We all have big hard drives that sometimes contain stuff that we usually forget about. It would be nice to see what is inside such a directory, and what the biggest file inside is. Although there are many more sophisticated and elaborate software products for this job, we want to demonstrate how this is achievable using Python and matplotlib.

How to do it...

Let's perform the following steps:

1. Implement a few helper functions to deal with folder discovery and internal data structures.

2. Implement the main function, draw(), that does the plotting.
3. Implement the main program body that verifies the user input arguments:

```
import os
import sys

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

def build_folders(start_path): folders = []

for each in get_directories(start_path):
size = get_size(each)
if size >= 25 * 1024 * 1024:

folders.append({'size' : size, 'path' : each})

for each in folders:
print "Path: " + os.path.basename(each['path']) print "Size: " + str(each['size'] /
1024 / 1024) + " MB" return folders

def get_size(path):
assert path is not None
total_size = 0
for dirpath, dirnames, filenames in os.walk(path): for f in filenames:

fp = os.path.join(dirpath, f)
try:
size = os.path.getsize(fp)
total_size += size
#print "Size of '{0}' is {1}".format(fp, size) except OSError as err:
```

```
print str(err)
pass
return total_size

def get_directories(path):
dirs = set()
for dirpath, dirnames, filenames in os.walk(path):

dirs = set([os.path.join(dirpath, x) for x in dirnames]) break # we just want the
first one
return dirs

def draw(folders):
""" Draw folder size for given folder"""
figsize = (8, 8) # keep the figure square
ldo, rup = 0.1, 0.8 # leftdown and right up normalized fig =
plt.figure(figsize=figsize)
ax = fig.add_axes([ldo, ldo, rup, rup], polar=True)

# transform data
x = [os.path.basename(x['path']) for x in folders] y = [y['size'] / 1024 / 1024 for y
in folders] theta = np.arange(0.0, 2 * np.pi, 2 * np.pi / len(x)) radii = y

bars = ax.bar(theta, radii)
middle = 90/len(x)
theta_ticks = [t*(180/np.pi)+middle for t in theta] lines, labels =
plt.thetagrids(theta_ticks, labels=x,

frac=0.5)
for step, each in enumerate(labels):
each.set_rotation(theta[step]*(180/np.pi)+ middle) each.set_fontsize(8)

# configure bars
colormap = lambda r:cm.Set2(r / len(x)) for r, each in zip(radii, bars):

each.set_facecolor(colormap(r)) each.set_alpha(0.5)
plt.show()
```

4. Next, we will implement the main program body where we verify the input arguments given by the user when the program is called from the command line:

if __name__ == '__main__':
if len(sys.argv) is not 2:
print "ERROR: Please supply path to folder."
sys.exit(-1)

start_path = sys.argv[1]

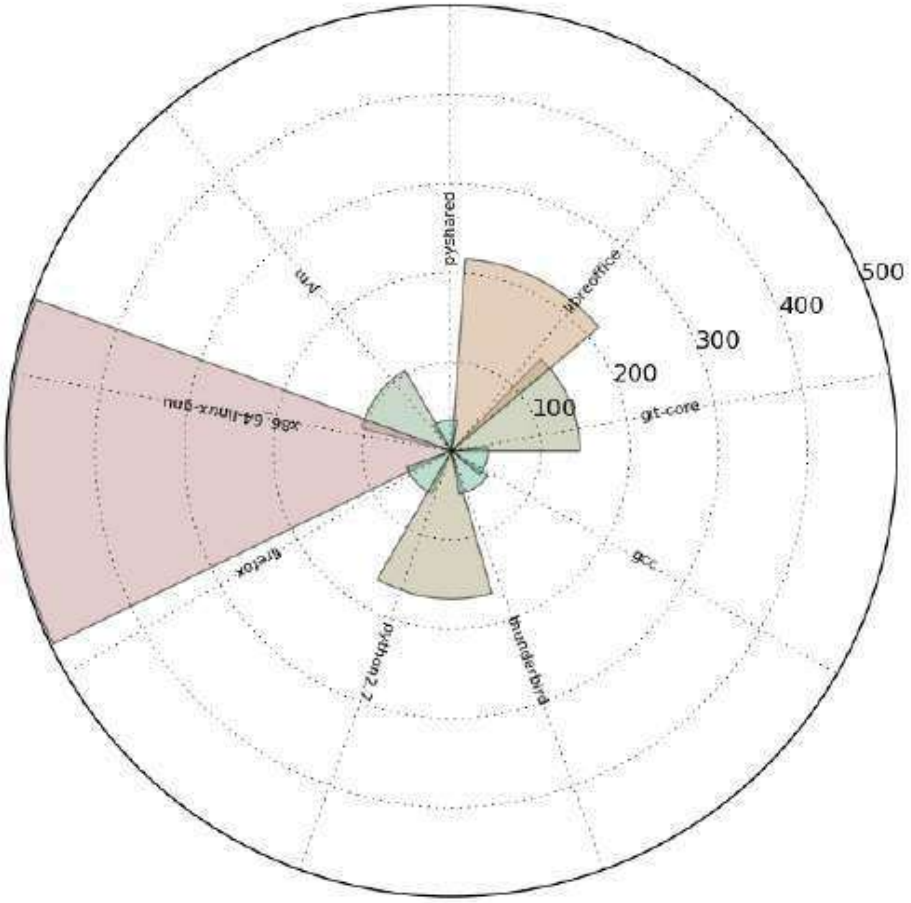
if not os.path.exists(start_path): print "ERROR: Path must exits." sys.exit(-1)

folders = build_folders(start_path)

if len(folders) < 1:
print "ERROR: Path does not contain any folders." sys.exit(-1)

draw(folders) You need to run the following from the command line: **$ python ch04_rec11_filesystem.py /usr/lib/**

It will produce a plot similar to this one:



How it works...

We will start from the bottom of the code, after if __name__ == '__main__' because that is the place where our program starts.
Using the module sys, we pick up command-line arguments; they represent the path to directory we want to visualize.

The function build_folders builds the list of dictionaries, each containing the size and path that it found inside the given start_path. This function calls get_directories, which returns a list of all the subdirectories in start_path. Later, for each found directory, we calculated the sizes in bytes using the get_size function.

For debugging purposes, we print our dictionary so that we are able to compare the figure to what our data looks like.

After we have built the folders as a list of dictionaries, we pass them to a function, draw, that performs all the work of transforming the data to the right dimensions (here, we are using the polar coordinate system), constructing the polar figure, and drawing all the bars, ticks, and labels.

Strictly speaking, we should divide this job into smaller functions, especially if this code is to be further developed.

5

Making 3D Visualizations

We will learn the following recipes in this chapter:
f Creating 3D bars

f Creating 3D histograms
f Animating in matplotlib
f Animating with OpenGL

Introduction

Visualization in 3D is sometimes effective, and sometimes inevitable. Here, we present some examples that will satisfy the most frequent requirements.
The content of this chapter will introduce and explain some topics on 3D visualizations.

Creating 3D bars

Although matplotlib is mainly focused on plotting and mainly in two dimensions, there are different extensions that enable us to plot over geographical maps, to integrate more with Excel and plot in 3D. These extensions are called toolkits in the matplotlib world. Toolkit is a collection of specific function that focuses on one topic, such as plotting in 3D.

Popular toolkits are Basemap, GTK Tools, Excel Tools, Natgrid, AxesGrid, and mplot3d. We will explore more of mplot3d in this recipe. The toolkit mpl_toolkits.mplot3d provides some basic 3D plotting. Plots supported are scatter, surf, line, and mesh. Although this is not the best 3D plotting library, it comes with matplotlib and we are already familiar with the interface.

Getting ready

Basically, we still need to create a figure and add desired axes to it. The difference is that we specify 3D projection for the figure, and the axes we add are Axes3D.

Now, we can use almost the same functions for plotting. Of course, what are different are the arguments, for we now have three axes that we need to provide data for.

For example, the function mpl_toolkits.mplot3d.Axes3D.plot specifies xs, ys, zs, and zdir arguments. All others are transferred directly to matplotlib.axes.Axes.plot. We will explain these specific arguments:

f xs and ys: These are coordinates for x and y axes
f zs: This is the value(s) for z axis. It can be one for all points or one for each point

f zdir: This will choose what will be the z-axis dimension (usually this is zs, but can be xs or ys)

There is a method rotate_axes in the module mpl_toolkits. mplot3d.art3d that contains 3D artist code and functions to convert 2D artists into 3D versions, which can be added to Axes3D to reorder coordinates so that the axes are rotated along with zdir. The default value is z. Prepending the axis with a '-' does the inverse transform, so zdir can be x, -x, y, -y, z, or -z.

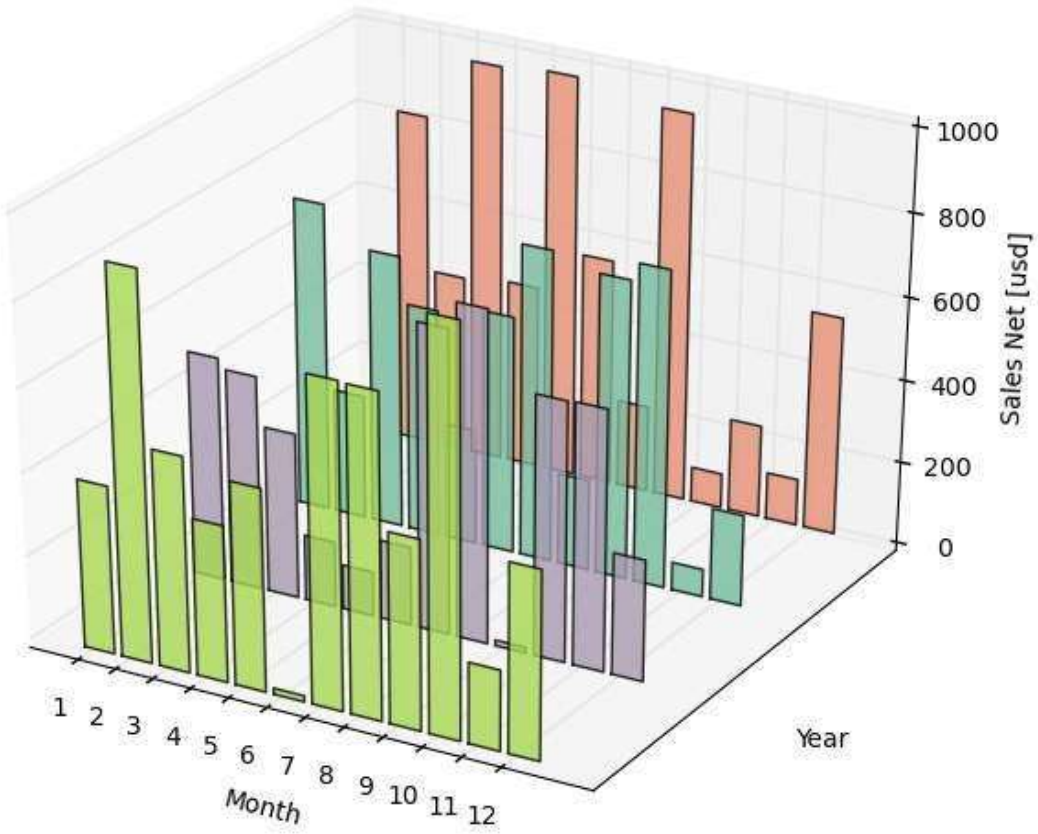
How to do it...

This is the code to demonstrate the concept explained:

```
import random

import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt import matplotlib.dates as mdates

from mpl_toolkits.mplot3d import Axes3D

mpl.rcParams['font.size'] = 10
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

for z in [2011, 2012, 2013, 2014]: xs = xrange(1,13)
ys = 1000 * np.random.rand(12)

color = plt.cm.Set2(random.choice(xrange(plt.cm.Set2.N))) ax.bar(xs, ys, zs=z, zdir='y', color=color, alpha=0.8)
```

```
ax.xaxis.set_major_locator(mpl.ticker.FixedLocator(xs))
ax.yaxis.set_major_locator(mpl.ticker.FixedLocator(ys))

ax.set_xlabel('Month')
ax.set_ylabel('Year')
ax.set_zlabel('Sales Net [usd]')

plt.show()
```

The preceding code produces the following diagram:



How it works...

We had to do the same preparation work as in the 2D world. The difference here is that we needed to specify the kind of backend. Then we generate some random data, for example, 4 years of sale (2011-2014).

We needed to specify the Z values to be the same for the 3D axis.
We picked a color randomly from the color-map set, and then we associated each Z-order collection of xs, ys pairs we would render the bar series.

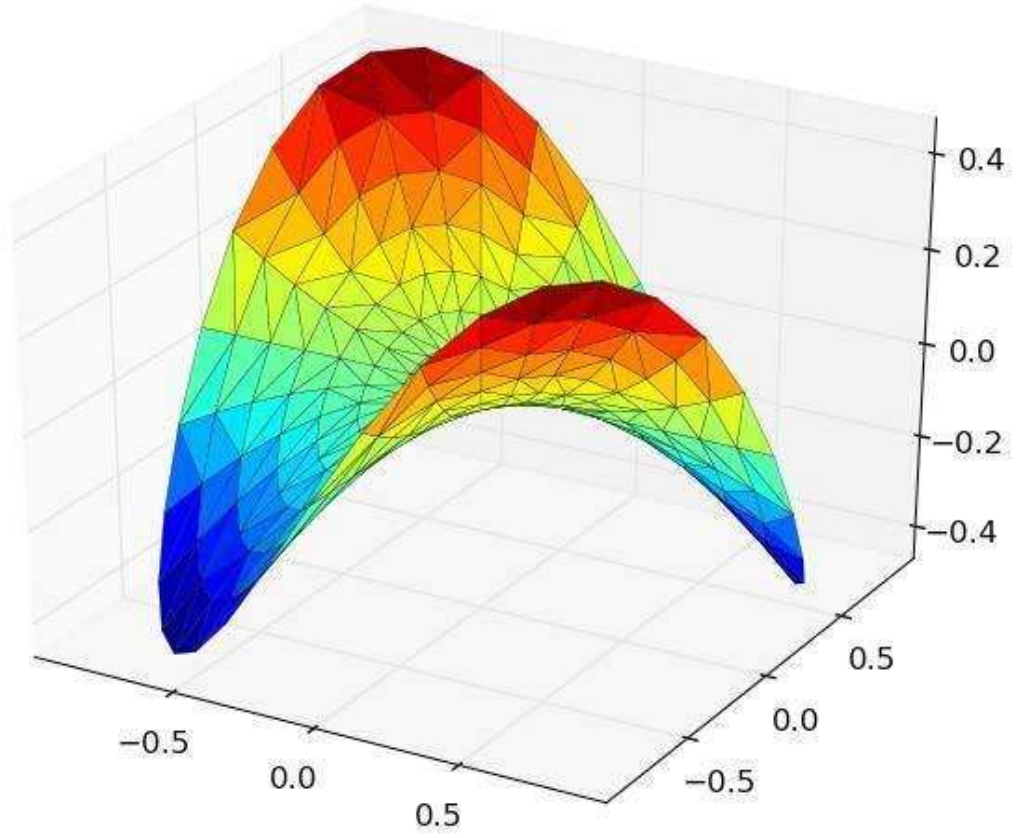
There's more...

Other plotting from 2D matplotlib is available here; for example, scatter() with similar interface to plot() but with added size of the point marker. We are also familiar with contour, contourf, and bar.

The new types that are available just in 3D are wireframe, surface, and tri-surface plots. The following code example plots tri-surface plot of popular Pringle functions or, more mathematically, hyperbolic paraboloid:

```
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
import matplotlib.pyplot as plt
import numpy as np

n_angles = 36 n_radii = 8

# An array of radii
# Does not include radius r=0, this is to eliminate duplicate points radii =
np.linspace(0.125, 1.0, n_radii)

# An array of angles
angles = np.linspace(0, 2*np.pi, n_angles, endpoint=False)
# Repeat all angles for each radius
angles = np.repeat(angles[...,np.newaxis], n_radii, axis=1)

# Convert polar (radii, angles) coords to cartesian (x, y) coords # (0, 0) is added
here. There are no duplicate points in the (x, y) plane
x = np.append(0, (radii*np.cos(angles)).flatten())
y = np.append(0, (radii*np.sin(angles)).flatten())
# Pringle surface

z = np.sin(-x*y)
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot_trisurf(x, y, z, cmap=cm.jet, linewidth=0.2)
plt.show()
```

The preceding code will give the following output:



Creating 3D histograms

Similar to 3D bars, we might want to create 3D histograms. They are used to easily spot correlation between three independent variables. They can be used to extract information from images where the third dimension could be the intensity of a channel in (x,y) space of the image under analysis.

In this recipe, we will learn how to create 3D histograms.

Getting ready

To recall, a histogram represents a number of occurrences of some value in a particular column (usually called "bin"). The three-dimensional histogram, then, represents a number of occurrences in a grid. This grid is rectangular, over two variables that are data in the two columns.

How to do it...

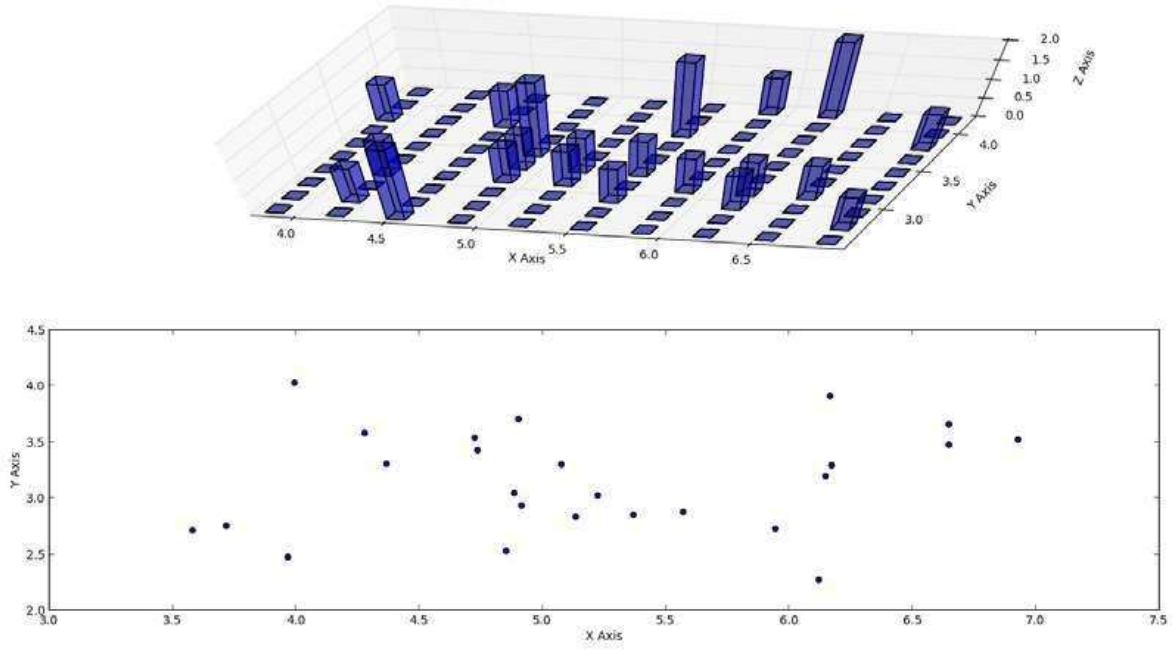
For this computation, we will:
1. Use NumPy because it has the function to compute a histogram of two variables.

2. Generate x and y from normal distributions but with different parameters to be able to distinct the correlation in the resulting histogram.
3. Plot the scatter plot of the same dataset to demonstrate how different is the display of scatter plot to 3D histogram.
Following is the code sample to implement the described steps:

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

from mpl_toolkits.mplot3d import Axes3D
mpl.rcParams['font.size'] = 10
samples = 25
x = np.random.normal(5, 1, samples) y = np.random.normal(3, .5, samples)
fig = plt.figure()
ax1 = fig.add_subplot(211, projection='3d')
# compute two-dimensional histogram
hist, xedges, yedges = np.histogram2d(x, y, bins=10)

# compute location of the x,y bar positions
elements = (len(xedges) - 1) * (len(yedges) - 1) xpos, ypos =
np.meshgrid(xedges[:-1]+.25, yedges[:-1]+.25)

xpos = xpos.flatten() ypos = ypos.flatten() zpos = np.zeros(elements)

# make every bar the same width in base dx = .1 * np.ones_like(zpos)
dy = dx.copy()

# this defines the height of the bar dz = hist.flatten()

ax1.bar3d(xpos, ypos, zpos, dx, dy, dz, color='b', alpha=0.4) ax1.set_xlabel('X
Axis')
ax1.set_ylabel('Y Axis')
ax1.set_zlabel('Z Axis')

# plot the same x,y correlation in scatter plot # for comparison
```

```
ax2 = fig.add_subplot(212)
ax2.scatter(x, y)
ax2.set_xlabel('X Axis')
ax2.set_ylabel('Y Axis')

plt.show()
```

The preceding code will give the following output:



How it works...

We prepare a computer histogram using np.histogram2d that returns our histogram (hist) and the x and y bin edges.

Because for the bard3d function we need coordinates in x, y space, so we need to compute the common matrix coordinates, and for that we use np.meshgrid that combines the x and y positional vectors into the 2D space grid (matrix). We can use this to plot bars in the xy plane locations.

The variables dx and dy represent width of the base of each bar and we want to make this constant, hence we give it a 0.1 point value for every position in the xy plane. The value in the z axis (dz) is actually our computer histogram (in variable hist) that represents the count of common x and y samples at a particular bin.
Scatter plot below (in preceding plots) displays the 2D axes that also visualize

correlation between two similar distributions but with a different set of starting parameters.

Sometimes 3D is what gives us more information and resonates what the data is containing in a better way. As more often 3D visualizations are more confusing than 2D, it is advised to think twice before we choose them over 2D.

Animating in matplotlib

In this recipe, we will explore how to animate our figures. Sometimes it is more descriptive to have pictures moving in animations to explain what is going on when we change values of variables. Our main library has limited but usually sufficient animation capabilities and we will explain how to use them.

Getting ready

A framework for animation is added to the standard matplotlib from version 1.1 and its main class is matplotlib.animation.Animation. This class is a base class; it is to be subclassed for specific behavior as is the case with the already provided classes: TimedAnimation, ArtistAnimation, and FuncAnimation. The following table gives the description of the classes:

Class name (parent class) Animation (object)

Description
This class wraps the creation of an animation using matplotlib. It is only a base class that should be subclassed to provide the needed behavior.
Class name (parent class)
TimedAnimation (Animation)

ArtistAnimation (TimedAnimation)
FuncAnimation (TimedAnimation)

Description
This animation subclass supports time-based animation, and drawing a new frame every interval* milliseconds.
Before calling this function, all plotting should have taken place and the relevant artists saved.
This makes an animation by repeatedly calling a function, passing in (optional)

arguments.

In order to be able to save animations in a video file, we must have ffmpeg or mencoder installer. Installation of these packages varies depending on the OS used, and the changes by different releases, so we must leave it to our dear reader to Google valid information.

How to do it...

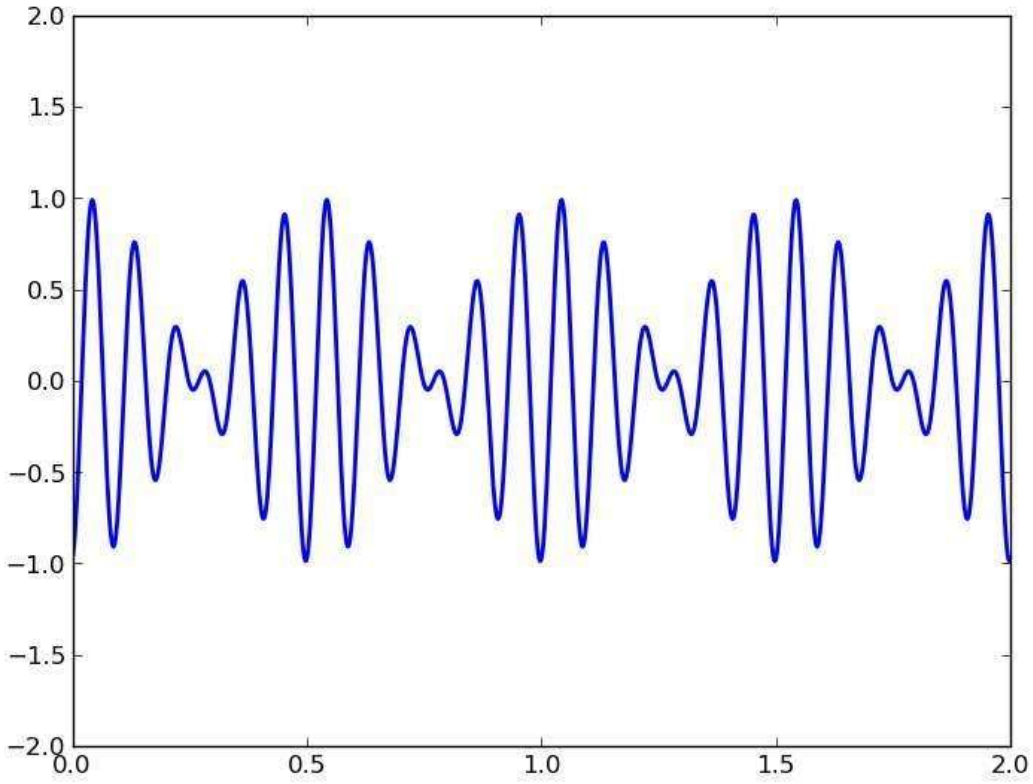
The following code listing demonstrates some matplotlib animations:

```
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import animation

fig = plt.figure()
ax = plt.axes(xlim=(0, 2), ylim=(-2, 2)) line, = ax.plot([], [], lw=2)

def init():
"""Clears current frame.""" line.set_data([], []) return line,

def animate(i):
"""Draw figure.
@param i: Frame counter
@type i: int
"""
x = np.linspace(0, 2, 1000)
y = np.sin(2 * np.pi * (x - 0.01 * i)) * np.cos(22 * np.pi * (x -

0.01 * i))
line.set_data(x, y)
return line,
# This call puts the work in motion
# connecting init and animate functions and figure we want to draw animator =
animation.FuncAnimation(fig, animate, init_func=init,

frames=200, interval=20, blit=True)

# This call creates the video file.
# Temporary, every frame is saved as PNG file # and later processed by ffmpeg
encoder into MPEG4 file # we can pass various arguments to ffmpeg via
```

extra_args animator.save('basic_animation.mp4', fps=30,

extra_args=['-vcodec', 'libx264'], writer='ffmpeg_file')
plt.show()

This will create the file basic_animation.mp4 in the folder from where you started working on this file, and also display a figure window with a running animation. The video file can be opened with most modern video players that support MPEG-4 format. The figure (frame) should look like the following graph:



How it works...

Most important are the functions init(), animate(), and save(). We first construct FuncAnimate by passing two callback functions to it, init and animate. Then we call its save() method to save our video file. More details on each function are in the following table:

Function name init

animate
matplotlib.animation.Animation. save

Usage
Passed to the matplotlib.animation. FuncAnimation constructor via parameter init_func to clear frame before next frame is drawn.
Passed to the matplotlib.animation. FuncAnimation constructor via func parameter.
The figure we want to animate is passed

via fig argument, which under the hood is passed to the matplotlib.animation. Animation constructor to connect
animation events with the figure we want to draw. This function gets (optional) parameters from frames, usually iterable, representing a number of frames.

Saves a movie file by drawing every frame. It creates temporary image files before processing them through encoder (ffmpeg or mencoder) to create a video file. This function also accepts various parameters that configure the video output, metadata (author...), codec to use, resolution/size, and so on. One of the parameters is the one that defines what video encoder to use. Currently supported are ffmpeg, ffmpeg_file, and mencoder.

There's more...

The usage of matplotlib.animation.ArtistAnimation differs from the usage of FuncAnimation, in that we must draw each artist beforehand and then instantiate the ArtistAnimation class with all artists' different frames. Artist animation is a kind of a wrapper of the matplotlib.animation.TimedAnimation class that draws frames every N milliseconds, thus supporting time-based animation. Unfortunately, for the Mac OS X users, animation framework can be troublesome on this platform, and sometimes simply does not work, which will improve with future releases of matplotlib.

Animating with OpenGL

The motivation to use OpenGL comes from the limitations of CPU processing power when we face a task to visualize millions of data points and do it fast (sometimes even in real time).

Modern computers have powerful GPUs that are made for fast visualization related computations (such as games), and there is no reason why they can't be used for scientific-related visualizations.

Actually, there is at least one drawback of writing the hardware accelerated software. As far as hardware dependency is concerned, modern graphic cards require proprietary drivers, sometimes not available on the target platform/machine (for example, user laptop); even when available, sometimes installing required dependencies on site is not what you want to spend your time on, while all you want is to present your findings and demonstrate your research results. This is not a show stopper but have this in mind, and measure benefits and costs of introducing this complexity in your project.

With caveats explained, we can say yes to hardware accelerated visualizations and say yes to OpenGL, industry standard for accelerated graphics.

We will be using OpenGL, as it is across platform, so the examples should work as presented on Linux, Mac, or Windows, given that you have the required hardware and OS level drivers installed.

Getting ready

If you have never used OpenGL, we will now try to give a quick intro, although really to know OpenGL, at least one full book needs to be read and understood. OpenGL is a specification, not an implementation, so OpenGL itself doesn't have any code, where the implementations are libraries developed according to this specification. Those are shipped with your operating system, or by vendors of graphic cards such as NVIDIA, or AMD/ATI.

Moreover, OpenGL is concerned only with graphics rendering and not animation, timing and other complex things that are left for additional library to pick up.

Basics of animating with OpenGL

Because OpenGL is a rendering library, it does not know what objects we draw on a screen. It doesn't care if we draw a cat, a ball, or a line, or all of those objects. So to move a rendered object, we need to clear and draw the whole image again. To animate something, we need a loop that draws and redraws everything very fast, and displays that to a user, so that the user thinks he/ she is seeing an animation.

Installing OpenGL on a machine is a platform dependent process. On Mac OS X, OpenGL implementations is part of OS upgrade, but development libraries (so called "headers") are part of Xcode development package.

On Windows, the best way would be to install vendor's latest graphic drivers for your graphic card. OpenGL may work without them, but you will probably be left without the latest features of stock drivers.

On Linux, if you are not against installing closed source software, there are vendor specific drivers downloadable from either Distro's own software manager, or from vendor site as installable binary. Standard implementation is almost always Mesa3D, the best known OpenGL implementation that uses Xorg to provide support for OpenGL for Linux, FreeBSD, and similar operating systems.

Basically, on Debian/Ubuntu, you should install the following packages and their dependencies: **$ sudo apt-get install libgl1-mesa-dev libgl-mesa-dri**

After this, you should be ready to use some development libraries and/or frameworks to actually write OpenGL backed applications.

We are focused here on Python, so we will overview some of the Python's most used libraries and frameworks that are built on top of OpenGL. We will mention matplotlib and its current and future support for OpenGL:

f Mayavi: This is a library specialized for 3D
f Pyglet: This is a pure Python library for graphics
f Glumpy: This is a fast rendering library built on top of NumPy
f Pyglet and OpenGL: This is used for visualizing Big Data (multi-million data points)

How to do it...

The specialized project Mayavi is a full feature 3D graphic library, and it is mainly used for advanced 3D rendering. It comes with the already mentioned Python packages such as EPD (though not with free license), which is a recommended way to install it on Windows and Mac OS X. On Linux; it can also be easily installed using pip:

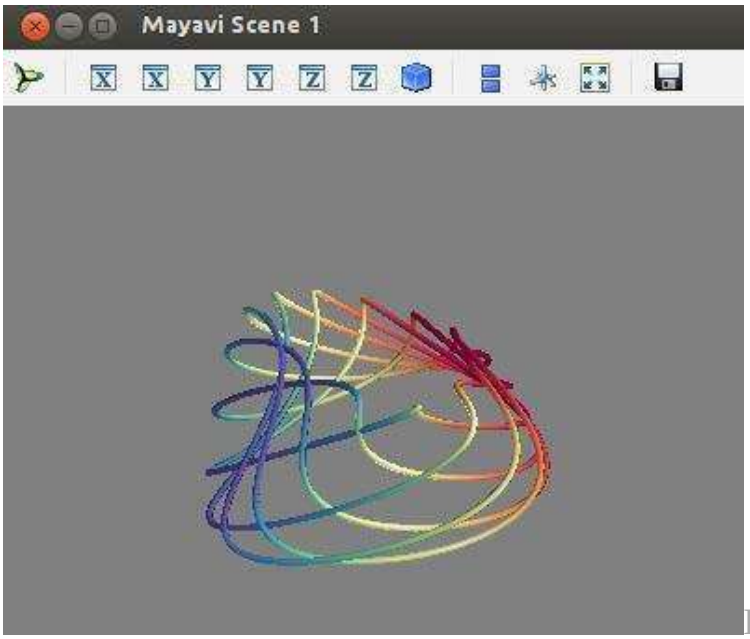
$ pip install mayavi

Mayavi can be used as a development library/framework or as an application. The Mayavi application comprises a visual editor for easy data explorations and somewhat interactive visualization.

As a library it can be used similarly to matplotlib, either from a script interface or as a full object-oriented library. Most of that interface is inside the module mlab, to be able to use that interface. For example, a simple animation with Mayavi can be done as follows:

```
import numpy
from mayavi.mlab import *

# Produce some nice data.
n_mer, n_long = 6, 11
pi = numpy.pi
dphi = pi/1000.0
phi = numpy.arange(0.0, 2*pi + 0.5*dphi, dphi, 'd') mu = phi*n_mer
x = numpy.cos(mu)*(1+numpy.cos(n_long*mu/n_mer)*0.5) y =
numpy.sin(mu)*(1+numpy.cos(n_long*mu/n_mer)*0.5) z =
numpy.sin(n_long*mu/n_mer)*0.5

# View it.
l = plot3d(x, y, z, numpy.sin(mu), tube_radius=0.025, colormap='Spectral')

# Now animate the data.
ms = l.mlab_source
for i in range(100):

x = numpy.cos(mu)*(1+numpy.cos(n_long*mu/n_mer + numpy.pi*(i+1)/5.)*0.5)
scalars = numpy.sin(mu + numpy.pi*(i+1)/5)
ms.set(x=x, scalars=scalars)
```

The preceding code will produce the following window with the rotating figure:



How it works...

We generate the dataset and create a set of functions for x, y, and z to be used in the plot3d function for the start position of the figure.

We then import the mlab_source object that enables us to manipulate our plot on the level of the points and scalars. We use this feature to then set particular points and scalars in for the loop to create a rotation animation with 100 frames.

There's more...

If you want to experiment more, the easiest way to do that is to load IPython, import mayavi.mlab, and run some test_* functions.
To see what is going on, you use IPython's ability to inspect and explore Python source, as shown in the following code:
In [1]: import mayavi.mlab

In [2]: mayavi.mlab.test_simple_surf??
Type: function
String Form:<function test_simple_surf at 0x641b410>
File: /usr/lib/python2.7/dist-packages/mayavi/tools/helper_ functions.py
Definition: mayavi.mlab.test_simple_surf()
Source:
def test_simple_surf():

"""Test Surf with a simple collection of points.""" x, y = numpy.mgrid[0:3:1,0:3:1]

```
return surf(x, y, numpy.asarray(x, 'd'))
```

We see here how by adding two question marks after function name ("??"), IPython found the source of the function and showed it to us. This is a true exploratory computing, and is often used within the visualization community because it is a fast way to get to know your data and code.

Using Pyglet Quickstart

Pyglet is another popular Python library that eases writing graphics and windowing-related applications. It supports OpenGL through its module pyglet.gl, but you don't have to directly talk to this module to use the power of Pyglet. Most convenient use is through pyglet.graphics.

Pyglet takes a different approach to Mayavi; there is no visual IDE, and you are responsible for doing everything from creating a window to issuing a low-level OpenGL call to configure OpenGL context. This is sometimes slower than Mayavi, but what you gain is the ability to control every piece of your application. Sometimes that also means more work hours involved, but usually it means better quality and performance for your applications.

The simplest application (image viewer) can be obtained using the following code:

```
import pyglet
window = pyglet.window.Window()
image = pyglet.resource.image('kitten.jpg')

@window.event
def on_draw():
window.clear()
image.blit(0, 0)

pyglet.app.run()
```

Here you can see that we create a window, load an image, and define what is going to happen when we draw a window object (that is, we define an event handler for the on_draw event). Lastly, we run our application (pyglet.app.run()).

Under the hood, OpenGL is used to draw in windows. This interface is accessible from the pyglet.gl module. Using it directly, though, is not efficient, so pyglet has a simpler interface at pyglet.graphics, where vertex arrays and

buffers are used internally to this interface.

Using Glumpy Quickstart
Glumpy is an OpenGL plus NumPy library for fast NumPy visualization using OpenGL. It is an open source project started by *Nicolas Rougier*, and aimed to be efficient. To use it, we need Python OpenGL bindings, SciPy, and of course Glumpy. Use following commands for this:

sudo apt-get install python-opengl
sudo pip install scipy
sudo pip install glumpy

Glumpy uses OpenGL textures to represent arrays since it is probably the fastest method of visualization on modern graphic hardware.
Pyprocessing introduction

Pyprocessing works in much the same fashion as Processing (http://processing.org). Most functions in Pyprocessing are equivalent to Processing functions. If you are familiar with Processing and Python, you already know almost everything you need to write Pyprocessing applications. To use it, the only thing we need to do is to import the pyprocessing package, write the rest of your code using Pyprocessing functions and data structures, and call run().

There are a lot of free tutorials on OpenGL and how to use it from C/C++ or any other language bindings. One list is provided here, on official OpenGL wiki at http://www. opengl.org/wiki/Getting_started#Tutorials_and_How_To_Guides.

There are many more projects that tackle Python, OpenGL, and 3D visualization in general. Some of them are young, some of them not maintained, but if you spot one that needs to be mentioned, let us know.

6

Plotting Charts with Images and Maps

This chapter contains recipes that will show:
f Processing images with PIL

f Plotting with images
f Displaying image with other plots in the figure
f Plotting data on a map using Basemap
f Plotting data on a map using Google Map API
f Generating CAPTCHA images

Introduction

This chapter explores how to work with images and maps. Python has some well-known image libraries that allow us to process images in both aesthetic and scientific ways. We will touch on PIL's capabilities by demonstrating how to process images by applying filters and by resizing them.
Furthermore, we will show how to use image files as annotation for our matplotlibs' charts. To deal with data visualization of geospatial datasets, we will cover the functionality of Python's available libraries and public APIs that we can use with map-based visual representations. The final recipe shows how Python can create CAPTCHA test images.

Processing images with PIL

Why use Python for image processing, if we could use WIMP (http://en.wikipedia.org/ wiki/WIMP_(computing)) or WYSIWYG (http://en.wikipedia.org/wiki/WYSIWYG) to achieve the same goal? This is used because we want to create an automated system to process images in real time without human support, thus, optimizing the image pipeline.

Getting ready
Note that the PIL coordinate system assumes that the (0,0) coordinate is in the upper-left corner.

The Image module has a useful class and instance methods to perform basic operations over a loaded image object (im):
f im = Image.open(filename): This opens a file and loads the image into im object.

f im.crop(box): This crops the image inside the coordinates defined by box. box defines left, upper, right, lower pixels coordinates (for example: box = (0, 100, 100,100)).

f im.filter(filter): This applies a filter on the image and returns a filtered image.

f im.histogram(): This returns a histogram list for this image, where each item represents the number of pixels. Number of items in the list is 256 for single channel images, but if the image is not a single channel image, there can be more items in the list. For an RGB image the list contains 768 items (one set of 256 values for each channel).

f im.resize(size, filter): This resizes the image and uses a filter for resampling. The possible filters are NEAREST, BILINEAR, BICUBIC, and ANTIALIAS. The default is NEAREST.

f im.rotate(angle, filter): This rotates an image in the counter clockwise direction.
f im.split(): This splits bands of image and returns a tuple of individual bands. Useful for splitting an RGB image into three single band images.

f im.transform(size, method, data, filter): This applies transformation on a given image using data and a filter. Transformation can be AFFINE, EXTENT, QUAD, and MESH. You can read more about transformation in the official documentation. Data defines the box in the original image where the transformation will be applied.

The ImageDraw module allows us to draw over the image, where we can use functions such as arc, ellipse, line, pieslice, point, and polygon to modify the pixels of the loaded image. The ImageChops module contains a number of image channel operations (hence the name Chops) that can be used for image composition, painting, special effects, and other processing operations. Channel operations are allowed only for 8-bit images. Here are some interesting channel operations:

f ImageChops.duplicate(image): This copies current image into a new i mage object
f ImageChops.invert(image): This inverts an image and returns a copy f ImageChops.difference(image1, image2): This is useful for verification that images are the same without visual inspection

The ImageFilter module contains the implementation of the kernel class that allows the creation of custom convolution kernels. This module also contains a set of healthy common filters that allows the application of well-known filters (BLUR and MedianFilter) to our image.

There are two types of filters provided by the ImageFilter module: fixed image enhancement filters and image filters that require certain arguments to be defined; for example, size of the kernel to be used.

We can easily get the list of all fixed filter names in IPython:
In [1]: import ImageFilter
In [2]: [f for f in dir(ImageFilter) if f.isupper()] Out[2]:
['BLUR',

'CONTOUR',
'DETAIL',
'EDGE_ENHANCE',
'EDGE_ENHANCE_MORE',
'EMBOSS',
'FIND_EDGES',
'SHARPEN',
'SMOOTH',
'SMOOTH_MORE']

The next example shows how we can apply all currently supported fixed filters on any supported image:

```
import os
import sys
from PIL import Image, ImageChops, ImageFilter

class DemoPIL(object):
def __init__(self, image_file=None): self.fixed_filters = [ff for ff in
dir(ImageFilter) if ff.isupper()]
```

```
assert image_file is not None
assert os.path.isfile(image_file) is True self.image_file = image_file
self.image = Image.open(self.image_file)

def _make_temp_dir(self):
from tempfile import mkdtemp
self.ff_tempdir = mkdtemp(prefix="ff_demo")

def _get_temp_name(self, filter_name):
name, ext = os.path.splitext(os.path.basename(self.image_ file))

newimage_file = name + "-" + filter_name + ext path =
os.path.join(self.ff_tempdir, newimage_file) return path

def _get_filter(self, filter_name):

# note the use Python's eval() builtin here to return function object
real_filter = eval("ImageFilter." + filter_name)
return real_filter

def apply_filter(self, filter_name):
print "Applying filter: " + filter_name filter_callable =
self._get_filter(filter_name) # prevent calling non-fixed filters for now if
filter_name in self.fixed_filters:

temp_img = self.image.filter(filter_callable) else:
print "Can't apply non-fixed filter now." return temp_img

def run_fixed_filters_demo(self):
self._make_temp_dir()
for ffilter in self.fixed_filters:

temp_img = self.apply_filter(ffilter)
temp_img.save(self._get_temp_name(ffilter)) print "Images are in:
{0}".format((self.ff_tempdir),)

if __name__ == "__main__": assert len(sys.argv) == 2 demo_image =
sys.argv[1]
demo = DemoPIL(demo_image)
# will create set of images in temporary folder demo.run_fixed_filters_demo()
```

We can run this easily from the command prompt:
$ pythonch06_rec01_01_pil_demo.py image.jpeg

We packed our little demo in the DemoPIL class, so we can extend it easily while sharing the common code around the demo function run_fixed_filters_demo. Common code here includes opening the image file, testing if the file is really a file, creating temporary directory to hold our filtered images, building the filtered image filename, and printing useful information to user. This way the code is organized in a better manner and we can easily focus on our demo function, without touching other parts of the code.

This demo will open our image file and apply every fixed filter available in ImageFilter to it and save that new filtered image in a unique temporary directory. The location of this temporary directory is retrieved, so we can open it with our OS's file explorer and view the created images.

As an optional exercise, try extending this demo class to perform other filters available in ImageFilter on the given image.

How to do it...

The example in this section shows how we can process all the images in a certain folder. We specify a target path, and the program reads all the image files in that target path (images folder) and resizes them to a specified ratio (0.1 in this example), and saves each one in a target folder called thumbnail_folder:

```
import os
import sys
from PIL import Image

class Thumbnailer(object):
def __init__(self, src_folder=None): self.src_folder = src_folder

self.ratio = .3
self.thumbnail_folder = "thumbnails"
def _create_thumbnails_folder(self):
thumb_path = os.path.join(self.src_folder, self.thumbnail_ folder)
if not os.path.isdir(thumb_path): os.makedirs(thumb_path)

def _build_thumb_path(self, image_path):
root = os.path.dirname(image_path)
```

```
name, ext = os.path.splitext(os.path.basename(image_path)) suffix =
".thumbnail"
return os.path.join(root, self.thumbnail_folder, name + suffix

+ ext)

def _load_files(self):
files = set()
for each in os.listdir(self.src_folder):

each = os.path.abspath(self.src_folder + '/' + each) if os.path.isfile(each):
files.add(each)
return files
def _thumb_size(self, size):
return (int(size[0] * self.ratio), int(size[1] * self.ratio))

def create_thumbnails(self):
self._create_thumbnails_folder() files = self._load_files()

for each in files:
print "Processing: " + each
try:

img = Image.open(each)
thumb_size = self._thumb_size(img.size) resized = img.resize(thumb_size,
Image.ANTIALIAS) savepath = self._build_thumb_path(each)
resized.save(savepath)

except IOError as ex:
print "Error: " + str(ex)

if __name__ == "__main__":
# Usage:
# ch06_rec01_02_pil_thumbnails.py my_images assert len(sys.argv) == 2
src_folder = sys.argv[1]

if not os.path.isdir(src_folder):
print "Error: Path '{0}' does not exits.".format((src_folder)) sys.exit(-1)
thumbs = Thumbnailer(src_folder)
```

```
# optionally set the name of theachumbnail folder relative to *src_folder*.
thumbs.thumbnail_folder = "THUMBS"

# define ratio to resize image to
# 0.1 means the original image will be resized to 10% of its size thumbs.ratio =
0.1

# will create set of images in temporary folder thumbs.create_thumbnails()
```

How it works...

For the given src_folder folder, we load all the files in this folder and try to load each file using Image.open(); this is the logic of the create_thumbnails() function. If the file we try to load is not an image, IOError will be thrown, and it will print this error and skip to next file in the sequence.

If we want to have more control over what files we load, we should change the _load_ files() function to only include files with certain extension (file type):

```
for each in os.listdir(self.src_folder):
if os.path.isfile(each) and os.path.splitext(each) is in
('.jpg','.png'):
self._files.add(each)
```

This is not foolproof, as file extension does not define file type, it just helps the operating system to attach a default program to the file, but it works in majority of the cases and is simpler than reading a file header to determine the file content (which still does not guarantee that the file really is the first couple of bytes, say it is).

There's more...

With PIL, although not used very often, we can easily convert images from one format to the other. This is achievable with two simple operations: first open an image in a source format using open(), and then save that image in the other format using save(). Format is defined either implicitly via filename extension (.png or .jpeg), or explicitly via the format of the argument passed to the save() function.

Plotting with images

Images can be used to highlight the strengths of your visualization in addition to pure data values. Many examples have proven that by using symbolic images, we map deeper into the viewer's mental model, thereby helping the viewer to remember the visualizations better and for a longer time. One way to do so is to place images where your data is, to map the values to what they represent. The matplotlib library is capable of delivering this functionality, thus we will demonstrate how to do it.

Getting ready

Use the fictional example from the story *The Gospel of the Flying Spaghetti Monster*, by *Bobby Henderson* where the author correlates number of pirates with sea-surface temperature. To highlight this correlation, we will display the size of the pirate ship proportional to the value representing the number of pirates in the year the sea-surface temperature is measured.

We will use Python matplotlib library's ability to annotate using images and text with advanced location settings, as well as arrow capabilities.
All the files required in the following recipe are available in the source code repository in the ch06 folder.

How to do it...

The following example shows how to add an annotation to a chart using images and text:

```
import matplotlib.pyplot as plt
from matplotlib._png import read_png
from matplotlib.offsetbox import TextArea, OffsetImage, \

AnnotationBbox

def load_data():
import csv
with open('pirates_temperature.csv', 'r') as f:

reader = csv.reader(f)
header = reader.next()
datarows = []
for row in reader:

datarows.append(row)
```

return header, datarows

def format_data(datarows): years, temps, pirates = [], [], [] for each in datarows:

years.append(each[0])
temps.append(each[1])
pirates.append(each[2])

return years, temps, pirates

After we have defined helper functions, we can approach the construction of the figure object and add subplots. We will annotate these for every year in the collection of years using the image of the ship, scaling the image to the appropriate size:

if __name__ == "__main__":
fig = plt.figure(figsize=(16,8))
ax = plt.subplot(111) # add sub-plot

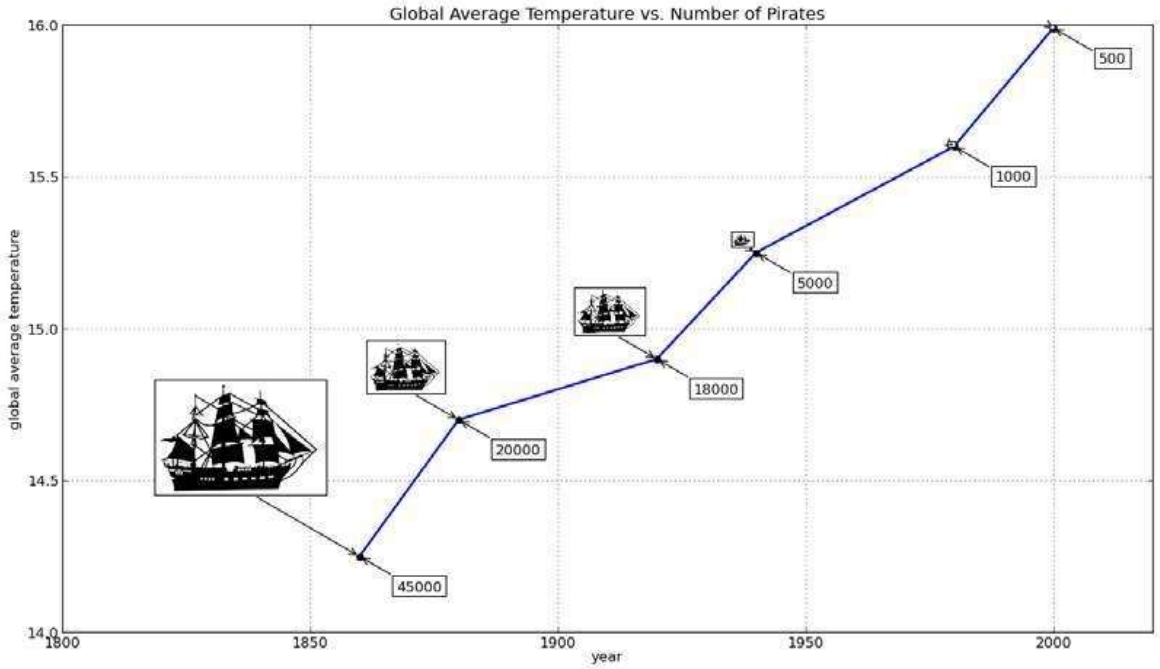
header, datarows = load_data()
xlabel, ylabel = header[0], header[1]
years, temperature, pirates = format_data(datarows) title = "Global Average Temperature vs. Number of Pirates"

plt.plot(years, temperature, lw=2) plt.xlabel(xlabel)
plt.ylabel(ylabel)

# for every data point annotate with image and number for x in xrange(len(years)):
# current data coordinate xy = years[x], temperature[x]
# add image
ax.plot(xy[0], xy[1], "ok")
# load pirate image
pirate = read_png('tall-ship.png')
# zoom coefficient (move image with size) zoomc = int(pirates[x]) * (1 / 90000.)
# create OffsetImage
imagebox = OffsetImage(pirate, zoom=zoomc)

create anotation bbox with image and setup properties ab = AnnotationBbox(imagebox, xy,

```
xybox=(-200.*zoomc, 200.*zoomc),
xycoords='data',
boxcoords="offset points",
pad=0.1,
arrowprops=dict(arrowstyle="->",
connectionstyle="angle,angleA=0,angleB=30,rad=3")
) ax.add_artist(ab)

# add text
no_pirates = TextArea(pirates[x], minimumdescent=False) ab =
AnnotationBbox(no_pirates, xy,

xybox=(50., -25.),
xycoords='data',
boxcoords="offset points",
pad=0.3,
arrowprops=dict(arrowstyle="->",

connectionstyle="angle,angleA=0,angleB=30,rad=3")
) ax.add_artist(ab)

plt.grid(1)
plt.xlim(1800, 2020) plt.ylim(14, 16)
plt.title(title)

plt.show()
```
The preceding code should give the following plot:



How it works...

We start by creating a figure of a decent size, that is, 16 x 8. We need this size to fit the images we want to display. Now, we load our data from the file, using the csv module. Instantiating the csv reader object, we can iterate over the data from the file row by row. Note how the first row is special, it is the header describing our columns. As we have plotted years on the x axis and temperature on the y axis, we read that:

xlabel, ylabel, _ = header
And use the following lines:
plt.xlabel(xlabel)
plt.ylabel(ylabel)

We used neat Python convention here to unpack the header into three variables, where by using _ for variable name, we indicate that we are not interested in the value of that variable.

We return the header and datarows lists from the load_data function to the main caller. Using the format_data() function, we read every item in the list, and add each separate entity (year, temperature, and number of pirates) into the relevant ID list for that entity. Year is displayed along the x axis, while temperature is on the y axis. The number of pirates is displayed as an image of a pirate ship, and

also to add precision, the value will be displayed. We plot year/temperature values using the standard plot() function, not adding anything more, apart from making the line a bit wider (2 pt).

We proceed then to add one image for every measurement and to illustrate the number of pirates for a given year. For this we loop over the range of values of length (range(len(years))) , plotting one black point on each year/temperature coordinate:

ax.plot(xy[0], xy[1], "ok")
The image of the ship is loaded from file into a suitable array format using the read_png helper function:
pirate = read_png('tall-ship.png')

We then compute zoom coefficient (zoomc) to enable us to scale the size of the image in proportion to the number of pirates for the current (pirates[x]) measurement. We also use the same coefficient to position the image along the plot.

The actual image is then instantiated inside OffsetImage—the image container with relative position to its parent (AnnotationBbox).
AnnotationBbox is an annotation-like class, but instead of displaying just text as with the Axes.annotate function, it can display other OffsetBox instances. This allows us to load an image or text object in an annotation and locate it at a particular distance from the data point, as well as to use the arrowing capabilities (arrowprops) to precisely point to an annotated data point.

We supply the AnnotateBbox constructor with certain arguments: f Imagebox: This must be an instance of OffsetBox (for example, OffsetImage); it is the content of the annotation box

f xy: This is the data point coordinate that the annotation relates to
f xybox: This defines the location of the annotation box

f xycoords: This defines what coordinating system is used by xy (for example, data coordinates)
f boxcoords: This defines what coordinating system is used by xybox (for example, offset from the xy location)
f pad: This specifies the amount of padding
f arrowprops: This is the dictionary of properties for drawing an arrow

connection from an annotation-bounding box to a data point

We add text annotation to this plot, using the same data items from the pirates list, with a slightly different relative position. Most of the arguments of the second AnnotationBbox are the same—we adjust xybox and pad to locate the text to the opposite side of the line. The text is inside the TextArea class instance, this is similar to what we do with image, but with texttime.TextArea and OffsetImage inherit from the same parent class, OffsetBox.

We set the text in this TextArea instance tono_pirates and put it in our AnnotationBbox.

Displaying an image with other plots in the figure

This recipe will show how we can make simple yet effective usage of Python matplotlib library to process image channels and display per-channel histogram of an external image.

Getting ready

We have provided some sample images, but the code is ready to load any image file, provided it is supported by matplotlib's imread function.

In this recipe, we will learn how to combine different matplotlib plots to achieve functionality of a simple image viewer that displays an image histogram for red, green, and blue channels.

How to do it...

To show how to build an image histogram viewer, we are going to implement a simple class named ImageViewer and that class will contain helper methods to:

1. Load image.
2. Separate RGB channels from image matrix.
3. Configure figure and axes (subplots).
4. Plot channel histograms.
5. Plot the image.

The following code shows how to build an image histogram viewer:

```
import matplotlib.pyplot as plt
import matplotlib.image as mplimage
import matplotlib as mpl
import os
```

```
class ImageViewer(object): def __init__(self, imfile): self._load_image(imfile)
self._configure()

self.figure = plt.gcf()
t = "Image: {0}".format(os.path.basename(imfile)) self.figure.suptitle(t,
fontsize=20)

self.shape = (3, 2)

def _configure(self):
mpl.rcParams['font.size'] = 10
mpl.rcParams['figure.autolayout'] = False mpl.rcParams['figure.figsize'] = (9, 6)
mpl.rcParams['figure.subplot.top'] = .9

def _load_image(self, imfile): self.im = mplimage.imread(imfile)

@staticmethod
def _get_chno(ch):
chmap = {'R': 0, 'G': 1, 'B': 2}
return chmap.get(ch, -1)
def show_channel(self, ch):
bins = 256
ec = 'none'
chno = self._get_chno(ch)
loc = (chno, 1)
ax = plt.subplot2grid(self.shape, loc)
ax.hist(self.im[:, :, chno].flatten(), bins, color=ch, ec=ec,\ label=ch, alpha=.7)
ax.set_xlim(0, 255)
plt.setp(ax.get_xticklabels(), visible=True)
plt.setp(ax.get_yticklabels(), visible=False)
plt.setp(ax.get_xticklines(), visible=True)
plt.setp(ax.get_yticklines(), visible=False)
plt.legend()
plt.grid(True, axis='y')
return ax

def show(self):
loc = (0, 0)
axim = plt.subplot2grid(self.shape, loc, rowspan=3) axim.imshow(self.im)
```

```
plt.setp(axim.get_xticklabels(), visible=False) plt.setp(axim.get_yticklabels(),
visible=False) plt.setp(axim.get_xticklines(), visible=False)
plt.setp(axim.get_yticklines(), visible=False) axr = self.show_channel('R')
axg = self.show_channel('G')
axb = self.show_channel('B')
plt.show()

if __name__ == '__main__':
im = 'images/yellow_flowers.jpg' try:

iv = ImageViewer(im)
iv.show()
except Exception as ex:
print ex
```

How it works...

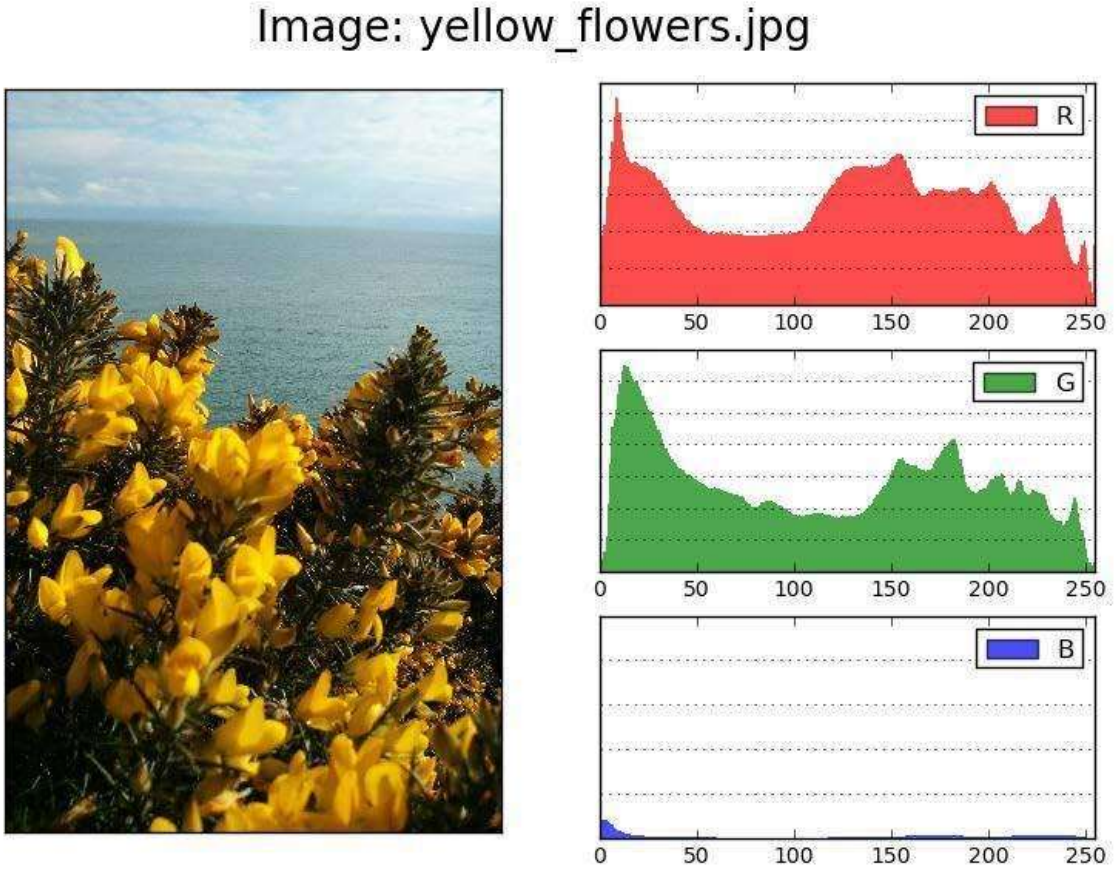
Reading from the end of the code, we see hard-coded filenames. These can be swapped by loading the argument from command line and parsing the given argument into the im variable using the sys.argv sequence.
We instantiate the ImageViewer class with the provided path to an image file. During object instantiation, we try to load an image file into an array, configure the figure via the rcParams dictionary, set the figure size and title, and define object fields (self.shape) to be used inside object's methods.

The main method here is show(), which creates a layout for the figure and loads the image arrays into the main (left column) subplot. We hide any ticks and tick labels as this is the actual image, where we don't have to use the ticks.

We then call the private method show_channel() for each of the red, green, and blue channels. This method also creates new subplot axes, this time in the right-hand side column, with each one in separate row. We plot the histogram for each channel in a separate subplot.

We also set up a little plot to remove unnecessary x ticks and add a legend in case we want to print this figure in a non-color environment. Therefore, we could discern channel representation even in those environments.

After we run this code we will get the following screenshot:



There's more...

The use of histogram plot type is just a choice for this image viewer example. We could have used any of the matplotlib supported plot types. Another real-world example would be to plot EEG or similar medical records where we would want to display slice as an image, the time series of EEG recorded as a line plot, and also additional meta information about the data shown, that would probably go into matplotlib.text.Text artists.

Having the ability to interact with the user GUI event, matplotlib's figure allows us also to implement interaction where we would want to zoom into all plots if we manually zoom on one plot only. That would be another usage where we want to display an image and zoom into it while also zoom into other displayed plots in the currently active figure. An idea would be to use motion_notify_event to call a function that will update x and y limits for all axes (subplots) in the current figure.

Plotting data on a map using Basemap

Probably the best geospatial visualizations are done by overlaying the data over the map. Whether the whole globe, a continent, a state, or even the sky, it is one of the easiest ways for a viewer to comprehend the relation between the data and geography it has displayed.

In this recipe we will be learning how to project data on a map using matplotlib's Basemap toolkit.

Getting ready

As we are already familiar with matplotlib as our plotting engine, we can extend that to matplotlib's capabilities to use other toolkits, one such example being the Basemap mapping toolkit.

Basemap itself doesn't do any plotting. It just transforms given geospatial coordinates to map projection and gives that data to matplotlib for plotting.

First, we need to install the Basemap toolkit. If you are using EPD, Basemap is already installed. If you are on Linux, it is best to use native package managers to install the package containing Basemap. On Ubuntu, for example, the package is called python-mpltoolkits. basemap and can be installed using standard package manager:

$ sudo apt-get install python-mpltoolkits.basemap

On Mac OS X it is recommended to use EPD, although installation using popular package managers such as Homebrew, Fink, and pip is also possible.

How to do it...

Here is a example on how to use the Basemap toolkit to plot simple Mercator projection within a specific region, specified by long, lat coordinate pairs:

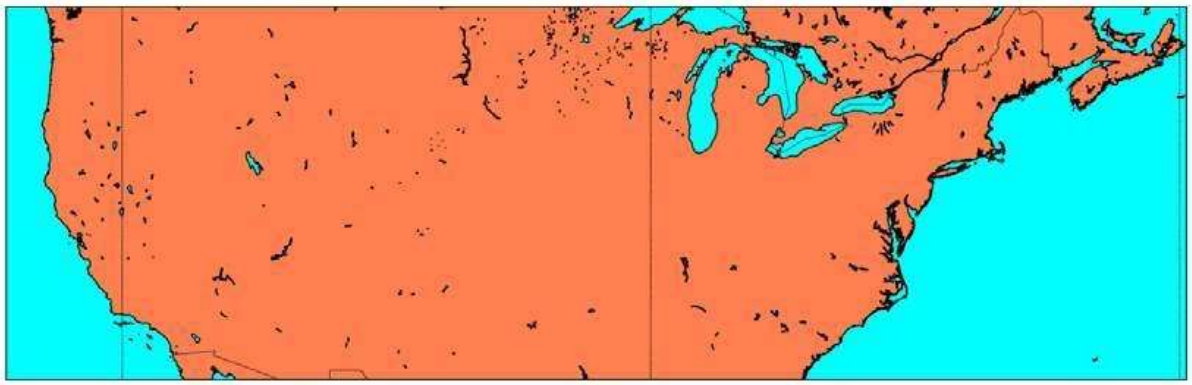
1. We instantiate Basemap defining the projection to be used (merc for Mercator). 2. We define (in the same Basemap constructor) longitude and latitude for the lower-left and upper-right corners of a map.

3. We set up the Basemap instance map, to draw coastlines and countries.
4. We set up the Basemap instance map to fill continents and draw the map boundary.
5. We instruct the Basemap instance map to draw meridians and parallels.

The following code shows how to use Basemap toolkit to plot a simple Mercator projection:

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
import numpy as np

map = Basemap(projection='merc',
resolution = 'h',
area_thresh = 0.1,

llcrnrlon=-126.619875, llcrnrlat=31.354158, urcrnrlon=-59.647219,
urcrnrlat=47.517613)

map.drawcoastlines()
map.drawcountries()
map.fillcontinents(color='coral', lake_color='aqua')
map.drawmapboundary(fill_color='aqua')

map.drawmeridians(np.arange(0, 360, 30)) map.drawparallels(np.arange(-90, 90,
30))
plt.show()
```

This will give a recognizable portion of our globe:



Now that we know how to plot a map, we need to know how to plot data on top of this map. If we recall that Basemap is a big transcoder of longitude and latitude pairs into current map projections, we recognize that all we need is a dataset that contains long/lat that we pass to Basemap for projecting, before plotting over with matplotlib. We use thecities.shp and cities.shx files to load the

coordinates of US cities and project them on to the map. The file is provided in the ch06 folder of the code repository. Here's the example on how to achieve this:

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
import numpy as np

map = Basemap(projection='merc',
resolution = 'h',
area_thresh = 100,

llcrnrlon=-126.619875, llcrnrlat=25,
urcrnrlon=-59.647219, urcrnrlat=55)
shapeinfo = map.readshapefile('cities','cities')
x, y = zip(*map.cities)
# build a list of US cities
city_names = []
for each in map.cities_info:

if each['COUNTRY'] != 'US': city_names.append("")
else:
city_names.append(each['NAME'])

map.drawcoastlines()
map.drawcountries()
map.fillcontinents(color='coral', lake_color='aqua')
map.drawmapboundary(fill_color='aqua')
map.drawmeridians(np.arange(0, 360, 30))
map.drawparallels(np.arange(-90, 90, 30))

# draw city markers
map.scatter(x,y,25, marker='o',zorder=10)
# plot labels at City coords.
for city_label, city_x, city_y in zip(city_names, x, y): plt.text(city_x, city_y,
city_label)
plt.title('Cities in USA')
plt.show()
```

How it works...

The basics of Basemap usage consists of importing the main module and instantiating a Basemap class with desired properties. What we must specify during instantiations are the projections to be used and the portion of a globe that we want to work with.

Additional configuration can be applied before drawing the map and displaying the figure window with matplotlib.pyplot.show().

More than a dozen (or 32, to be precise) of different projections are supported in Basemap. Most of them are very narrow-usage oriented, but some are more general and applied for most common map visualizations.
We can easily see what projections are available by asking the Basemap module itself:

```
In [5]: import mpl_toolkits.basemap

In [6]: print mpl_toolkits.basemap.supported_ projections
mbtfpq McBryde-Thomas Flat-Polar Quartic
aeqd Azimuthal Equidistant
sinu Sinusoidal
poly Polyconic
omerc Oblique Mercator
gnom Gnomonic
moll Mollweide
lcc Lambert Conformal
tmerc Transverse Mercator
nplaea North-Polar Lambert Azimuthal
gall Gall Stereographic Cylindrical
npaeqd North-Polar Azimuthal Equidistant
mill
merc
stere
eqdc
cyl
npstere
spstere
hammer
```

geos
nsper
eck4
aea
kav7
spaeqd South-Polar Azimuthal Equidistant
ortho Orthographic
cass Cassini-Soldner
vandg van der Grinten
laea Lambert Azimuthal Equal Area
splaea South-Polar Lambert Azimuthal
robin Robinson
Miller Cylindrical Mercator Stereographic Equidistant Conic
Cylindrical Equidistant North-Polar Stereographic
South-Polar Stereographic
Hammer Geostationary Near-Sided Perspective Eckert IV Albers Equal Area
Kavrayskiy VII

Usually, we will plot the whole projection, if nothing is specified, some reasonable defaults are used.

To zoom in on a specific region of the map, we will specify the latitude and longitude of the lower-left and upper-right corners of the region you want to show. For this example, we will use the Mercator projection.

Here we see how the arguments names are shortened descriptions: f llcrnrlon: This is lower-left corner longitude f llcrnrlat: This is lower-left corner latitude f urcrnrlon: This is upper-right corner longitude f urcrnrlat: This is upper-right corner latitude

There's more...

We have just scratched the surface of the capabilities of Basemap toolkit, more examples can be found in the official documentation at http://matplotlib.org/basemap/users/ examples.html.

Most of the data used in the examples in the official Basemap documentation is located on remote servers and in a specific format. To efficiently fetch this data, NetCDF data format is used. NetCDF is a common data format designed with

network efficiency in mind. It allows a program to fetch as much data as is needed, even when the whole dataset is very large, which makes using this format very practical. We don't have to download and store large datasets locally every time we want to use them and every time they change.

Plotting data on a map using Google Map API

In this recipe, we will diverge from the desktop environment and show how we can output for the Web. Although, the main language for the web frontend is not Python but HTML, CSS, and JavaScript, we can still use Python for heavy lifting: fetch data, process it, perform intensive computations, and render data in a format(s) suitable for web output, that is, create HTML pages with the required JavaScript version to render our visualization(s).

Getting ready

We will use Google Data Visualization Library for Python to help us prepare data for the frontend interface, where we will use another Google Visualization API to render data in the desired visualization, that is, a map and a table.

Before we start, we need to install the google-visualization-python module. Download the latest stable version from https://code.google.com/p/google-visualizationpython/downloads/detail?name=gviz_api_py-1.8.2.tar.gz&can=2&q=, unpack the archive and install the module. The following actions demonstrate how to do this:

```
$ tar xfv gviz_api_py-1.8.2.tar.gz
$ cd gviz_api_py
$ sudo python ./setup.py install
```

On Windows and Mac OS X use the appropriate software to unpack the tar.gz archive, while the other steps should remain the same. Note that we have to become a super user (that is, gain administrator privileges) to install this module on our system.

A better option, if you don't want to pollute your OS packages, is to create a virtualenv environment to install the packages just for this recipe. We explained how to deal with virutalenv environments in *Chapter 1, Preparing Your Working Environment*.

For the frontend library we don't have to install anything, as that library will be loaded from the web page directly from the Google servers.

We need active access to Internet for this recipe because the output of it will be a web page that will, when opened in a web browser, pull the JavaScript libraries directly from remote servers.

In this recipe, we will be learning how to use Google Data Visualization Library for Python and JavaScript to combine them for creating web visualization.

How to do it...

The following example shows how to visualize Disposable Median Monthly Salary per Country on the world map projection using Google Geochart and Table Visualization, loading the data from a CSV file using Python and the gdata_viz module. We will:

1. Implement a function to act as a template generator.
2. Use the csv module to load the data from the local CSV file.

3. Use DataTable to describe the data and LoadData to load the data from the Python dictionary.
4. Render the output to a web page.

This can be achieved with the following code:

```
import csv
import gviz_api

def get_page_template():
page_template = """
<html>

<script src="https://www.google.com/jsapi" type="text/ javascript"></script>
<script>
google.load('visualization', '1', {packages:['geochart', 'table']});

google.setOnLoadCallback(drawMap); function drawMap() {
var json_data = new google.visualization.DataTable(%s, 0.6);
var options = {colorAxis: {colors: ['#eee', 'green']}}; var mymap = new
google.visualization.GeoChart( document.getElementById('map_div'));
mymap.draw(json_data, options);
```

```
var mytable = new google.visualization.Table(
document.getElementById('table_div'));
mytable.draw(json_data, {showRowNumber: true}) }
</script>
<body>
<H1>Median Monthly Disposable Salary World Countries</H1>

<div id="map_div"></div> <hr />
<div id="table_div"></div>

<div id="source">
<hr />
<small>
Source:
<a href="http://www.numbeo.com/cost-of-living/prices_by_

country.jsp?displayCurrency=EUR&itemId=105">
http://www.numbeo.com/cost-of-living/prices_by_country.jsp?dis
playCurrency=EUR&itemId=105
</a>
</small>
</div>
</body>
</html>
"""
return page_template

def main():
# Load data from CVS file
afile = "median-dpi-countries.csv" datarows = []
with open(afile, 'r') as f:

reader = csv.reader(f)
reader.next() # skip header for row in reader:

datarows.append(row)
# Describe data
description = {"country": ("string", "Country"), "dpi": ("number", "EUR"), }

# Build list of dictionaries from loaded data data = []
```

for each in datarows:

```
data.append({"country": each[0],
"dpi": (float(each[1]), each[1])})
# Instantiate DataTable with structure defined in 'description' data_table =
gviz_api.DataTable(description)
# Load it into gviz_api.DataTable data_table.LoadData(data)
# Creating a JSon string
json = data_table.ToJSon(columns_order=("country", "dpi"),
order_by="country", )

# Put JSON string into the template
# and save to output.html
with open('output.html', 'w') as out:

out.write(get_page_template() % (json,))
if __name__ == '__main__': main()
```

This will produce the output.html file, which we can open in our favorite web browser. The page should look like the following screenshot:

How it works...

The main entry point here is our main() function. First we use the csv module to load our data. This data is obtained from the public website www.numbeo.com, and data is put in the CSV format. The final file is available in the repository for this chapter in the ch06 folder. To be able to use from Google Data Visualization Library, we need to describe the data to it. We describe data using the Python dictionaries, where we define the ID of the columns, their data type, and optional label. In the following example the data is defined in this constraint:

{"name": ("data_type", "Label")}:
description = {"country": ("string", "Country"),
"dpi": ("number", "EUR"), }
Then we need to fit our loaded CSV rows in this format. We will build a list of dictionaries in the data variable.

Now we have everything to instantiate our data_table with gviz_data.DataTable with the described structure. Then we load the data into it and output in the JSON format to our page_template.

The get_page_template()function contains the other part of this equation. It contains a client (frontend) code to produce an HTML web page and a JavaScript code to load Google Data Visualization Library from Google servers. The line that loads the Google's JavaScript API is:

<script src="https://www.google.com/jsapi"
type="text/javascript"></script>

After this, follows another pair of <script>...</script> tags that contains an additional setup. First we load Google Data Visualization Library and the required package—geochart and table:

google.load('visualization', '1', {packages:['geochart', 'table']});
Then we set up a function that will be called when the pages are loaded. This event in the web world is registered as onLoad, so callback is set up via setOnLoadCallback function: google.setOnLoadCallback(drawMap);

This defines that when a page is loaded, the google instance will call the custom function drawMap() that we defined. The drawMap function loads a JSON string into the JavaScript version of the DataTable instance:

var json_data = new google.visualization.DataTable(%s, 0.6); Following that, we create a geochart instance in an HTML element with the ID map_div: var mymap = new google.visualization.GeoChart(
document.getElementById('map_div'));

Draw the map using json_data and provided custom options:
mymap.draw(json_data, options);
Similarly, Google's JavaScript table is rendered below the map:

var mytable = new google.visualization.Table(
document.getElementById('table_div'));
mytable.draw(json_data, {showRowNumber: true})

We save this output as an HTML file that we can open in a browser. This is not so useful for the dynamic rendering of a web service. There is a better option for

this—to output the HTTP Response directly from Python, and thus build a background service responding to client web requests with JSON that client can load and render.

If you want to understand more on reading HTTP responses, please read more on HTTP Protocol and Response messages at http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Response_message.

We do this by replacing the ToJson() call with the ToJSonResponse() with the same signature. This call will respond with a proper HTTP response containing the payload—our JSON-ified data_table ready to be consumed by our JavaScript client.

There's more...

This, of course, is just one example of how we can combine Python as a backend language, sitting on our server, doing the data fetch and processing, while the frontend is left to the universal HTML/JavaScript/CSS set of languages. This enables us to provide interactive and dynamic interfaces with visualizations to a wide audience without requiring them to install anything (well, apart from a web browser, but that is usually installed on a computer or smartphone). Saying that, we must note that the quality of these outputs is not as high as that of matplotlib, whereas the strength of matplotlib lies in the high quality output.

To work more with the Web (and Python) you would have to learn more about the web technologies and languages used. This book does not cover such topics but does give an insight into how to achieve one possible solution using well known third-party libraries that produce pleasing web outputs, with as little web coding as possible.

More documentation is available on the Google Developer portal at https://developers.google.com/chart/interactive/docs/dev/gviz_api_lib.

Generating CAPTCHA images

Although this is not strictly data visualization in terms that we usually refer to, the ability to generate images using Python comes in handy in many cases, and this is one of them. In this recipe, we will be covering the generation of random images to tell humans and computers apart—CAPTCHA image.

Getting ready

CAPTCHA stands for Completely Automated Public Turing test to tell Computers and Humans Apart, and is trademarked by Carnegie Mellon University. This test is used to challenge computer programs (usually referred to as bots) that automatically fill various web forms that are primarily targeted to humans and that should not be automated. Usual examples are sign-up forms, login forms, surveys, and similar.

CAPTCHA itself can take various forms, but the most common form consists of a challenge where humans should read an image with distorted characters and numbers and type in the result in the related response field.

In this recipe, we will learn how to harness Python's Imaging Library to generate images, rendering lines and points, and also rendering text.

How to do it...

We will show what would be involved in creating a personal and simple CAPTCHA generator by performing the following steps:

1. Define size, text, font size, background color, and CAPTCHA length.
2. Pick random characters from the English alphabet.
3. Draw those on the image using defined font and colors.
4. Add some noise in the form of lines and arcs.
5. Return the image object to the caller together with the CAPTCHA challenge.
6. Show the generated image to the user.

The following code shows how to create a personal and simple CAPTCHA generator:

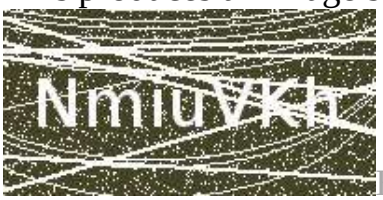
```
from PIL import Image, ImageDraw, ImageFont
import random
import string

class SimpleCaptchaException(Exception): pass
class SimpleCaptcha(object):
def __init__(self, length=5, size=(200, 100), fontsize=36, random_text=None,
random_bgcolor=None):

self.size = size
self.text = "CAPTCHA"
```

```
self.fontsize = fontsize
self.bgcolor = 255 self.length = length

self.image = None # current captcha image
if random_text:
self.text = self._random_text()
if not self.text:
raise SimpleCaptchaException("Field text must not be empty.")
if not self.size:
raise SimpleCaptchaException("Size must not be empty.")
if not self.fontsize:
raise SimpleCaptchaException("Font size must be defined.")
if random_bgcolor:
self.bgcolor = self._random_color()

def _center_coords(self, draw, font):
width, height = draw.textsize(self.text, font)
xy = (self.size[0] - width) / 2., (self.size[1] - height) / 2. return xy

def _add_noise_dots(self, draw):
size = self.image.size
for _ in range(int(size[0] * size[1] * 0.1)):

draw.point((random.randint(0, size[0]), random.randint(0, size[1])), fill="white")

return draw

def _add_noise_lines(self, draw):
size = self.image.size
for _ in range(8):

width = random.randint(1, 2)
start = (0, random.randint(0, size[1] - 1)) end = (size[0],
random.randint(0,size[1]-1)) draw.line([start, end], fill="white", width=width)

for _ in range(8):
start = (-50, -50)
end = (size[0] + 10, random.randint(0, size[1]+10)) draw.arc(start + end, 0, 360,
fill="white")
```

```
return draw
def get_captcha(self, size=None, text=None, bgcolor=None): if text is not None:
self.text = text

if size is not None:
self.size = size
if bgcolor is not None:
self.bgcolor = bgcolor

self.image = Image.new('RGB', self.size, self.bgcolor) # Note that the font file
must be present
# or point to your OS's system font
# Ex. on Mac the path should be '/Library/Fonts/Tahoma.ttf' font =
ImageFont.truetype('fonts/Vera.ttf', self.fontsize) draw =
ImageDraw.Draw(self.image)
xy = self._center_coords(draw, font)
draw.text(xy=xy, text=self.text, font=font)

# Add some dot noise
draw = self._add_noise_dots(draw)
# Add some random lines
draw = self._add_noise_lines(draw)
self.image.show()
return self.image, self.text

def _random_text(self):
letters = string.ascii_lowercase + string.ascii_uppercase random_text = ""
for _ in range(self.length):

random_text += random.choice(letters)
return random_text

def _random_color(self):
r = random.randint(0, 255)
g = random.randint(0, 255)
b = random.randint(0, 255)
return (r, g, b)

if __name__ == "__main__":
sc = SimpleCaptcha(length=7, fontsize=36, random_text=True,
```

random_bgcolor=True)
sc.get_captcha()

This produces an image similar to the following:



How it works...

This example gives a process on how to use Python's imaging library to generate predefined images, to create a simple, yet effective CAPTCHA generator.

We wrapped the functionality into one class SimpleCaptcha, because it gives us a safe space for future development. We also created a custom SimpleCaptchaException to accommodate future exception hierarchies.

If you are writing anything more than trivial and quick and dirty scripts, it is always good to start writing and designing custom exception hierarchies for your domain, rather than using generic Python's standard exceptions. You will gain a lot in the readability and maintenance of the software.

Start reading from the main section, at the end of code listing, where we instantiate class giving settings of our future image as arguments to the constructor. Following that, we call the get_captcha method on the sc object. For this recipe's purposes, get_captcha shows the image object as a result, but we also return the image object to the potential caller of this method, so it could make use of the result. The usage can vary, the caller could either save the image on the file, or if this was a web application, return the image stream and written challenge to the client requesting this CAPTCHA.

The important thing to note is that, in order to finish the challenge-response process of the CAPTCHA test, we must return the CAPTCHA string generated on the image as text so that the caller can compare the user's response with the expected values.

The get_captcha method first verifies the input arguments, in order to override the class' defaults if the user provides custom values. After that, a new image object is instantiated by Image.new. This object is saved in self.image, where we use it to draw and write text. Having written the text to the image, we add the noise of randomly placed points and lines, as well as some arc segments.

These tasks are carried out by the _add_noise_points and_add_noise_lines methods. The first one loops a few times and adds a point to a random location on the image, not too close to the edges of the image, and the latter one draws lines from the left-hand side of the image to the right-hand side of the image.

There's more...

We constructed this class using some assumptions about its use. We assumed that user will just want to accept our default settings (that is, random seven characters on a random background color) and receive the result from it. That is the reasoning behind placing helper functions in the constructor to set random text and random background color. If the most frequent and effective usage is to always override configuration, then we want to remove these operations from the constructor and place them in separate calls.

For example, maybe a user wants to always use English words as the CAPTCHA challenge. If this is the case, we want to be able to just call a method to provide us with results like that. This method could be get_english_captcha and with the random logic of this constructor, we would then construct that method to pick random words from the provided English dictionary. On a Unix system, there is a common English dictionary inside /usr/share/ dict/words that we could use for this:

```
def get_english_captcha(self):
words = '/usr/share/dict/words'
with open(words, 'r') as wf:

words = wf.readlines()
aword = random.choice(words)
aword = aword.strip() # remove newline and spaces

return self.get_captcha(text=aword)
```

Overall, the example of CAPTCHA generation is not production quality and should not be used without adding more protection and randomness, such as letter rotation.

If you need to protect your web forms from bots, there are already third-party Python modules and libraries that you should reuse. There are even specialized modules built for the existing web frameworks.

There are event web services such as reCAPTCHA (http://www.google.com/recaptcha) with already proven Python module recaptcha-client (https://pypi.python.org/ pypi/recaptcha-client) that you can sign up and use. It does not require any imaging libraries because the image is pulled directly from the reCAPTCHA web service but it has other dependencies such as pycrypto. By using this web service and library, you are also helping books scanned using Optical Character Recognition (OCR) from the Google Books project or old editions of the New York Times. Read more on the reCAPTCHA website.

7

Using Right Plots to Understand Data

In this chapter we will cover the following recipes:
f Understanding logarithmic plots

f Understanding spectrograms
f Creating a stem plot
f Drawing streamlines of vector flow
f Using colormaps
f Using scatter plots and histograms
f Plotting the cross-correlation between two variables
f Importance of autocorrelation

Introduction

In this chapter we will focus more on understanding what we want to say with the data we are presenting, and how to say it effectively. We will present some new techniques and plots, but all will be underlined by an understanding of the information we want to convey to the user. Let's ask the question, "Why do we want to present information in this state?". This is the most important question that should be asked during the data exploration phase. If we miss the opportunity to understand the data and present it in a certain way, the viewer, then, is not going to understand the data correctly for sure.

Understanding logarithmic plots

More often than not, reading daily newspapers and similar articles, one can find charts that are used by media organizations to misrepresent the facts. One common example is using linear scales to create so called panic charts, where a constantly growing value is followed for a long period of time (years) and starting values are smaller than the latest one by several magnitudes. These values when visualized correctly would (and usually should) produce linear or almost linear charts, taking some panic out of the articles they illustrate.

Getting ready

With the logarithmic scale, the ratio of consecutive values is constant. This is important when we are trying to read log plots. With linear (arithmetic) scales, the constant is the distance between consecutive values. In other words, logarithmic plots have a constant distance in orders of magnitude. We will see this illustrated in the following plots. The code used to produce this figure is explained next.

As a general rule of thumb, logarithmic scales should be used for the following conditions: f When the data presented has values that span several orders of magnitude f When the data presented has skewness towards large values (some points are much larger than the rest of the data)
f When you want to show the rate of change (growth rate), and not the value of change

Don't go blindly following these rules; they are more like hints, than rules. Always use your own judgment about the data in hand and requirements presented to you by the project or customer.

Depending on the data range, different log bases should be used. The standard base for logs is 10, but if the range of the data is smaller, a base of 2 can prove to be more useful as it will show more resolution within the smaller range.

If we have the range of data suitable for display on logarithmic scales, we will notice that the values previously being too close to judge any difference are now well apart. This allows us to read the chart much easier than if we would present the data in linear scale.

The growth rate charts, where long range time-series data is collected, are where we want to see not the absolute value measured at time point, but the growth in time. We will still get the absolute value information, but that information is of lower priority.

Also, if the data distribution has a positive skew, for example, salaries, taking the logarithm of the value (salary) will help us fit the data into the model, as the logarithm transformation will give us a more normal data distribution.

How to do it...

We will exemplify this with a sample code that shows the same two datasets (one linear, and one logarithmic in nature) on two different plots (in the same figure) using different scales (linear and logarithmic).

We will be performing the following steps with the help of the code mentioned after these steps:

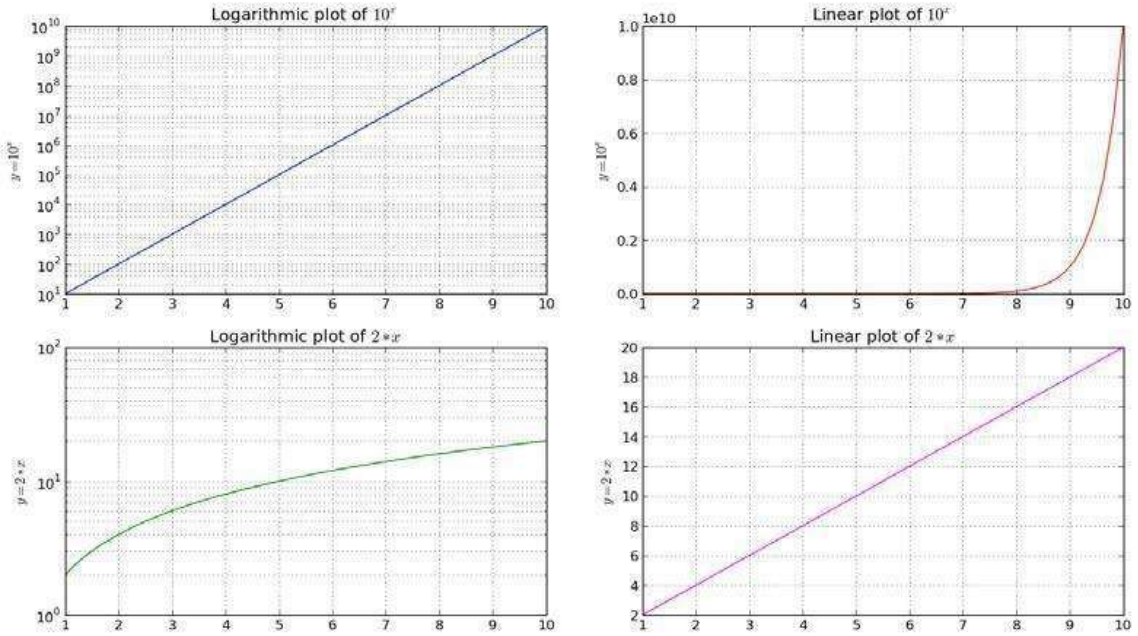
1. Generate two simple datasets: y—exponential/logarithmic in nature and z—linear.
2. Create a figure containing a grid of four subplots.

3. Create two subplots containing the y dataset: one in logarithmic scale and one in linear scale.
4. Create two other subplots containing the z dataset, again one logarithmic and the other linear.
The following is the code:

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

x = np.linspace(1, 10) y = [10 ** el for el in x] z = [2 * el for el in x]

fig = plt.figure(figsize=(10, 8))

ax1 = fig.add_subplot(2, 2, 1)
ax1.plot(x, y, color='blue')
ax1.set_yscale('log')
ax1.set_title(r'Logarithmic plot of $ {10}^{x} $ ') ax1.set_ylabel(r'$ {y} =
{10}^{x} $')
plt.grid(b=True, which='both', axis='both')

ax2 = fig.add_subplot(2, 2, 2)
ax2.plot(x, y, color='red')
ax2.set_yscale('linear')
ax2.set_title(r'Linear plot of $ {10}^{x} $ ') ax2.set_ylabel(r'$ {y} = {10}^{x}
$') plt.grid(b=True, which='both', axis='both')

ax3 = fig.add_subplot(2, 2, 3) ax3.plot(x, z, color='green')
ax3.set_yscale('log')
ax3.set_title(r'Logarithmic plot of $ {2}*{x} $ ') ax3.set_ylabel(r'$ {y} = {2}*
{x} $')
```

```
plt.grid(b=True, which='both', axis='both')

ax4 = fig.add_subplot(2, 2, 4)
ax4.plot(x, z, color='magenta')
ax4.set_yscale('linear')
ax4.set_title(r'Linear plot of $ {2}*{x} $ ') ax4.set_ylabel(r'$ {y} = {2}*{x} $')
plt.grid(b=True, which='both', axis='both')

plt.show()
```

This code will produce the following output:



How it works...

We generate some sample data and two dependent variables: y and z. The variable y is expressed as an exponential function of x (data), and the variable z is a simple linear function of x. This helps us illustrate different looks of linear and exponential charts. We then create a grid of four subplots, where the top row subplots are of data (x, y) and bottom row are of data (x, z) pairs.

Looking from the left-hand side, the columns have logarithmic scales on the y axis, while from the right-hand side, the columns are in the linear scale. We set this using set_yscale('log') for every axis separately.

For every subplot we set the title and label, where the label also describes the

function plotted. With plt.grid(b=True, which='both', axis='both'), we turn the grid on for both the axes and both major and minor ticks.
We observe how linear functions are straight lines on linear plots, while logarithmic functions are straight lines on logarithmic plots.

Understanding spectrograms

A spectrogram is a time-varying spectral representation that shows how the spectral density of a signal varies with time.

A spectrogram represents a spectrum of frequencies of the sound or other signal in a visual manner. It is used in various science fields, from sound fingerprinting such as voice recognition, to radar engineering and seismology.

Usually, a spectrogram layout is as follows: the x axis represents time, the y axis represents frequency, and the third dimension is the amplitude of a frequency-time pair, which is color coded. This is three-dimensional data; therefore, we can also create 3D plots where the intensity is represented as the height on the z axis. The problem with 3D charts is that humans are bad at understanding and comparing them. Also, they tend to take more space than 2D charts.

Getting ready

For serious signal processing, we would go into low-level details to be able to detect patterns and autofingerprint certain specifics; but for this data visualization recipe we will leverage a couple of well-known Python libraries to read an audio file, sample it, and plot a spectrogram.

In order to read WAV files to visualize sound, we need to do some prep work. We need to install the libsndfile1 system library for reading/writing audio files. This is done via your favorite package manager. For Ubuntu, use:

$ sudo apt-get install libsndfile1-dev
It is important to install the dev package, which contains header files, so pip can build the scikits.audiolab module.

We also can install libasound and ALSA (Advanced Linux Sound Architecture) headers to avoid runtime warning. This is optional, as we are not going to use the features provided by the ALSA library. For Ubuntu Linux, issue the

following command:

$ sudo apt-get install libasound2-dev
To install scikits.audiolab, which we will use to read WAV files, we will use pip:
$ pip install scikits.audiolab

Always remember to enter the virtual environment for the current project, as you don't want to dirty the system libraries.

How to do it...

For this recipe, we will use the prerecorded sound file, test.wav, that can be found in the file repository of this book. But we could also generate a sample, which we will try later.

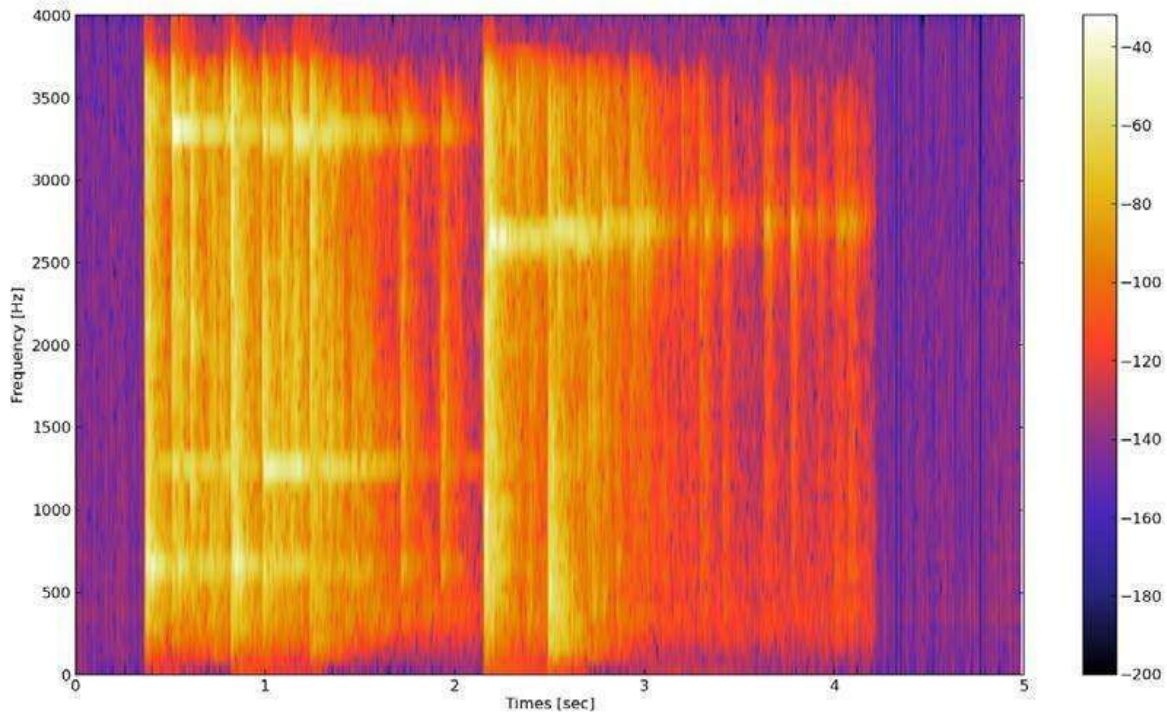
In this example, we perform the following steps in order:
1. Read the WAV file that contains a recorded sound sample.

2. Define the length of the window used for the Fourier Transform using NFFT.
3. Define the overlapping data points using noverlap while sampling.

```
import os
from math import floor, log

from scikits.audiolab import Sndfile
import numpy as np
from matplotlib import pyplot as plt

# Load the sound file in Sndfile instance
soundfile = Sndfile("test.wav")

# define start/stop seconds and compute start/stop frames start_sec = 0
stop_sec = 5
start_frame = start_sec * soundfile.samplerate stop_frame = stop_sec *
soundfile.samplerate

# go to the start frame of the sound object soundfile.seek(start_frame)

# read number of frames from start to stop delta_frames = stop_frame -
start_frame sample = soundfile.read_frames(delta_frames) map = 'CMRmap'

fig = plt.figure(figsize=(10, 6), )
```

```
ax = fig.add_subplot(111)
# define number of data points for FT
NFFT = 128
# define number of data points to overlap for each block noverlap = 65

pxx, freq, t, cax = ax.specgram(sample, Fs=soundfile.samplerate, NFFT=NFFT,
noverlap=noverlap, cmap=plt.get_cmap(map))

plt.colorbar(cax)
plt.xlabel("Times [sec]") plt.ylabel("Frequency [Hz]")

plt.show()
```

This generates the following spectrogram, with visible white-like traces for separate notes:



NFFT defines the number of data points used for computing the Discrete Fourier Transform in each block. The most efficient computation is when NFFT is a value raised to the power of 2. The windows can overlap and the number of data points that are overlapped (that is, repeated) is defined by the noverlap argument.

How it works...

We need to load a sound file first. To do that, we use the scikits.audiolab.SndFile method and provide it with a filename. This will instantiate a sound object, which we can then query for data and call the function on.

To read the data needed for the spectrogram, we need to read the desired frames of data from our sound object. This is done by read_frames(), which accepts the start and end frames. We calculate the frame number by multiplying the sample rate with the time points (start, end) we want to visualize.

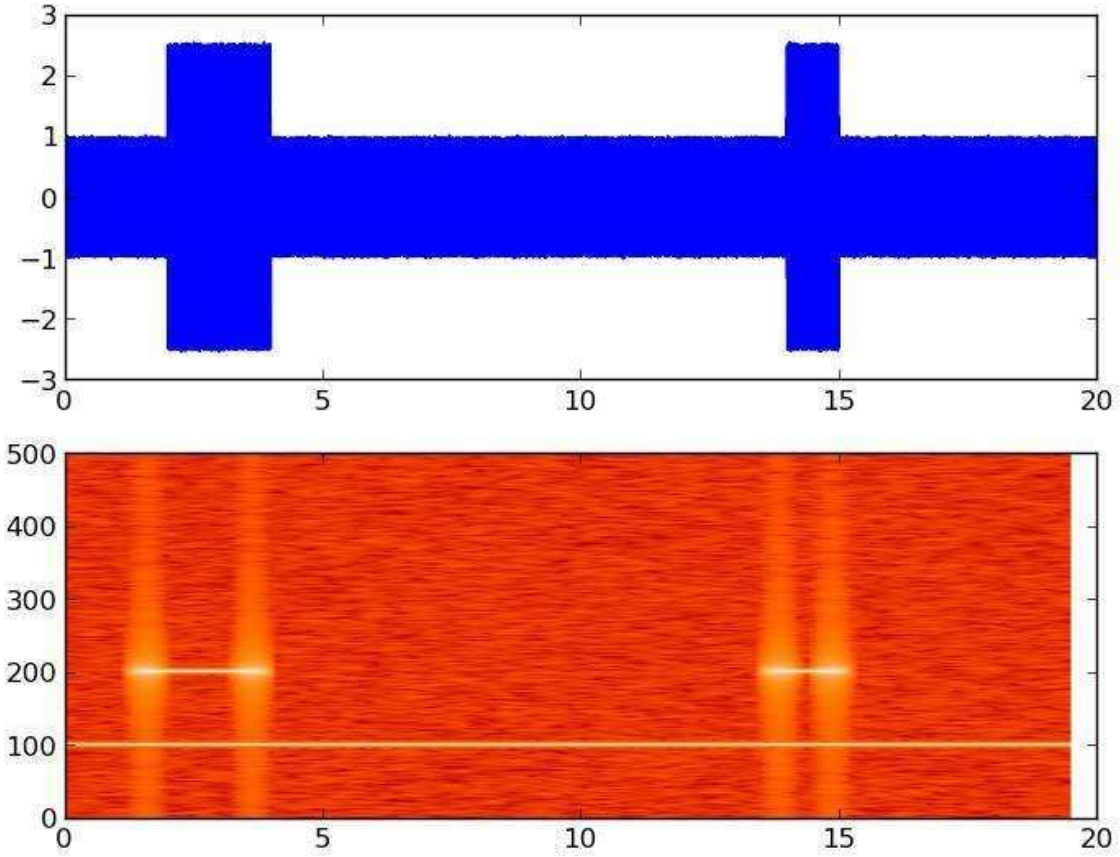
There's more...

If you can't find audio (wave), you can easily generate one. Here's how to generate it:

```
import numpy
def _get_mask(t, t1, t2, lvl_pos, lvl_neg): if t1 >= t2:
raise ValueError("t1 must be less than t2") return
numpy.where(numpy.logical_and(t > t1, t < t2), lvl_pos, lvl_neg)

def generate_signal(t):
sin1 = numpy.sin(2 * numpy.pi * 100 * t) sin2 = 2 * numpy.sin(2 * numpy.pi *
200 * t)

# add interval of high pitched signal sin2 = sin2 * _get_mask(t, 2, 5, 1.0, 0.0)

noise = 0.02 * numpy.random.randn(len(t)) final_signal = sin1 + sin2 + noise
return final_signal
if __name__ == '__main__':

step = 0.001
sampling_freq=1000
t = numpy.arange(0.0, 20.0, step) y = generate_signal(t)

# we can visualize this now
# in time
ax1 = plt.subplot(211)
plt.plot(t, y)
# and in frequency
plt.subplot(212)
plt.specgram(y, NFFT=1024, noverlap=900,
```

Fs=sampling_freq, cmap=plt.cm.gist_heat)
plt.show()

This will give you the following signal where the top subplot represents the signal we generated. Here, the x axis represents time and the y axis represents the signal's amplitude. The bottom subplot represents the same signal in the frequency domain. Here, while the x axis represents time as in the top subplot (we matched the time by selecting the sampling rate), the y axis represents the frequency of the signal.



Creating a stem plot

A two-dimensional stem plot displays data as lines extending from a baseline along the x axis. A circle (the default) or other marker, whose y axis represents the data value, terminates each stem.

In this recipe we will be discussing how to create a stem plot.
Do not confuse stem with stem and leaf plots, which is a method of representing data by separating the last important digit of values as leaves and higher order values as stems.

```
steam | leaf
==========================
    0 | 6 7 8
    1 | 0 2 3 4 7 7 7 8 9
    2 | 1 3 4 4 5 7
    3 | 3 1 1 2 6 6 9
    4 | 1 5 5 6 9
    5 | 0
```

Getting ready

For this kind of plot we want to use a sequence of discrete data, where ordinary line plots will not make sense anyway.
Plot discrete sequences as stems, where data values are represented as markers at the end of each stem. Stems extend from the baseline (usually at y = 0) to the data point value.

How to do it...

We will use matplotlib to plot stem plots using the stem() function. This function can use just a series of y values when x values are generated as a simple sequence from 0 to len(y) - 1. If we provide the stem() function with both x and y sequences, they will be used for both axes.

What we want to configure with the stem plot is several formatters:

f linefmt: This is the line formatter for the stem line
f markerfmt: Stems at the end of the line are formatted using this argument f basefmt: This formats the look of the base line
f label: This defines the label for legend for the stem plot
f hold: This holds the current graphs on the current axes

f bottom: This sets up the location of baseline position on the y axis, and the default value is 0
The argument hold is used as a usual feature for plots. If it is on (True), all the following plots are added to the current axes. Otherwise, each plot will create new figures and axes.

To create a stem plot, perform the following steps:

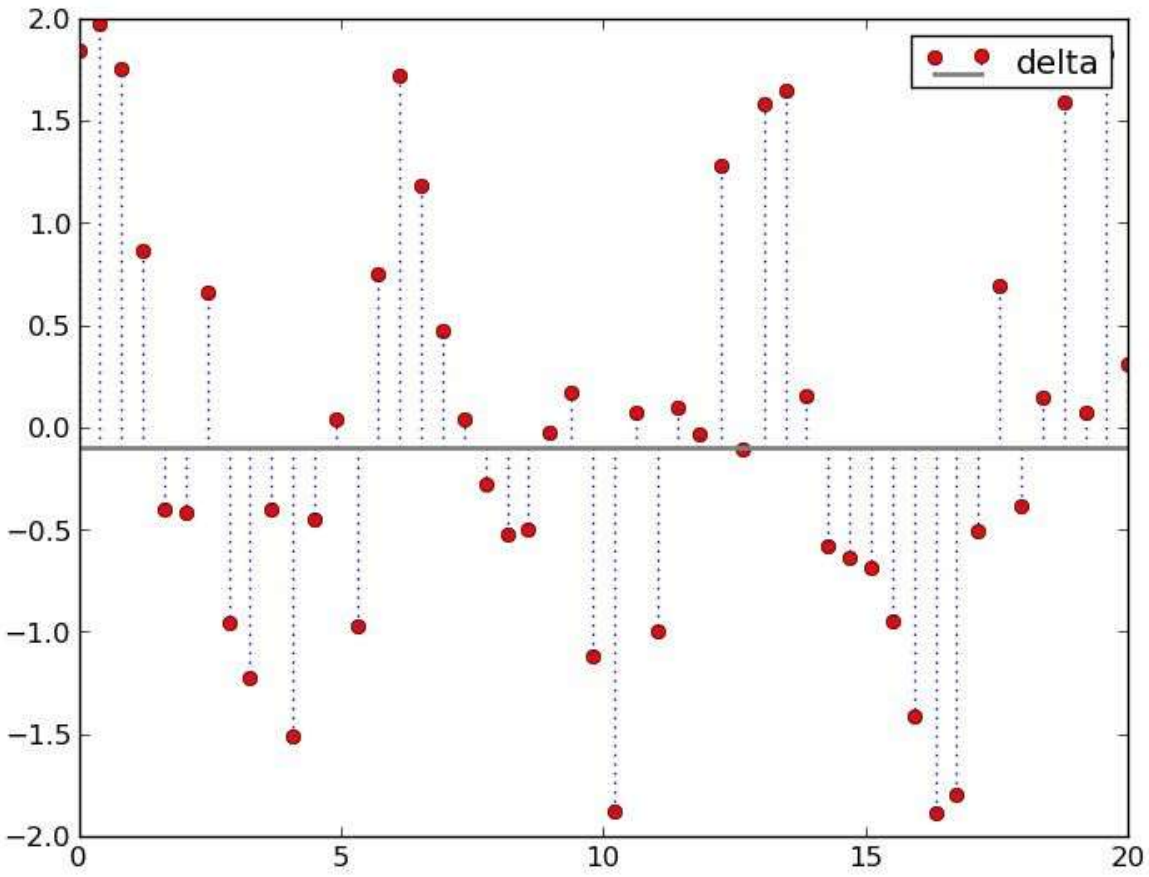
1. Generate random noise data.

2. Configure stem options.
3. Plot the stem.

The following is the code to do it:

```
import matplotlib.pyplot as plt
import numpy as np
# time domain in which we sample x = np.linspace(0, 20, 50)
# random function to simulate sampled signal y = np.sin(x + 1) + np.cos(x ** 2)
# here we can setup baseline position bottom = -0.1

# True -- hold current axes for further plotting # False -- opposite. clear and use
new figure/plot hold = False

# set label for legend. label = "delta"
markerline, stemlines, baseline = plt.stem(x, y, bottom=bottom, label=label,
hold=hold)

# we use setp() here to setup
# multiple properties of lines generated by stem() plt.setp(markerline,
color='red', marker='o')
plt.setp(stemlines, color='blue', linestyle=':')
plt.setp(baseline, color='grey', linewidth=2, linestyle='-')

# draw a legend plt.legend()
plt.show()
```

The previous code produces the following plot:



How it works...

First, we need some data. For this recipe, the generated sampled pseudo-signal will suffice. In the real world, any discrete sequential data can be properly visualized using the stem plot. We generate this signal using Numpy's numpy.linspace, numpy.cos, and numpy.sin functions.

We then set up a label for the stem plot and the position of the baseline, which defaults to 0.0.
If we want to draw multiple stem plots, we will set hold toTrue and the resulting plot calls will be rendered over the same set of axes.

A call to matplotlib.stem returns three objects. First is markerline, an instance of Line2D. This holds the reference to a line representing stems themselves, rendering only markers and not the line connecting the markers. This line can be made visible by editing the property of that Line2D instance; the process for this will be explained soon. The final one is also a Line2D instance, baseline, holding

a reference to a horizontal line that represents the source of stemlines. The second object returned is stemlines, which is a collection (Python list at the moment) of Line2D instances representing stem lines, of course. We use these returned objects to manipulate the visual appeal of a stem plot using the setp function to apply properties to all the lines (the Line2D instances) in those objects, or a collections of objects.

Experiment with the desired settings until you understand how setp changes your plot's style.

Drawing streamlines of vector flow

Stream plots are used to visualize a flow in vector fields. Examples from Science and Nature include fields of magnetic and gravitational forces and movement of liquid materials.

A vector field can be visualized in such a way where we assign a line and one or more arrows to every point. The intensity can be represented by the line length, and the direction by an arrow pointing in a particular direction.

Usually, the intensity of the force is visualized with the length of a particular streamline, but the density can also be used for the same purpose.

Getting ready

To visualize vector fields, we will use matplotlib's matplotlib.pyplot.streamplot function. This function creates plots from streamlines of a flow, uniformly filling the domain. The velocities field is interpolated and streamlines are integrated. The original source for this function is to visualize wind patterns or liquid flow; hence, we don't need strict vector lines but a uniform representation of the vector field.

The most important arguments for this function are (X, Y), which are the evenly spaced grids of one-dimensional NumPy arrays, and (U, V), which match two-dimensional NumPy arrays of (X, Y) velocities. Matrices U and V must be of such dimensions that the number of rows must be of equal length as Y, and the number of columns must match the length of X.

The line width of the stream plot can be controlled per line, if the linewidth argument is given a two-dimensional array matching the shape of u and v

velocities, or it simply can be just one integer value that all the lines will accept.

Color, also, can be just one value for all streamlines and a matrix shaped like the linewidth argument.

Arrows (the class FancyArrowPatch) are used to indicate the vector direction and we can control them using two parameters: arrowsize to change the size of the arrow, and arrowstyle to change the format of the arrow (for example, "simple", "->"...).

How to do it...

We will start with a simple example, just to get a sense of what's going on here. Perform the following steps:

1. Create data vectors.
2. Print intermediate values.
3. Plot the stream plot.
4. Show the figure with streamlines visualizing our vectors.

The following is the code sample:

```
import matplotlib.pyplot as plt
import numpy as np
Y, X = np.mgrid[0:5:100j, 0:5:100j]
U = X
V = Y

from pprint import pprint
print "X"
pprint(X)

print "Y"
pprint(Y)
plt.streamplot(X, Y, U, V)
```

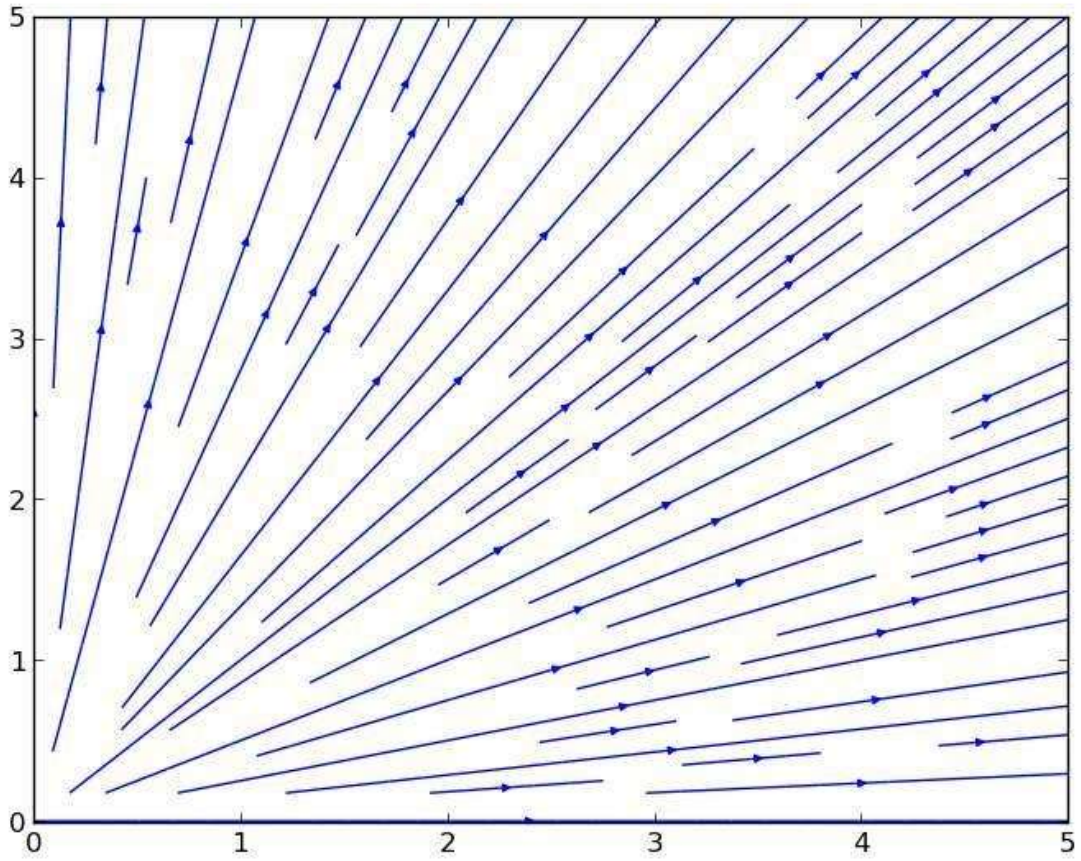
plt.show() The previous code will give the following textual output:

X
array([[0. , 0.05050505, 0.1010101 , ..., 4.8989899 ,

4.94949495, 5.],

[0. , 0.05050505, 0.1010101 , ..., 4.8989899 ,
4.94949495, 5.],
[0. , 0.05050505, 0.1010101 , ..., 4.8989899 ,
4.94949495, 5.],
...,
[0. , 0.05050505, 0.1010101 , ..., 4.8989899 ,
4.94949495, 5.],
[0. , 0.05050505, 0.1010101 , ..., 4.8989899 ,
4.94949495, 5.],
[0. , 0.05050505, 0.1010101 , ..., 4.8989899 ,
4.94949495, 5.]])
Y
array([[0. , 0. , 0. , ..., 0. ,
0. , 0.],
[0.05050505, 0.05050505, 0.05050505, ..., 0.05050505,
0.05050505, 0.05050505],
[0.1010101 , 0.1010101 , 0.1010101 , ..., 0.1010101 ,
0.1010101 , 0.1010101],
...,
[4.8989899 , 4.8989899 , 4.8989899 , ..., 4.8989899 ,
4.8989899 , 4.8989899],
[4.94949495, 4.94949495, 4.94949495, ..., 4.94949495,
4.94949495, 4.94949495],
[5. , 5. , 5. , ..., 5. , 5. , 5.]])

And it also generates the following streamline flow figure:



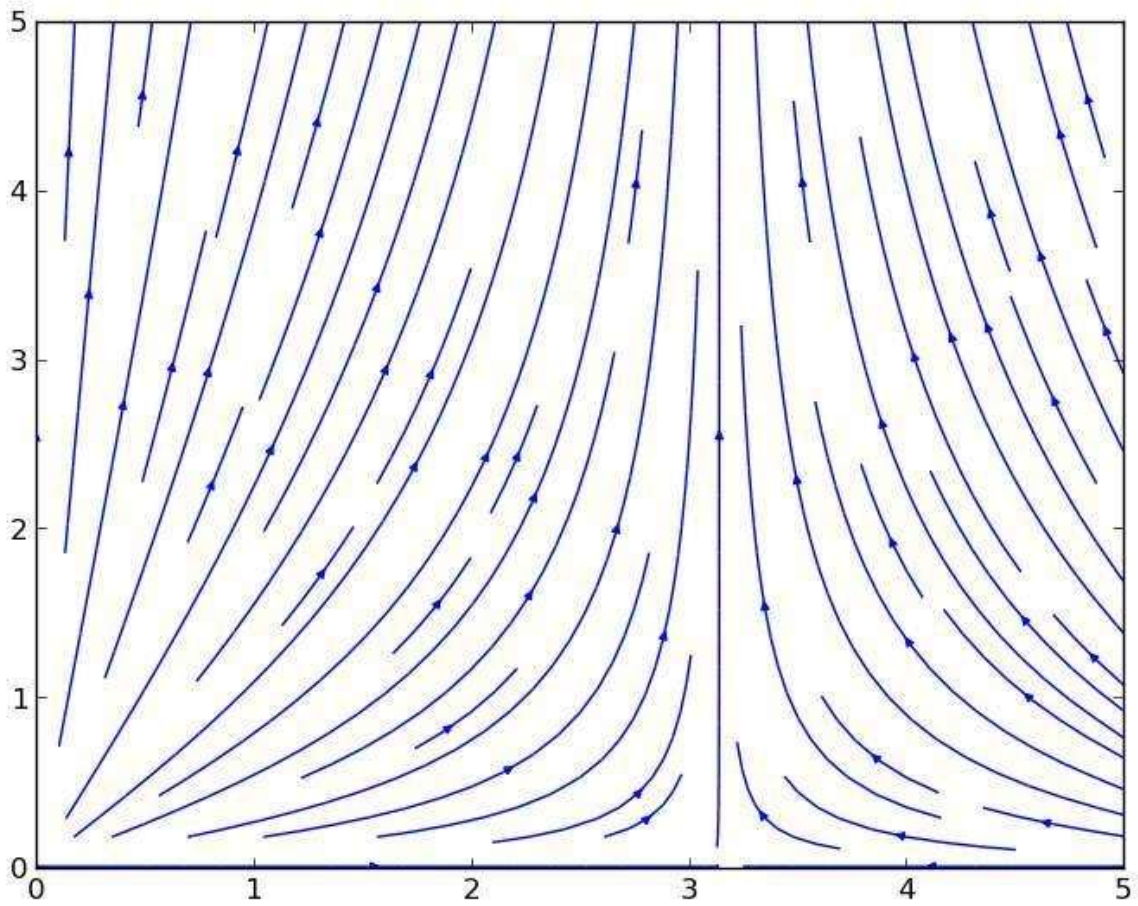
How it works...

We create a vector field of X and Y by indexing the two-dimensional mesh grid using NumPy's mgrid instance. We specify a range of the grid as start and stop (-2 and 2, respectively). The third index represents a step length. The step length represents the number of points to include between start and stop. If we want to include the stop value, we use a complex number for the step length, where the magnitude is used for a number of points required between start and stop, stop being inclusive.

The mesh grid, fleshed out like this, is then used to compute vector velocities. Here, for the sake of an example, we just use the same meshgrid property as vector velocities. This generates a plot that clearly shows plain linear dependency and the flow of the represented vector field.

Play with the values of U and V to get a sense of how the values of U and V influence the stream plot. For example, make U = np.sin(X) or V = np.sin(Y).

Following that, try to change the start and stop values. Check out the following plot for U = np.sin(X):



Bare in mind that the plot is a generated set of lines and arrow patches; hence, there is no way (currently, at least) to update the existing plot because lines and arrows know nothing about vectors and fields. Future implementations might include that, but at the moment this is a known limitation in the current version of matplotlib.

There's more...

Of course, this example gives just an opportunity to get to know and understand matplotlib's stream plot features and capabilities.

The real power comes when you have the real data at hand to play with. And after understanding this recipe, you will be able to recognize what are the tools you have, so that when you are given the data and you know its domain, you will

be able to pick the best tool for the job.

Using colormaps

Color coding the data can have a great impact on how your visualizations are perceived by the viewer, as they come with assumptions about color and what that color represents. Being explicit, if the color is used to add additional information to the data, it is always good. To know when and how to use color in your visualizations is even better.

Getting ready

If your data is not naturally color coded (such as earth/terrain altitudes or an object's temperature), it's better not to make any artificial mappings to natural coloring. We want to understand the data appropriately and so make a choice of color to help the reader decode data easily. We don't want readers constantly trying to suppress learned mapping of color for temperatures if we are representing financial data that has no connection with Kelvins or Celsius.

If possible, avoid usual red/green associations if there is no strong correlation in the data to associate them with those colors.

To help you pick the right color mapping, we will explain some colormaps available in the matplotlib package that can save a lot of time and help you, if you know what they are used for and how to find them.

Colormaps in general can be categorized as follows:

f Sequential: This represents monochromatic colormaps of two color tones from low to high saturation of the same color, for example, from white to bright blue. This is ideal for most cases, as they clearly show the change from low to high values.

f Diverging: This represents the central point and is the median value (some light color usually), but then ranges go to two different color tones in direction for high and low values. This can be ideal for data with a significant median value; for example, when the median is at 0, it clearly shows the difference between negative and positive values.

f Qualitative: For cases where data has no inherent ordering and all you want is

to make sure different categories are easily discernible from each other, this is the colormap to choose.

f Cyclic: This is handy to use where data can wrap around endpoint values, for example, representing time of the day, wind direction, or phase angle.

matplotlib comes with a lot of predefined maps, and we can divide them into several categories. We will suggest when to use some of those colormaps. The most common and base colormaps are autumn, bone, cool, copper, flag, gray, hot, hsv, jet, pink, prism, sprint, summer, winter, and spectral.

We have another set of colormaps coming from the Yorick scientific visualization package. This is an evolution from the GIST package, so all colormaps in this collection have gist_ as the prefix in their names.

The Yorick scientific visualization package is also an interpreted language written in C, not quite active lately. You can find more information on its official website at http://yorick.sourceforge.net/index.php.

These colormap sets contain the following maps: gist_earth, gist_heat, gist_ncar, gist_rainbow, and gist_stern.
Then we have the colormaps based on ColorBrewer (http://colorbrewer.org), where we can categorize them into the following:
f Diverging: This is where luminance is highest at the midpoint and decreases towards different endpoints
f Sequential: This is where luminance decreases monotonically
f Qualitative: This is where different sets of colors are used to differentiate data categories
Also, there are some miscellaneous colormaps that are available:

Colormap brg
bwr
coolwarm rainbow

seismic terrain

Description
This will represent a diverging blue-red-green colormap.
This will represent a diverging blue-white-red colormap.
This is useful for 3D shading, color blindness, and ordering of colors.

This represents a spectral purple-blue-green-yellow-orange-red colormap with diverging luminance.
This represents a diverging blue-white-red colormap.

This represents Mapmaker's colors—blue, green, yellow, brown, and white—originally from the IGOR Pro software.
Most of the maps presented here can be reversed by putting the _r postfix after the name of the colormap, for example, hot_r is an inversed cycle colormap of hot.

How to do it...

We can set colormaps on many items in matplotlib. For example, a colormap can be set on image, pcolor, and scatter. This is usually accomplished via an argument to a function call, cmap. This argument accepts an instance of colors.Colormap.

We can also use matplotlib.pyplot.set_cmap to set cmap for the latest object plotted on the axes.
You can get all the available colormaps easily with matplotlib.pyplot.colormaps.
Fire up IPython and type in the following:
In [1]: import matplotlib.pyplot as plt

In [2]: plt.colormaps() Out[2]:
['Accent',

'Accent_r',
'Blues',
'Blues_r',

...
'winter',
'winter_r']

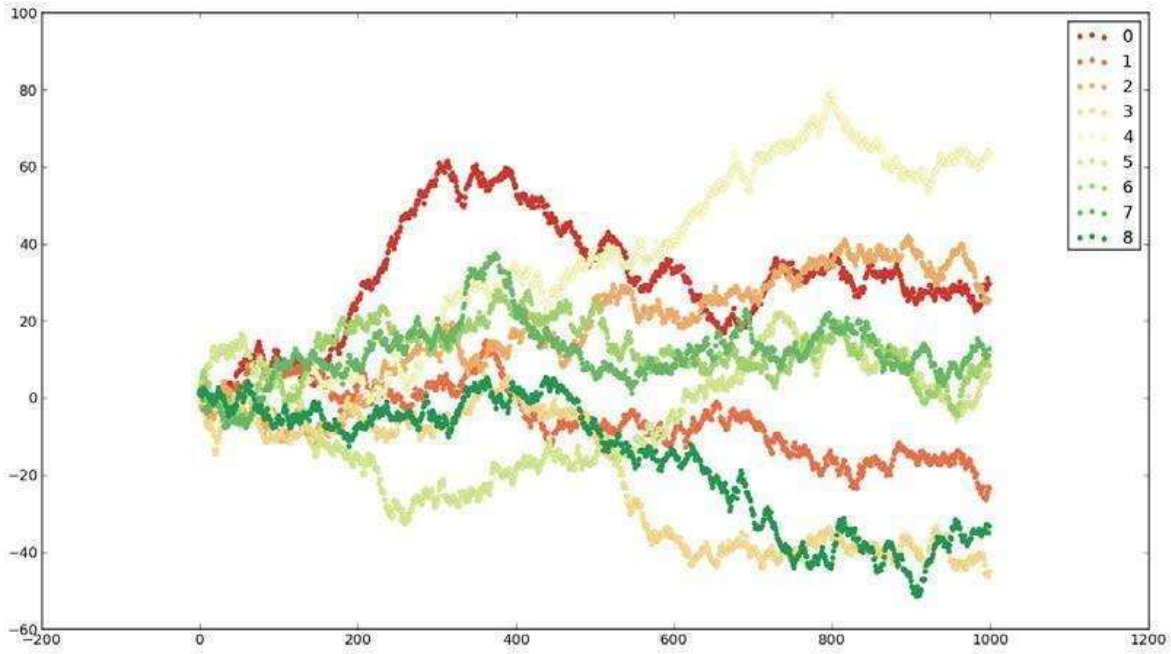
Note that we have shortened the preceding list because it contains around 140 items and will span across several pages here.
This will import the pyplot function interface and allow us to call the colormaps function, which returns a list of all the registered colormaps.
Finally, we want to show you how to make a nice looking colormap. In the following example we need to:

1. Navigate to the ColorBrewer website to get divergent colormap color values in the Hex format.
2. Generate random samples of x and y, where y is the cumulative sum of values (simulate stock price variations).

3. Apply customization to the scatter plot function of matplotlib.
4. Tweak the scatter marker line color and width to make the plot more readable and pleasant for the viewer.

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

# Red Yellow Green divergent colormap
red_yellow_green = ['#d73027', '#f46d43', '#fdae61', '#fee08b', '#ffffbf',
'#d9ef8b',
'#a6d96a', '#66bd63', '#1a9850']

sample_size = 1000
fig, ax = plt.subplots(1)

for i in range(9):
y = np.random.normal(size=sample_size).cumsum() x = np.arange(sample_size)
ax.scatter(x, y, label=str(i), linewidth=0.1,

edgecolors='grey',
facecolor=red_yellow_green[i])
ax.legend() plt.show()
```

The previous code will render a nice looking figure:



How it works...

We used the ColorBrewer website to find out colors in a red-yellow-green diverging colormap. Then, we listed those colors in our code and applied them to our scatter plot.

ColorBrewer is a web tool built by *Cynthia Brewer*, *Mark Harrower*, and *The Pennsylvania State University* to explore colormaps. It is a very handy tool to pick up colormaps of different ranges and see them applied on a map using slight variations, so that you immediately sense what they will look like on a chart. This particular map is at http://colorbrewer2.org/index.ph p?type=diverging&scheme=RdYlGn&n=9.

Sometimes, we will have to make our customization on matplotlib.rcParams, which is the first thing we want to do before we create a figure or any of the axes.
For example, matplotlib.rcParams['axes.cycle_color'] is the configuration setting we want to change in order to set up the default colormap for most of the matplotlib functions.

There's more...

Using matplotlib.pyplot.register_cmap, we can register a new colormap to matplotlib, so it can be found using the get_cmap function. We can use it in two different ways. Here are both the signatures:

f register_cmap(name='swirly', cmap=swirly_cmap)
f register_cmap(name='choppy', data=choppydata, lut=128)

The first signature allows us to specify a colormap as an instance of colors.Colormap and register it using the name argument. The argument name can be omitted in which case it will be inherited from the name attribute of the cmap instance provided.

For the latter one, we are passing three arguments to the linear-segmented colormap constructor and registering that colormap afterwards.
Using maplotlib.pyplot.get_cmap we can get the colors.Colormap instance using the name argument.
The following is how to make your own map using matplotlib.colors.LinearSegmentedColormap:

```
from pylab import *
cdict = {'red': ((0.0, 0.0, 0.0),
(0.5, 1.0, 0.7),
(1.0, 1.0, 1.0)),
'green': ((0.0, 0.0, 0.0),
(0.5, 1.0, 0.0),
(1.0, 1.0, 1.0)),
'blue': ((0.0, 0.0, 0.0),
(0.5, 1.0, 0.0),
(1.0, 0.5, 1.0))}
my_cmap = matplotlib.colors.LinearSegmentedColormap('my_
colormap',cdict,256)
pcolor(rand(10,10),cmap=my_cmap)
colorbar()
```

Executing this method is the simplest part, while the hardest part is to actually come up with a combination of colors that are informative, that do not take any information away from the data we want to visualize, and that are also pleasant to the eyes of the viewer.

For the base map list (the colormaps listed in the table earlier), we can use the pylab shortcut to set the colormap. For example:

```
imshow(X)
hot()
```

This would set the colormap of the image X to cmap='hot'.

Using scatter plots and histograms

Scatter plots are very often encountered, as they are the most common plot to visualize the relation between two variables. If we want to have a quick look at the data of these two variables and see if there is any relation between them (that is, correlation), we will draw a quick scatter plot. For a scatter plot to exist, we must have one variable that can be systematically changed by, for example, experimenter, so we can inspect the possibilities of influencing another variable.

That's why, in this recipe, we will learn how to understand the scatter plots.

Getting ready

We want to see, for example, how two events are affected by each other or if they are affected at all. This visualization is especially useful on large sets of data, where we cannot make any conclusions by looking at the data in the native form, when it is just numbers.

Correlation between values, if there is any, can be positive and negative. Positive correlation is for increasing X values, we have Y values increasing too. In negative correlation for increasing X values, Y values are decreasing. In an ideal case, positive correlation is a line starting from the bottom-left corner of the axes to the top-right corner. An ideal negative correlation is a line starting from the top-left corner to the bottom-right corner of the axes.
An ideal positive correlation between two data points is given the value of 1, and an ideal negative is given the value of -1. Everything inside this interval represents a weaker correlation between the two values. Usually, everything inside -0.5 to 0.5 is not considered valuable from the perspective of two variables being in real connection.

An example of positive correlation would be the amount of money put in a charity jar being directly positively correlated to the number of people seeing the jar. Negative correlation is between the time required to reach place B from place A, depending on the distance between the locations A and B. The bigger the distance, more will be the time we need to complete the travel.

The example we have presented here is of a positive correlation but this is not perfect, as different people might put different amounts of money per visit. But

in general, we can assume that the more the number of people who see the jar, more will be the money left inside.

Keep in mind, though, that even if the scatter plot displays a correlation between two variables, that correlation might not be a direct one. There might be a third variable that influences both the plotted variables, so the correlation is just a case that plotted values are correlated with that third variable. In the end, the correlation might be just apparent and no real relation exists behind it.

How to do it...

With the following code sample, we will demonstrate how the scatter plot can explain the relation between variables.
The data we use is obtained using the Google Trends web portal, where one can download the CSV file containing normalized values of relative search volumes for the given parameters. We will store our data in the ch07_search_data.py Python module, so we can import it in subsequent code recipes. The following is the content of it:

```
# ch07_search_data
# daily search trend for keyword 'flowers' for a year

DATA = [
1.04, 1.04, 1.16, 1.22, 1.46, 2.34, 1.16, 1.12, 1.24, 1.30, 1.44,
1.22, 1.26,
1.34, 1.26, 1.40, 1.52, 2.56, 1.36, 1.30, 1.20, 1.12, 1.12, 1.12,
1.06, 1.06,
1.00, 1.02, 1.04, 1.02, 1.06, 1.02, 1.04, 0.98, 0.98, 0.98, 1.00,
1.02, 1.02,
1.00, 1.02, 0.96, 0.94, 0.94, 0.94, 0.96, 0.86, 0.92, 0.98, 1.08,
1.04, 0.74,
0.98, 1.02, 1.02, 1.12, 1.34, 2.02, 1.68, 1.12, 1.38, 1.14, 1.16,
1.22, 1.10,
1.14, 1.16, 1.28, 1.44, 2.58, 1.30, 1.20, 1.16, 1.06, 1.06, 1.08,
1.00, 1.00,
0.92, 1.00, 1.02, 1.00, 1.06, 1.10, 1.14, 1.08, 1.00, 1.04, 1.10,
1.06, 1.06,
1.06, 1.02, 1.04, 0.96, 0.96, 0.96, 0.92, 0.84, 0.88, 0.90, 1.00,
1.08, 0.80,
0.90, 0.98, 1.00, 1.10, 1.24, 1.66, 1.94, 1.02, 1.06, 1.08, 1.10,
1.30, 1.10,
```

1.12, 1.20, 1.16, 1.26, 1.42, 2.18, 1.26, 1.06, 1.00, 1.04, 1.00,
0.98, 0.94,
0.88, 0.98, 0.96, 0.92, 0.94, 0.96, 0.96, 0.94, 0.90, 0.92, 0.96,
0.96, 0.96,
0.98, 0.90, 0.90, 0.88, 0.88, 0.88, 0.90, 0.78, 0.84, 0.86, 0.92,
1.00, 0.68,
0.82, 0.90, 0.88, 0.98, 1.08, 1.36, 2.04, 0.98, 0.96, 1.02, 1.20,
0.98, 1.00,
1.08, 0.98, 1.02, 1.14, 1.28, 2.04, 1.16, 1.04, 0.96, 0.98, 0.92,
0.86, 0.88,
0.82, 0.92, 0.90, 0.86, 0.84, 0.86, 0.90, 0.84, 0.82, 0.82, 0.86,
0.86, 0.84,
0.84, 0.82, 0.80, 0.78, 0.78, 0.76, 0.74, 0.68, 0.74, 0.80, 0.80,
0.90, 0.60,
0.72, 0.80, 0.82, 0.86, 0.94, 1.24, 1.92, 0.92, 1.12, 0.90, 0.90,
0.94, 0.90,
0.90, 0.94, 0.98, 1.08, 1.24, 2.04, 1.04, 0.94, 0.86, 0.86, 0.86,
0.82, 0.84,
0.76, 0.80, 0.80, 0.80, 0.78, 0.80, 0.82, 0.76, 0.76, 0.76, 0.76,
0.78, 0.78,
0.76, 0.76, 0.72, 0.74, 0.70, 0.68, 0.72, 0.70, 0.64, 0.70, 0.72,
0.74, 0.64,
0.62, 0.74, 0.80, 0.82, 0.88, 1.02, 1.66, 0.94, 0.94, 0.96, 1.00,
1.16, 1.02,
1.04, 1.06, 1.02, 1.10, 1.22, 1.94, 1.18, 1.12, 1.06, 1.06, 1.04,
1.02, 0.94,
0.94, 0.98, 0.96, 0.96, 0.98, 1.00, 0.96, 0.92, 0.90, 0.86, 0.82,
0.90, 0.84,
0.84, 0.82, 0.80, 0.80, 0.76, 0.80, 0.82, 0.80, 0.72, 0.72, 0.76,
0.80, 0.76,
0.70, 0.74, 0.82, 0.84, 0.88, 0.98, 1.44, 0.96, 0.88, 0.92, 1.08,
0.90, 0.92,
0.96, 0.94, 1.04, 1.08, 1.14, 1.66, 1.08, 0.96, 0.90, 0.86, 0.84,
0.86, 0.82,
0.84, 0.82, 0.84, 0.84, 0.84, 0.84, 0.82, 0.86, 0.82, 0.82, 0.86,
0.90, 0.84,
0.82, 0.78, 0.80, 0.78, 0.74, 0.78, 0.76, 0.76, 0.70, 0.72, 0.76,
0.72, 0.70,
0.64]

We need to perform the following steps: 1. Use a clean dataset of the Google Trend search volume for one year for the keyword flowers; we will import this dataset into the variable d.
2. Use a random normal distribution of the same length (365 data points) as our Google Trend dataset; this will be the dataset d1.

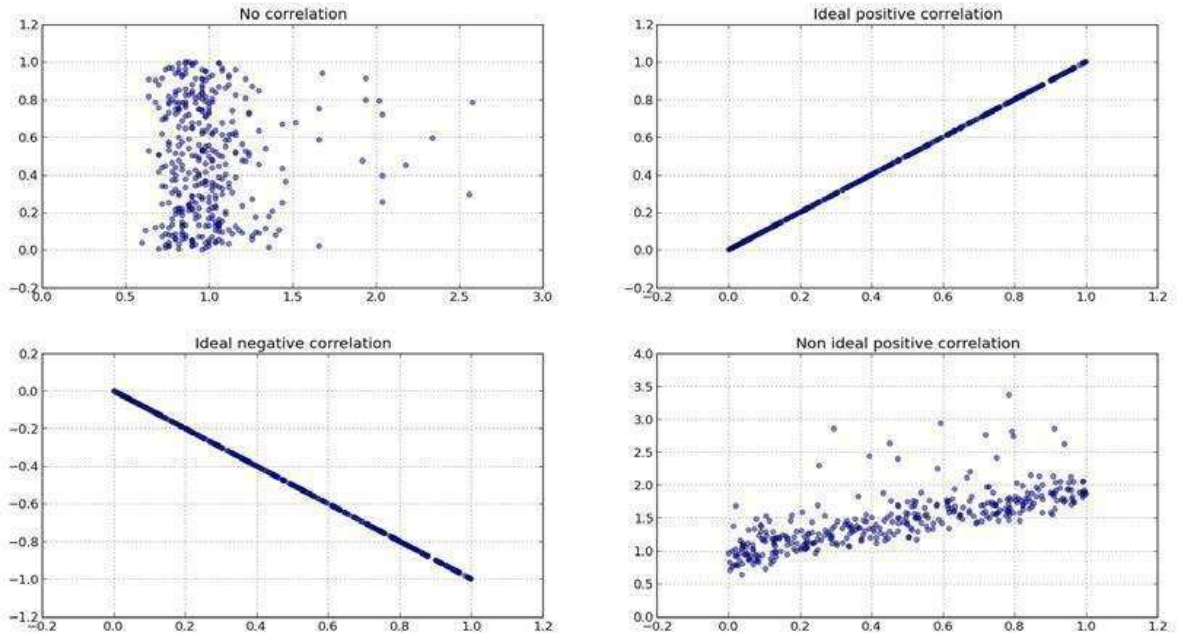
3. Create a plot containing four subplots.
4. In the first subplot, plot a scatter plot of d and d1.
5. In the second subplot, plot a scatter plot of d1 with d1.
6. In the third subplot, render a scatter plot of d1 with inverted d1.

7. And in the fourth subplot, render a scatter plot of d1 with a similar dataset constructed with a combination of d1 and d.
The following code will illustrate the relation as explained earlier in this recipe:

```
import matplotlib.pyplot as plt
import numpy as np
# import the data
from ch07_search_data import DATA
d = DATA

# Now let's generate random data for the same period d1 =
np.random.random(365)
assert len(d) == len(d1)

fig = plt.figure()

ax1 = fig.add_subplot(221) ax1.scatter(d, d1, alpha=0.5) ax1.set_title('No
correlation') ax1.grid(True)

ax2 = fig.add_subplot(222)
ax2.scatter(d1, d1, alpha=0.5)
ax2.set_title('Ideal positive correlation')

ax2.grid(True)

ax3 = fig.add_subplot(223)
ax3.scatter(d1, d1*-1, alpha=0.5) ax3.set_title('Ideal negative correlation')
ax3.grid(True)

ax4 = fig.add_subplot(224)
```

```
ax4.scatter(d1, d1+d, alpha=0.5)
ax4.set_title('Non ideal positive correlation') ax4.grid(True)

plt.tight_layout()
plt.show()
```

The following is the output we should get when the preceding code is executed:



How it works...

The sample we see in the preceding output clearly displays if there is any correlation between different datasets. While the second (top-right) subplot shows an ideal, or perfect, positive correlation of the dataset d1 with d1 itself (obviously), we can see that the fourth subplot (bottom right) hints that there is a positive correlation, although not ideal. We constructed this dataset from d1 and d (random) to simulate two similar signals (events), while the second subplot plotted using d and d1 has certain randomness (or noise) in it, but still can be compared with the original (d) signal.

There's more...

We can also add histograms to scatter plots in such a way that they can tell us more about the data plotted. We can add horizontal and vertical histograms to show frequencies of data points on the x and y axes. Using this, we can at the same time see the summary of the whole dataset (histogram) and individual data

points (the scatter plot).

The following is an example of the code to generate the scatter-histogram combination using the same two datasets we introduced in this recipe. The meat of the code is the function scatterhist() that is given here for reuse with different datasets, trying to set some of the variables based on the dataset provided (the number of bins in histogram, limits for axes, and so on).

We start with the usual imports:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable
```

The following code is the definition of our function to generate scatter histograms given an (x, y) dataset and optionally, a figsize parameter:

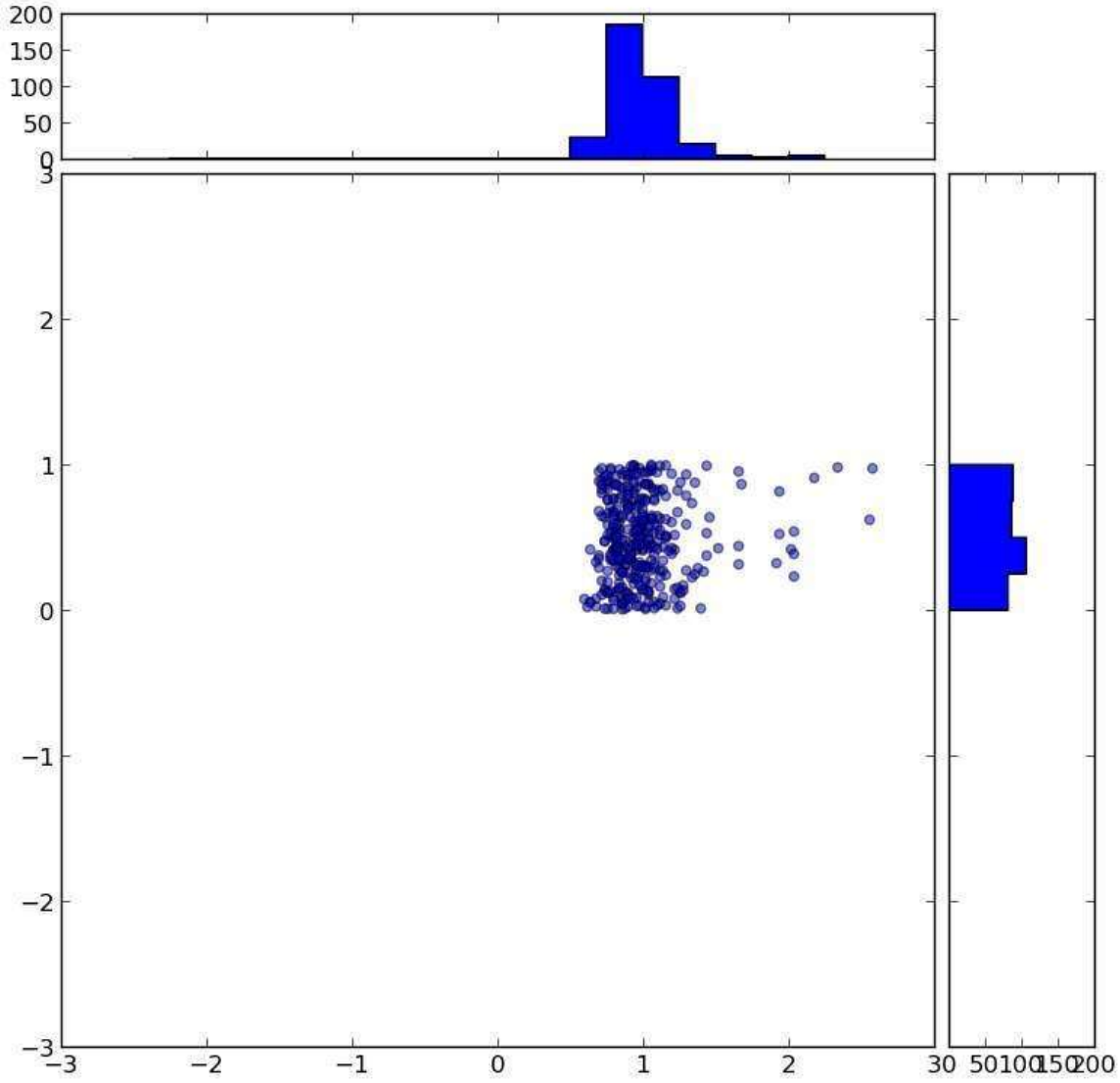
```
def scatterhist(x, y, figsize=(8,8)):
"""
Create simple scatter & histograms of data x, y inside given plot

@param figsize: Figure size to create figure
@type figsize: Tuple of two floats representing size in inches
@param x: X axis data set
@type x: np.array

@param y: Y axis data set
@type y: np.array
"""

_, scatter_axes = plt.subplots(figsize=figsize)

# the scatter plot:
scatter_axes.scatter(x, y, alpha=0.5) scatter_axes.set_aspect(1.)

divider = make_axes_locatable(scatter_axes)
axes_hist_x = divider.append_axes(position="top", sharex=scatter_ axes, size=1,
pad=0.1)
axes_hist_y = divider.append_axes(position="right", harey=scatter_axes, size=1,
pad=0.1) # compute bins accordingly binwidth = 0.25
```

```
# global max value in both data sets
xymax = np.max([np.max(np.fabs(x)), np.max(np.fabs(y))]) # number of bins
bincap = int(xymax / binwidth) * binwidth

bins = np.arange(-bincap, bincap, binwidth) nx, binsx, _ = axes_hist_x.hist(x,
bins=bins, histtype='stepfilled',
orientation='vertical') ny, binsy, _ = axes_hist_y.hist(y, bins=bins,
histtype='stepfilled',
orientation='horizontal')

tickstep = 50
ticksmax = np.max([np.max(nx), np.max(ny)]) xyticks = np.arange(0, ticksmax
+ tickstep, tickstep)

# hide x and y ticklabels on histograms for tl in axes_hist_x.get_xticklabels():
tl.set_visible(False)
axes_hist_x.set_yticks(xyticks)

for tl in axes_hist_y.get_yticklabels(): tl.set_visible(False)
axes_hist_y.set_xticks(xyticks)

plt.show()
```

Now we proceed with loading of the data and function call to generate and render the desired chart.

```
if __name__ == '__main__': # import the data
from ch07_search_data import DATA as d

# Now let's generate random data for the same period d1 =
np.random.random(365)
assert len(d) == len(d1)

# try with the random data # d = np.random.randn(1000) # d1 =
np.random.randn(1000)

scatterhist(d, d1)
```

The previous code should generate the following output:



Plotting the cross-correlation between two variables

If we have two different datasets from two different observations, we want to know if those two event sets are correlated. We want to cross correlate them and see if they match in any way. We are looking for a pattern of a smaller data sample in a larger data sample. The pattern does not have to be an obvious or trivial pattern.

Getting ready

We can use matplotlib's matplotlib.pyplot.xcorr function from the pyplot lab. This function can plot the correlation between two datasets in such a way that we can see if there is any significant pattern between the plotted values. It is assumed that x and y are of the same length.

If we pass the argument normed as True, we can normalize by cross-correlation at 0th lag (that is, when there is no time delay or time lag).
Behind the scenes, correlation is done using NumPy's numpy.correlate function.

Using the argument usevlines (setting it to True), we can instruct matplotlib to use vlines() instead of plot() to draw the lines of the correlation plot. The main difference is if we are using plot(), we can style the lines using the standard Line2D properties passed in the **kwargs argument to the matplotlib.pyplot.xcorr function.

How to do it...

In this following example we need to perform the following steps:

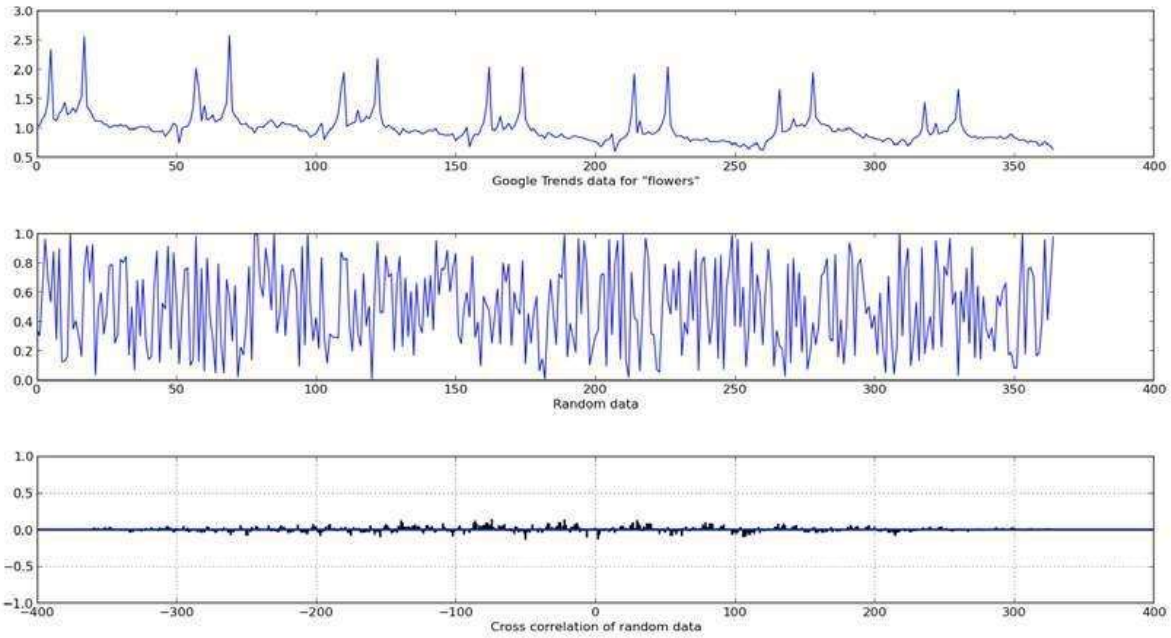
1. Import the matplotlib.pyplot module.
2. Import the numpy package.
3. Use a clean dataset of the Google search volume trend for a year for the keyword flowers.
4. Plot the datasets (the real and artificial ones) and the cross-correlation diagram.
5. Tighten the layout in order to have a better overview of labels and ticks.
6. Add appropriate labels and grids for easier understanding of the plot.

The following is the code that will perform the previously mentioned steps:

```
import matplotlib.pyplot as plt
import numpy as np
# import the data
from ch07_search_data import DATA as d

total = sum(d)
av = total / len(d)
z = [i - av for i in d]

# Now let's generate random data for the same period d1 =
np.random.random(365)
assert len(d) == len(d1)
```

```
total1 = sum(d1)
av1 = total1 / len(d1) z1 = [i - av1 for i in d1]

fig = plt.figure()

# Search trend volume
ax1 = fig.add_subplot(311)
ax1.plot(d)
ax1.set_xlabel('Google Trends data for "flowers"')

# Random: "search trend volume" ax2 = fig.add_subplot(312) ax2.plot(d1)
ax2.set_xlabel('Random data')

# Is there a pattern in search trend for this keyword? ax3 = fig.add_subplot(313)
ax3.set_xlabel('Cross correlation of random data')
ax3.xcorr(z, z1, usevlines=True, maxlags=None, normed=True, lw=2)
ax3.grid(True)
plt.ylim(-1, 1)

plt.tight_layout()
plt.show()
```

The previous code will render the following output:



How it works...

We used a real dataset with a recognizable pattern in it (two peaks repeating in a similar manner across the dataset; refer to the preceding plot). The other dataset is just some random normal-distributed data of the same length as the real accrued data from the public service, Google Trends.

We plotted both datasets over the top half of the output to visualize the data. Using matplotlib's xcorr, which in turn uses NumPy's correlate() function, we computed cross-correlation and plotted it on the bottom half of the screen.

Cross-correlation computation in NumPy returns a correlation coefficient's array that represents a degree of similarity of two datasets (or signals, as usually referred to if used in the signal processing field).

The cross-correlation diagram, correlogram, tells us that these two signals are not correlated, which is represented by the height of the correlation values (vertical lines that appear at certain time lags). We can see that there is more than one vertical line (the correlation coefficient at time lag *n*) that is above 0.5.

If, for example, two datasets would have the correlation at a time lag of 100 (that is, a 100-second shift between the same object observed by two different sensors), we will see a vertical line (representing the correlation coefficient) at x = 100 in the preceding output.

Importance of autocorrelation

Autocorrelation represents the degree of similarity between a given time series and a lagged (that is, delayed in time) version of itself over successive time intervals. It occurs in time series studies when the errors associated with a given time period carry over into future time periods. For example, if we are predicting the growth of stock dividends, an overestimate in one year is likely to lead to overestimates in the succeeding years.

The time series analysis data arises in lots of different scientific applications and financial processes. Some of the examples include: generated reports of financial performance, prices over time, computing volatility, and others.

If we are analyzing unknown data, autocorrelation can help us detect whether the data is random or not. For that we can use correlogram. It can help provide answers to questions such as: Is the data random? Is this time series data a white

noise signal? Is it sinusoidal? Is it autoregressive? What is the model of this time series data?

Getting ready

We will use matplotlib to compare two sets of data. One is the Google day trend of search volume for a certain keyword for one year (365 days). The other set is 365 random measurements (the generated random data) with normal distribution.

We will autocorrelate both datasets and compare how the correlograms visualize patterns in data.

How to do it...

In this section we will perform the following steps:

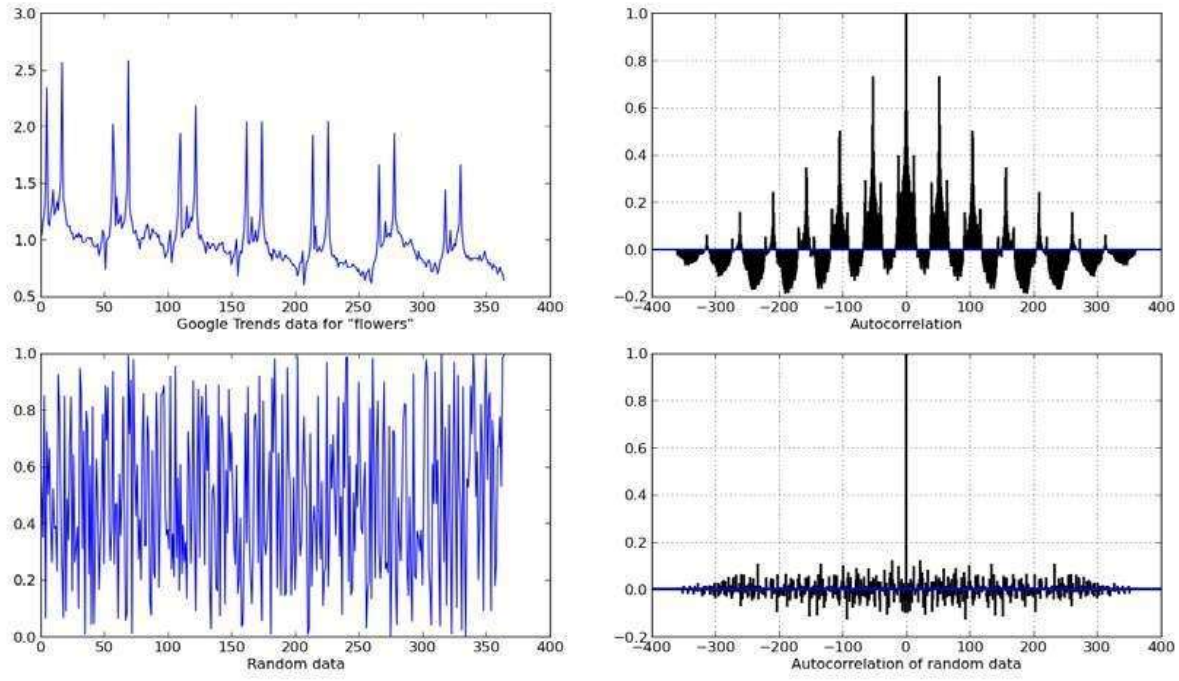
1. Import the matplotlib.pyplot module.
2. Import the numpy package.
3. Use a clean dataset of Google search volume for a year.
4. Plot the dataset and plot its autocorrelation diagram.
5. Generate a same-length random dataset using NumPy.
6. Plot the random dataset on the same figure, and plot its autocorrelation diagram.
7. Add appropriate labels and grids for easier understanding of the plot.

The following is the code:

```
import matplotlib.pyplot as plt
import numpy as np
# import the data
from ch07_search_data import DATA as d

total = sum(d)
av = total / len(d)
z = [i - av for i in d]

fig = plt.figure()
# plt.title('Comparing autocorrelations')

# Search trend volume
ax1 = fig.add_subplot(221)
ax1.plot(d)
ax1.set_xlabel('Google Trends data for "flowers"')
```

```
# Is there a pattern in search trend for this keyword? ax2 = fig.add_subplot(222)
ax2.acorr(z, usevlines=True, maxlags=None, normed=True, lw=2)
ax2.grid(True)
ax2.set_xlabel('Autocorrelation')

# Now let's generate random data for the same period d1 =
np.random.random(365)
assert len(d) == len(d1)

total = sum(d1)
av = total / len(d1) z = [i - av for i in d1]

# Random: "search trend volume" ax3 = fig.add_subplot(223) ax3.plot(d1)
ax3.set_xlabel('Random data')

# Is there a pattern in search trend for this keyword? ax4 = fig.add_subplot(224)
ax4.set_xlabel('Autocorrelation of random data')
ax4.acorr( z, usevlines=True, maxlags=None, normed=True, lw=2)
ax4.grid(True)

plt.show()
```

The previous code will render the following output:



How it works...

Looking at the left-hand side plots, it is easy to spot patterns in the search volume data, where the bottom-left plot has normally distributed random data with patterns that are not obvious, but still might exist.

Computing and plotting autocorrelation over random data, we can see that there is a high correlation at 0, which is expected, and data is correlated with itself without any time lag. But going before or after no time lag, the signal is almost 0. So we can safely conclude that there is no correlation between the signal in original time and any time lags examined.

Looking at the real data, Google search volume trend, we can see the same behavior at zero time lag, still something we can expect for any autocorrelated signal. But we have strong signals at around 30, 60, and 110 days after zero time lag. This indicates that there is a pattern with this particular search term and the way people search for it on the Google search engine.

We will leave the exercise of explaining why is this a very different story to the reader. Remember that correlation and causation are two very different things.

There's more...

Autocorrelation is used very often when we want to identify a model for unknown data. When we try to fit data into a model, how data correlates to itself is sometimes the first step to identifying the appropriate model for a dataset we are presented with. This requires more than Python; it requires knowledge of mathematical modeling and various statistical tests (LjungBox test, Box-Pierce test, and so on) that will help us answer any questions we may have.

8

More on matplotlib Gems

In this chapter we will cover:
f Drawing barbs

f Making a box and whisker plot
f Making Gantt charts
f Making errorbars
f Making use of text and font properties
f Rendering text with LaTeX
f Understanding the difference between pyplot and OO API

Introduction

In this chapter we will explore some less frequently used features of the matplotlib package. Some of these examples stretch the matplotlib original target, but they show what can be done with a little creativity, and prove that matplolib is full featured and generically oriented.

Drawing barbs

A barb is a representation of the speed and direction of wind, and is mainly deployed by meteorology scientists. In theory they can be used to visualize any type of two-dimensional vector quantities. They are similar to arrows (quivers), but the difference is that arrows represent vector magnitude by the length of the arrow, while barbs give more information about the vector's magnitude by employing lines or triangles as increments of magnitude. We will explain what barbs are, how to read them, and how to visualize them using Python and matplotlib. Here's a typical set of barbs:

0 5 10 15 30 40 50 60 100
In the preceding diagram, the triangle, also known as flag, represents the largest increment. A full line or barb, represents a smaller increment; a half line is the smallest increment.

The increments are in the order of 5, 10, and 65 for a half-line, line, and triangle respectively. The values here represent, for meteorologists at least, wind speed in nautical miles per hour (knots).

We ordered the barbs from left to right to represent the following magnitudes: 0, 5, 10, 15, 30, 40, 50, 60, and 100 knots. The direction here is the same for each barb and is from north to south, because the east-west speed component is 0 for each barb.

Getting ready

A barb can be created using a matplotlib function from matplotlib.pyplot.barbs.

The barbs function accepts various arguments, but the main use case is that we specify X and Y coordinates, representing locations of observed data points. The second pair of arguments—U, V—represents the magnitude of the vector in north-south and east-west directions in knots.

Other arguments that can be useful are pivots, sizes, and various coloring arguments.

A pivot argument (pivot) represents the part of the arrow represented on the grid point. We get a pivot argument when the arrow rotates around this point. The arrow can rotate around the tip or middle, which are valid values for the pivot argument.

Because barbs consist of several parts, we can set up the coloring of any of those parts. So we have a few color-related arguments that we can set up:

f barbcolor: This defines the color of all the parts for a barb, except for flags f flagcolor This defines the color of any flag on the barb

f facecolor: This argument is used if none of the preceding color arguments are specified (or the default value is read from rcParams)

If any of the preceding color-related arguments are specified, the argument facecolor is overridden. The argument facecolor is the one used in coloring polygons. The size argument (sizes) specifies the ration of a feature to the length of the barb. This is a collection of coefficients that can be specified by using any or all of the following keys:

f spacing: This defines the space among features of the flag/barb
f height: This defines the distance from the shaft to the top of a flag or barb f width: This defines the width of a flag
f emptybarb: This defines the circle radius used for low magnitudes

How to do it...

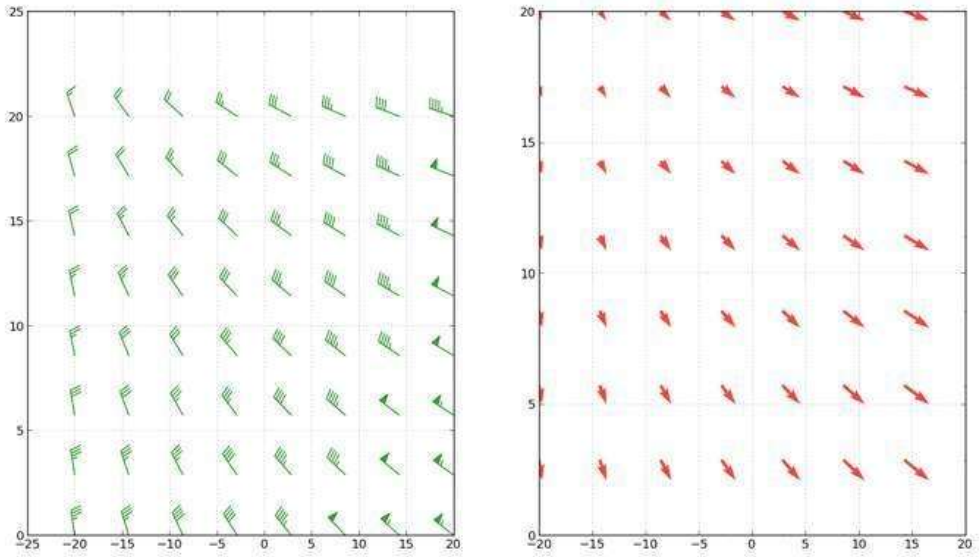
Let's demonstrate how to use a barb function by performing the following steps:

1. Generate a grid of coordinates to simulate observations.
2. Simulate observational values for wind speed.
3. Plot barb diagrams.
4. Plot quivers to demonstrate different appearances.

The following code will generate the figure:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(-20, 20, 8)
y = np.linspace( 0, 20, 8)
# make 2D coordinates
X, Y = np.meshgrid(x, y)
U, V = X+25, Y-35

# plot the barbs
plt.subplot(1,2,1)
plt.barbs(X, Y, U, V, flagcolor='green', alpha=0.75) plt.grid(True, color='gray')

# compare that with quiver / arrows
plt.subplot(1,2,2)
plt.quiver(X, Y, U, V, facecolor='red', alpha=0.75)

# misc settings
plt.grid(True, color='grey') plt.show()
```

The preceding code renders two subplots as shown in the following figure:



How it works...

To illustrate how the same data can bring different information to light, we used barbs and quiver plots from matplotlib to visualize simulated observed wind data.

First, we used NumPy to generate samples of variations for x and y arrays. Then we used NumPy's meshgrid() function to create a 2D grid of coordinates where our observed data is sampled at certain coordinates. Finally, U and V are wind speed values in NS (north-south) and EW (east-west) directions, in knots (nautical miles per hours). For the purpose of the recipe, we adjusted some values from the already available X and Y matrices.

We then divided the figure into two subplots, plotting barbs in the leftmost plot and arrow-patches in the rightmost plot. We adjusted the color and transparency of both the subplots slightly, as well as turned the grid on both the subplots.

There's more...

This is all fine on the northern hemisphere where the wind rotates in a counter-clockwise direction and the feathers (triangles, full lines and half lines of the barb) point in the direction of lower pressure. On the southern hemisphere, this is inverted so our wind barb graph would not represent the data we are visualizing correctly.
We have to invert this direction of feathers. Luckily, the barbs function has the

argument flip_barb. This argument can be of one single Boolean value (True or False) or a sequence of Boolean values such as the shape of other data arrays, when each item in the sequence specifies a flip decision for each barb.

Making a box and a whisker plot

Do you want to visualize a series of data measurement (or observations) to show several properties of the data series (such as the median value, the spread of the data, and the distribution of the data) in one plot? And would you want to do that in a way where you can visually compare several similar data series? How would you visualize them? Welcome to the box-and-whisker plot! Probably the best plot type for comparing distributions, if you are talking to people used to information density.

The box-and-whisker plot usage examples range from comparing test scores between schools to comparing process parameters before and after changes (optimization).

Getting ready

What are the elements of box and whisker plots? As we see in the following diagram, we have several important elements that carry information in the box-and-whisker plot. The first component is the box that carries information about the interquartile range going from lower to upper quartile values. The median value of the data is represented by a line across the box.

The whiskers extend from the box on both sides going from the first quartile (25 percentile) to the last quartile (75 percentile) of the data. In other words, the whiskers extend 1.5 times from the base of the inter-quartile range. In the case of a normal distribution, whiskers will cover 99.3 percent of the total data range.

If there are values outside the whiskers range, they will be displayed as fliers. Otherwise, the whiskers will cover the total range of the data.
Optionally, the box can also carry information about confidence intervals around the median. This is represented by a notch in the box. This information can be used to indicate whether the data in the two series is of the similar distribution. However, this is not rigorous and is just an indication that can be visually inspected.

How to do it...

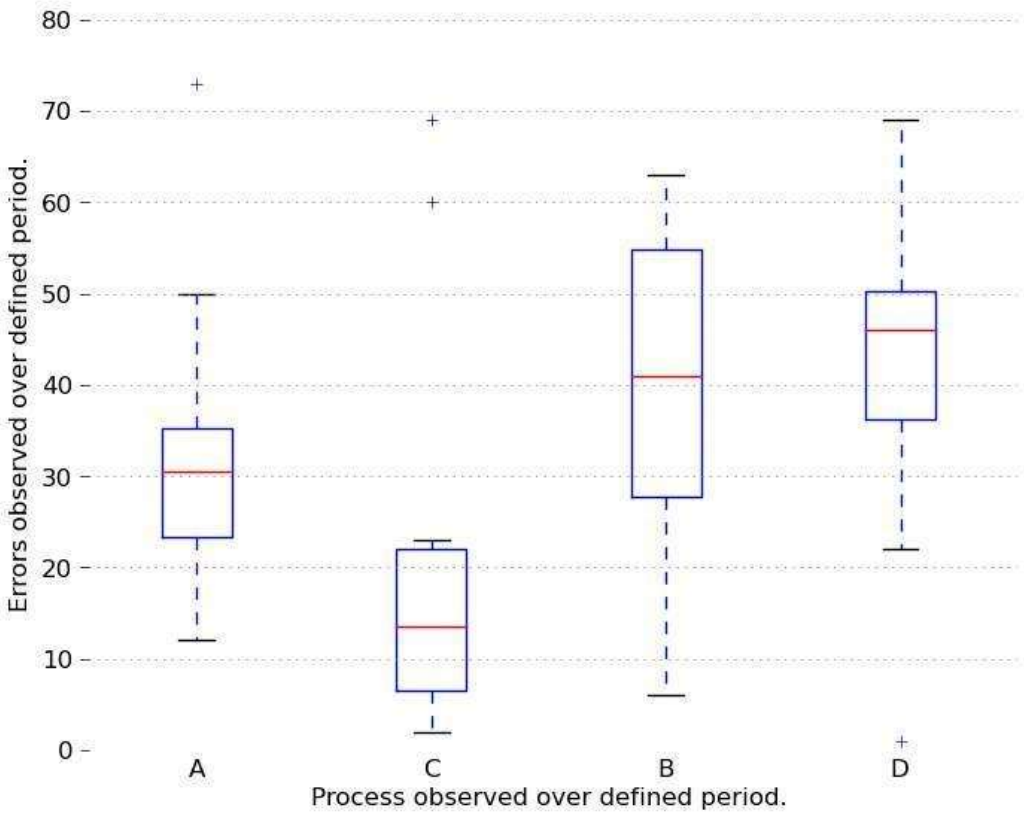
In the following recipe we will learn how to create a box-and-whisker plot using matplotlib. We will perform the following steps:
1. Sample some comparative process data, where a single integer number represents the occurrence of an error during the observed period of the running process.

2. Read data from the PROCESSES dictionary into DATA.
3. Read labels from the PROCESSES dictionary into LABELS.
4. Render the box-and-whisker plot using matplotlib.pyplot.boxplot.
5. Remove some chart junk from the figure.
6. Add axes labels.
7. Show the figure.

The following code implements these steps:

```
import matplotlib.pyplot as plt
# define data
PROCESSES = {

"A": [12, 15, 23, 24, 30, 31, 33, 36, 50, 73],
"B": [6, 22, 26, 33, 35, 47, 54, 55, 62, 63],
"C": [2, 3, 6, 8, 13, 14, 19, 23, 60, 69],
"D": [1, 22, 36, 37, 45, 47, 48, 51, 52, 69],
}

DATA = PROCESSES.values() LABELS = PROCESSES.keys()
plt.boxplot(DATA, notch=False, widths=0.3)
# set ticklabel to process name plt.gca().xaxis.set_ticklabels(LABELS)
# some clean up(removing chartjunk) # turn the spine off
for spine in plt.gca().spines.values(): spine.set_visible(False)

# turn all ticks for x-axis off
plt.gca().xaxis.set_ticks_position('none') # leave left ticks for y-axis on
plt.gca().yaxis.set_ticks_position('left')

# set axes labels
plt.ylabel("Errors observed over defined period.") plt.xlabel("Process observed
over defined period.")

plt.show()
```

The preceding code generates the following figure:



How it works...

The box and whisker plot is rendered by first computing quartiles for the given data in DATA. These quartile values are used to compute lines to draw boxes and whiskers.

We adjusted the plot to be more visually pleasing and not contain any unnecessary lines (referring to superfluous lines such as "chart junk", as mentioned in the famous book *The Visual Display of Quantitative Information* by *Edward R. Tufte*). Those lines do not carry information and just put more pressure on the mental models in a viewer's brain to decode all the lines before discovering real valuable information.

Making Gantt charts

One form of very widely-used visualization of time-based data is a Gantt chart. Named after the mechanical engineer *Henry Gantt* who invented it in 1910s, it is

almost exclusively used to visualize work breakdown structures in project management. This chart is loved by managers for its descriptive value and not so loved by employees, especially when the project deadline is near.

Because it is very common, almost every one can understand and read it, even if it is overloaded with additional (related and unrelated) information.

A basic Gantt chart has a time series on the x axis, and a set of labels that represent tasks or subtasks on the y axis. Task duration is usually visualized either as a line or as a bar chart, extending from the start to end time of a given task.

If subtasks are present, one or many subtasks have a parent task, in which the case total time of a task is aggregated from subtasks in such a way that overlapping and gap time is accounted for. This is useful to perform critical path analysis.

Critical path analysis is a mathematical analysis that computes a path containing all the tasks required, taking into account tasks interdependencies so that the total start to completion time of a project can be calculated. This is a very important tool in the project management field and can be universally applied to any type of project for scheduling and resource planning.

So, in this recipe we will be covering the creation of the Gantt chart using Python.

Getting ready

There are many full-fledged software applications and services that allow you to make very flexible and complicated Gantt charts. We will try to demonstrate how you could do it in pure Python, not relying on external applications, yet achieving neat looking and informative Gantt charts.
The Gantt chart shown in the example does not support nested tasks, but it is sufficient for simple work breakdown structures.

How to do it...

The following code example will allow us to demonstrate how Python can be used together with matplotlib to render the Gantt chart. We will perform the following steps: 1. Load TEST_DATA that contains a set of tasks and instantiate the Gantt class with TEST_DATA.

2. Each task contains a label and the start and end time.
3. Process all tasks by plotting horizontal bars on the axes.
4. Format x and y axes for the data we are rendering.
5. Tighten the layout.
6. Show the Gantt chart.
The following is a sample code:

```
from datetime import datetime
import sys

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.font_manager as font_manager
import matplotlib.dates as mdates

import logging

class Gantt(object):
'''
Simple Gantt renderer.
Uses *matplotlib* rendering capabilities. '''

# Red Yellow Green diverging colormap # from http://colorbrewer2.org/
RdYlGr = ['#d73027', '#f46d43', '#fdae61',

'#fee08b', '#ffffbf', '#d9ef8b', '#a6d96a', '#66bd63', '#1a9850']
POS_START = 1.0

POS_STEP = 0.5 def __init__(self, tasks):
self._fig = plt.figure()
self._ax = self._fig.add_axes([0.1, 0.1, .75, .5])

self.tasks = tasks[::-1]

def _format_date(self, date_string):
'''
Formats string representation of *date_string* into

*matplotlib.dates*
instance.
```

'''
try:

date = datetime.strptime(date_string, '%Y-%m-%d %H:%M:%S') except
ValueError as err:
logging.error("String '{0}' can not be converted to datetime object: {1}"
.format(date_string, err))
sys.exit(-1)
mpl_date = mdates.date2num(date)
return mpl_date

def _plot_bars(self):
'''
Processes each task and adds *barh* to the current *self._ax*

(*axes*).
'''
i = 0
for task in self.tasks:

start = self._format_date(task['start'])
end = self._format_date(task['end'])
bottom = (i * Gantt.POS_STEP) + Gantt.POS_START width = end - start
self._ax.barh(bottom, width, left=start, height=0.3,

align='center', label=task['label'], color = Gantt.RdYlGr[i])
i += 1

def _configure_yaxis(self):
'''y axis'''
task_labels = [t['label'] for t in self.tasks] pos = self._positions(len(task_labels))
ylocs = self._ax.set_yticks(pos)
ylabels = self._ax.set_yticklabels(task_labels) plt.setp(ylabels, size='medium')

def _configure_xaxis(self): ''x axis'''
make x axis date axis self._ax.xaxis_date()

# format date to ticks on every 7 days
rule = mdates.rrulewrapper(mdates.DAILY, interval=7) loc =
mdates.RRuleLocator(rule)

formatter = mdates.DateFormatter("%d %b")

self._ax.xaxis.set_major_locator(loc)
self._ax.xaxis.set_major_formatter(formatter) xlabels =
self._ax.get_xticklabels()
plt.setp(xlabels, rotation=30, fontsize=9)

def _configure_figure(self): self._configure_xaxis() self._configure_yaxis()

self._ax.grid(True, color='gray') self._set_legend()
self._fig.autofmt_xdate()

def _set_legend(self):
'''
Tweak font to be small and place *legend* in the upper right corner of the figure
'''
font = font_manager.FontProperties(size='small') self._ax.legend(loc='upper
right', prop=font)

def _positions(self, count):
'''
For given *count* number of positions, get array for the

positions.
'''
end = count * Gantt.POS_STEP + Gantt.POS_START pos =
np.arange(Gantt.POS_START, end, Gantt.POS_STEP) return pos

The main function that drives the Gantt chart generation is defined in the following code. In this function, we load the data into an instance, plot bars accordingly, set up the date formatter for the time axis (x axis), and set values for the y axis (the project's tasks).

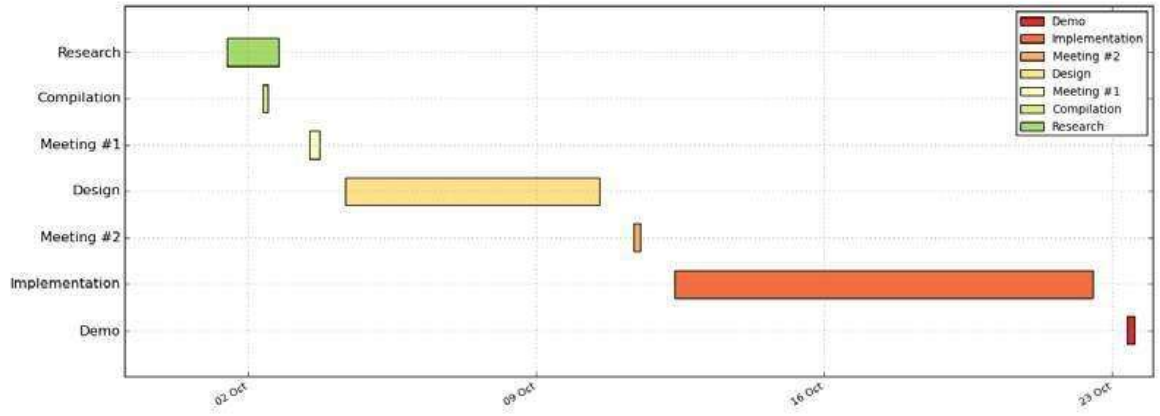
def show(self):
self._plot_bars()
self._configure_figure()
plt.show()

if __name__ == '__main__': TEST_DATA = (

{ 'label': 'Research', 'start':'2013-10-01
12:00:00', 'end': '2013-10-02 18:00:00'}, # @IgnorePep8 { 'label': 'Compilation',
'start':'2013-10-02
09:00:00', 'end': '2013-10-02 12:00:00'}, # @IgnorePep8 { 'label': 'Meeting #1',
'start':'2013-10-03
12:00:00', 'end': '2013-10-03 18:00:00'}, # @IgnorePep8 { 'label': 'Design',
'start':'2013-10-04
09:00:00', 'end': '2013-10-10 13:00:00'}, # @IgnorePep8 { 'label': 'Meeting #2',
'start':'2013-10-11
09:00:00', 'end': '2013-10-11 13:00:00'}, # @IgnorePep8 { 'label':
'Implementation', 'start':'2013-10-12
09:00:00', 'end': '2013-10-22 13:00:00'}, # @IgnorePep8 { 'label': 'Demo',
'start':'2013-10-23
09:00:00', 'end': '2013-10-23 13:00:00'}, # @IgnorePep8)

gantt = Gantt(TEST_DATA) gantt.show()

This code will render a simple, neat looking Gantt chart like the following one:



How it works...

We can start reading the preceding code from the bottom after the condition that checks if we are in "__main__".

After we instantiate the Gantt class giving it TEST_DATA, we set up the necessary fields of our instance. We save TASK_DATA in the self.tasks field, and we create our figure and axes to hold the charts we create in future.

Then we call show() on the instance that walks us through the steps required to render the Gantt chart:

```
def show(self):
self._plot_bars()
self._configure_figure()
plt.show()
```

Plotting bars requires an iteration where we apply the data about the name and duration of each task to the matplotlib.pyplot.barh function, adding it to the axes at self._ax. We place each task in a separate channel by giving it a different (incremental) bottom argument value.

Also, to make it easy to map tasks to their names, we cycle over the divergent color maps that we generated using the colorbrewer2.org tool.

The next step is to configure the figure, which means that we set up the format date on the x axis and tickers' positions and labels on the y axis to match the tasks plotted by matplotlib.pyplot.barh.

We do a final tweaking of grid and legend.
At the end, we call plt.show() to show the figure.

Making errorbars

Error bars are useful to display the dispersion of data on a plot. They are relatively simple as a form of visualization; however, they are also a bit problematic because what is shown as an error varies across different sciences and publications. This does not lessen the usefulness of error bars, it just imposes the need to always be careful and explicitly state the nature of the error visualized as an error bar.

Getting ready

To be able to plot an error bar in the raw observed data, we need to compute the mean and the error we want to display.
The error we compute represents the 95 percent confidence interval that the mean we get from our observation is stable, which means our observations are good estimates of the whole population.

matplotlib supports these type of plots via matplotlib.pyplot.errorbar function. It offers several capabilities around error bars. They can be vertical (yerr) or

horizontal (xerr), and symmetrical or asymmetrical.

How to do it...

In the following code we will:
1. Use some sample data that consists of four sets of observations.

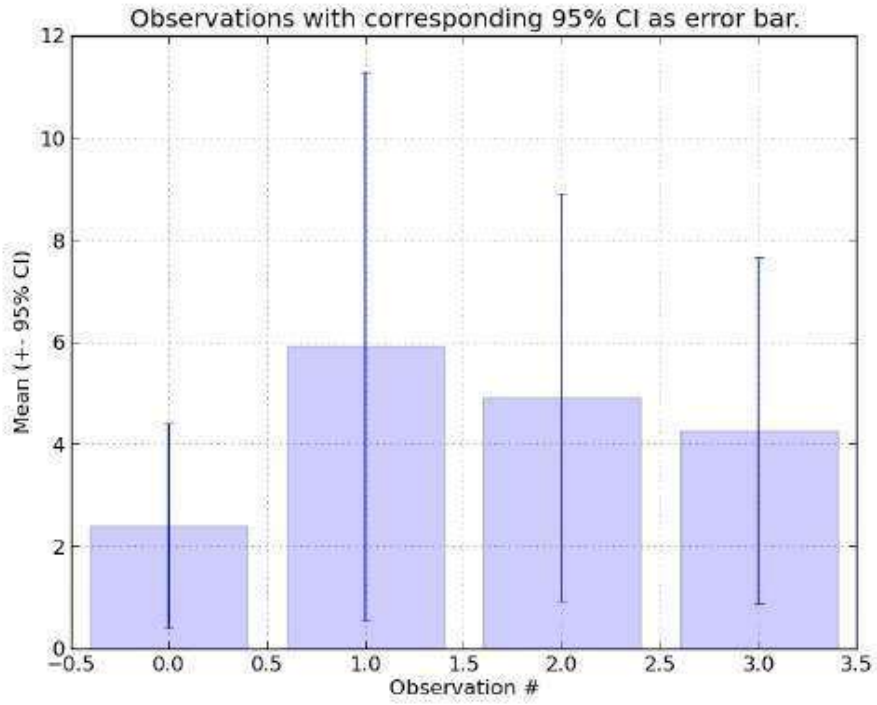
2. For each set of observations, compute the mean value.
3. For each set of observations, compute the 95 percent confidence interval.
4. Render bars with vertical symmetrical error bars.

Here is the code for this:

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

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

# find mean for each of our observations
y = np.mean(TEST_DATA, axis=1, dtype=np.float64) # and the 95% confidence
interval
ci95 = np.abs(y - 1.96 * sc.sem(TEST_DATA, axis=1))

# each set is one try
tries = np.arange(0, len(y), 1.0)

# tweak grid and setup labels, limits
plt.grid(True, alpha=0.5)
plt.gca().set_xlabel('Observation #')
plt.gca().set_ylabel('Mean (+- 95% CI)')
plt.title("Observations with corresponding 95% CI as error bar.") plt.bar(tries, y,
align='center', alpha=0.2) plt.errorbar(tries, y, yerr=ci95, fmt=None)

plt.show()
```

The preceding code will render a plot with error bars that display 95 percent confidence intervals as whiskers extending along the y axis. Remember, the

wider the whiskers, the lesser is the probability that the observed mean is true. The following graph is the output for the preceding code:



How it works...

In order to avoid iterating over each set of observations, we use NumPy's vectorized methods to compute means and standard errors, which we use for plotting and computing error values. Using NumPy's vectorized implementations that are written in C language (and called from Python) allows us to speed up computations by several magnitudes.
This is not very important for a few data points but, for millions of data points, it can either make or break our efforts to create responsive applications.

Also, you may notice that we explicitly specified dtype=np.float 64 in the np.mean function call. According to the official NumPy documentation reference (http://docs. scipy.org/doc/numpy/reference/generated/numpy.mean.html), np.mean can be inaccurate if used in single precision; it's better to compute it with np.float32, or if performance is not an issue, use np.float 64.

There's more...

There is an ongoing issue with what to show on error bars. Some advise on using

SD, *2SD*, *SE*, or *95%CI*. We must understand what the difference between all these values and what they are used for, in order to be able to give reasoning on what to use and when.

Standard Deviation informs us about the distribution of individual data points around the mean value. If we assume normal distribution, then we know that *68.2% (~2/3)* of data values will fall between ±*SD*, and *95.4%* of values will be between ±*2*SD*.

Standard Error is calculated as *SD* divided by the square root of *N* (*SD/√N*), where *N* is the number of data points. Standard Error (SE) informs us about variability of mean values, if we are able to perform the same sampling more than once (like performing the same study hundreds of times).

The confidence interval is calculated from SE, similar to how the range of values is calculated from Standard Deviation. To calculate 95 percent confidence interval, we must add/subtract *1.96 * SE* to/from our mean value or use proper notation: *95% CI = M ± (1.96 * SE)*. The wider the confidence interval, the lesser we would be sure that we are right.

We see that in order to be sure that our estimation is correct and that we are giving its proof to our reader, we should display the confidence interval, which in turn carries the standard error; this, if small, proves that our means are stable.

Making use of text and font properties

We already learned how to annotate the plot by adding legends, but sometimes we want more with text. This recipe will explain and demonstrate more features of text manipulation in matplotlib, giving a powerful toolkit for even advanced typesetting needs.

We will not cover LaTeX support in this recipe, as there is a recipe named *Rendering text with LaTeX* in this chapter.

Getting ready

We start with listing of the most useful set of functions that matplotlib offers. Most of the functions are available via pyplot module's interface, but we map their origin function here to allow you to explore more if a particular text feature is not covered in this recipe. Basic text manipulations and their mapping in

matplotlib OO API is presented in the following table:

matplotlib.pyplot text
Matplotlib API
matplotlib.axes.Axes.text
xlabel matplotlib.axes.Axes.set_xlabel
ylabel matplotlib.axes.Axes.set_ylabel
title matplotlib.axes.Axes.set_title
suptitle matplotlib.figure.Figure.suptitle
figtext matplotlib.figure.Figure.text
Description

Adds text to the axes at the location specified by (x, y). Argument fontdict allows us to override generic font properties, or we can use kwargs to override specific property.

Sets the label for the x
axis. Specifies the spacing between the label and the x axis in accordance with labelpad.

Similar to xlabel, but
intended for the y axis.

Sets the title for the axes. Accepts all the usual
text properties such as
fontdict and kwargs.

Adds a centered title to the figure. Accepts all the usual text properties via kwargs. Uses figure coordinates.

Puts text anywhere on the figure. The location is defined using x,y, using figure's normalized coordinates. Override font properties using fontdict, but also support kwargs to override any text-related property.

The base class for text storing and drawing inside windows or data coordinates is the matplotlib.text.Text class. It supports the definition of the location of text objects as well as a range of properties that we can define, to tune how our strings are going to appear on a figure or a window.
The font properties supported by the matplotlib.text.Text instances are:

Property family
size or
fontsize
style or fontstyle
variant
weight or fontweight
Values

'serif',
'sans-serif', 'cursive',
'fantasy',
'monospace'
12, 10,... or 'xx-small', 'x-small',
'small',
'medium',
'large',
'x-large', 'xxlarge'
'normal',
'italic',
'oblique'
'normal',
'small-caps' 0-1000 or
'ultralight', 'light',
'normal',
'regular',
'book',
'medium',
'roman',
'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black'

Description

Specifies the font name or font family. If this is a list, then it is ordered by priority, so the first matched name will be used.

Specifies the size in relative or absolute points, or specifies the relative size as a size string.
Specifies the font style as a string.
Specifies the font variant.

Specifies the font weight or using a specific weight string. Font weight is defined as the thickness of character outline relative to its height.
Property

stretch or fontstretch
Values

0-1000 or
'ultra
condensed', 'extra
condensed', 'condensed', 'semi
condensed', 'normal',
'semi
expanded', 'expanded', 'extra
expanded', 'ultra
expanded'

Description

Specifies the stretch of the font. Stretch is defined as horizontal condensation or expansion. This property is not currently implemented.

fontproperties Defaults to the matplotlib.font_manager.

FontProperties instance. This class stores and manages font properties as described in W3C CSS Level1 specification at http://www.w3.org/TR/1998/REC-CSS2-19980512/.

We can also specify the background box that will contain the text, and which can be further specified in color, borders, and transparency.
The basic text color is read from rcParams['text.color'], if not specified on the current instance, of course.
Specified text can also be aligned according to visual needs. There are the following alignment properties:

f horizontalalignment or ha: This allows alignment of text horizontally to center, left, and right.
f verticalalignment or va: Allowed values for this are center, top, bottom, and baseline.
f multialignment: This allows alignment of text strings that span multilines.

Allowed values are: left, right, and center.

How to do it...

So far all is good, but we have a hard time visualizing all these variations in the fonts we can create. So this is going to illustrate what we can do. In the next code we will be performing the following steps:

1. List all the possible properties we want to vary on the font.
2. Iterate over the first set of variations: font family and size.
3. Iterate over the second set of variations: weight and style.

4. Render text samples for both the iterations and print the variation combination as a text on the plot.
5. Remove axes from the figure, as they serve no purpose.
The following is the code:

```
import matplotlib.pyplot as plt
from matplotlib.font_manager import FontProperties

# properties:
families = ['serif', 'sans-serif', 'cursive', 'fantasy', 'monospace'] sizes = ['xx-small',
'x-small', 'small', 'medium', 'large',

'x-large', 'xx-large']
styles = ['normal', 'italic', 'oblique']
weights = ['light', 'normal', 'medium', 'semibold', 'bold', 'heavy', 'black']
variants = ['normal', 'small-caps']

fig = plt.figure(figsize=(9,17))
ax = fig.add_subplot(111)
ax.set_xlim(0,9)
ax.set_ylim(0,17)

# VAR: FAMILY, SIZE
y = 0
size = sizes[0]
style = styles[0]
weight = weights[0]
variant = variants[0]
```

```
for family in families:
x = 0
y = y + .5
for size in sizes:

y = y + .4
sample = family + " " + size
ax.text(x, y, sample, family=family, size=size,

style=style, weight=weight, variant=variant) # VAR: STYLE, WEIGHT
y = 0
family = families[0]
size = sizes[4]
variant = variants[0]

for weight in weights: x = 5
y = y + .5
for style in styles:

y = y + .4
sample = weight + " " + style
ax.text(x, y, sample, family=family, size=size,

style=style, weight=weight, variant=variant)
ax.set_axis_off() plt.show()
```

The preceding code will produce the following screenshot:

How it works...

The code is really straightforward, as we just iterate twice over tuples of

properties printing their values.
The only trick employed here is the positioning of text on the figure canvas, as that allows us to have nice layout of text samples we can easily compare.

Keep in mind that the default font matplotlib will use is dependent on the operating system you are running, so the preceding screenshot might look slightly different. This screenshot was rendered using standard Ubuntu 13.04 installed fonts.

Rendering text with LaTeX

If we want to plot more scientific graphics and explain math as it should be using scientific notations and complex equations on the figures, we need support from the best. Although matplotlib has support for math text rendering, the best support comes from the LaTeX community, proven in the task being used for many decades.

LaTeX is a high-quality typesetting system for the production of scientific and technical documentation, being a *de facto* standard for scientific typesetting or publication. It is a free software, available on the majority of desktop platforms used today as prepackages binary installation; hence, it is easy to install.

The basic syntax of LaTeX is similar to markup languages; so to produce satisfactory content, one would write focusing more on the structure than on the look and style. For example:

```
\documentclass{article}
\title{This here is a title of my document}
\author{Peter J. S. Smith}
\date{September 2013}
\begin{document}

\maketitle
Hello world, from LaTeX!
\end{document}
```

We see how this is different from the usual word processor, where you have the WYSIWYG editor environment and the style is already applied to your text. Sometimes this is good but, for scientific publications, style is a secondary

concern; the primary focus is having the right, correct, and valid content. Here, by content we also mean mathematical notations (usually a lot of it), including graphs.

Apart from this, there are many more features such as automatic generation of bibliographies and indexes, which are important for medium to large publications. And those are the main focus points of the LaTeX system. Because this is not a book about LaTeX, we will stop with the quick introduction here. A lot more documentation is available on the project's website at http://latex-project.org/.

Getting ready

Before we start demonstrating matplotlib's support for rendering text using LaTeX, we need to have the following packages installed on our system:
f LaTeX system: The most common one is the TeX Live prepackaged distribution f DVI to PNG converter: This makes PNG graphics from DVI files as obtained from TeX, by producing anti-aliased screen-resolution images
f Ghost script: This is required, unless already installed by TeX Live distribution

There are different prepackaged systems of the LaTeX environment for different operating systems. For Linux-based systems, TeX Live is a complete TeX system. For Mac OS, the recommended environment is the MacTeX distribution; for the Windows environment, the proTeX system is going to install all the TeX supports, including LaTeX.

Whichever package you install, make sure it comes with font libraries and programs for typesetting, previewing, and printing of TeX documents in many different languages. We will install our package for Linux using the texlive and dvipng packages for Ubuntu. We can install this by using the following command:
$ sudo apt-get install texlive dvipng

The next step is to tell our matplotlib to use LaTeX by setting text.usetex toTrue. We can do that either in our custom .matplotlibrc inside our home directory (/home/<user>/. matplotlibrc on Unix-based systems, or C:\Documents and Settings\<user>\. matplotlibrc) via rcParams['text'], or by using the following code:

```
matplotlib.pyplot.rc('text', usetex=True)
```

The start of the code will tell matplotlib to go back to LaTeX for all text rendering. It is important to do this before we add any figure and axis.
Not all backends support LaTeX rendering. Only the Agg, PS and PDF backends support text rendering via LaTeX.

How to do it...

What we want to do here is demonstrate the basic usage properties of LaTeX. We will perform the following steps:

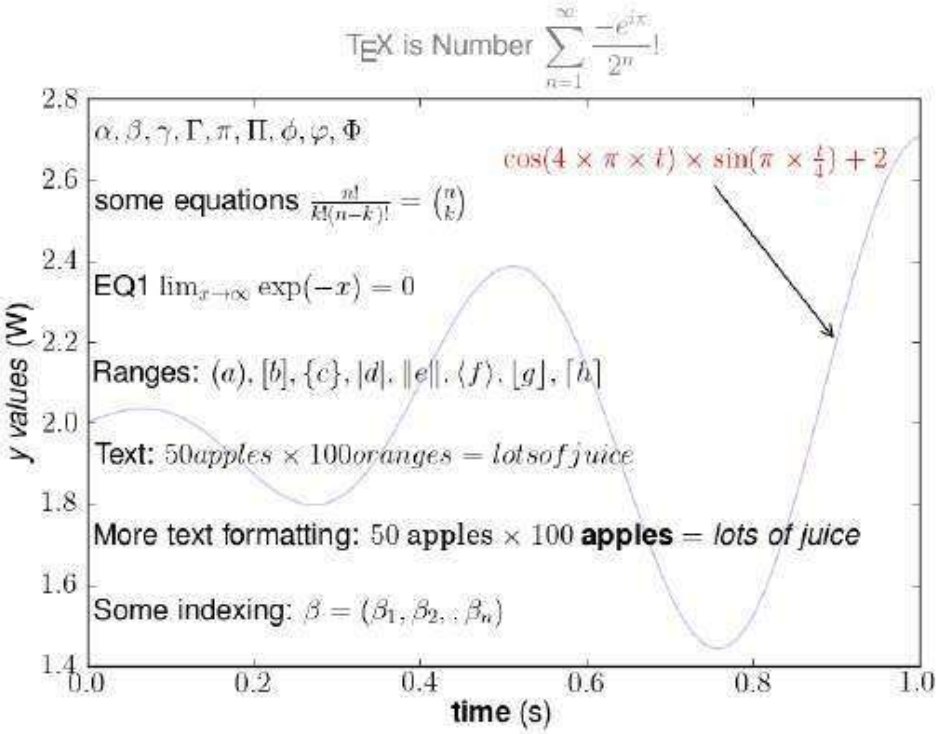
1. Generate some sample data.
2. Set up matplotlib to use LaTeX for this plotting session.
3. Set up the font and font properties to be used.
4. Write out the equation syntax.
5. Demonstrate the usage of Greek symbols' syntax.
6. Draw math notations of fractions and fractals.
7. Write some limits and exponential expressions.
8. Write possible range expressions.
9. Write expressions with text and formatted text in them.
10. Write some math expressions on x and y labels as figure titles.

The following code will perform these steps:

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

# Example data
t = np.arange(0.0, 1.0 + 0.01, 0.01)
s = np.cos(4 * np.pi * t) * np.sin(np.pi*t/4) + 2

plt.rc('text', usetex=True)
plt.rc('font',**{'family':'sans-serif','sans-serif':['Helvetica'], 'size':16})

plt.plot(t, s, alpha=0.25)

# first, the equation for 's'
# note the usage of Python's raw strings
plt.annotate(r'$\cos(4 \times \pi \times {t}) \times \sin(\pi \times \ frac {t} 4) +
2$', xy=(.9,2.2), xytext=(.5, 2.6), color='red', arrowpr ops={'arrowstyle':'->'})

# some math alphabet
plt.text(.01, 2.7, r'$\alpha, \beta, \gamma, \Gamma, \pi, \Pi, \phi, \ varphi, \Phi$')
# some equation
```

```
plt.text(.01, 2.5, r'some equations $\frac{n!}{k!(n-k)!} = {n \choose k}$')
# more equations
plt.text(.01, 2.3, r'EQ1 $\lim_{x \to \infty} \exp(-x) = 0$') # some ranges...
plt.text(.01, 2.1, r'Ranges: $( a ), [ b ], \{ c \}, | d |, \| e \|, \ langle f \rangle, \lfloor
g \rfloor, \lceil h \rceil$')
# you can multiply apples and oranges
plt.text(.01, 1.9, r'Text: $50 apples \times 100 oranges = lots of juice$')
plt.text(.01, 1.7, r'More text formatting: $50 \textrm{ apples} \times 100 \textbf{
apples} = \textit{lots of juice}$')
plt.text(.01, 1.5, r'Some indexing: $\beta = (\beta_1,\beta_2,\dotsc,\ beta_n)$')
# we can also write on labels
plt.xlabel(r'\textbf{time} (s)')
plt.ylabel(r'\textit{y values} (W)')
# and write titles using LaTeX
plt.title(r"\TeX\ is Number "

r"$\displaystyle\sum_{n=1}^\infty\frac{-e^{i\pi}}{2^n}$!", fontsize=16,
color='gray')
# Make room for the ridiculously large title.
plt.subplots_adjust(top=0.8)

plt.savefig('tex_demo') plt.show()
```

The preceding code will render the following text-saturated figure that demonstrates LaTeX rendering:



How it works...

After we set up the rendering engine and font properties, we basically used standard matplotlib calls for text rendering, such as matplotlib.pyplot.annotate, matplotlib. pyplot.text, matplotlib.pyplot.xlabel, matplotlib.pyplot.ylabel, and matplotlib.pyplot.title.

The difference here is that all the strings are so-called raw strings, meaning Python will not interpret them and no string substitution will occur; hence the LaTeX engine is going to receive exactly the same strings as commands to act upon.

More examples of the TeX syntax and how to use it in matplotlib can be found on official matplotlib documentation at http://matplotlib.org/users/mathtext. html#writing-mathematical-expressions.

Note that this URL is not on LaTeX but on matplotlib's own integrated TeX parser. This parser supports almost the same syntax, and it can even be sufficient for your needs.

There's more...

If you run into a problem while setting up this environment or have different problems with fonts that either look bad or are not able to produce the LaTeX rendering, make sure that you have installed all required packages, your $PATH environment variable (if on Windows) is set up to include all the required binaries, and matplotlib is set to use LaTeX for text rendering.

If all of the given instructions are followed and the results cannot be replicated, refer to the official matplotlib website at http://matplotlib.org/users/usetex.html#possible-hangups, and the LaTeX community on http://tex.stackexchange.com/ for further assistance.

It is known that this setup is not as streamlined as it should be, and some quirks may occur for various reasons.

Understanding the difference between pyplot and OO API

This recipe will try to explain some of the programming interfaces in matplotlib and make a comparison of pyplot and object-oriented API (Application Programming Interface). Depending on the task at hand, this will allow us to decide why and when to use either of those interfaces.

Getting ready

When the matplotlib library started, it was similar to many open source projects—there was no proper (free) solution to the problem a person had, so he wrote one. The problem encountered with MATLAB® was with performance for the task in hand (http://www.aosabook.org/ en/matplotlib.html), and the original author already had knowledge of both MATLAB® and Python, so he started writing matplotlib as a solution for his need for the current project.

This is the main reason matplotlib has a MATLAB®-like interface that allows one to quickly plot data without worrying about background details, which platform matplotlib is running on, which are the underlying rendering libraries (is it Linux with GTK, Qt, Tk, or wxWidgets either on Linux or Windows), or are we running on Mac OS with the help of Cocoa toolkits. This is all hidden inside matplotlib under a nice procedural interface in the matplotlib.pyplot module, a stateful interface handling logic for creating figures and axes to connect them with the configured backend. It also keeps data structures for the current figure and axes, which are called upon with plot commands.

This is the interface (matplotlib.pyplot) we were using throughout most of this book as it is simple, straightforward, and good enough for most of the tasks we were trying to accomplish. The matplotlib library was designed with this philosophy in mind. We must be able to draw plots with as few commands as possible, even just one command (for example, plt.plot([1,2,3,4,5]); plt.show() works!). For these tasks we don't want to be forced into thinking about objects, instances, methods, properties, rendering backends, figures, canvases, lines, and other graphical primitives.

If you are reading this book from the start, you probably noticed that some classes started appearing in various examples; such as, FontProperties or AxesGrid, where we needed more than is provided by the matplotlib.pyplot module.

This is the object-oriented programming interface that implements all the hidden hard stuff such as rendering graphical elements, rendering those to the platform's graphical toolkit, and handling user inputs (mouse and keystrokes). There is nothing to stop us from using OO API, and that is what we are going to do.

So if we take a look at matplotlib as a software, it consists of three parts:
f matplotlib.pylab interface: This is a set of functions for the user to create plots like in MATLAB®
f matplotlib API (also called matplotlib frontend): This is a set of classes for the creation and management of figures, text, lines, plots, and so on

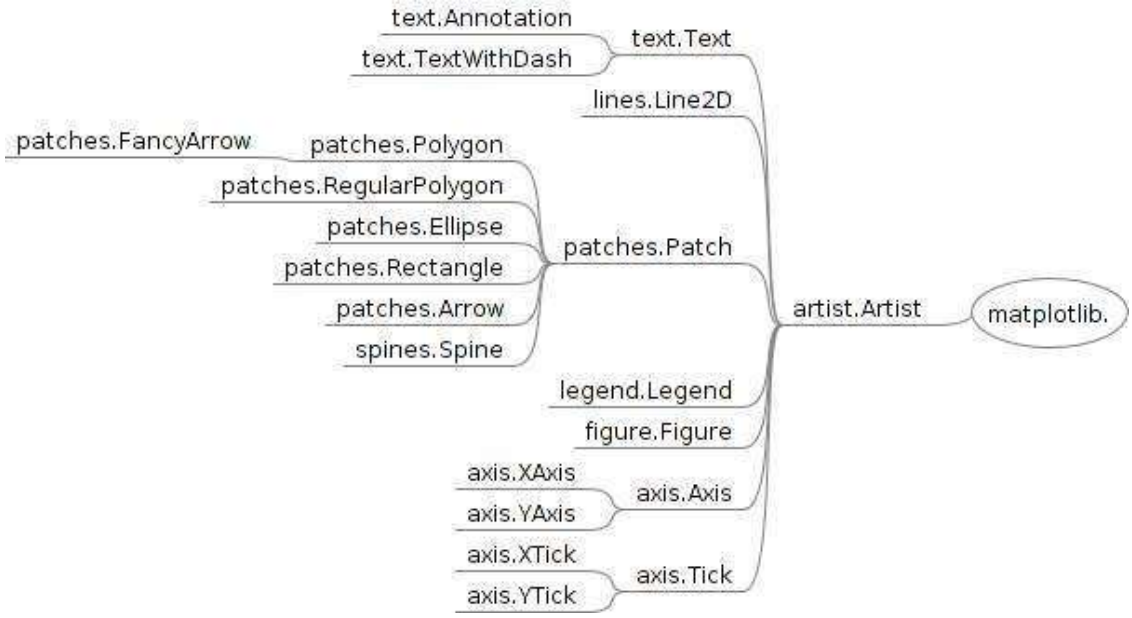
f backends: These are drawing drivers; they transform front abstract representation into a file or a display device
This backend layer contains concrete implementations of abstract interface classes. There are classes such as FigureCanvas (a surface to draw onto paper), Renderer (a paintbrush that does the drawing on the canvas), and Event (a class that handles the user's keystrokes and mouse events).

The code is also separated. The base abstract classes are in matplotlib.backend_bases and every concrete implementation is in a separate module. For example, the GTK 3 backend is in matplotlib.backends.backend_gkt3agg.

In this stack there is an Artist classes' hierarchy where most of the hard stuff is done. Artist knows about Renderer and how to use it to draw images on

FigureCanvas. Most of the stuff we are interested in (text, lines, ticks, tick labels, images, and so on) are Artist or subclasses of the Artist class (located in the matplotlib.artist module).

The class, matplotlib.artist.Artist, contains all the shared properties of its children: coordinates transformation, clip box, label, user event handlers, and visibility.



In this figure Artist is the base for most of the other classes. There are two basic categories of classes that inherit from Artist. The first category is of primitive artists, which are visible objects such as Line2D, Rectangle, Circle, and Text. The second category is of composite artists, which are collections of other Artists such as Axis, Tick, Axes, and Figure. For example, Figure has the background of the primitive artist Rectangle, but also contains at least one composite artist, Axes.

Most of the plotting is happening on the Axes class (matplotlib.axes.Axes). Figure background elements such as ticks, axis lines, and the grid and color of the background patch is contained in Axes. Another important feature of Axes is that all the helper methods create other primitive artists and add them to the Axes instance; for example, plot, hist, and imshow.
Axes.hist, for example, creates many matplotlib.patch.Rectangle instances and stores them in the Axes.patches collection.

Axes.plot creates one or more matplotlib.lines.Line2D and stores them in the

Axes.lines collection.

How to do it...

As an illustration we will:

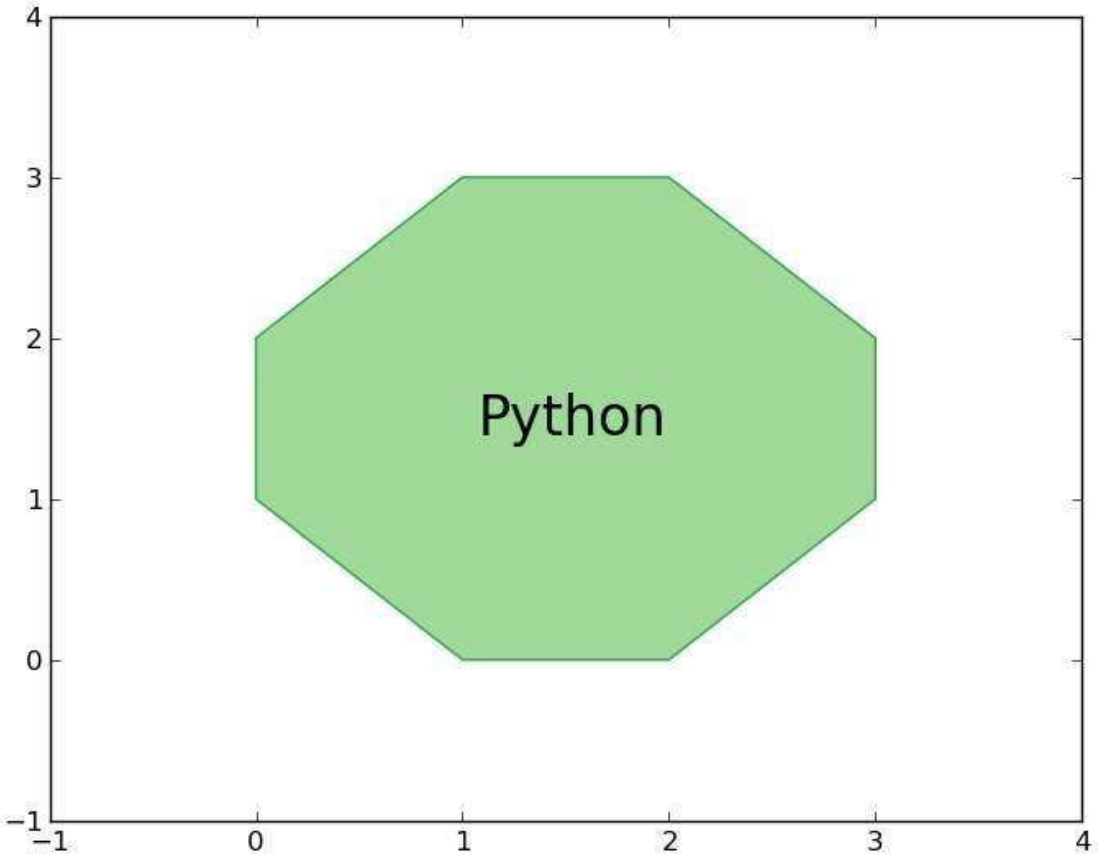
1. Instantiate the matplotlib Path object for custom drawing.
2. Construct the vertices of our object.
3. Construct the path's command codes to connect those vertices.
4. Create a patch.
5. Add it to the Axes instance of figure.

The following code implements our intentions:

```
import matplotlib.pyplot as plt
from matplotlib.path import Path
import matplotlib.patches as patches

# add figure and axes
fig = plt.figure()
ax = fig.add_subplot(111)

coords = [
(1., 0.), # start position (0., 1.),
(0., 2.), # left side
(1., 3.),
(2., 3.),
(3., 2.), # top right corner (3., 1.), # right side (2., 0.),
(0., 0.), # ignored
]

line_cmds = [Path.MOVETO, Path.LINETO, Path.LINETO, Path.LINETO,
Path.LINETO, Path.LINETO, Path.LINETO, Path.LINETO, Path.CLOSEPOLY,
]

# construct path
path = Path(coords, line_cmds)
# construct path patch
patch = patches.PathPatch(path, lw=1,

facecolor='#A1D99B', edgecolor='#31A354') # add it to *ax* axes
ax.add_patch(patch)
```

```
ax.text(1.1, 1.4, 'Python', fontsize=24) ax.set_xlim(-1, 4)
ax.set_ylim(-1, 4)
plt.show()
```

The preceding code will generate the following:



How it works...

For this octagon we used the base patch matplotlib.path.Path, which supports the basic set of primitives for drawing lines and curves (moveto and lineto). These can be used to draw simple and also more advanced polygons using Bezier curves.

First, we specified a set of coordinates in the data coordinates that we match with a set of path commands to act upon those coordinates (or vertices, if you like). With that we instantiate matplotlib.path.Path. We then construct the patch instance matplotlib. patched.PathPatch with that path, which is a general polycurve path patch.

This patch can now be added to the figure's axes (the fig.axes collection) and we can render the figure to show the polygon.

What we didn't want to do in this example is use matplotlib.figure.Figure directly in place of the matplotlib.pyplot.figure() call. The reason for this is that the pyplot. figure() call does a lot in the background, such as reading the rc parameters from the matplotlibrc file (to load default figsize, dpi and figure color settings), setting up the figure manager class (Gcf), and so on. We could do all that, but until we really know what we are doing, this is the recommended way to create the figure.

As a general rule of thumb, unless we cannot achieve something via the pyplot interface, we should not reach for direct classes such as Figure, Axes, and Axis, because there is a lot of state managing going on in the background; so, unless we are developing matplotlib, we should avoid bothering about that.

There's more...

If you want interactivity and exploration, it would be the best to use matplotlib via the Python interactive shell. For this purpose, probably the most well known is the IPython pylab mode. This gives you all the matplotlib features in a powerful and introspective shell with rich set features such as history, inline plotting, and the possibility to share your work if you use IPython Notebook.

IPython Notebook is a web-based interface to the IPython shell, where the work can be shared and converted into HTML or PDF. Matplotlib plots are embedded and inlined, so they can also be saved and shared.

Symbols

3D bars
creating 139-143
3D histograms
creating 143-146
3D visualization
about 139
3D bars, creating 139-143
3D histograms, creating 143-146
.rc file 14

Index

A

Advanced Linux Sound Architecture (ALSA) 194
animate() function 149
animation
with OpenGL 150-154
animation, matplotlib 146-150
Animation (object) class 146
annotations
adding 92-94
array slicing 54
ArtistAnimation (TimedAnimation) class 147
autocorrelation
about 220
importance 220
plotting 221-223
Axes3D 140
AxesGrid 139
axis() function 82
axis label
size, setting 110-112
transparency, setting 110-112
axis lengths
defining 81-83
axis limits
defining 81-83
B
backends 251
background color
defining 89

barb
about 225
drawing 225-229

barbs function 226

bar charts
creating 99-101

bard3d function 146
bar plot 72-78
Basemap

about 139
used, for plotting data on map 172-176

Basemap toolkit
URL, for documentation 177
box plot
about 77
making 229-232
brg colormap 206
bwr colormap 206

C

CAPTCHA image
about 183, 184
generating 183-188

cell() method 24
chart line
shadow, adding to 113-116
color
defining 88

ColorBrewer
URL 206
Colorbrewer2
URL 85
colored markers
used, for drawing scatter plots 105, 107
colormaps
using 205-210
colormaps, categories
cyclic 206

diverging 205
qualitative 206
sequential 205
colormaps, ColorBrewer
category 206
colormaps function 207
Comma Separated Values. *See* CSV
contour() function
about 125
call signature 126
contourf(..., V) call signature 126
contour plot
about 125
creating 125-128
contour(X,Y,Z) call signature 126
contour(X,Y,Z,N) call signature 126
contour(X,Y,Z,V) call signature 126
contour(Z) call signature 126
contour(Z, **kwargs) call signature 126
contour(Z,N) call signature 126
contour(Z,V) call signature 126
controlled random datasets
generating 56-64
convolve function 64
coolwarm colormap 206
Coordinate system
Axes 113
Data 113
Display 113
Figure 113
correlate() function 220
cosine plot
drawing 78-81
create_thumbnails() function 163
cross-correlation
plotting, between two variables 217-220 CSV 20
CSV file
data, exporting to 31-35 data, importing from 20-22
csv.reader() method 21

D

data
cleaning up, from outliers 40-46
exporting, to CSV file 31-35
exporting, to JSON file 31-35
exporting, to Microsoft Excel file 31-35 importing, from CSV file 20-22
importing, from database 36-40
importing, from fixed-width datafile 25, 26 importing, from JSON resource 28-30 importing, from Microsoft Excel file 22-25 importing, from tab-delimited file 27, 28 plotting on map, Basemap used 172-176 plotting on map, Google Map API used

177-183
database
data, importing from 36-40

data table
adding, to figure 116-118
data visualization, types
bar charts 72
histograms 72
line graphs 72
pie charts 72
deactivate command 9
dialect 27
Distutis 8
Django 1.1 8
drawMap() function 182
dump() method 35
DVI to PNG converter 247

E

Enthought Python Distribution (EPD) 10 equential colormap 205
error bars

about 99, 237
drawing 99-101

making 237-239

F
figtext function 241
figure
data table, adding 116-118
figure() function 76, 89
file
reading 46-48

filesystem tree
visualizing, polar bar used 134-137
fill_between() function 104, 128
filled areas
plotting 103, 104
fill() function 105
fixed-width datafile
data, importing from 25, 26
font properties
family 242
fontproperties 243
fontsize 242
fontstretch 243
fontstyle 242
fontweight 242
size 242
stretch 243
style 242
using 240-246
variant 242
weight 242
format_data() function 167
freetype 6
freetype 1.4+ 7
FuncAnimation (TimedAnimation) 147

G

Gantt chart

about 232
making 232-237

get_captcha method 187
get() method 30
get_size function 137
Ghost script 247
GitHub

URL 29
Glumpy 151
Glumpy Quickstart

used, for animating with OpenGL 155 Google Data Visualization Library 177
Google Developer

URL 183
Google Geochart 178
Google Map API

used, for plotting data on map 177-183 Google Visualization API 177
grids

about 121-123
customizing 121-124
setting 89, 90

GTK Tools 139
H

hist() function 98
histograms
about 97
making 97-99
using 210-217
hold property 74
Homebrew project 10
horizontalalignment property 243 HTTP Protocol and Response message
URL 183

I
ImageChops module 159
image data
importing, into NumPy arrays 50-56

ImageDraw module 158
ImageFilter module 159
Image module
im.crop(box) method 158
im.filter(filter) method 158
im.histogram() method 158
im = Image.open(filename) method 158 im.resize(size, filter) method 158
im.rotate(angle, filter) method 158 im.split() method 158
im.transform(size, method, data, filter) method 158
image processing
example 56
with PIL 158-163
image processing, Python 50-56 images
displaying, with other plots 168-172 plotting 164-168
imread function 168
init() function 149
installation, matplotlib
steps 6-8
on Mac OS X 10, 11
on Windows 11, 12
installation, NumPy 6-8
installation, PIL
for image processing 13
installation, requests module 14 installation, SciPy 6-8
installation, SQLite library 36 installation, virtualenv 8, 9
installation, virtualenvwrapper 8, 9 intsallation, Python 11, 12
IPython 8, 153
IPython Notebook 255
isolines 125

J
JavaScript Object Notation (JSON) 28

JSON file

data, exporting to 31-35
json.loads() function 31
JSON resource
data, importing from 28-30

L

labels
setting 89, 90
LaTeX
about 246, 247
syntax 246
used, for rendering text 246-250
LaTex syntax 81
LaTeX system 247
legend
adding 92-94
legend() function 93
libpng 6
libpng 1.2 7
line markers
list 87
line plot
about 72-78
format strings, defining 84-86 properties 86
properties, defining 84-86 styles, defining 84-86
linestyles
list 87
linspace function 64
load_data function 167
load_files() function 163
loadtxt() function 21
logarithmic plot
about 190-193
rules 190

M

map

data plotting, Basemap used 172-176 data plotting, Google Map API used 177-183

matplotlib
about 72
animation 146-149
installing 6-8
installing, on Mac OS X 10, 11
installing, on Windows 11, 12
plot, elements 73

matplotlib API 251
matplotlib parameters
customizing, in code 14, 16
customizing, per project 16, 17
matplotlib.pylab interface 251
matplotlibrc configuration file
location 17
settings 17
matplotlib software
backends 251
matplotlib API 251
matplotlib.pylab interface 251
Mayavi 151
Median absolute deviation (MAD) 40 Median Filter 69
meshgrid() function 228
meshgrid property 204
Microsoft Excel file
data, exporting to 31-35
data, importing from 22-25 mkvirtualenv ENV command 9 mplot3d 139
multialignment property 243

N

Natgrid 139
NetCDF 177
next() function 47
noise signal

smoothing, in real-world data 64-70

NumPy
about 6
installing 6-8
URL 6

NumPy arrays
image data, importing into 50-56
O

object-oriented API (OO API)
about 250
differentiating, with pyplot 250-255

open() function 163

OpenGL
about 151
animating with 150-154
animating with, Glumpy Quickstart used 155 animating with, Pyglet Quickstart used 154

OpenRefine
URL 46
Optical Character Recognition (OCR) 188
outliers
data, cleaning up 40-46

P

pie charts
about 101
making 101, 103

PIL
about 12, 53
installing, for image processing 13
URL 13
used, for image processing 158-163

Pillow
URL 13
plot
background color, defining 89 color, defining 88
elements 73
plot() function 73, 142, 167
plot types
bar plot 72-78
box plot 77
contour plot 125
cosine plot 78-81
defining 72-78
line plot 72-78
logarithmic plot 190
polar plot 132
scatter plots 105
sine plot 78-81
stacked charts 72-78
stem plot 198
stream plot 201
whiskers plot 77
polar() function 132
polar plot
about 132
drawing 131-133
probability distribution 57
Processing
URL 155
proTeX system 247
Pyglet 151
Pyglet Quickstart
used, for animating with OpenGL 154
PyPi
URL 25
pyplot
differentiating, with OO API 250-255
pyplot function 207
Pyprocessing 155
Python

CSV module 20
file, reading 46, 47, 48
image processing 50-56
intsalling 11, 12
Python 2.3 8
Python 2.6 8
Python 2.7+ Version 6
Python 3.3+ Version 6
Python Distribution Utilities 8 Python Imaging Library. *See* PIL

R

rainbow colormap 206
random module 57
read() function 47
read_png() function 167
real-world data

noise signal, smoothing 64-70 reCAPTCHA
URL 188
recaptcha-client
URL 188
requests module
about 29
installing 14
Response.json() method 30 run() function 155

S

save() function 149, 163
scatter() function 107, 142
scatterhist() function 215
scatter plots

about 105
drawing, with colored markers 105-107 using 210-217

scikit-image
URL 56
SciPy

about 6
installing 6-8
SciPy Cookbook
URL 66
SciPy signal toolbox
implementation 69
seek() function 47
seismic colormap 206
setp() function 85
setstate() function 63
shadow
adding, to chart line 113-116
sheets() method 24
show() function 171, 237
sine plot
drawing 78-81
spectrogram
about 193-197
spines
about 95
moving, to center 95, 96
split() function 28
SQLite db file 38
SQLite library
installing 36
stacked chart plot 72-78
standard deviation 57
statistical population 57
stem() function 198
stem plot
about 198
creating 198-201
formatters 198
streaming data source
reading 48, 49
stream plot
about 201
drawing 201-205
used, for visualizing vector field 201

struct module
about 25
URL 25
subplot
using 118-120
suptitle function 241

T

tab-delimited file
data, importing from 27, 28
Table Visualization 178
tell() function 47
terrain colormap 206
test_* function 153
text
alignment properties 243 rendering, with LaTeX 246-250
text function 241
text properties
using 240-246
ticks
about 89
setting 89, 90
TimedAnimation (Animation) class 147
title function 241
toolkit 139
transformations 113

U
Ubuntu 13.04 246
under-plot area
filling 128-131
urllib2 module 14
usevlines() function 218
V

variance 57
verticalalignment property 243 virtualenv

about 8
installing 8, 9
virtualenvwrapper

about 9
installing 8, 9
URL 9

W

whiskers plot
about 77
making 229-232

WIMP
URL 158
windowing algorithm 68
workon ENV command 9
write_data() function 35
WYSIWYG
URL 158

X
xlabel function 241
Y
ylabel function 241
Thank you for buying

Python Data Visualization Cookbook
About Packt Publishing

Packt, pronounced 'packed', published its first book " *Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the

software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cuttingedge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

Python Geospatial
Development - Second
Edition

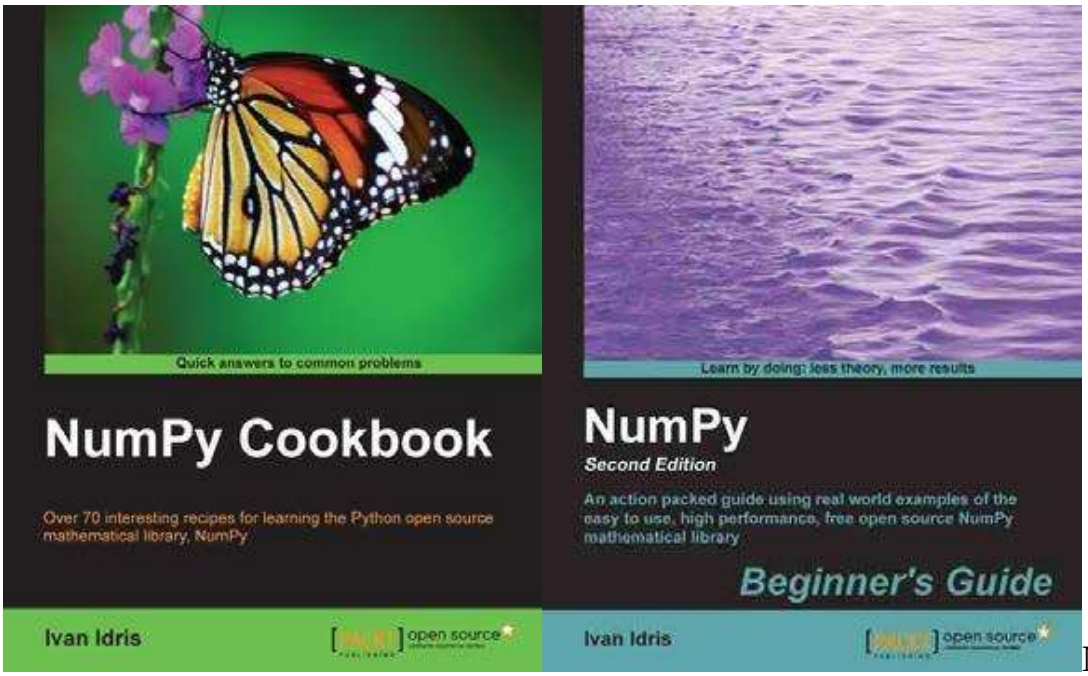
ISBN: 978-1-78216-152-3 Paperback: 508 pages Learn to build sophisticated mapping applications from scratch using Python tools for geospatial development 1. Build your own complete and sophisticated mapping applications in Python

2. Walks you through the process of building your own online system for viewing and editing geospatial data

3. Practical, hands-on tutorial that teaches you all about geospatial development in Python
Programming ArcGIS 10.1 with Python Cookbook
ISBN: 978-1-84969-444-5 Paperback: 304 pages

Over 75 recipes to help you automates geoprocessing tasks, create solutions, and solve problem for ARcGIS with Python

1. Learn how to create geoprocessing scripts with ArcPy
2. Customize and modify ArcGIS with Python 3. Create time-saving tools and scripts for ArcGIS



NumPy Cookbook
ISBN: 978-1-84951-892-5 Paperback: 226 pages Over 70 interesting recipes for learning the Python open source mathematical library, NumPy
1. Do high performance calculations with clean and efficient NumPy code
2. Analyze large sets of data with statistical functions 3. Execute complex linear algebra and mathematical computations
NumPy Beginner's Guide Second Edition
ISBN: 978-1-78216-608-5 Paperback: 310 pages

An action packed guide using real world examples of the easy to use, high performance, free open source NumPy mathematical library

1. Perform high performance calculations with clean and efficient NumPy code
2. Analyze large datasets with statistical functions 3. Execute complex linear algebra and
mathematical computations