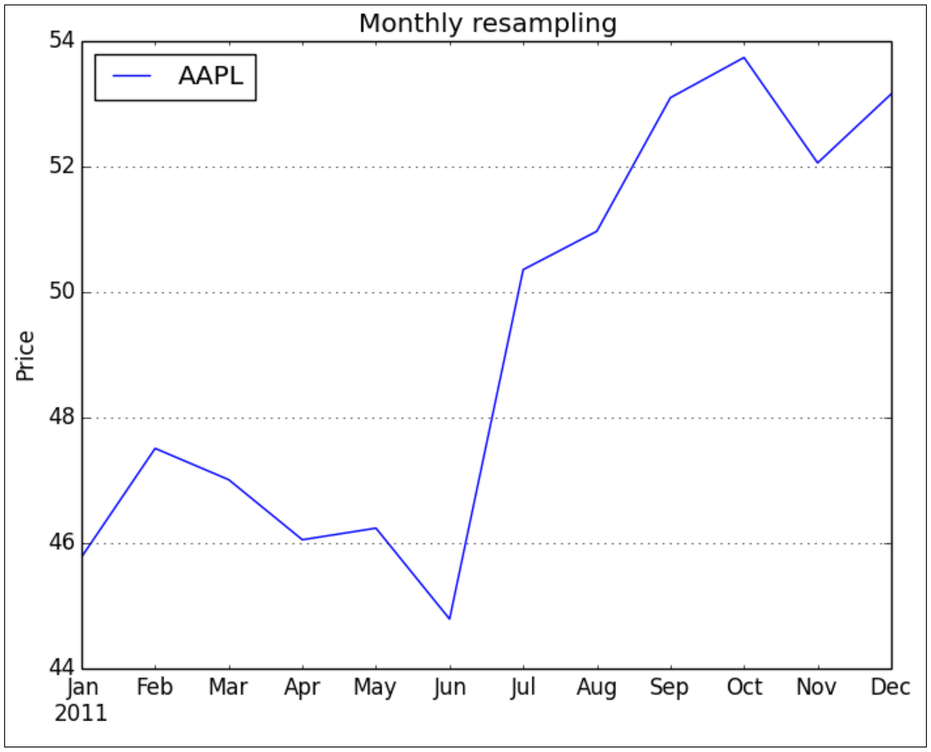
4. Use the `DataFrame plot()` method to plot the data:

```
df.plot()
resampled.plot()
plt.show()
```

The plot for the original time series is as follows:



The resampled data has less data points, and therefore, the resulting plot is choppier, as shown in the following screenshot:



The complete resampling code is as follows:

```
from __future__ import print_function
import pandas
import matplotlib.pyplot as plt
from datetime import datetime
from matplotlib import finance
import numpy as np

# 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=np.mean)
print(resampled)

# Plot
df.plot()
plt.title('AAPL prices')
plt.ylabel('Price')

resampled.plot()
plt.title('Monthly resampling')
plt.ylabel('Price')
plt.grid(True)
plt.show()
```

How it works...

We created a date-time index from a list of dates and times. This index was then used to create a pandas `DataFrame`. We then resampled our time series data. A single character gives the resampling frequency, as listed here:

- `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.

See also

- The related pandas documentation is at `http://pandas.pydata.org/pandas-docs/dev/generated/pandas.DataFrame.resample.html`

11
Latest and Greatest NumPy

In this chapter, we cover the following recipes:

- Fancy indexing in place of ufuncs with the `at()` method
- Partial sorting via selection of the fast median with the `partition()` function
- Skipping NaNs with the `nanmean()`, `nanvar()`, and `nanstd()` functions
- Creating value-initialized arrays with the `full()` and `full_like()` functions
- Random sampling with `numpy.random.choice()`
- Using the `datetime64` type and the related API

Introduction

Since the first edition of *NumPy Cookbook*, the NumPy team has introduced new features; I will describe them in this chapter. It's probably unlikely that you read the first edition of this book and are now reading the second edition. I wrote the first edition in 2012 and used the then available features. NumPy has many features, so you can't expect coverage of all of them, but the functionality I've described in this chapter is relatively important.

Fancy indexing in place for ufuncs with the at() method

The `at()` method was added to the NumPy universal function class in NumPy 1.8. This method allows fancy indexing in-place. Fancy indexing is indexing that does not involve integers or slices, which is normal indexing. "In-place" means that the data of the input array will be altered.

The signature for the `at()` method is `ufunc.at(a, indices[, b])`. The indices array corresponds to the elements to operate on. We must specify the `b` array only for universal functions with two operands.

How to do it...

The following steps demonstrate how the `at()` method works:

1. Create an array with `7` random integers from `-4` to `4` with a seed of `44`:

   ```
   np.random.seed(44)
   a = np.random.random_integers(-4, 4, 7)
   print(a)
   ```

   The array appears as follows:

   ```
   [ 0 -1 -3 -1 -4  0 -1]
   ```

2. Apply the `at()` method of the `sign` universal function to the third and fifth array elements:

   ```
   np.sign.at(a, [2, 4])
   print(a)
   ```

   We get the following altered array:

   ```
   [ 0 -1 -1 -1 -1  0 -1]
   ```

See also

- The NumPy universal function documentation is at `http://docs.scipy.org/doc/numpy/reference/ufuncs.html`

Partial sorting via selection for fast median with the partition() function

The `partition()` subroutine does partial sorting. This should be less work than normal sorting.

> Refer to http://en.wikipedia.org/wiki/Partial_sorting for more information. A useful scenario is selecting the top five (or some other number) items of a group. Partial sorting doesn't preserve the right order within the set of the top elements.

The first parameter of the subroutine is the input array to sort. The second parameter is an integer or a list of integers corresponding to the indices of the array elements. The `partition()` subroutine sorts items at those indices correctly. One specified index gives two partitions. Multiple indices result in more than two partitions. The algorithm guarantees that items in partitions smaller than a correctly sorted item come before that item. Otherwise, they are put behind that item.

How to do it...

Let's illustrate this explanation with an example:

1. Create an array with random numbers to sort:

   ```
   np.random.seed(20)
   a = np.random.random_integers(0, 7, 9)
   print(a)
   ```

   The array has the following elements:

   ```
   [3 2 7 7 4 2 1 4 3]
   ```

2. Partially sort the array by partitioning it into two roughly equal parts:

   ```
   print(np.partition(a, 4))
   ```

   We get the following result:

   ```
   [2 3 1 2 3 7 7 4 4]
   ```

How it works...

We partially sorted a nine-element array. The function guaranteed only that one element in the middle, at index `4`, is at the right place. This corresponds to attempting to select the top five items of the array without caring about the order within the top five set. Since the correctly sorted item is in the middle, this also returns the median of the array.

See also

- The relevant NumPy documentation is at `http://docs.scipy.org/doc/numpy/reference/generated/numpy.partition.html`

Skipping NaNs with the nanmean(), nanvar(), and nanstd() functions

It is common to attempt to estimate how variable the arithmetic mean, variance, and standard deviation of a set of data are.

A simple, but effective, method is called **jackknife resampling** (refer to `http://en.wikipedia.org/wiki/Jackknife_resampling`). The idea behind jackknife resampling is to create datasets from the original data by leaving out one value each time. In essence, we are attempting to estimate what will occur if at least one of the values is incorrect. For every new dataset, we recalculate the statistical estimator we are interested in. This helps us understand how the estimator varies.

How to do it...

We will apply jackknife resampling to random data. We will skip every array element once by setting it to NaN (Not a Number). The `nanmean()`, `nanvar()`, and `nanstd()` can then be used to compute the arithmetic mean, variance, and standard deviation:

1. First initialize a 30 x 3 array for the estimates, as follows:

   ```
   estimates = np.zeros((len(a), 3))
   ```

2. Loop through the array and create a new dataset by setting one value to NaN at every iteration of the loop. For every new dataset, calculate the estimates:

   ```
   for i in xrange(len(a)):
      b = a.copy()
      b[i] = np.nan
      estimates[i,] = [np.nanmean(b), np.nanvar(b),
        np.nanstd(b)]
   ```

3. Print the variance for every estimator:

   ```
   print("Estimator variance", estimates.var(axis=0))
   ```

   The following output appears on the screen:

   ```
   Estimator variance [ 0.00079905  0.00090129  0.00034604]
   ```

How it works...

We estimated the variances of the arithmetic mean, variance, and standard deviation of a data set with jackknife resampling. This indicates how much the arithmetic mean, variance and standard deviation vary. The code for this recipe is in the `jackknife.py` file in this book's code bundle:

```
from __future__ import print_function
import numpy as np

np.random.seed(46)
a = np.random.randn(30)
estimates = np.zeros((len(a), 3))

for i in xrange(len(a)):
   b = a.copy()
   b[i] = np.nan

   estimates[i,] = [np.nanmean(b), np.nanvar(b), np.nanstd(b)]

print("Estimator variance", estimates.var(axis=0))
```

See also

- The documentation page for `nanmean()` is at `http://docs.scipy.org/doc/numpy-dev/reference/generated/numpy.nanmean.html`
- The documentation page for `nanvar()` is at `http://docs.scipy.org/doc/numpy-dev/reference/generated/numpy.nanvar.html`
- The documentation page for `nanstd()` is at `http://docs.scipy.org/doc/numpy-dev/reference/generated/numpy.nanstd.html`

Creating value initialized arrays with the full() and full_like() functions

The `full()` and `full_like()` functions are new additions to NumPy meant to facilitate initialization. Here's what the documentation says about them:

```
>>> help(np.full)
Return a new array of given shape and type, filled with `fill_value`.
>>> help(np.full_like)
Return a full array with the same shape and type as a given array.
```

How to do it...

Let's see how `full()` and `full_like()` function:

1. Create a `1` by `2` array with `full()`, filled with the lucky number `7`:

   ```
   print(np.full((1, 2), 7))
   ```

   Accordingly, we get the following array:

   ```
   array([[ 7.,  7.]])
   ```

   The array elements are floating-point numbers.

2. Specify an integer data type, as follows:

   ```
   print(np.full((1, 2), 7, dtype=np.int))
   ```

   The output changes accordingly:

   ```
   array([[7, 7]])
   ```

3. The `full_like()` function checks the metadata of an array and reuses it for the new array. For example, create an array using `linspace()`, and apply it as a template for the `full_like()` function:

   ```
   a = np.linspace(0, 1, 5)
   print(a)
   array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])
   print(np.full_like(a, 7))
   array([ 7.,  7.,  7.,  7.,  7.])
   ```

4. Again, we filled the array with the lucky number `7`. To modify the data type to integer, use the following line:

   ```
   print(np.full_like(a, 7, dtype=np.int))
   array([7, 7, 7, 7, 7])
   ```

How it works...

We produced arrays with `full()` and `full_like()`. The `full()` function filled the array with the number `7`. The `full_like()` function reused the metadata of an array for the creation of a new array. Both functions let you specify the data type of the array.

Random sampling with numpy.random.choice()

Bootstrapping is a procedure similar to jackknifing. The basic bootstrapping method has the following steps:

1. Generate samples from the original data of size *N*. Visualize the original data sample as a bowl of numbers. We create new samples by taking numbers at random from the bowl. After taking a number, we return it to the bowl.
2. For each generated sample, we compute the statistical estimator of interest (for example, the arithmetic mean).

How to do it...

We will apply `numpy.random.choice()` to do bootstrapping:

1. Generate a data sample following the binomial distribution that simulates flipping a fair coin five times:

```
N = 400
np.random.seed(28)
data = np.random.binomial(5, .5, size=N)
```

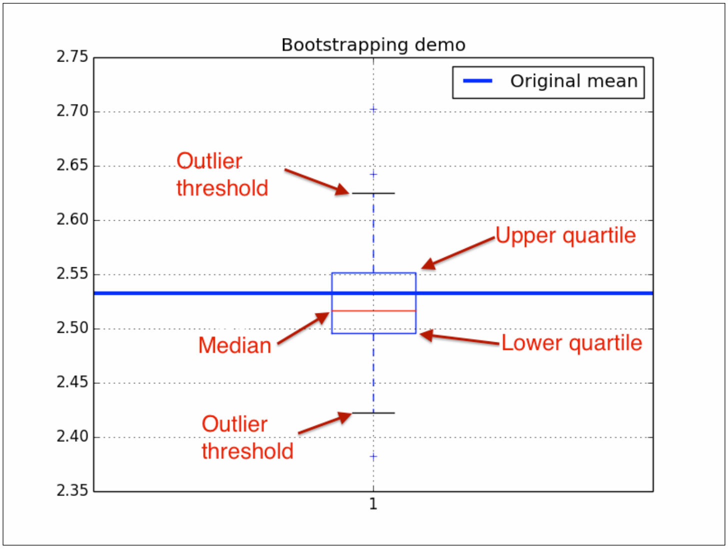
2. Generate 30 samples and compute their means (more samples will give a better result):

```
bootstrapped = np.random.choice(data, size=(N, 30))
means = bootstrapped.mean(axis=0)
```

3. Visualize the arithmetic means distribution with a `matplotlib` box plot:

```
plt.title('Bootstrapping demo')
plt.grid()
plt.boxplot(means)
plt.plot(3 * [data.mean()], lw=3, label='Original mean')
plt.legend(loc='best')
plt.show()
```

Refer to the following annotated plot for the end result:



How it works...

We simulated an experiment involving flipping a fair coin five times. We bootstrapped the data by creating samples and computing the corresponding means. Then we used `numpy.random.choice()` to bootstrap. We visualized the means with a `matplotlib` box plot. If you are not familiar with box plots, the annotations in the plot will hopefully help you. The following elements of the box plot are of importance:

- The median represented by a line in a box.
- Upper and lower quartiles shown as edges of the box.
- Whiskers indicating boundaries for outliers. By default, these are set at *1.5 * (Q3 – Q1)* from the edges of the box, which is also known as the **interquartile range**.

See also

- The NumPy `numpy.random.choice()` documentation is at `http://docs.scipy.org/doc/numpy-dev/reference/generated/numpy.random.choice.html`
- The `matplotlib boxplot()` function documentation is at `http://matplotlib.org/api/pyplot_api.html`
- The Wikipedia page about box plots is at `http://en.wikipedia.org/wiki/Box_plot`

Using the datetime64 type and related API

The `datetime64` type represents a date and the corresponding time. You need NumPy 1.7.0 or later versions to use this data type.

How to do it...

To get acquainted with `datetime64`, follow these steps:

1. Create a `datetime64` from a string, as follows:

   ```
   print(np.datetime64('2015-05-21'))
   ```

   The preceding line prints the following output:

   ```
   numpy.datetime64('2015-05-21')
   ```

   We created a `datetime64` type for May 21, 2015, using the `YYYY-MM-DD` format, where `Y` corresponds to the year, `M` corresponds to the month, and `D` corresponds to the day of the month. NumPy complies with the ISO 8601 standard—an international standard for representing dates and times.

2. ISO 8601 also defines the `YYYY-MM-DD`, `YYYY-MM`, and `YYYYMMDD` formats. Check these out for yourself, as follows:

   ```
   print(np.datetime64('20150521'))
   print(np.datetime64('2015-05'))
   ```

   The code prints the following lines:

   ```
   numpy.datetime64('20150521')
   numpy.datetime64('2015-05')
   ```

3. By default, ISO 8601 uses the local time zone. The time can be specified using the `T[hh:mm:ss]` format. For example, we can define January 1, 1578, and the time 9:18 p.m. as follows:

   ```
   local = np.datetime64('1578-01-01T21:18')
   print(local)
   ```

   The following line shows the result:

   ```
   numpy.datetime64('1578-01-01T21:18Z')
   ```

4. A string in the `-[hh:mm]` format defines an offset relative to the UTC time zone. We can create a `datetime64` type with 8 hours of offset, as follows:

   ```
   with_offset = np.datetime64('1578-01-01T21:18-0800')
   print(with_offset)
   ```

   We then see the following line on the screen:

   ```
   numpy.datetime64('1578-01-02T05:18Z')
   ```

   The `Z` at the end stands for Zulu time, which is how UTC is sometimes referred to.

5. Subtract the two `datetime64` objects from each other:

   ```
   print(local - with_offset)
   ```

   The result appears as follows:

   ```
   numpy.timedelta64(-480,'m')
   ```

   Subtracting creates a `timedelta64` NumPy object, which in this case indicates a 480-minute delta.

How it works...

You learned about the `datetime64` NumPy type. This data type allows us to manipulate dates and times with ease. Its features include simple arithmetic and creation of arrays using the normal NumPy capabilities. Please refer to the `datetime_demo.py` file in this book's code bundle:

```
import numpy as np

print(np.datetime64('2015-05-21'))
#numpy.datetime64('2015-05-21')

print(np.datetime64('20150521'))
print(np.datetime64('2015-05'))
#numpy.datetime64('20150521')
#numpy.datetime64('2015-05')

local = np.datetime64('1578-01-01T21:18')
print(local)
#numpy.datetime64('1578-01-01T21:18Z')

with_offset = np.datetime64('1578-01-01T21:18-0800')
print(with_offset)
#numpy.datetime64('1578-01-02T05:18Z')

print(local - with_offset)
```

See also

- The relevant NumPy documentation is at `http://docs.scipy.org/doc/numpy/reference/arrays.datetime.html`
- The relevant Wikipedia page is at `http://en.wikipedia.org/wiki/ISO_8601`

12
Exploratory and Predictive Data Analysis with NumPy

In this chapter, we cover the following recipes:

- Exploring atmospheric pressure
- Exploring the day-to-day pressure range
- Studying annual atmospheric pressure averages
- Analyzing maximum visibility
- Predicting pressure with an autoregressive model
- Predicting pressure with a moving average model
- Studying intrayear average pressure
- Studying extreme values of atmospheric pressure

Introduction

Data analysis is one of the most important use cases of NumPy. Depending on our goals, we can distinguish between many phases and types of data analysis. In this chapter, we will talk about **exploratory** and **predictive data analysis**. Exploratory data analysis probes the data for clues. At this stage, we are probably unfamiliar with the dataset. Predictive analysis tries to predict something about the data using a model.

The data comes from the Dutch meteorological institute KNMI. It is specifically about the weather station at De Bilt, where the KNMI headquarters is located. In these recipes, we will inspect atmospheric pressure and maximum visibility (see `http://www.knmi.nl/climatology/daily_data/download.html`).

I modified and converted the textual data from the KNMI to the NumPy-specific `.npy` format, saved as a 40996 x 5 array. The array contains daily values for five variables:

- The date in the `YYYYMMDD` format
- The average daily atmospheric pressure
- The highest daily atmospheric pressure
- The lowest daily atmospheric pressure
- The maximum daily visibility

Exploring atmospheric pressure

In this recipe, we will take a look at the daily mean sea level pressure (in 0.1 hPa) calculated from 24 hourly values. This includes printing descriptive statistics and visualizing the probability distribution. In nature, we often deal with the normal distribution, so the normality test from *Chapter 10*, *Fun with Scikits*, will come in handy.

The complete code is in the `exploring.py` file in this book's code bundle:

```
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.stats.adnorm import normal_ad

data = np.load('cbk12.npy')

# Multiply to get hPa values
meanp = .1 * data[:,1]

# Filter out 0 values
meanp = meanp[ meanp > 0]

# Get descriptive statistics
print("Max", meanp.max())
print("Min", meanp.min())
mean = meanp.mean()
print("Mean", mean)
print("Median", np.median(meanp))
std = meanp.std()
```

```
print("Std dev", std)

# Check for normality
print("Normality", normal_ad(meanp))

#histogram with Gaussian PDF
plt.subplot(211)
plt.title('Histogram of average atmospheric pressure')
_, bins, _ = plt.hist(meanp, np.sqrt(len(meanp)), normed=True)
plt.plot(bins, 1/(std * np.sqrt(2 * np.pi)) * np.exp(- (bins -
mean)**2/(2 * std**2)), 'r-', label="Gaussian PDF")
plt.grid()
plt.legend(loc='best')
plt.xlabel('Average atmospheric pressure (hPa)')
plt.ylabel('Frequency')

# boxplot
plt.subplot(212)
plt.boxplot(meanp)
plt.title('Boxplot of average atmospheric pressure')
plt.ylabel('Average atmospheric pressure (hPa)')
plt.grid()

# Improves spacing of subplots
plt.tight_layout()
plt.show()
```

Getting ready

Install `statsmodels`, if you haven't installed already, for the normality test (see the *Installing scikits-statsmodels* recipe *Chapter 10*, *Fun with Scikits*).

How to do it...

Follow these steps to explore the daily atmospheric pressure:

1. Load the data with the `load()` function:

   ```
   data = np.load('cbk12.npy')
   ```

2. Normally data needs to be processed and cleaned up. In this case, multiply the values to get values in `hPa` and remove `0` values corresponding to the missing values:

   ```
   # Multiply to get hPa values
   meanp = .1 * data[:,1]

   # Filter out 0 values
   meanp = meanp[ meanp > 0]
   ```

3. Get descriptive statistics, including maximum, minimum, arithmetic mean, median, and standard deviation:

```
print("Max", meanp.max())
print("Min", meanp.min())
mean = meanp.mean()
print("Mean", mean)
print("Median", np.median(meanp))
std = meanp.std()
print("Std dev", std)
```

You should see the following values:

```
Max 1048.3
Min 962.1
Mean 1015.14058231
Median 1015.8
Std dev 9.85889134337
```

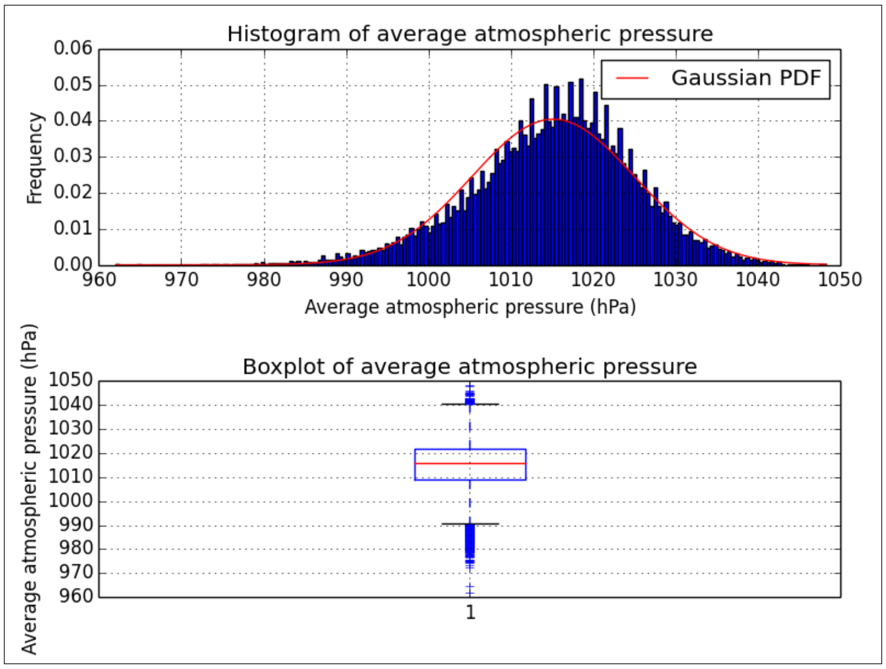
4. Apply the normality test from *Chapter 10*, *Fun with Scikits*, as follows:

```
print("Normality", normal_ad(meanp))
```

The following values appear on the screen:

```
Normality (72.685781095773564, 0.0)
```

It is also nice to visualize the distribution of values with a histogram and a box plot. Refer to the following plot for the end result:



See also

- The *Performing a normality test with statsmodels* recipe from *Chapter 10, Fun with Scikits*
- For an explanation of box plots, see the *Random sampling with numpy.random.choice()* recipe from *Chapter 11*, *Latest and Greatest NumPy*
- The documentation of the `load()` function is at `http://docs.scipy.org/doc/numpy/reference/generated/numpy.load.html`

Exploring the day-to-day pressure range

The daily pressure range is the difference of the daily highs and lows. With real-world data, we sometimes have missing values. Here, we can potentially lack values for the high and/or low pressures of a given day. It's possible to fill those gaps with a smart algorithm. However, let's keep it simple and just ignore them. After calculating the ranges, we will do a similar analysis as in the previous recipe, but we will use functions that can deal with `NaN` values. Also, we will look at the relation between months and ranges.

The corresponding code is in the `day_range.py` file in this book's code bundle:

```
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt
import calendar as cal

data = np.load('cbk12.npy')

# Multiply to get hPa values
highs = .1 * data[:,2]
lows = .1 * data[:,3]

# Filter out 0 values
highs[highs == 0] = np.nan
lows[lows == 0] = np.nan

# Calculate range and stats
ranges = highs - lows
print("Minimum daily range", np.nanmin(ranges))
print("Maximum daily range", np.nanmax(ranges))

print("Average daily range", np.nanmean(ranges))
print("Standard deviation", np.nanstd(ranges))
```

```
# Get months
dates = data[:,0]
months = (dates % 10000)/100
months = months[~np.isnan(ranges)]

monthly = []
month_range = np.arange(1, 13)

for month in month_range:
   indices = np.where(month == months)
   monthly.append(np.nanmean(ranges[indices]))

plt.bar(month_range, monthly)
plt.title('Monthly average of daily pressure ranges')
plt.xticks(month_range, cal.month_abbr[1:13])
plt.ylabel('Monthly Average (hPa)')
plt.grid()
plt.show()
```

How to do it...

The first steps of this recipe are almost the same as those of the previous recipe, so we will skip them. Follow along for the analysis of the daily pressure range:

1. We could leave missing values at their current `0` value. However, it is usually safer to set them to `NaN` to avoid confusion. Set the missing values to `NaN`, as follows:

   ```
   highs[highs == 0] = np.nan
   lows[lows == 0] = np.nan
   ```

2. Compute the ranges, minima, maxima, mean, and standard deviations with the `nanmin()`, `nanmax()`, `nanmean()`, and `nanstd()` functions:

   ```
   ranges = highs - lows
   print("Minimum daily range", np.nanmin(ranges))
   print("Maximum daily range", np.nanmax(ranges))

   print("Average daily range", np.nanmean(ranges))
   print("Standard deviation", np.nanstd(ranges))
   ```

   The result appears on the screen:

   ```
   Minimum daily range 0.4
   Maximum daily range 41.7
   Average daily range 6.11945360571
   Standard deviation 4.42162136692
   ```

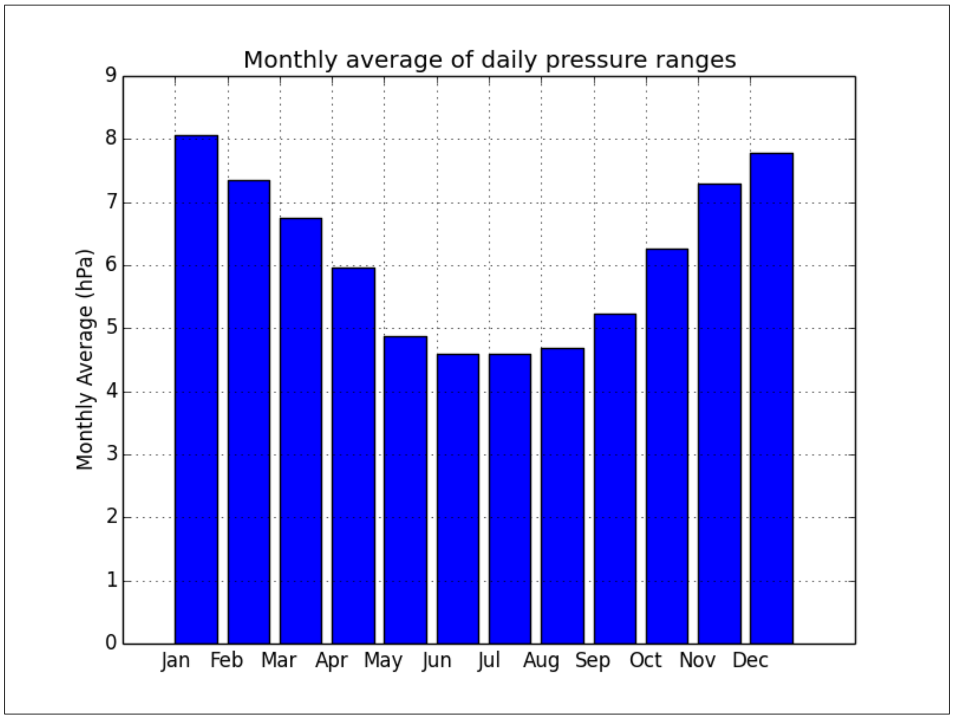
3. As I mentioned previously, the dates are given in the `YYYYMMDD` format. With a bit of arithmetic, we can easily get the months. Also, we ignore the month values corresponding to the `NaN` range values:

   ```
   dates = data[:,0]
   months = (dates % 10000)/100
   months = months[~np.isnan(ranges)]
   ```

4. Average the ranges by month, as follows:

   ```
   monthly = []
   month_range = np.arange(1, 13)

   for month in month_range:
      indices = np.where(month == months)
      monthly.append(np.nanmean(ranges[indices]))
   ```

   In the last step, we draw a `matplotlib` bar chart of monthly average values of daily pressure ranges. Refer to the following plot for the end result:



How it works...

We analyzed the daily ranges of atmospheric pressure. Further, we visualized the monthly averages of the daily range. There seems to be a pattern leading to smaller daily atmospheric pressure ranges in summer. Of course, more work is necessary to make certain.

See also

- The *Exploring atmospheric pressure* recipe

Studying annual atmospheric pressure averages

You may have heard of global warming, which claims that temperature is rising steadily each year. Since pressure is another thermodynamic variable, we may expect pressure also to follow a trend. The complete code for this recipe is in the `annual.py` file in this book's code bundle:

```
import numpy as np
import matplotlib.pyplot as plt


data = np.load('cbk12.npy')

# Multiply to get hPa values
avgs = .1 * data[:,1]
highs = .1 * data[:,2]
lows = .1 * data[:,3]

# Filter out 0 values
avgs = np.ma.array(avgs, mask = avgs == 0)
lows = np.ma.array(lows, mask = lows == 0)
highs = np.ma.array(highs, mask = highs == 0)

# Get years
years = data[:,0]/10000

# Initialize annual stats arrays
y_range = np.arange(1901, 2014)
nyears = len(y_range)
y_avgs = np.zeros(nyears)
```

```
y_highs = np.zeros(nyears)
y_lows = np.zeros(nyears)

# Compute stats
for year in y_range:
   indices = np.where(year == years)
   y_avgs[year - 1901] = np.mean(avgs[indices])
   y_highs[year - 1901]  = np.max(highs[indices])
   y_lows[year - 1901] = np.min(lows[indices])

plt.title('Annual atmospheric pressure for De Bilt(NL)')
plt.ticklabel_format(useOffset=900, axis='y')

plt.plot(y_range, y_avgs, label='Averages')

# Plot ignoring NaNs
h_mask = np.isfinite(y_highs)
plt.plot(y_range[h_mask], y_highs[h_mask], '^', label='Highs')

l_mask = np.isfinite(y_lows)
plt.plot(y_range[l_mask], y_lows[l_mask], 'v', label='Lows')

plt.xlabel('Year')
plt.ylabel('Atmospheric pressure (hPa)')
plt.grid()
plt.legend(loc='best')
plt.show()
```

How to do it...

To check for a trend, let's plot the average, maximum, and minimum annual atmospheric pressures with the following steps:

1. Initialize the annual statistics arrays:

   ```
   y_range = np.arange(1901, 2014)
   nyears = len(y_range)
   y_avgs = np.zeros(nyears)
   y_highs = np.zeros(nyears)
   y_lows = np.zeros(nyears)
   ```

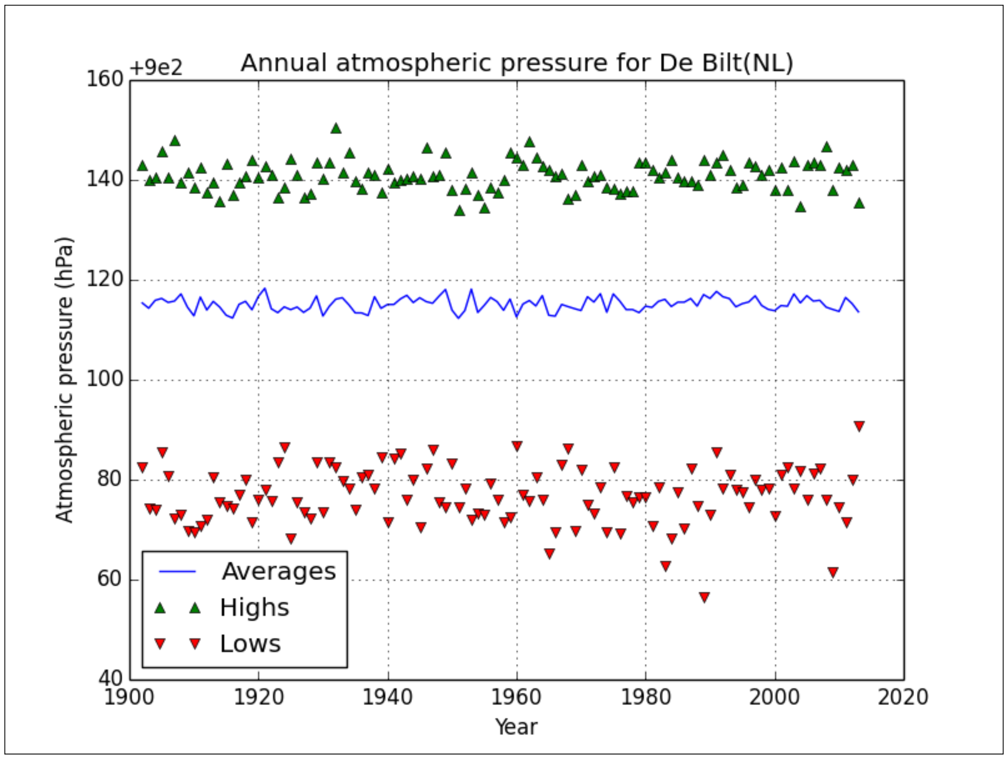
2. Compute the annual statistics:

```
for year in y_range:
   indices = np.where(year == years)
   y_avgs[year - 1901] = np.mean(avgs[indices])
   y_highs[year - 1901] = np.max(highs[indices])
   y_lows[year - 1901] = np.min(lows[indices])
```

3. Plot, ignoring the `NaN` values, as follows:

```
h_mask = np.isfinite(y_highs)
plt.plot(y_range[h_mask], y_highs[h_mask], '^', label='Highs')

l_mask = np.isfinite(y_lows)
plt.plot(y_range[l_mask], y_lows[l_mask], 'v', label='Lows')
```

Refer to the following plot for the end result:



How it works...

The average annual pressure seems to be flat or fluctuating a bit, but without any trend. We used the `isfinite()` function to ignore the `NaN` values in the final plot. This function checks for infinite and `NaN` values.

See also

- The *Exploring atmospheric pressure* recipe
- The `isfinite()` function documentation is at `http://docs.scipy.org/doc/numpy/reference/generated/numpy.isfinite.html`

Analyzing maximum visibility

If you've gone through all the recipes in this chapter so far, you might need a break from atmospheric pressure. So let's look into visibility instead. The data file has a column for a maximum visibility, which the KNMI describes as follows:

> *"Maximum visibility; 0: <100 m, 1:100-200 m, 2:200-300 m,..., 49:4900-5000 m, 50:5-6 km, 56:6-7 km, 57:7-8 km,..., 79:29-30 km, 80:30-35 km, 81:35-40 km,..., 89: >70 km)"*

Visibility here is a discrete variable, so averaging values may not make sense. Also, it seems that we have a lot of `0` values for the period between 1901 and 1950 for almost every day. I don't believe that De Bilt was extra foggy in that period. For the purpose of this recipe, we define mist as visibility between 1 and 2 km, which corresponds to the values of `10` and `20` in the data file. Let's also define haze as visibility between 2 and 5 km. This in turn corresponds to `20` and `50` in our data file.

Air pollution could reduce visibility, especially on clear days. We can define clear days as those with visibility higher than 30 km, or the value of `79` in our data file. Ideally, we should use air pollution data, but unfortunately, we don't have that. As far as I know, the air pollution levels around this particular weather station are not very high. It is interesting to know the number of clear days per year. The code for the analysis is in the `visibility.py` file in this book's code bundle:

```
import numpy as np
import matplotlib.pyplot as plt

data = np.load('cbk12.npy')

# Get minimum visibility
visibility = data[:,4]
```

```
# doy
doy = data[:,0] % 10000

doy_range = np.unique(doy)

# Initialize arrays
ndoy = len(doy_range)
mist = np.zeros(ndoy)
haze = np.zeros(ndoy)


# Compute frequencies
for i, d in enumerate(doy_range):
   indices = np.where(d == doy)
   selection = visibility[indices]

   mist_truth = (10 < selection) & (selection < 20)
   mist[i] = len(selection[mist_truth])/(1. * len(selection))

   haze_truth = (20 < selection) & (selection < 50)
   haze[i] = len(selection[haze_truth])/(1. * len(selection))

# Get years
years = data[:,0]/10000

# Initialize annual stats arrays
y_range = np.arange(1901, 2014)
nyears = len(y_range)
y_counts = np.zeros(nyears)

# Get annual counts
for year in y_range:
   indices = np.where(year == years)
   selection = visibility[indices]
   y_counts[year - 1901] = len(selection[selection > 79])

plt.subplot(211)
plt.plot(np.arange(1, 367), mist, color='.25', label='mist')
plt.plot(np.arange(1, 367), haze, color='0.75', lw=2, label='haze')
plt.title('Probability of mist and haze')
plt.xlabel('Day of the year')
plt.ylabel('Probability')
plt.grid()
plt.legend(loc='best')

plt.subplot(212)
plt.plot(y_range, y_counts)
plt.xlabel('Year')
plt.ylabel('Number of clear days')
plt.title('Annual counts of clear days')
plt.grid()
plt.tight_layout()
plt.show()
```

How to do it...

Follow these steps to plot annual counts of clear days, the day of year (1-366) against the probability of haze and mist:

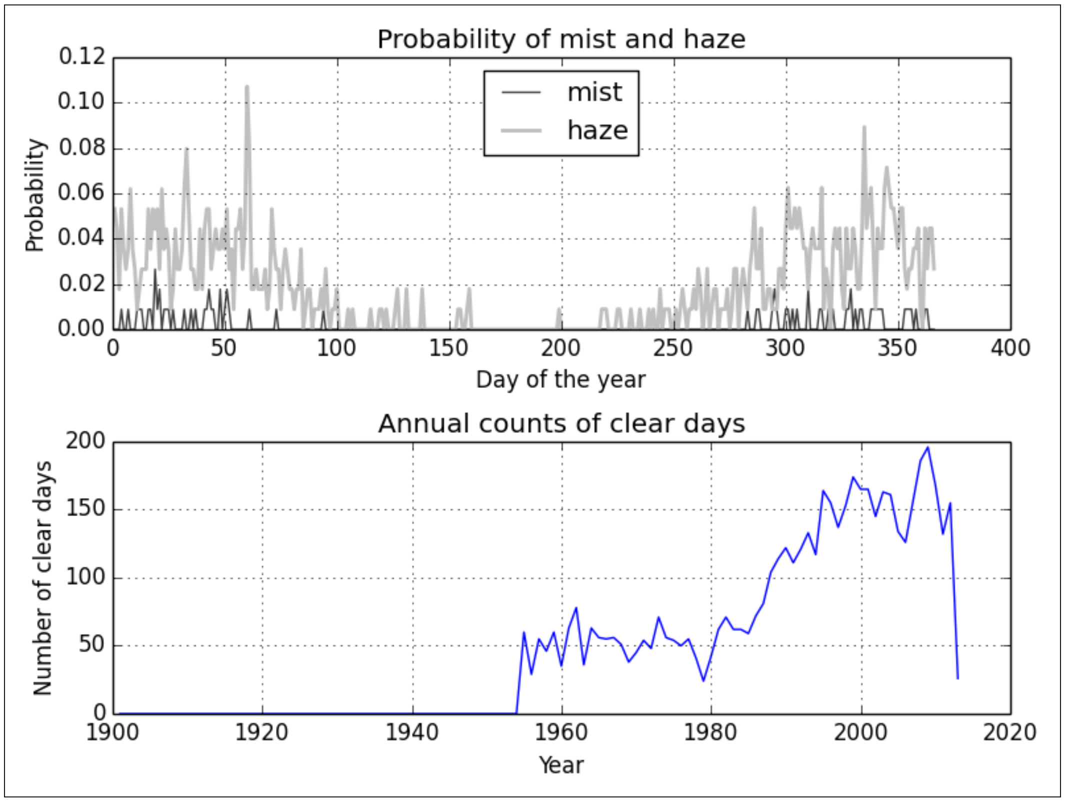
1. Compute the probability of haze and mist with following code block:

   ```
   for i, d in enumerate(doy_range):
      indices = np.where(d == doy)
      selection = visibility[indices]

      mist_truth = (10 < selection) & (selection < 20)
      mist[i] = len(selection[mist_truth])/(1. * len(selection))

      haze_truth = (20 < selection) & (selection < 50)
      haze[i] = len(selection[haze_truth])/(1. * len(selection))
   ```

2. Get the annual counts using this snippet:

   ```
   for year in y_range:
      indices = np.where(year == years)
      selection = visibility[indices]
      y_counts[year - 1901] = len(selection[selection > 79])
   ```

Refer to the following plot for the end result:



How it works...

As you can see, we start getting clear days after 1950. This is not due to extra foggy weather before 1950, but because of the phenomenon of missing or invalid data. The drop in the last year is also due to incomplete data. After 1980, we see a definite rise of clear days. This is supposed to be the period when global warming and climate change increased too. Unfortunately, we don't have data directly linked to air pollution, but our exploratory analysis indicates the existence of a trend.

Mist seems to occur mostly in the first and last two months of the year. You can draw similar conclusions about haze. Obviously, haze is more probable than mist, which is probably a good thing. You could also plot a histogram to make sure. However, keep in mind that you need to ignore `0` values as I mentioned earlier.

See also

- The *Exploring atmospheric pressure* recipe
- The *Studying annual atmospheric pressure averages* recipe
- The relevant Wikipedia page at `http://en.wikipedia.org/wiki/Visibility`

Predicting pressure with an autoregressive model

A very simple predictive model takes the current value of a variable and extrapolates it to the next period. To extrapolate, we can use a simple mathematical function. Since a variety of functions can be approximated by polynomials as in the Taylor series, polynomials of low degree might do the trick. What this boils down to is regression of the previous values to the next values. The corresponding models are therefore called **autoregressive**.

We have to be careful about **overfitting**. **Cross-validation** is a common approach to split the data into **train** and **test** sets. We fit the data using the train set and test the fit with the test set. This should reduce bias (see the `autoregressive.py` file in this book's code bundle):

```
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt


data = np.load('cbk12.npy')

# Load average pressure
meanp = .1 * data[:,1]

# Split point for test and train data
cutoff = 0.9 * len(meanp)

for degree, marker in zip(xrange(1, 4), ['o', 'x','.']):
   poly = np.polyfit(meanp[:cutoff - 1], meanp[1:cutoff], degree)
   print('Polynomial coefficients', poly)

   fit = np.polyval(poly, meanp[cutoff:-1])
   error = np.abs(meanp[cutoff + 1:] - fit)/fit
   plt.plot(error, marker, color=str(.25* degree), label='Degree ' +
str(degree))
   plt.plot(np.full(len(error), error.mean()), lw=degree, label='Mean
for degree ' + str(degree))
```

```
    print("Absolute mean relative error", error.mean(), 'for polynomial
of degree', degree)
    print()

plt.title('Relative test errors for polynomial fits')
plt.ylabel('Relative error')
plt.grid()
plt.legend(loc='best')
plt.show()
```

How to do it...

With the following steps, we will fit atmospheric pressure using polynomials of varying degrees:

1. Define a cutoff for the test and train sets:

   ```
   cutoff = 0.9 * len(meanp)
   ```

2. Fit the data with the `polyfit()` and `polyval()` functions:

   ```
   poly = np.polyfit(meanp[:cutoff - 1], meanp[1:cutoff], degree)
   print('Polynomial coefficients', poly)

   fit = np.polyval(poly, meanp[cutoff:-1])
   ```

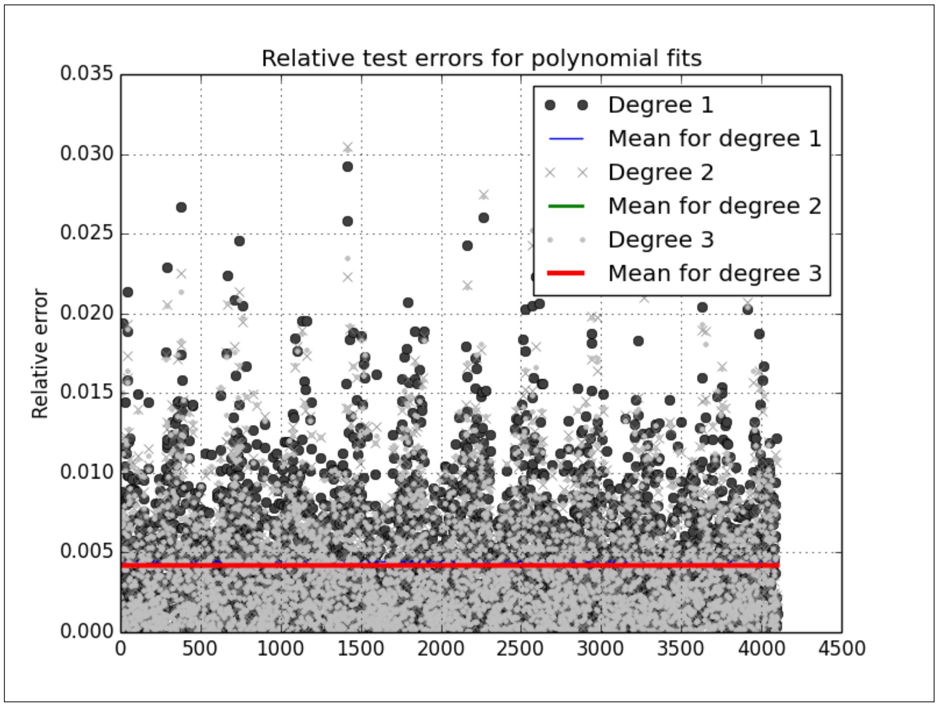
3. Calculate the relative error:

   ```
   error = np.abs(meanp[cutoff + 1:] - fit)/fit
   ```

   This code prints the following output:

   ```
   Polynomial coefficients [ 0.995542    4.50866543]
   Absolute mean relative error 0.00442472512506 for polynomial of
   degree 1


   Polynomial coefficients [ -1.79946321e-04   1.17995347e+00
   2.77195814e+00]
   Absolute mean relative error 0.00421276856088 for polynomial of
   degree 2


   Polynomial coefficients [  3.17914507e-06  -6.62444552e-03
   4.44558056e+00   2.76520065e+00]
   Absolute mean relative error 0.0041906802632 for polynomial of
   degree 3
   ```

Refer to the following plot for the end result:



How it works...

The mean relative errors for the three polynomials are very close—around .004—so we see a single line in the plot (it would be interesting to know what the typical measurement error is for atmospheric pressure), which is smaller than a percent. We see some potential outliers, but not too many. Most of the heavy lifting was done by the `polyfit()` and `polyval()` functions, which respectively fit data to a polynomial and evaluate the polynomial.

See also

- The *Exploring atmospheric pressure* recipe
- The Wikipedia page about cross-validation at `http://en.wikipedia.org/wiki/Cross-validation_%28statistics%29`
- The documentation for `polyfit()` is at `http://docs.scipy.org/doc/numpy/reference/generated/numpy.polyfit.html`
- The documentation for `polyval()` is at `http://docs.scipy.org/doc/numpy/reference/generated/numpy.polyval.html`

Predicting pressure with a moving average model

A simple way to model atmospheric pressure is to assume that values dance around a mean **μ**. We then assume in the simplest case that **deviations of consecutive values ε** from the mean follow this equation:

$$P_t = \mu + \varepsilon_t + \theta\varepsilon_{t-1}$$

The relation is linear and in the simplest case, we need to estimate only one parameter—**θ**. To do so, we will require SciPy functionality. The full code for this recipe is in the `moving_average.py` file in this book's code bundle:

```
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt
from datetime import datetime as dt
from scipy.optimize import leastsq


data = np.load('cbk12.npy')

# Load average pressure
meanp = .1 * data[:,1]

cutoff = 0.9 * len(meanp)

def model(p, ma1):
   return p * ma1

def error(p, t, ma1):
   return t - model(p, ma1)

p0 = [.9]
mu = meanp[:cutoff].mean()
params = leastsq(error, p0, args=(meanp[1:cutoff] - mu,
meanp[:cutoff-1] - mu))[0]
print(params)

abs_error = np.abs(error(params, meanp[cutoff+1:] - mu,
meanp[cutoff:-1] - mu))
```

```
plt.plot(abs_error, label='Absolute error')
plt.plot(np.full_like(abs_error, abs_error.mean()), lw=2,
label='Absolute mean error')
plt.title('Absolute error for the moving average model')
plt.ylabel('Absolute error (hPa)')
plt.grid()
plt.legend(loc='best')
plt.show()
```

Getting started

If necessary, install SciPy by following the instructions in the *Installing SciPy* recipe of *Chapter 2*, *Advanced Indexing and Array Concepts*.

How to do it...

The following steps apply the moving average model to atmospheric pressure.

1. Define the following functions:

   ```
   def model(p, ma1):
      return p * ma1

   def error(p, t, ma1):
      return t - model(p, ma1)
   ```

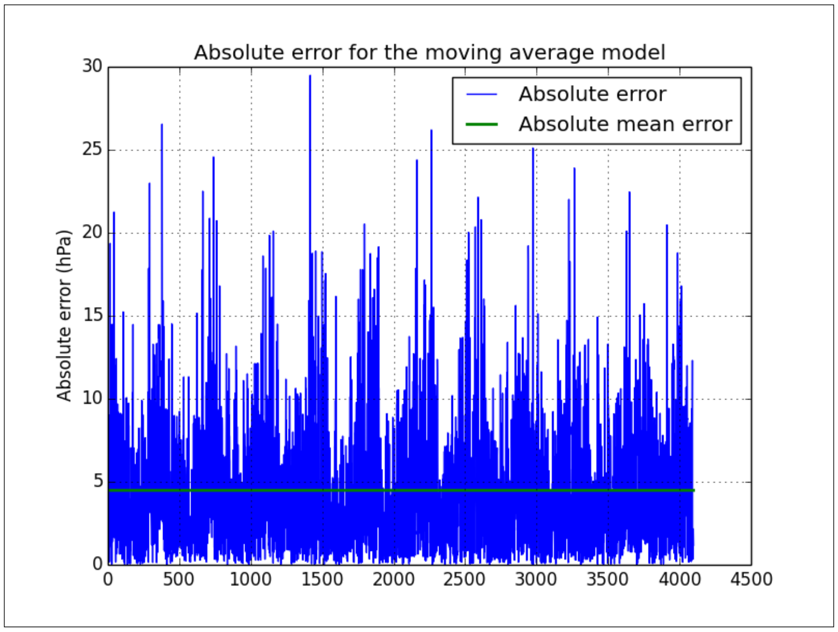
2. Use the functions from the previous step to fit a moving average model with the `leastsq()` function and initial guess of `0.9` for the model parameter:

   ```
   p0 = [.9]
   mu = meanp[:cutoff].mean()
   params = leastsq(error, p0, args=(meanp[1:cutoff] - mu,
   meanp[:cutoff-1] - mu))[0]
   ```

3. Compute the absolute error after fitting using the test dataset:

   ```
   abs_error = np.abs(error(params, meanp[cutoff+1:] - mu,
   meanp[cutoff:-1] - mu))
   ```

Refer to the following plot of the absolute error for each data point in the dataset:



How it works...

The `leastsq()` function fits a model by minimizing errors. It requires a function that computes the error of the fit and an initial guess for model parameters.

See also

- The *Exploring atmospheric pressure* recipe
- The Wikipedia page about the moving average model at `http://en.wikipedia.org/wiki/Moving-average_model`
- The documentation for `leastsq()` is at `http://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.leastsq.html`

Studying intrayear average pressure

It's interesting to explore the atmospheric pressure within a year. In particular, it may be informative to check for patterns related with variability, and therefore, predictability. The reason is that atmospheric pressure in some months can vary much, and reduce predictability. In this recipe, we will plot monthly box plots and monthly variance of atmospheric pressure.

The recipe code is in the `intrayear.py` file in this book's code bundle. Please pay extra attention to the highlighted sections:

```
import numpy as np
import matplotlib.pyplot as plt
import calendar as cal

data = np.load('cbk12.npy')

# Multiply to get hPa values
meanp = .1 * data[:,1]


# Get months
dates = data[:,0]
months = (dates % 10000)/100

monthly = []
vars = np.zeros(12)
month_range = np.arange(1, 13)

for month in month_range:
   indices = np.where(month == months)
   selection = meanp[indices]

   # Filter out 0 values
   selection = selection[selection > 0]

   monthly.append(selection)
   vars[month - 1] = np.var(selection)

def plot():
    plt.xticks(month_range, cal.month_abbr[1:13])
    plt.grid()
    plt.xlabel('Month')

plt.subplot(211)
plot()
plt.title('Atmospheric pressure box plots')
plt.boxplot(monthly)
plt.ylabel('Atmospheric pressure (hPa)')

plt.subplot(212)
```

```
plot()

# Display error bars using standard deviation
plt.errorbar(month_range, vars, yerr=vars.std())
plt.plot(month_range, np.full_like(month_range, np.median(vars)),
lw=3, label='Median')

# Shades the region above the median
plt.fill_between(month_range, vars, where=vars>np.median(vars),
color='0.5')
plt.title('Variance of atmospheric pressure')
plt.ylabel('Variance')
plt.legend(loc='best')

plt.show()
```

How to do it...

While we are exploring, the steps tend to be repeated, and there is an overlap between this recipe and the other recipes in this book. The following steps are new in this recipe:

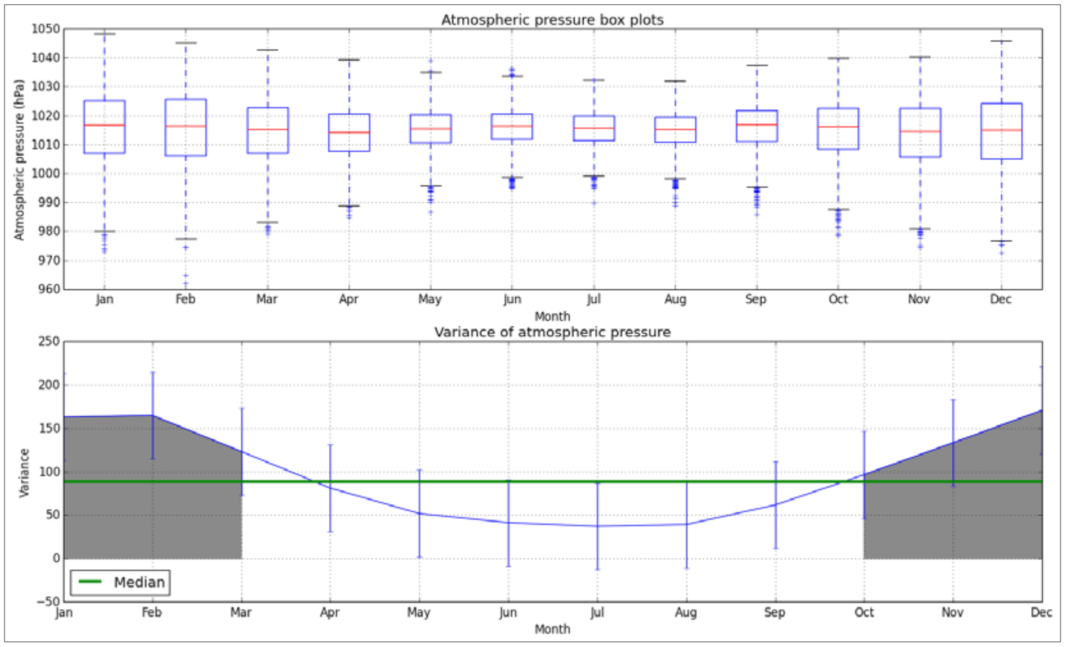
1. Display error bars using the standard deviation:

   ```
   plt.errorbar(month_range, vars, yerr=vars.std())
   ```

2. Shade the region of the plot with values above the median:

   ```
   plt.fill_between(month_range, vars,
   where=vars>np.median(vars), color='0.5')
   ```

   Refer to the following plot for the end result:



How it works...

We matched months to measurements of atmospheric pressure. We used the matches to draw box plots and visualize monthly variance. This study shows that the atmospheric pressure variance is above the median in the coldest months of January, February, November, and December. From the plots, we see that the pressure ranges narrow in the warm summer months. This is consistent with the results from the other recipes.

See also

- The *Exploring atmospheric pressure* recipe
- The *Studying annual atmospheric pressure averages* recipe
- The documentation for `var()` is at `http://docs.scipy.org/doc/numpy/reference/generated/numpy.var.html`

Studying extreme values of atmospheric pressure

Outliers are a problem because they influence our understanding of data. In this recipe, we define outliers to be away from the first or third quartile of the data by at least 1.5 times the **interquartile range**. The interquartile range is the distance between the first and third quartiles. Let's count the outliers for each month of the year. The complete code is in the `extreme.py` file in this book's code bundle:

```
import numpy as np
import matplotlib.pyplot as plt
import calendar as cal

data = np.load('cbk12.npy')

# Multiply to get hPa values
meanp = .1 * data[:,1]

# Filter out 0 values
meanp = np.ma.array(meanp, mask = meanp == 0)

# Calculate quartiles and irq
q1 = np.percentile(meanp, 25)
median = np.percentile(meanp, 50)
q3 = np.percentile(meanp, 75)

irq = q3 - q1

# Get months
dates = data[:,0]
months = (dates % 10000)/100

m_low = np.zeros(12)
m_high = np.zeros(12)
month_range = np.arange(1, 13)

for month in month_range:
   indices = np.where(month == months)
   selection = meanp[indices]
   m_low[month - 1] = len(selection[selection < (q1 - 1.5 * irq)])
   m_high[month - 1] = len(selection[selection > (q3 + 1.5 * irq)])

plt.xticks(month_range, cal.month_abbr[1:13])
plt.bar(month_range, m_low, label='Low outliers', color='.25')
plt.bar(month_range, m_high, label='High outliers', color='0.5')
plt.title('Atmospheric pressure outliers')
plt.xlabel('Month')
plt.ylabel('# of outliers')
plt.grid()
plt.legend(loc='best')
plt.show()
```

How to do it...

To plot the number of outliers for each month of the year, do the following steps:

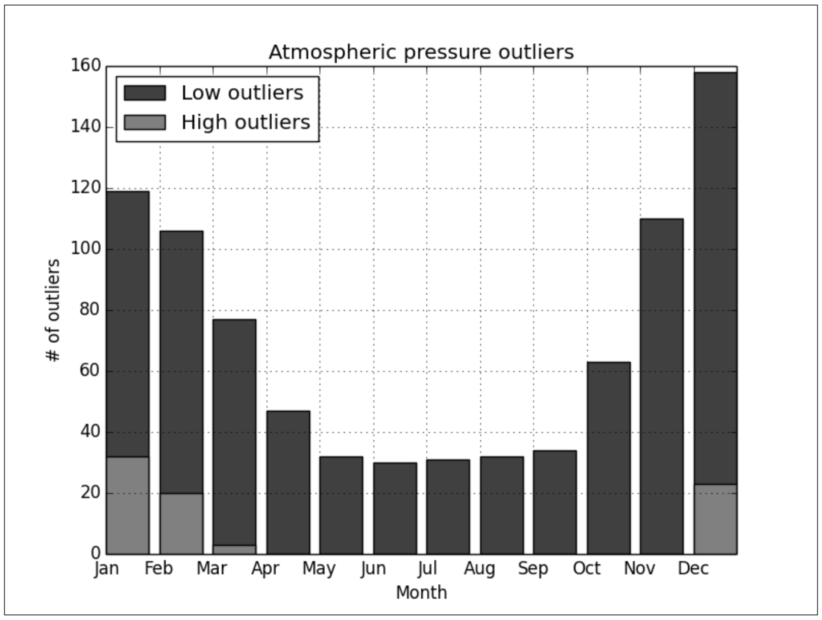
1. Compute the quartiles and the interquartile range with the `percentile()` function:

```
q1 = np.percentile(meanp, 25)
median = np.percentile(meanp, 50)
q3 = np.percentile(meanp, 75)

irq = q3 - q1
```

2. Count the number of outliers, as follows:

```
for month in month_range:
   indices = np.where(month == months)
   selection = meanp[indices]
   m_low[month - 1] = len(selection[selection < (q1 - 1.5 * irq)])
   m_high[month - 1] = len(selection[selection > (q3 + 1.5 *
irq)])
```

Refer to the following plot for the end result:



How it works...

It looks like we got outliers mostly on the lower side and they are less probable in summer. The outliers on the higher side seem to occur only during certain months. We found the quartiles with the `percentile()` function, using the fact that a quarter corresponds to 25 percent.

See also

- The *Exploring atmospheric pressure* recipe
- The documentation for the `percentile()` function is at `http://docs.scipy.org/doc/numpy-dev/reference/generated/numpy.percentile.html`

Index