Chetan Giridhar

Automate it!

Recipes to upskill your business

Put your life on autopilot and make your business processes efficient with the magic and power of Python!

Packt

Automate it!

Recipes to upskill your business

Put your life on autopilot and make your business processes efficient with the magic and power of Python!

Chetan Giridhar

BIRMINGHAM - MUMBAI

Automate it!

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

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

First published: January 2017

Production reference: 1240117

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.
ISBN 978-1-78646-051-6

www.packtpub.com

Credits

Author

Chetan Giridhar

Reviewer

Maurice HT Ling

Commissioning Editor

Edward Gordon

Acquisition Editor

Denim Pinto

Content Development Editor

Nikhil Borkar

Technical Editor

Madhunikita Sunil Chindarkar

Copy Editor

Muktikant Garimella

Project Coordinator

Ulhas Kambali

Proofreader

Safis Editing

Indexer

Rekha Nair

Graphics

Abhinash Sahu

Production Coordinator

Arvindkumar Gupta

About the Author

Chetan Giridhar is a technology leader and an open source evangelist. He is the author of *Learning Python Design Patterns, Second Edition*, has been an invited speaker at international PyCon conferences, and is an associate editor at the Python Papers journal. He takes keen interest in platform engineering, distributed systems, mobile app development, and real-time cloud applications. You can take a look at his experiments at `https://github.com/cjgiridhar` and his website `https://technobeans.com`.

In his current role as the chief technology officer, Chetan owns the product strategy and drives technology for CallHub. Prior to CallHub, he was associated with BlueJeans Networks and NetApp, where he worked on cloud, video, and enterprise storage products.

Chetan believes that the world is full of knowledge; he's always curious to learn new things and share them with open source community, friends, and colleagues. You can connect with him on LinkedIn (`https://www.linkedin.com/in/cjgiridhar`).

> *I'd like to thank Packt team to have worked with me on this book. This book wouldn't have been possible without the support and encouragement of my family(Jayant, Jyotsana, Deepti, and Pihu) and friends.*

About the Reviewer

Maurice HT Ling has been programming in Python since 2003. Having completed his PhD in Bioinformatics and BSc (Hons) in Molecular and Cell Biology at the University of Melbourne, he is an honorary fellow in the University of Melbourne, Australia. Maurice is the chief editor of Computational and Mathematical Biology, and coeditor of the Python Papers. Recently, Maurice cofounded the first synthetic biology startup in Singapore, AdvanceSyn Pte. Ltd, as a director (technology). He is also the principal partner of Colossus Technologies LLP, Singapore. His research interests lie in life--biological life, artificial life, and artificial intelligence--using computer science and statistics as the tools to understand life and its numerous aspects. In his free time, Maurice likes to read, enjoy a cup of coffee, write his personal journal, or philosophize on the various aspects of life. His website and LinkedIn profile are `http://maurice.vodien.com` and `http://www.linkedin.com/in/mauriceling`, respectively. Last but not least, Maurice has started and maintains a few open source projects (`https://github.com/mauriceling`), notably:

- Digital Organism Simulation Environment (DOSE)
- Collection of Python Algorithms and Data Structures (COPADS)
- Technical (Analysis) and Applied Statistics System (TAPPS)

Customer Feedback

Thank you for purchasing this Packt book. We take our commitment to improving our content and products to meet your needs seriously--that's why your feedback is so valuable. Whatever your feelings about your purchase, please consider leaving a review on this book's Amazon page. Not only will this help us, more importantly it will also help others in the community to make an informed decision about the resources that they invest in to learn. You can also review for us on a regular basis by joining our reviewers' club. If you're interested in joining, or would like to learn more about the benefits we offer, please contact us: `customerreviews@packtpub.com`.

Table of Contents

Preface

Business process automation is the technology-enabled automation of activities that accomplish a specific function or workflow. It is an organizational transformation that aims to drive process efficiency, consistency, and repeatability. Many business processes, such as HR onboarding, lead management, financial reports, and invoicing, among others, can be easily automated to achieve these goals across the organization using Python modules.

The Python recipes covered in the chapters of this book will help you gain knowledge and encourage you to think of automating your business flows. With a classical problem-solving pattern, we look at different areas, such as HR, marketing, customer support, and many more, where we can automate and innovate with the help of Python recipes.

What this book covers

`Chapter 1`, *Working with the Web*, talks about the interesting world of the World Wide Web and covers the numerous ways in which one can interact with the Web using Python modules.

It starts by covering the basics of HTTP Web requests and slowly takes the readers to more advanced topics such as web scraping and downloading content from the Web. The chapter also equips you with writing your own asynchronous web servers and helps you understand and build web automation.

In the last section, the chapter helps you automate lead generation--a classical situation that marketers face on a regular basis--using Python recipes.

`Chapter 2`, *Working with CSV and Excel Worksheets*, takes you through Python recipes that will help you simplify and automate redundant tasks with CSV and Excel sheets. Until computers became a daily part of our lives, office records were stored in files and managed on office desks. Thanks to the world of Excel sheets we can manage our data in a much better way.

The first part of this chapter helps you perform read/write operations on CSV files. It also helps you write your own CSV dialects. Don't know what we mean by dialects? Well, find it out in this chapter! You will also learn how to automate the management of employee information, an integral part of HR processes, with CSV files and Python code.

The second part of the chapter equips you with the knowledge of performing operations such as retrieving and inserting data in Excel worksheets. It also covers some advanced operations, such as formatting cells, performing mathematical tasks with formulas, and inserting charts. Lastly, with a nice example, it explains how you could automate the income statement analysis across different years for a finance team.

`Chapter 3`, *Getting Creative with PDF Files and Documents*, talks about how Word documents and PDF files have become some of the most commonly used file formats by business professionals and how you can automate mundane tasks in PDFs and Word documents using Python. Want to send an invoice to your customer or send a set of requirements to your vendor? Businesses often end up using PDF files and Word documents for such needs.

This chapter starts by covering operations that you can perform with PDF files using Python recipes. You will gain knowledge on generating and reading PDF files, copying them, and even manipulating them to build your own header and footer formats. Do you know you can merge many PDF files with a simple Python recipe? Would you like to automate the process of generating pay slips for your organization? Sounds interesting? Then this chapter is definitely for you!

This chapter also takes you on a journey of working with Word documents. It helps you build knowledge on reading and writing data into Word files. Adding tables, images, charts; you name it, and this chapter covers it. If that’s not enough, this chapter also takes an example from HR processes and helps you build a customized new-hire orientation program schedule for new employees based on their Business Unit.

`Chapter 4`, *Playing with SMS and Voice Notifications*, opens up a whole new world of automation with SMS and voice notifications. It starts with a nice introduction to cloud telephony and covers its practical use cases.

In the initial section of this chapter, we discuss how SMS text messages are useful in certain situations. You will learn how to send SMS messages and receive incoming texts with Python recipes. The section on SMS notifications ends by showing how one could automate a customer service process for a Domino’s Pizza store.

The chapter also covers voice notifications in detail; you will be familiarized with voice workflows, such as sending voice messages and receiving incoming voice calls with Python code snippets. And did you know you could automate customer support by building our own contact center? Interested? You've got to take a look at this chapter.

`Chapter 5`, *Fun with E-mails*, covers interesting Python recipes, such as sending e-mail messages, beautifying e-mail content with MIME, and even working with attachments. E-mails have been ubiquitous for the last 2-3 decades; you see them everywhere! You are so used to working with them and for so many reasons. But did you ever realize you could manipulate your inbox with Python snippets?

This chapter will also help you fetch and read e-mail conversations and clear up your inbox by deleting messages. Interested in adding labels to select messages, or would you like to understand more about e-mail encryption? Well, this chapter covers it for you. And, of course, the chapter ends with an example of automating customer support flows with a Python recipe.

`Chapter 6`, *Presentations and More*, explores the various ways in which you can generate presentations of your own in an automated way using Python. This chapter shows how you can create a new presentation and add content or slides to it. It also shows you ways of reading or updating existing presentations and inserting charts, tables, and pictures—basically, all the operations you need. And, of course, we automate a weekly sales report for a sales manager using Python recipes. This one's dedicated to all the sales managers out there.

`Chapter 7`, *Power of APIs*, takes you on a journey to the interesting world of APIs. APIs are a critical part of the World Wide Web today. Talking across services, sharing information, and many other operations rely on APIs and Webhooks.

The chapter starts with an introduction to REST APIs, covers the fundamentals of the REST philosophy, and provides you with the knowledge required to develope your own APIs. It also shows how you could automate the scheduling of product updates on social media--an essential use case for marketing teams--with Python recipes and Twitter REST APIs.

In the next section, the chapter covers Webhooks, a critical aspect of the Web today. It helps you implement Webhooks and manifests how a business professional can leverage Web hooks to automate lead management using Python recipes.

`Chapter 8`, *Talking to Bots*, takes you to the brand new world of bots. It starts by classifying bots based on their capabilities and shows how you could build and use bots on apps such as Telegram.

The chapter also gives you a brief introduction to the concepts of stateless and stateful, and it gives you the taste of integrating artificial intelligence algorithms in bots. In the end, the chapter takes an example of a book publishing website and shows how bots can be used to achieve relevant interactions with customers--a problem that the customer success teams deal with on a day-to-day basis.

`Chapter 9`, *Imagining Images*, provides you a jumpstart for working with images. It shows you how to convert your images to different formats (considering compression), resize and crop images, and also how to generate thumbnails with Python. I think you're already thinking of some use cases.

Not just that, it also you provides you with a foundation of finding the difference between images and comparing them, which can be very useful for you to build an image-based search algorithm. Lastly, the chapter motivates you to move your company to a paperless mode by automating the process of scanning documents and indexing them using Python.

`Chapter 10`, *Data Analysis and Visualizations*, begins with an introduction to the data analysis process and covers essential aspects in a simplistic way. It shows how you could read and select relevant data with techniques such as filtering and data aggregation.

It also helps you interpret data and generate insights with visualizations using Python recipes. Lastly, the chapter covers a business use case of analyzing social media data and generating insights for a magazine article. Interesting use case, right? You must read the chapter to know more.

`Chapter 11`, *Time in the Zone*, covers a nice set of Python recipes to work with date and time objects. It shows you how you could add time or days to your date, compare dates, and represent date and time in multiple formats. It also teaches you how to work with daylight savings and perform time zone calculations with Python.

In the end, the chapter takes an example of automated invoicing and covers issues with time zones to highlight the need for considering time zones while automating business processes.

What you need for this book

This books covers Python recipes that aim to help you automate business processes. All you need is a decent computer configuration (1 GB RAM, 10 GB HDD) with Python v2.7.10 installed.

I'm sure you will find at least a couple of use cases that you can automate with the ideas covered in this book. So, keep looking for possibilities of automation in your area of expertise in the office, and surely, the Python recipes in this book will be very helpful.

Who this book is for

Automate It! is a book for business professionals and developers. It gives you a great selection of recipes to automate your business processes and development tasks with Python. It provides a platform for you to automate repetitive and time-consuming business tasks and make them efficient.

As the book also targets business professionals from HR, sales, marketing, and customer support backgrounds, technical topics are covered in a detailed way so that they can understand technology and make use of Python recipes to put their life on autopilot.

Sections

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

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

Getting ready

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

How to do it…

This section contains the steps required to follow the recipe.

How it works…

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

There's more…

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

See also

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

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We got the tree with the help of the `fromstring()` method that converts the contents of the page (string format) to the HTML format."

A block of code is set as follows:

```
try:
    r = requests.get("http://www.google.com/")
except requests.exceptions.RequestException as e:
    print("Error Response:", e.message)
```

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

```
pip install -U requests
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Now that we have entered **Email** and **Password**, the last thing to do is submit the form and click on the **Log In** button."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

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

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

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

Customer support

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

Downloading the example code

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

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the **Search** box. Please note that you need to be logged in to your Packt account.

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

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

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Automate-it`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Downloading the color images of this book

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

Errata

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

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

Piracy

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

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

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

Questions

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

1
Working with the Web

Can you image a life without the Internet? For almost everything, right from exchanging information to ordering food, we rely heavily on the Internet today. Let's go through the interesting world of the World Wide Web and cover numerous ways with which we can interact with it using Python modules.

In this chapter, we will cover the following recipes:

- Making HTTP requests
- A brief look at web scraping
- Parsing and extracting web content
- Downloading content from the Web
- Working with third-party REST APIs
- Asynchronous HTTP server in Python
- Web automation with selenium bindings
- Automating lead generation with web scraping

Introduction

Internet has made life so easy that sometimes you just don't realize the power of it. Checking out your friend's status, calling your parents, responding to an important business e-mail, or playing a game–we rely on the **World Wide Web** (**WWW**) today for almost everything.

Thankfully, Python has a rich set of modules that help us perform various tasks on the Web. Phew! Not only could you make simple HTTP requests retrieve data from websites or download pages and images, you could also parse the page content to gather information and analyze it to generate meaningful insights with Python. And wait; did I mention that you could spawn a browser in an automated fashion to perform a daily mundane task?

The recipes in this chapter will primarily focus on the Python modules that can be treated as the tool of choice while performing the preceding operations on the Web. Specifically, we will focus on the following Python modules in this chapter:

- `requests` (`http://docs.python-requests.org/en/master/`)
- `urllib2` (`https://docs.python.org/2/library/urllib2.html`)
- `lxml` (`https://pypi.python.org/pypi/lxml`)
- `BeautifulSoup4` (`https://pypi.python.org/pypi/beautifulsoup4`)
- `selenium` (`http://selenium-python.readthedocs.org/`)

> While the recipes in this chapter will give you an overview of how to interact with the Web using Python modules, I encourage you to try out and develop code for multiple use cases, which will benefit you as an individual and your project on an organizational scale.

Making HTTP requests

Throughout the following recipes in this chapter, we will use Python v2.7 and the `requests` (v2.9.1) module of Python. This recipe will show you how to make HTTP requests to web pages on the Internet.

But before going there, let's understand the **Hypertext Transfer Protocol** (**HTTP**) in brief. HTTP is a stateless application protocol for data communication on the WWW. A typical HTTP Session involves a sequence of request or response transactions. The client initiates a **TCP** connection to the Server on a dedicated IP and Port; when the Server receives the request, it responds with the response code and text. HTTP defines request methods (HTTP verbs like `GET`, `POST`), which indicate the desired action to be taken on the given Web URL.

In this recipe, we'll learn how to make HTTP `GET`/`POST` requests using Python's `requests` module. We'll also learn how to POST `json` data and handle HTTP exceptions. Cool, let's jump in.

Getting ready

To step through this recipe, you will need to install Python v2.7. Once installed, you will need to install Python `pip`. **PIP** stands for **Pip Installs Packages** and is a program that can be used to download and install the required Python packages on your computer. Lastly, we'll need the `requests` module to make HTTP requests.

We will start by installing the `requests` module (I'll leave the Python and `pip` installation for you to perform on your machine, based on your operating system). No other prerequisites are required. So, hurry up and let's get going!

How to do it…

1. On your Linux/Mac computer, go to Terminal and run the following command:

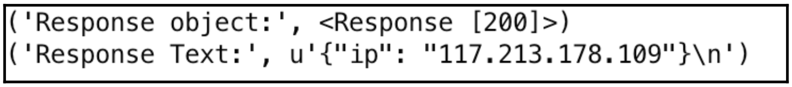
   ```
   pip install -U requests
   ```

   You only need to use `sudo` if you don't have permissions to Python site packages, else `sudo` is not required.

2. The following code helps you make a HTTP `GET` request with Python's `requests` module:

   ```
   import requests r =
   requests.get('http://ip.jsontest.com/')
   print("Response object:", r)
   print("Response Text:", r.text)
   ```

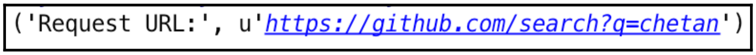
3. You will observe the following output:



```
('Response object:', <Response [200]>)
('Response Text:', u'{"ip": "117.213.178.109"}\n')
```

4. Creating a HTTP `GET` request with data payload is also trivial with requests. The following code helps you in achieving this. This is how you can also check the URL request that will be sent:

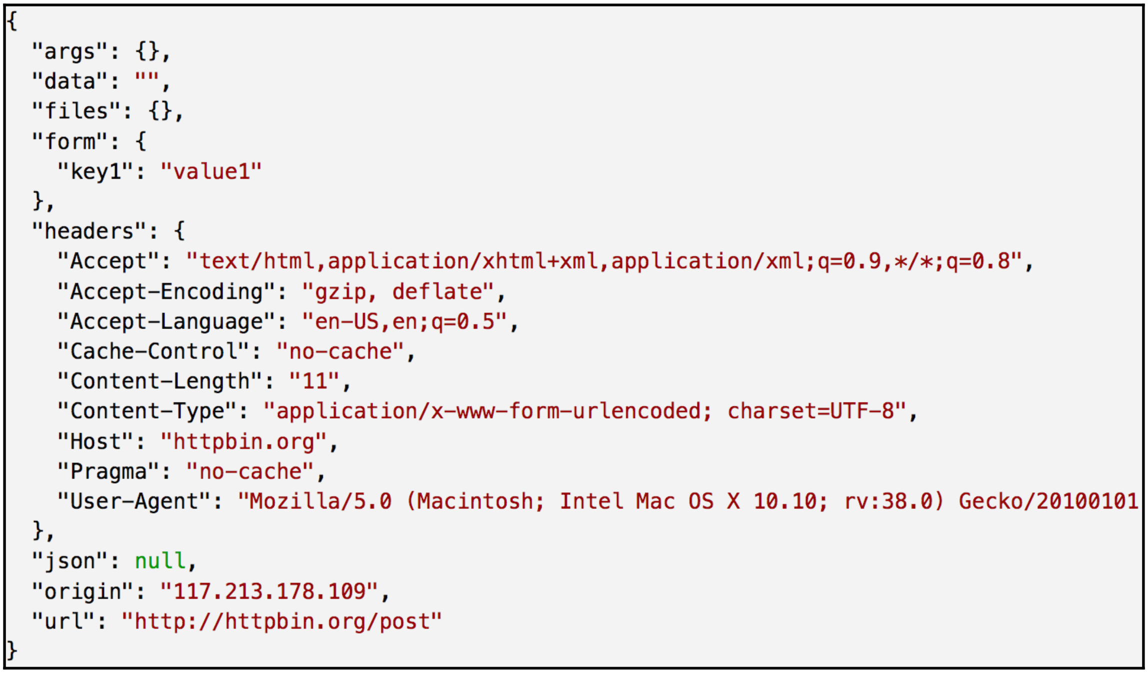
   ```
   payload = {'q': 'chetan'} r =
   requests.get('https://github.com/search', params=payload)
   print("Request URL:", r.url)
   ```



```
('Request URL:', u'https://github.com/search?q=chetan')
```

5. Let's now make a HTTP `POST` call using the `requests` module. This is similar to filling up and posting a login or signup form on a website:

   ```
   payload = {'key1': 'value1'} r =
   requests.post("http://httpbin.org/post", data=payload)
   print("Response text:", r.json())
   ```



```
{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "key1": "value1"
  },
  "headers": {
    "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
    "Accept-Encoding": "gzip, deflate",
    "Accept-Language": "en-US,en;q=0.5",
    "Cache-Control": "no-cache",
    "Content-Length": "11",
    "Content-Type": "application/x-www-form-urlencoded; charset=UTF-8",
    "Host": "httpbin.org",
    "Pragma": "no-cache",
    "User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:38.0) Gecko/20100101
  },
  "json": null,
  "origin": "117.213.178.109",
  "url": "http://httpbin.org/post"
}
```

6. Handling errors and exceptions is also very convenient with requests. The following code snippet shows an example of error handling. If you run this code without an Internet connection on your machine, it will result in an exception. The exception handler catches the exception and states that it failed to establish a new connection, as expected:

```
try:
    r = requests.get("http://www.google.com/")
except requests.exceptions.RequestException as e:
    print("Error Response:", e.message)
```

How it works…

In the this recipe, we looked at how to make different types of HTTP requests with Python's `requests` module. Let's look at how this code works:

- In the first example, we made a `GET` request to `http://ip.jsontest.com` and got the response code and response text. It returns the current IP address of our computer on the Internet.
- In the second example, we made a HTTP `GET` request with the payload data. Look how the request URL contains `?q=chetan`, and it searches all the repositories by the name, Chetan, on GitHub.
- Next, we made a `POST` request with the payload data being `{'key1', 'value1'}`. This is like submitting an online form, as we observed in the *How to do it* section.
- The `requests` module has a `Response` object, `r`, which includes various methods. These methods help in extracting response, status code and other information required while working with the Web:
    - `r.status_code` – Returns the response code
    - `r.json()` – Converts the response to `.json` format
    - `r.text` – Returns the response data for the query
    - `r.content` – Includes the HTML and XML tags in the response content
    - `r.url` – Defines the Web URL of the request made
- We also looked at the exception handling with the `requests` module, wherein, if there was no Internet, an exception occurred and the `requests` module could easily catch this exception. This was achieved with the `requests.exceptions` class of the `requests` module.

There's more…

Cool, that was neat! Making HTTP requests on the Web is just the beginning. There's still more in terms of what we can do with the Web, such as working with page contents. So, let's see what's next.

A brief look at web scraping

Before we learn how to perform **web scraping**, let's understand what scraping means. In the Web world, scraping is a way to sift through the pages of a website with the intention of extracting the required information in the said format with the help of a computer program. For example, if I want to get the title and date of all the articles published on a blog, I could write a program to scrape through the blog, get the required data, and store it in a database or a flat file, based on the requirement.

Web scraping is often confused with web crawling. The **web crawler** is a bot that systematically browses the Web with the purpose of web indexing and is used by search engines to index web pages so that users can search the Web more effectively.

But scraping is not easy. The data, which is interesting to us, is available on a blog or website in a particular format, say XML tags or embedded in HTML tags. So, it is important for us to know the format before we begin extracting the data we need. Also, the web scraper should know the format in which the extracted data needs to be stored in order to act on it later. It is also important to understand that the scraping code will fail should the HTML or XML format change, even though the browser display may be the same.

Legality of web scraping

Web scraping has always been under the scanner in legal terms. Can you do web scraping? How legal or ethical is it? Can we use the data obtained from scraping for profit?

This subject has been under a lot of discussion, but at a high level, you may get into issues with web scraping if you scrape the Web for copyright information, violate the Computer Fraud and Abuse Act, or violate a website's terms of service. For instance, if you're scraping the Web to get public data, you should still be fine. However, it is very contextual and you need to be careful about what you're scraping and how you are using the data.

Here are a few pointers on the Web on data scraping:

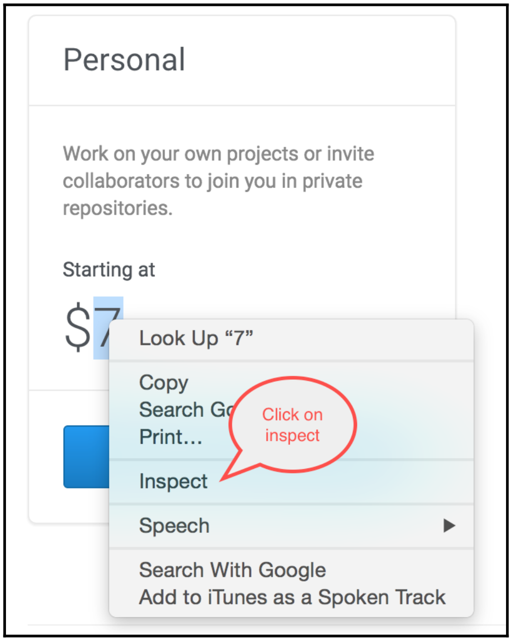
- `https://en.wikipedia.org/wiki/Web_scraping#Legal_issues`
- `https://www.quora.com/What-is-the-legality-of-web-scraping`

Getting ready

We take an example of pricing data from the `https://github.com/` website to demonstrate web scraping with Python. This is a really trivial example but gets us up to speed with scraping. Let's get started and scrape some interesting data with this Python recipe.

How to do it...

1. Open the Google Chrome browser on your computer and open the `https://github.com/pricing/` web page. On this page, you will notice multiple pricing plans namely, **Personal**, **Organization**, and **Enterprise**.
2. Now, on your browser, right-click on the pricing of the **Personal** plan and click on the **Inspect** element, as shown in the following screenshot:



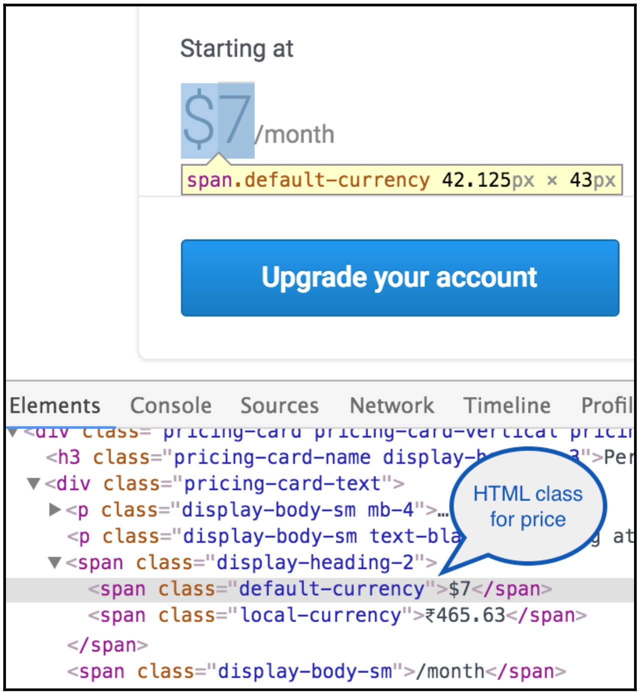
Personal

Work on your own projects or invite collaborators to join you in private repositories.

Starting at

$7

3. Once you click on **Inspect**, the Chrome browser's console log opens up, which will help you understand the HTML structure of GitHub's pricing page, as follows:



```
▼<div class="pricing-card pricing-card-vertical pricin
   <h3 class="pricing-card-name display-h        3">Per
 ▼<div class="pricing-card-text">
   ▶<p class="display-body-sm mb-4">…
    <p class="display-body-sm text-bla                 g at
   ▼<span class="display-heading-2">
      <span class="default-currency">$7</span>
      <span class="local-currency">₹465.63</span>
    </span>
    <span class="display-body-sm">/month</span>
```

4. If you look at the highlighted HTML `span` – `<span class="default-currency">$7</span>`, you'll know that this web page uses the `default-currency` class to list down the pricing of plans. We'll now use this property to extract the prices of multiple GitHub plans.
5. But before doing that, let's install the Python module, `lxml`, which will be needed to extract content from the preceding HTML document. Install the `lxml` and `requests` modules:

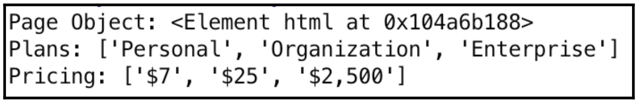
```
pip install lxml
pip install requests
```

6. Now, open your favorite editor and type this code snippet:

```
from lxml import html
import requests
```

```
page = requests.get('https://github.com/pricing/')
tree = html.fromstring(page.content)
print("Page Object:", tree)
plans = tree.xpath('//h2[@class="pricing-card-name
alt-h3"]/text()')
pricing = tree.xpath('//span[@class="default-
currency"]/text()')
print("Plans:", plans, "\nPricing:", pricing)
```

7. If you look at the preceding code, we used the `default-currency` class and `pricing-card-name display-heading-3` to get the pricing and pricing plan. If you run the code snippet, the output of the program will be as follows:



```
Page Object: <Element html at 0x104a6b188>
Plans: ['Personal', 'Organization', 'Enterprise']
Pricing: ['$7', '$25', '$2,500']
```

> With web scrapping you will see issues when the HTML tags for the web content has changed. For instance, if a CSS class name gets changed or an anchor is replaced with a button, the scraping code may not fetch the data you need. So, make sure you change your Python code accordingly.

How it works…

As we discussed earlier, we need to find out an appropriate way of extracting information. So, in this example, we first got the HTML tree for the `https://github.com/pricing/` page. We got the tree with the help of the `fromstring()` method that converts the contents of the page (string format) to the HTML format.

Then, using the `lxml` module and the `tree_xpath()` method, we looked for the `default-currency` class and `pricing-card-name display-heading-3` to get the pricing and pricing plans.

See how we used the complete XPath, `h3[@class='class-name']`, to locate the pricing plans and the `//span[@class="default-currency"]` XPath to select the actual pricing data. Once the elements were selected, we printed the text data that was returned to us as a Python list.

That's it; we scraped the GitHub page for the required data. Nice and simple.

There's more…

You learnt what web scrapers are, and how they go ahead and extract interesting information from the Web. You also understood how they are different from web crawlers. But then, there's always something more!

Web scraping involves extraction, which cannot happen until we parse the HTML content from the web page to get the data interesting to us. In the next recipe, we'll learn about parsing HTML and XML content in detail.

Parsing and extracting web content

Well, now we're confident about making HTTP requests to multiple URLs. We also looked at a simple example of web scraping.

But WWW is made up of pages with multiple data formats. If we want to scrape the Web and make sense of the data, we should also know how to parse different formats in which data is available on the Web.

In this recipe, we'll discuss how to s.

Getting ready

Data on the Web is mostly in the HTML or XML format. To understand how to parse web content, we'll take an example of an HTML file. We'll learn how to select certain HTML elements and extract the desired data. For this recipe, you need to install the `BeautifulSoup` module of Python. The `BeautifulSoup` module is one of the most comprehensive Python modules that will do a good job of parsing HTML content. So, let's get started.

How to do it…

1. We start by installing `BeautifulSoup` on our Python instance. The following command will help us install the module. We install the latest version, which is `beautifulsoup4`:

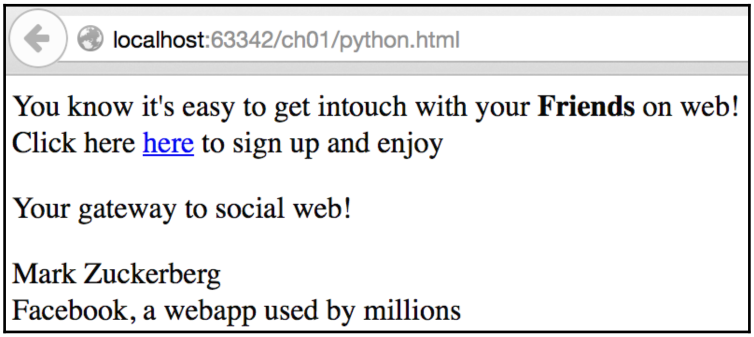
   ```
   pip install beautifulsoup4
   ```

2. Now, let's take a look at the following HTML file, which will help us learn how to parse the HTML content:

   ```
   <html xmlns="http://www.w3.org/1999/html">
   <head>
       <title>Enjoy Facebook!</title>
   </head>
   <body>
       <p>
         <span>You know it's easy to get intouch with
         your <strong>Friends</strong> on web!<br></span>
         Click here <a href="https://facebook.com">here</a>
         to sign up and enjoy<br>
       </p>
       <p class="wow"> Your gateway to social web! </p>
       <div id="inventor">Mark Zuckerberg</div>
       Facebook, a webapp used by millions
   </body>
   </html>
   ```

3. Let's name this file as `python.html`. Our HTML file is hand-crafted so that we can learn the multiple ways of parsing it to get the required data from it. `Python.html` has typical HTML tags given as follows:
   - `<head>` – It is the container of all head elements like `<title>`.
   - `<body>` – It defines the body of the HTML document.
   - `<p>` – This element defines a paragraph in HTML.
   - `<span>` – It is used to group inline elements in a document.
   - `<strong>` – It is used to apply a bold style to the text present under this tag.
   - `<a>` – It represents a hyperlink or anchor and contains `<href>` that points to the hyperlink.
   - `<class>` – It is an attribute that points to a class in a style sheet.
   - `<div id>` – It is a container that encapsulates other page elements and divides the content into sections. Every section can be identified by attribute `id`.

4. If we open this HTML in a browser, this is how it'll look:



You know it's easy to get intouch with your **Friends** on web!
Click here here to sign up and enjoy

Your gateway to social web!

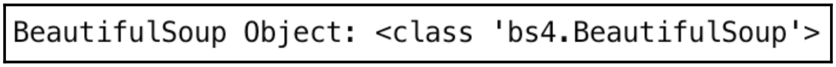
Mark Zuckerberg
Facebook, a webapp used by millions

5. Let's now write some Python code to parse this HTML file. We start by creating a `BeautifulSoup` object.

We always need to define the parser. In this case we used `lxml` as the parser. The parser helps us read files in a designated format so that querying data becomes easy.

```
import bs4
myfile = open('python.html')
soup = bs4.BeautifulSoup(myfile, "lxml")
#Making the soup
print "BeautifulSoup Object:", type(soup)
```

The output of the preceding code is seen in the following screenshot:

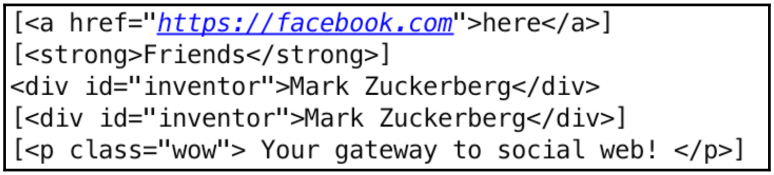


```
BeautifulSoup Object: <class 'bs4.BeautifulSoup'>
```

6. OK, that's neat, but how do we retrieve data? Before we try to retrieve data, we need to select the HTML elements that contain the data we need.
7. We can select or find HTML elements in different ways. We could select elements with ID, CSS, or tags. The following code uses `python.html` to demonstrate this concept:

```
#Find Elements By tags
print soup.find_all('a')
print soup.find_all('strong')
#Find Elements By id
print soup.find('div', {"id":"inventor"})
print soup.select('#inventor')
#Find Elements by css print
soup.select('.wow')
```

The output of the preceding code can be viewed in the following screenshot:

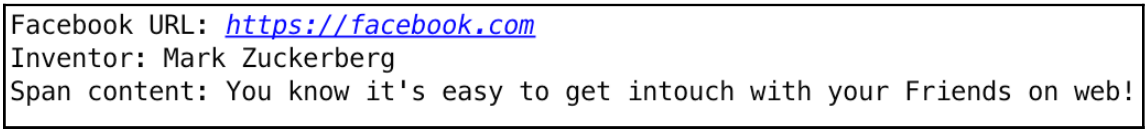


```
[<a href="https://facebook.com">here</a>]
[<strong>Friends</strong>]
<div id="inventor">Mark Zuckerberg</div>
[<div id="inventor">Mark Zuckerberg</div>]
[<p class="wow"> Your gateway to social web! </p>]
```

8. Now let's move on and get the actual content from the HTML file. The following are a few ways in which we can extract the data of interest:

```
print "Facebook URL:", soup.find_all('a')[0]['href']
print "Inventor:", soup.find('div', {"id":"inventor"}).text
print "Span content:", soup.select('span')[0].getText()
```

The output of the preceding code snippet is as follows:



```
Facebook URL: https://facebook.com
Inventor: Mark Zuckerberg
Span content: You know it's easy to get intouch with your Friends on web!
```

Whoopie! See how we got all the text we wanted from the HTML elements.

How it works…

In this recipe, you learnt the skill of finding or selecting different HTML elements based on ID, CSS, or tags.

In the second code example of this recipe, we used `find_all('a')` to get all the anchor elements from the HTML file. When we used the `find_all()` method, we got multiple instances of the match as an array. The `select()` method helps you reach the element directly.

We also used `find('div', <divId>)` or `select(<divId>)` to select HTML elements by `div Id`. Note how we selected the `inventor` element with `div` ID `#inventor` in two ways using the `find()` and `select()` methods. Actually, the select method can also be used as `select(<class-name>)` to select HTML elements with a CSS class name. We used this method to select element `wow` in our example.

In the third code example, we searched for all the anchor elements in the HTML page and looked at the first index with `soup.find_all('a')[0]`. Note that since we have only one anchor tag, we used the index 0 to select that element, but if we had multiple anchor tags, it could be accessed with index 1. Methods like `getText()` and attributes like `text` (as seen in the preceding examples) help in extracting the actual content from the elements.

There's more…

Cool, so we understood how to parse a web page (or an HTML page) with Python. You also learnt how to select or find HTML elements by ID, CSS, or tags. We also looked at examples of how to extract the required content from HTML. What if we want to download the contents of a page or file from the Web? Let's see if we can achieve that in our next recipe.

Downloading content from the Web

So, in the earlier recipe, we saw how to make HTTP requests, and you also learnt how to parse a web response. It's time to move ahead and download content from the Web. You know that the WWW is not just about HTML pages. It contains other resources, such as text files, documents, and images, among many other formats. Here, in this recipe, you'll learn ways to download images in Python with an example.

Getting ready

To download images, we will need two Python modules, namely `BeautifulSoup` and `urllib2`. We could use the `requests` module instead of `urrlib2`, but this will help you learn about `urllib2` as an alternative that can be used for HTTP requests, so you can boast about it.

How to do it…

1. Before starting this recipe, we need to answer two questions. What kind of images would we like to download? From which location on the Web do I download the images? In this recipe, we download *Avatar* movie images from Google (`https://google.com`) images search. We download the top five images that match the search criteria. For doing this, let's import the Python modules and define the variables we'll need:

```
        from bs4 import BeautifulSoup
        import re
        import urllib2
        import os
        ## Download paramters
        image_type = "Project"
        movie = "Avatar"
        url =
"https://www.google.com/search?q="+movie+"&source=lnms&tbm=isch"
```

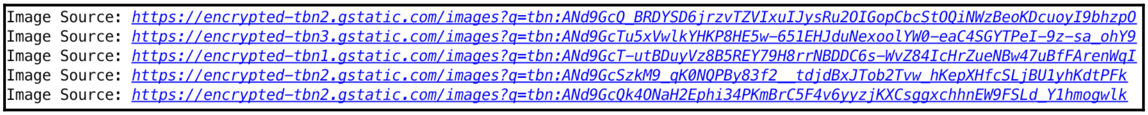
2. OK then, let's now create a `BeautifulSoup` object with URL parameters and appropriate headers. See the use of `User-Agent` while making HTTP calls with Python's `urllib` module. The `requests` module uses its own `User-Agent` while making `HTTP` calls:

```
header = {'User-Agent': 'Mozilla/5.0'}
soup = BeautifulSoup(urllib2.urlopen
(urllib2.Request(url,headers=header)))
```

3. Google images are hosted as static content under the domain name `http://www.gstatic.com/`. So, using the `BeautifulSoup` object, we now try to find all the images whose source URL contains `http://www.gstatic.com/`. The following code does exactly the same thing:

```
images = [a['src'] for a in soup.find_all("img", {"src":
re.compile("gstatic.com")})][:5]
for img in images:
print "Image Source:", img
```

The output of the preceding code snippet can be seen in the following screenshot. Note how we get the image source URL on the Web for the top five images:

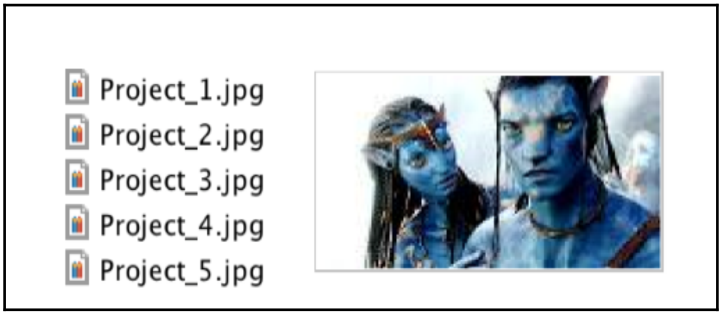


```
Image Source: https://encrypted-tbn2.gstatic.com/images?q=tbn:ANd9GcQ_BRDYSD6jrzvTZVIxuIJysRu2OIGopCbcStOQiNWzBeoKDcuoyI9bhzpO
Image Source: https://encrypted-tbn3.gstatic.com/images?q=tbn:ANd9GcTu5xVwlkYHKP8HE5w-651EHJduNexoolYW0-eaC4SGYTPeI-9z-sa_ohY9
Image Source: https://encrypted-tbn1.gstatic.com/images?q=tbn:ANd9GcT-utBDuyVz8B5REY79H8rrNBDDC6s-WvZ84IcHrZueNBw47uBfFArenWqI
Image Source: https://encrypted-tbn2.gstatic.com/images?q=tbn:ANd9GcSzkM9_qK0NQPBy83f2__tdjdBxJTob2Tvw_hKepXHfcSLjBU1yhKdtPFk
Image Source: https://encrypted-tbn2.gstatic.com/images?q=tbn:ANd9GcQk4ONaH2Ephi34PKmBrC5F4v6yyzjKXCsggxchhnEW9FSLd_Y1hmogwlk
```

4. Now that we have the source URL of all the images, let's download them. The following Python code uses the `urlopen()` method to `read()` the image and downloads it onto the local file system:

```
for img in images:
  raw_img = urllib2.urlopen(img).read()
  cntr = len([i for i in os.listdir(".") if image_type in i]) + 1
f = open(image_type + "_"+ str(cntr)+".jpg", 'wb')
f.write(raw_img)
f.close()
```

5. When the images get downloaded, we can see them on our editor. The following snapshot shows the top five images we downloaded and `Project_3.jpg` looks as follows:



How it works…

So, in this recipe, we looked at downloading content from the Web. First, we defined the parameters for download. Parameters are like configurations that define the location where the downloadable resource is available and what kind of content is to be downloaded. In our example, we defined that we have to download *Avatar* movie images and, that too, from **Google**.

Then we created the `BeautifulSoup` object, which will make the URL request using the `urllib2` module. Actually, `urllib2.Request()` prepares the request with the configuration, such as headers and the URL itself, and `urllib2.urlopen()` actually makes the request. We wrapped the HTML response of the `urlopen()` method and created a `BeautifulSoup` object so that we could parse the HTML response.

Next, we used the soup object to search for the top five images present in the HTML response. We searched for images based on the `img` tag with the `find_all()` method. As we know, `find_all()` returns a list of image URLs where the picture is available on **Google**.

Finally, we iterated through all the URLs and again used the `urlopen()` method on URLs to `read()` the images. `Read()` returns the image in a raw format as binary data. We then used this raw image to write to a file on our local file system. We also added a logic to name the image (they actually auto-increment) so that they're uniquely identified in the local file system.

That's nice! Exactly what we wanted to achieve! Now let's up the ante a bit and see what else we can explore in the next recipe.

Working with third-party REST APIs

Now that we've covered ground on scraping, crawling, and parsing, it's time for another interesting work that we can do with Python, which is working with third-party APIs. I'd assume many of us are aware and might have a basic understanding of **REST API**. So, let's get started!

Getting ready

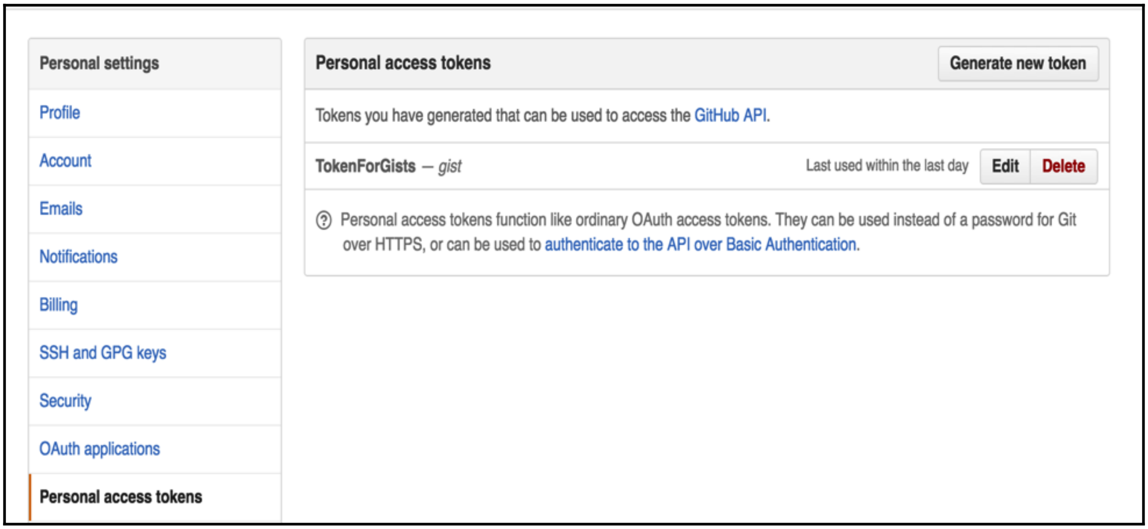
To demonstrate the understanding, we take the case of GitHub gists. Gists in GitHub are the best way to share your work, a small code snippet that helps your colleague or a small app with multiple files that gives an understanding of a concept. GitHub allows the creation, listing, deleting, and updating of gists, and it presents a classical case of working with GitHub REST APIs.

So, in this section, we use our very own `requests` module to make HTTP requests to GitHub REST API to create, update, list, or delete gists.

The following steps will show you how to work with GitHub REST APIs using Python.

How to do it…

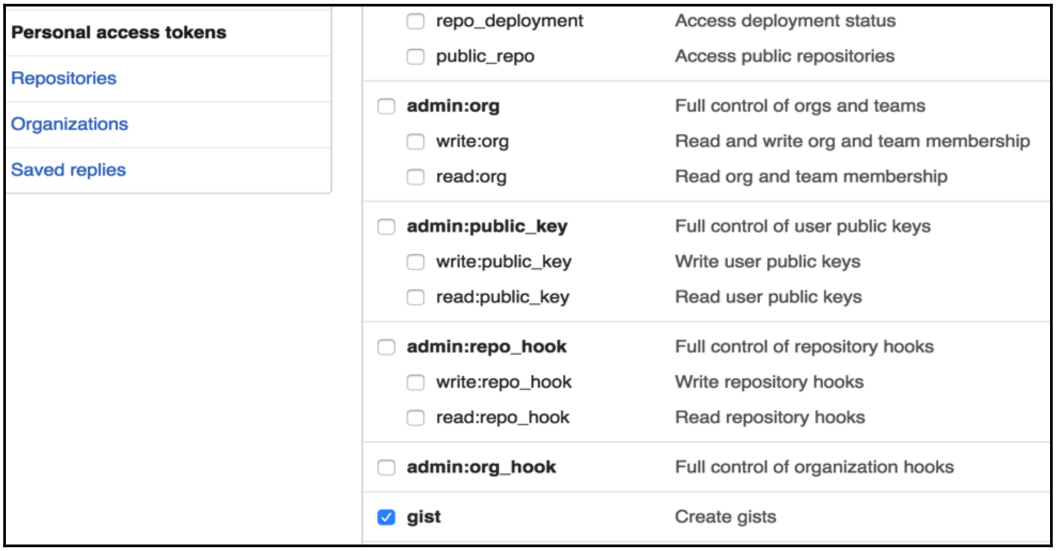
1. To work with GitHub REST APIs, we need to create a **Personal access token**. For doing that, log in to `https://github.com/` and browse to `https://github.com/settings/tokens` and click on **Generate new token**:



Tokens you have generated that can be used to access the GitHub API.

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to authenticate to the API over Basic Authentication.

2. You'll now be taken to the **New personal access token** page. Enter a description at the top of the page and check the **gists** option among the scopes given out. Note that scope represents the access for your token. For instance, if you just select **gists**, you can use GitHub APIs to work on the **gists** resource but not on other resources such as **repo** or users. For this recipe, the **gists** scope is just what we need:



Personal access tokens

Repositories

Organizations

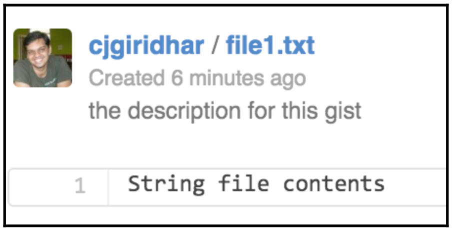
Saved replies

□ repo_deployment	Access deployment status
□ public_repo	Access public repositories
□ **admin:org**	Full control of orgs and teams
□ write:org	Read and write org and team membership
□ read:org	Read org and team membership
□ **admin:public_key**	Full control of user public keys
□ write:public_key	Write user public keys
□ read:public_key	Read user public keys
□ **admin:repo_hook**	Full control of repository hooks
□ write:repo_hook	Write repository hooks
□ read:repo_hook	Read repository hooks
□ **admin:org_hook**	Full control of organization hooks
☑ **gist**	Create gists

3. Once you click on **Generate token**, you'd be presented with a screen containing your personal access token. Keep this token confidential with you.
4. With the access token available, let's start working with APIs and create a new gist. With create, we add a new resource, and for doing this, we make an HTTP `POST` request on GitHub APIs, such as in the following code:

```
import requests
import json
BASE_URL = 'https://api.github.com'
Link_URL = 'https://gist.github.com'
username = '<username>' ## Fill in your github username
api_token = '<api_token>'  ## Fill in your token
header = {  'X-Github-Username': '%s' % username,
            'Content-Type': 'application/json',
            'Authorization': 'token %s' % api_token,
}
url = "/gists"
data ={
        "description": "the description for this gist",
        "public": True,
        "files": {
          "file1.txt": {
            "content": "String file contents"
          }
        }
      }
      r = requests.post('%s%s' % (BASE_URL, url),
          headers=header,
         data=json.dumps(data))
    print r.json()['url']
```

5. If I now go to my `gists` page on GitHub, I should see the newly created gist. And voila, it's available!



cjgiridhar / file1.txt
Created 6 minutes ago
the description for this gist

```
1   String file contents
```

6. Hey, we were successful in creating the gist with the GitHub APIs. That's cool, but can we now view this `gist`? In the preceding example, we also printed the URL of the newly created gist. It will be in the format, `https://gist.github.com/<username>/<gist_id>`. We now use this **gist_id** to get the details of the gist, which means we make a HTTP `GET` request on the **gist_id**:

```
import requests
import json
BASE_URL = 'https://api.github.com'
Link_URL =
'https://gist.github.com'

username = '<username>'
api_token = '<api_token>'
gist_id = '<gist id>'

header = { 'X-Github-Username': '%s' % username,
           'Content-Type': 'application/json',
           'Authorization': 'token %s' % api_token,
}
```

```
url = "/gists/%s" % gist_id
r = requests.get('%s%s' % (BASE_URL, url),
                  headers=header)
print r.json()
```

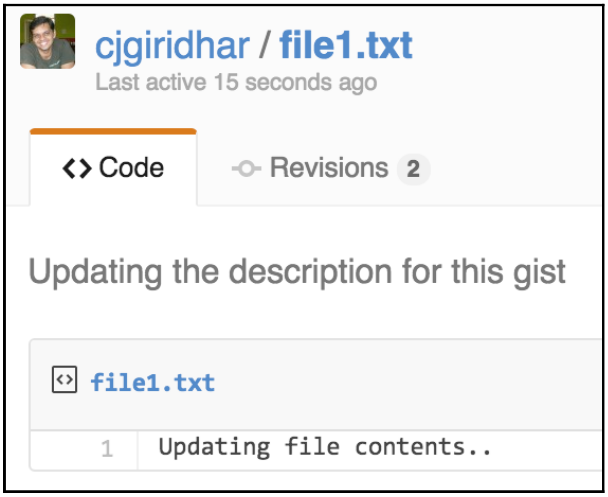
7. We created a new gist with the HTTP `POST` request and got the details of the gist with the HTTP `GET` request in the previous steps. Now, let's update this gist with the HTTP `PATCH` request.

> Many third-party libraries choose to use the `PUT` request to update a resource, but HTTP `PATCH` can also be used for this operation, as chosen by GitHub.

8. The following code demonstrates updating the gist:

```
import requests
import json

BASE_URL = 'https://api.github.com'
Link_URL = 'https://gist.github.com'

username = '<username>'
api_token = '<api_token>'
gist_id = '<gist_id>'

header = { 'X-Github-Username': '%s' % username,
           'Content-Type': 'application/json',
           'Authorization': 'token %s' % api_token,
}
data = {   "description": "Updating the description
           for this gist",
           "files": {
             "file1.txt": {
               "content": "Updating file contents.."
             }
           }
}
url = "/gists/%s" % gist_id
r = requests.patch('%s%s' %(BASE_URL, url),
                   headers=header,
                   data=json.dumps(data))
print r.json()
```

9. Now, if I look at my GitHub login and browse to this gist, the contents of the gist have been updated. Awesome! Don't forget to see the **Revisions** in the screenshot–see it got updated to revision **2**:



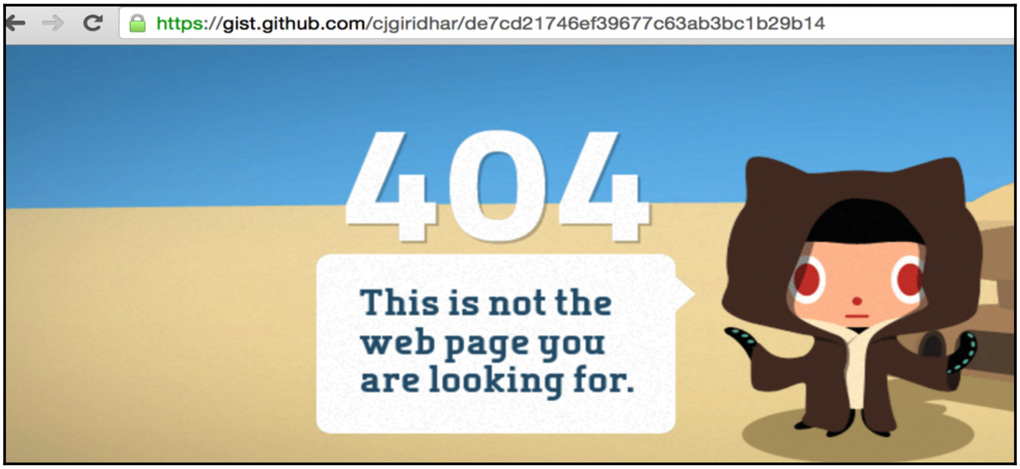
Updating the description for this gist

```
Updating file contents..
```

10. Now comes the most destructive API operation–yes deleting the gist. GitHub provides an API for removing the gist by making use of the HTTP `DELETE` operation on its `/gists/<gist_id>` resource. The following code helps us delete the `gist`:

```
import requests
import json
BASE_URL = 'https://api.github.com'
Link_URL = 'https://gist.github.com'
username = '<username>'
api_token = '<api_token>'
gist_id = '<gist_id>'

header = {  'X-Github-Username': '%s' % username,
            'Content-Type': 'application/json',
            'Authorization': 'token %s' % api_token,
}
url = "/gists/%s" % gist_id
r = requests.delete('%s%s' %(BASE_URL, url),
                    headers=header, )
```

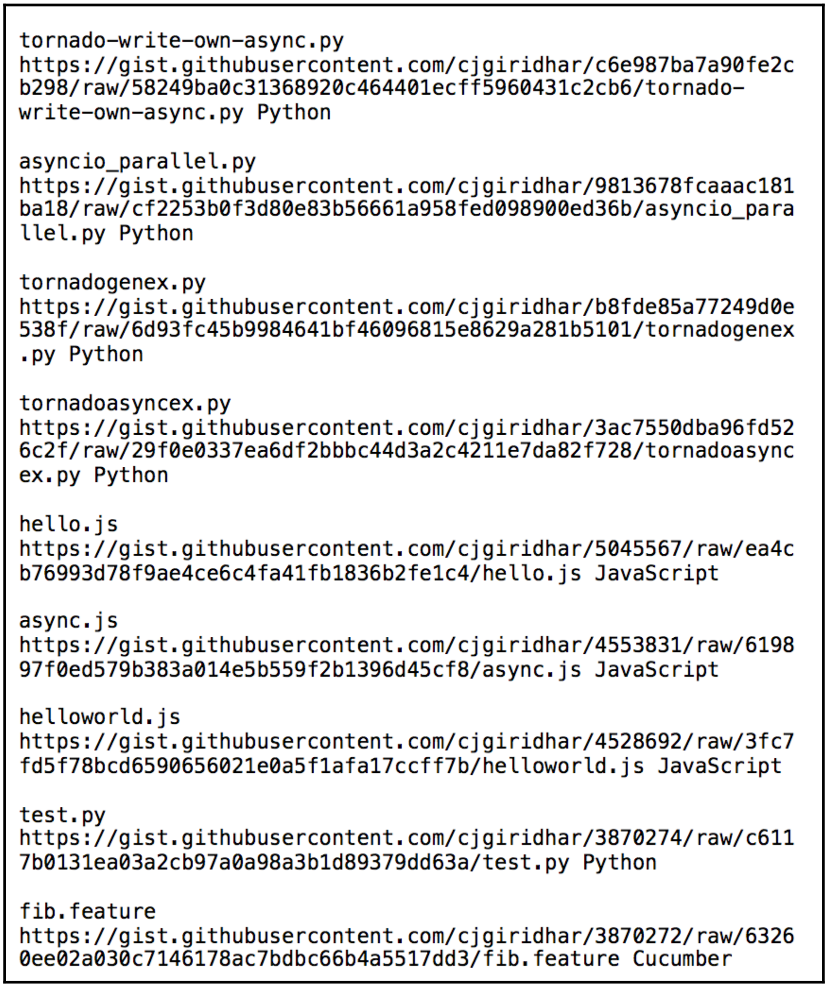
11. Let's quickly find out if the gist is now available on the GitHub website? We can do that by browsing the gist URL on any web browser. And what does the browser say? It says **404** resource not found, so we have successfully deleted the gist! Refer to the following screenshot:


https://gist.github.com/cjgiridhar/de7cd21746ef39677c63ab3bc1b29b14
404
This is not the web page you are looking for.

12. Finally, let's list all the gists in your account. For this we make an HTTP `GET` API call on the `/users/<username>/gists` resource:

```
import requests

BASE_URL = 'https://api.github.com'
Link_URL = 'https://gist.github.com'

username = '<username>'      ## Fill in your github username
api_token = '<api_token>'  ## Fill in your token

header = {  'X-Github-Username': '%s' % username,
            'Content-Type': 'application/json',
            'Authorization': 'token %s' % api_token,
}
url = "/users/%s/gists" % username
r = requests.get('%s%s' % (BASE_URL, url),
                  headers=header)
gists = r.json()
for gist in gists:
    data = gist['files'].values()[0]
    print data['filename'],
    data['raw_url'], data['language']
```

The output of the preceding code for my account is as follows:



```
tornado-write-own-async.py
https://gist.githubusercontent.com/cjgiridhar/c6e987ba7a90fe2c
b298/raw/58249ba0c31368920c464401ecff5960431c2cb6/tornado-
write-own-async.py Python

asyncio_parallel.py
https://gist.githubusercontent.com/cjgiridhar/9813678fcaaac181
ba18/raw/cf2253b0f3d80e83b56661a958fed098900ed36b/asyncio_para
llel.py Python

tornadogenex.py
https://gist.githubusercontent.com/cjgiridhar/b8fde85a77249d0e
538f/raw/6d93fc45b9984641bf46096815e8629a281b5101/tornadogenex
.py Python

tornadoasyncex.py
https://gist.githubusercontent.com/cjgiridhar/3ac7550dba96fd52
6c2f/raw/29f0e0337ea6df2bbbc44d3a2c4211e7da82f728/tornadoasync
ex.py Python

hello.js
https://gist.githubusercontent.com/cjgiridhar/5045567/raw/ea4c
b76993d78f9ae4ce6c4fa41fb1836b2fe1c4/hello.js JavaScript

async.js
https://gist.githubusercontent.com/cjgiridhar/4553831/raw/6198
97f0ed579b383a014e5b559f2b1396d45cf8/async.js JavaScript

helloworld.js
https://gist.githubusercontent.com/cjgiridhar/4528692/raw/3fc7
fd5f78bcd6590656021e0a5f1afa17ccff7b/helloworld.js JavaScript

test.py
https://gist.githubusercontent.com/cjgiridhar/3870274/raw/c611
7b0131ea03a2cb97a0a98a3b1d89379dd63a/test.py Python

fib.feature
https://gist.githubusercontent.com/cjgiridhar/3870272/raw/6326
0ee02a030c7146178ac7bdbc66b4a5517dd3/fib.feature Cucumber
```

How it works…

Python's `requests` module helps in making HTTP `GET`/`POST`/`PUT`/`PATCH` and `DELETE` API calls on GitHub's resources. These operations, also known as HTTP verbs in the REST terminology, are responsible for taking certain actions on the URL resources.

As we saw in the examples, the HTTP `GET` request helps in listing the gists, `POST` creates a new gist, `PATCH` updates a gist, and `DELETE` completely removes the gist. Thus, in this recipe, you learnt how to work with third-party REST APIs–an essential part of WWW today–using Python.

See also

There are many third-party applications that are written as REST APIs. You may want to try them out the same way we did for GitHub. For example, both Twitter and Facebook have great APIs and the documents are also easy to understand and use. Of course, they do have Python bindings.

Asynchronous HTTP server in Python

If you realize, many web applications that we interact with, are by default synchronous. A client connection gets established for every request made by the client and a callable method gets invoked on the server side. The server performs the business operation and writes the response body to the client socket. Once the response is exhausted, the client connection gets closed. All these operations happen in sequence one after the other–hence, synchronous.

But the Web today, as we see it, cannot rely on synchronous modes of operations only. Consider the case of a website that queries data from the Web and retrieves the information for you. (For instance, your website allows for integration with **Facebook** and every time a user visits a certain page of your website, you pull data from his **Facebook** account.) Now, if we develop this web application in a synchronous manner, for every request made by the client, the server would make an I/O call to either the database or over the network to retrieve information and then present it back to the client. If these I/O requests take a longer time to respond, the server gets blocked waiting for the response. Typically web servers maintain a thread pool that handles multiple requests from the client. If a server waits long enough to serve requests, the thread pool may get exhausted soon and the server will get stalled.

Solution? In comes the asynchronous ways of doing things!

Getting ready

For this recipe, we will use Tornado, an asynchronous framework developed in Python. It has support for both Python 2 and Python 3 and was originally developed at FriendFeed (`http://blog.friendfeed.com/`). Tornado uses a non-blocking network I/O and solves the problem of scaling to tens of thousands of live connections (`C10K problem`). I like this framework and enjoy developing code with it. I hope you'd too! Before we get into the *How to do it* section, let's first install tornado by executing the following command:

```
pip install -U tornado
```

How to do it…

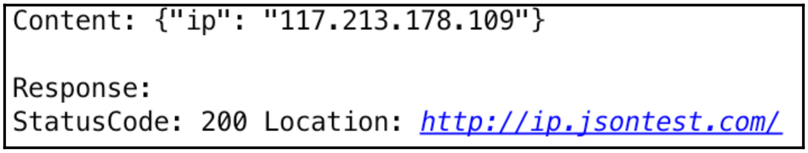
1. We're now ready to develop our own HTTP server that works on an asynchronous philosophy. The following code represents an asynchronous server developed in the `tornado` web framework:

```
 import tornado.ioloop
 import tornado.web
 import httplib2

 class AsyncHandler(tornado.web.RequestHandler):
     @tornado.web.asynchronous
     def get(self):
       http = httplib2.Http()
       self.response, self.content =
         http.request("http://ip.jsontest.com/", "GET")
       self._async_callback(self.response, self.content)

     def _async_callback(self, response, content):
     print "Content:", content
     print "Response:\nStatusCode: %s Location: %s"
       %(response['status'], response['content-location'])
     self.finish()
     tornado.ioloop.IOLoop.instance().stop()
 application = tornado.web.Application([
       (r"/", AsyncHandler)], debug=True)
if __name__ == "__main__":
  application.listen(8888)
  tornado.ioloop.IOLoop.instance().start()
```

2. Run the server as:

```
python tornado_async.py
```

3. The server is now running on port 8888 and ready to receive requests.
4. Now, launch any browser of your choice and browse to `http://localhost:8888/`. On the server, you'll see the following output:



```
Content: {"ip": "117.213.178.109"}

Response:
StatusCode: 200 Location: http://ip.jsontest.com/
```

How it works…

Our asynchronous web server is now up and running and accepting requests on port 8888. But what is asynchronous about this? In fact, tornado works on the philosophy of a single-threaded event loop. This event loop keeps polling for events and passes it on to the corresponding event handlers.

In the preceding example, when the app is run, it starts by running the `ioloop`. The `ioloop` is a single-threaded event loop and is responsible for receiving requests from the clients. We have defined the `get()` method, which is decorated with `@tornado.web.asynchronous`, which makes it asynchronous. When a user makes a HTTP GET request on `http://localhost:8888/`, the `get()` method is triggered that internally makes an I/O call to `http://ip.jsontest.com`.

Now, a typical synchronous web server would wait for the response of this I/O call and block the request thread. But tornado being an asynchronous framework, it triggers a task, adds it to a queue, makes the I/O call, and returns the thread of execution back to the event loop.

The event loop now keeps monitoring the task queue and polls for a response from the I/O call. When the event is available, it executes the event handler, `async_callback()`, to print the content and its response and then stops the event loop.

There's more…

Event-driven web servers such as tornado make use of kernel-level libraries to monitor for events. These libraries are `kqueue`, `epoll`, and so on. If you're really interested, you should do more reading on this. Here are a few resources:

- `https://linux.die.net/man/4/epoll`
- `https://www.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2`

Web automation with selenium bindings

In all the recipes so far, we had a dedicated URL to make HTTP requests, be it calling a REST API or downloading content from the Web. But then, there are services that don't have a defined API resource or need to log in to the Web to perform operations. In such cases, you don't have much control over the requests, as it is the same URL that serves multiple different content, based on the user session or cookie. Then what do we do?

Well, how about controlling the browser itself to achieve tasks in such scenarios? Controlling the browser itself? Interesting, isn't it?

Getting ready

For this recipe, we'll use Python's selenium module. Selenium (`http://www.seleniumhq.org/`) is a portable software framework for web applications and automates browser actions. You could automate mundane tasks with selenium. Selenium spawns a browser and helps you perform tasks as though a human is doing them. Selenium supports some of the most popularly used browsers like Firefox, Chrome, Safari, and Internet Explorer, among others. Let's take an example of logging in to Facebook with Python's selenium in this recipe.

How to do it…

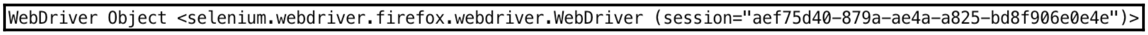
1. We start by installing selenium bindings for Python. Installing selenium can be done with the following command:

```
pip install selenium
```

2. Let's start by first creating a browser object. We use the Firefox browser for spawning the browser instance:

```
from selenium import webdriver browser =
webdriver.Firefox()
print "WebDriver Object", browser
```

3. The following screenshot shows how a selenium web driver object got created. It also has a unique session ID:

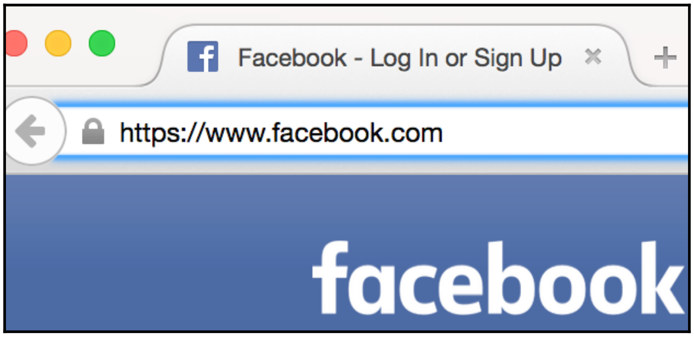


```
WebDriver Object <selenium.webdriver.firefox.webdriver.WebDriver (session="aef75d40-879a-ae4a-a825-bd8f906e0e4e")>
```

4. Next, we ask the browser to browse to the Facebook home page. The following code helps us achieve this:

```
browser.maximize_window()
browser.get('https://facebook.com')
```

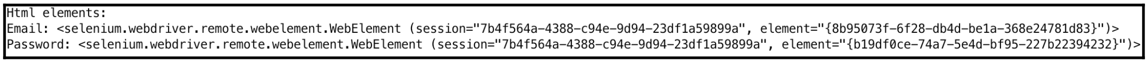
5. Once you run the preceding code, you will see a Firefox browser opened, and it connects to the Facebook login page, as in the following screenshot:



6. For the next step, we locate the e-mail and password elements and enter the appropriate data:

```
email = browser.find_element_by_name('email')
password = browser.find_element_by_name('pass')
print "Html elements:"
print "Email:", email, "\nPassword:", password
```

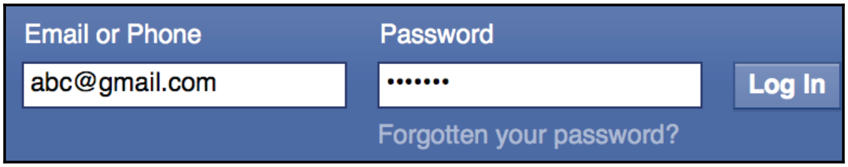
The output of the preceding code is as follows:



```
Html elements:
Email: <selenium.webdriver.remote.webelement.WebElement (session="7b4f564a-4388-c94e-9d94-23df1a59899a", element="{8b95073f-6f28-db4d-be1a-368e24781d83}")>
Password: <selenium.webdriver.remote.webelement.WebElement (session="7b4f564a-4388-c94e-9d94-23df1a59899a", element="{b19df0ce-74a7-5e4d-bf95-227b22394232}")>
```

7. Once we have selected the **Email** and **Password** text inputs, we now fill them with the correct **Email** and **Password**. The following code will enable entering **Email** and **Password**:

```
email.send_keys('abc@gmail.com') #Enter correct email
addresspassword.send_keys('pass123') #Enter correct password
```



8. Now that we have entered **Email** and **Password**, the last thing to do is submit the form and click on the **Log In** button. We do this by finding the element by ID and clicking on the element:

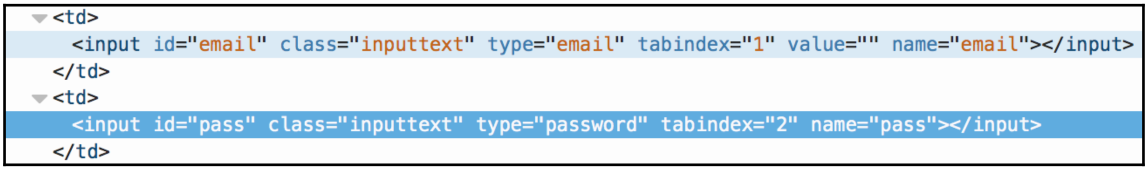
```
browser.find_element_by_id('loginbutton').click()
```

If you have entered the correct e-mail ID and password, you'd have logged in to Facebook!

How it works…

For this recipe, we used the selenium WebDriver Python APIs. **WebDriver** is the latest inclusion in selenium APIs and drives browsers natively like a user. It can drive locally or on a remote machine using the selenium server. In this example, we ran it on the local machine. Basically, the selenium server runs on a local machine on a default port 4444 and selenium WebDriver APIs interact with the selenium server to take actions on the browser.

In this recipe, we first created a WebDriver instance using the Firefox browser. We then used the WebDriver API to browse to the Facebook homepage. We then parsed the HTML page and located the **Email** and **Password** input elements. How did we find the elements? Yes, similar to what we did in the web scraping example. As we have the developer console in Chrome, we can install the firebug plugin in Firefox. Using this plugin, we can get the HTML elements for **Email** and **Password**. See the following screenshot:



```
▼<td>
   <input id="email" class="inputtext" type="email" tabindex="1" value="" name="email"></input>
 </td>
▼<td>
   <input id="pass" class="inputtext" type="password" tabindex="2" name="pass"></input>
 </td>
```

Once we figured the HTML element names, we programmatically created an HTML element object using WebDriver's `find_element_by_name()` method. WebDriver API has a method `send_keys()` that can work on element objects and enter the required text (in this case `email` and `password`). The last operation is to submit the form, and we performed it by finding the **Log In** object and clicking on it.

There's more…

We looked at a very basic example with the selenium WebDriver Python bindings. Now it's up to your imagination what you can achieve with selenium, automating mundane tasks.

Automating lead generation with web scraping

Ryan is a marketing manager at *Dely Inc*. Dely is a food delivery start-up and is trying to establish itself in the city of London. Dely is good at logistics and wants to aggregate restaurants on their platform, so when consumers order food from these restaurants, Dely will be responsible for the actual delivery. Dely is hoping that with every delivery they do, they will get a percentage cut from the restaurants. In return, restaurants have to think about their kitchen and not the logistical aspects. If you carefully think, virtually, every restaurant, big or small, is their probable lead. Dely wants to reach out to these restaurants and hopes to add them to their platform and fulfill their delivery needs.

Ryan is responsible for getting in touch with restaurants and wants to run a marketing campaign on all the target restaurants. But before he can do this, he needs to create a database of all the restaurants in London. He needs details, such as the name of the restaurant, the street address, and the contact number so that he can reach these restaurants. Ryan knows all his leads are listed on Yelp, but doesn't know where to start. Also, if he starts looking at all restaurants manually, it will take him a huge amount of time. With the knowledge you gained in this chapter, can you help Ryan with lead generation?

Legality of web scraping

We covered the legal aspects of web scraping in the initial parts of the chapter. I would like to warn you again on this. The example covered in this chapter, again, is for you to understand how to perform web scraping. Also, here we're scraping Yelp for public data, which is commonly available, as in this case, it is available on the restaurant's website itself.

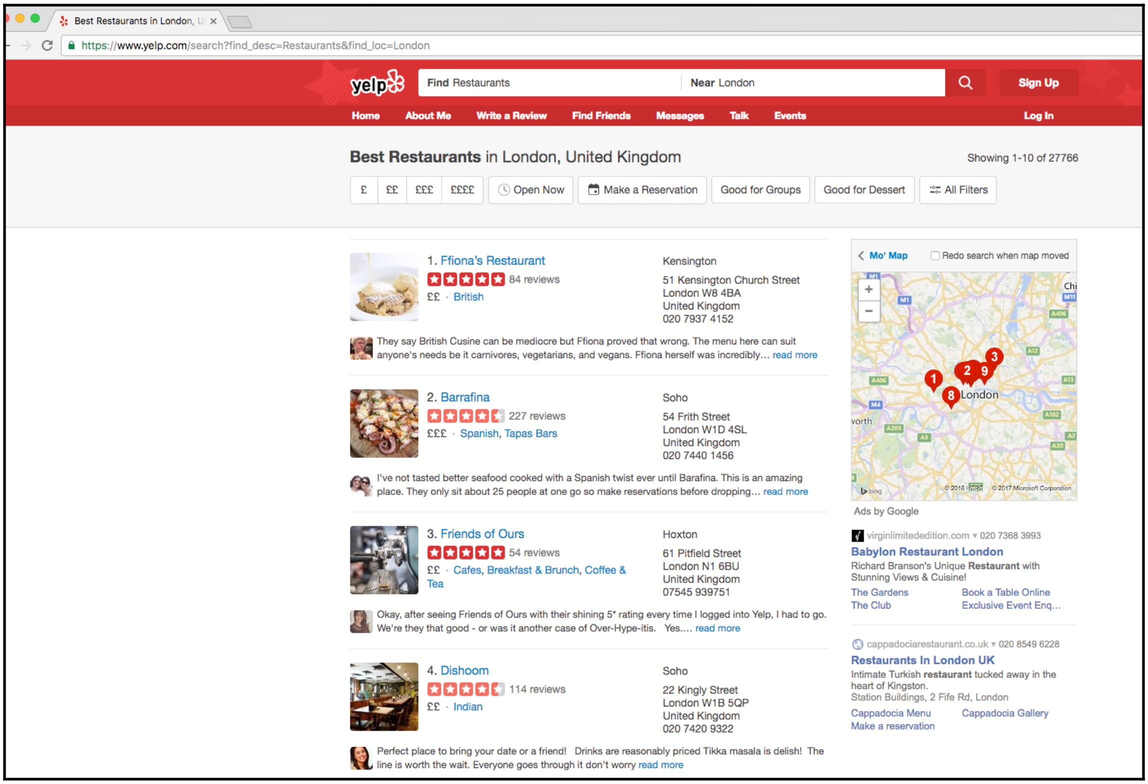
Getting ready

Now, if you look at Ryan's problem, he needs an automated way of collecting the database of all the restaurants listed in London. Yes, you got it right. Web scraping can help Ryan build this database. Can it be that easy? Let's see in this recipe.

For this recipe, we don't need any extra modules. We'll use the `BeautifulSoup` and `urllib` Python modules that we used in the previous recipes of this chapter.

How to do it…

1. We start by going to the Yelp website (`https://yelp.com/`) and searching for all the restaurants in the city of London. When you do that, you'll get a list of all the restaurants in London. Observe the URL that displays the search criteria. It is `https://www.yelp.com/search?find_desc=Restaurants&find_loc=London`. See the following screenshot for reference:



Best Restaurants in London, United Kingdom

Showing 1-10 of 27766

1. Ffiona's Restaurant
84 reviews
££ · British
Kensington
51 Kensington Church Street
London W8 4BA
United Kingdom
020 7937 4152

They say British Cusine can be mediocre but Ffiona proved that wrong. The menu here can suit anyone's needs be it carnivores, vegetarians, and vegans. Ffiona herself was incredibly… read more

2. Barrafina
227 reviews
£££ · Spanish, Tapas Bars
Soho
54 Frith Street
London W1D 4SL
United Kingdom
020 7440 1456

I've not tasted better seafood cooked with a Spanish twist ever until Barafina. This is an amazing place. They only sit about 25 people at one go so make reservations before dropping… read more

3. Friends of Ours
54 reviews
££ · Cafes, Breakfast & Brunch, Coffee & Tea
Hoxton
61 Pitfield Street
London N1 6BU
United Kingdom
07545 939751

Okay, after seeing Friends of Ours with their shining 5* rating every time I logged into Yelp, I had to go. We're they that good - or was it another case of Over-Hype-itis. Yes.… read more

4. Dishoom
114 reviews
££ · Indian
Soho
22 Kingly Street
London W1B 5QP
United Kingdom
020 7420 9322

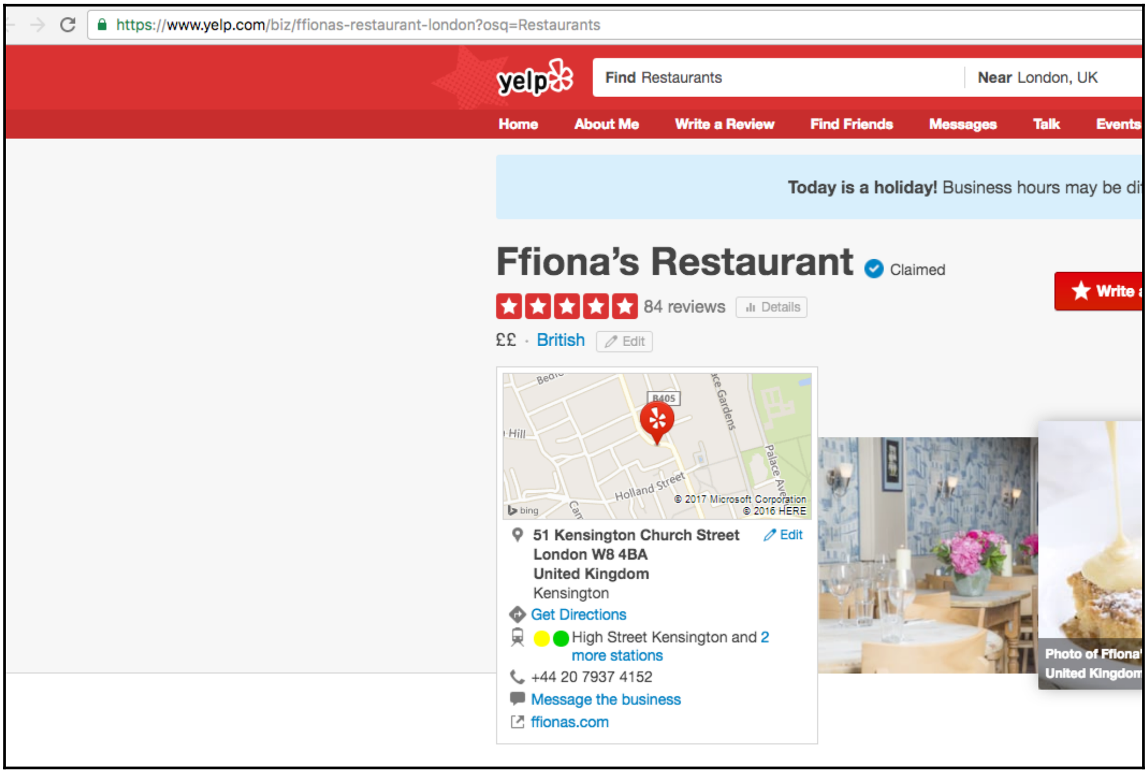
Perfect place to bring your date or a friend! Drinks are reasonably priced Tikka masala is delish! The line is worth the wait. Everyone goes through it don't worry read more

Ads by Google

virginlimitededition.com ▾ 020 7368 3993
Babylon Restaurant London
Richard Branson's Unique **Restaurant** with Stunning Views & Cuisine!
The Gardens · Book a Table Online
The Club · Exclusive Event Enq…

cappadociarestaurant.co.uk ▾ 020 8549 6228
Restaurants In London UK
Intimate Turkish **restaurant** tucked away in the heart of Kingston.
Station Buildings, 2 Fife Rd, London
Cappadocia Menu · Cappadocia Gallery
Make a reservation

2. Now, if you click on any of the restaurants' link that shows up in the search results, we should get the details that Ryan needs. See the following screenshot, where we get the details of *Ffiona's Restaurant*. Note how every restaurant has a dedicated URL; in this case, it is `https://www.yelp.com/biz/ffionas-restaurant-london?osq=Restaurants`. Also note that on this page, we have the name of the restaurant, the street address, and even the contact number. All the details that Ryan needs for his campaign; that's cool!



https://www.yelp.com/biz/ffionas-restaurant-london?osq=Restaurants

Today is a holiday! Business hours may be di

Ffiona's Restaurant

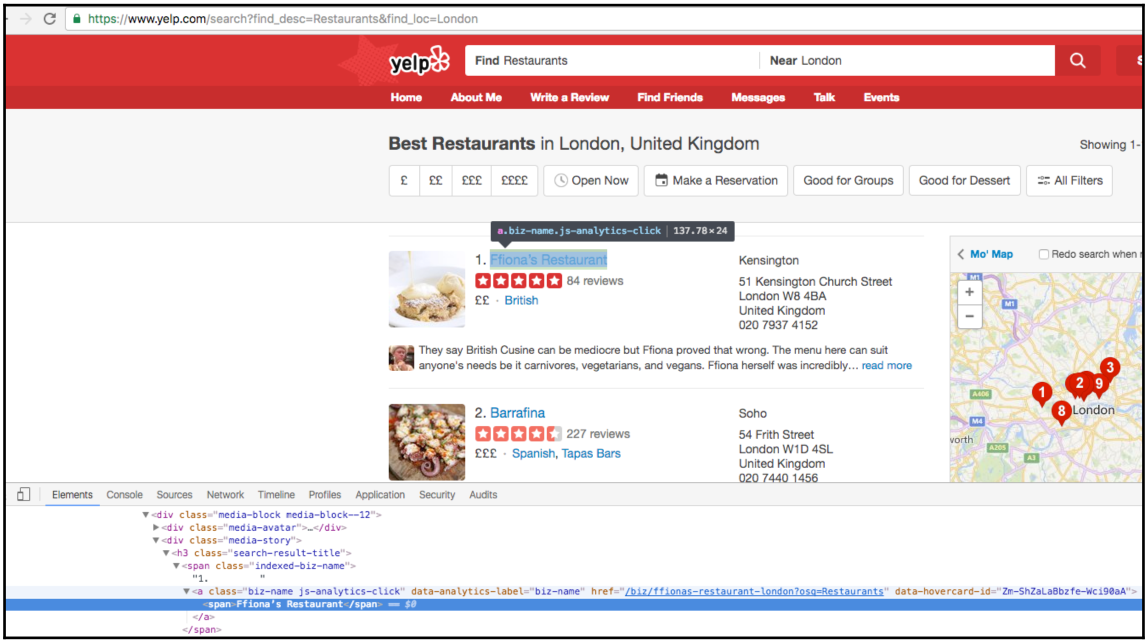
51 Kensington Church Street
London W8 4BA
United Kingdom
Kensington

High Street Kensington and 2 more stations

+44 20 7937 4152

3. OK nice, so we now know how to get the list of restaurants and also fetch the relevant details for a restaurant. But how do we achieve this in an automated way? As we saw in the web scraping example, we need to look for the HTML elements on the web pages from where we can collect this data.

4. Let's start with the search page. Open the search page (`https://www.yelp.com/search?find_desc=Restaurants&find_loc=London`) on your Chrome browser. Now, right-click on the first restaurant's URL and click on **Inspect** to get the HTML elements. If you notice, in the following screenshot, all the restaurants that are listed on the search page have a common CSS class name, `biz-name`, which indicates the name of the restaurant. It also contains the `href` tag, which points to the dedicated URL of the restaurant. In our screenshot, we get the name, **Ffiona's Restaurant**, and the `href` points to the restaurant's URL, `https://yelp.com/biz/ffionas-restaurant-london?osq=Resturants`.



Best Restaurants in London, United Kingdom

Showing 1-

£ | ££ | £££ | ££££ | Open Now | Make a Reservation | Good for Groups | Good for Dessert | All Filters

a.biz-name.js-analytics-click | 137.78×24

1. Ffiona's Restaurant
84 reviews
££ · British

Kensington
51 Kensington Church Street
London W8 4BA
United Kingdom
020 7937 4152

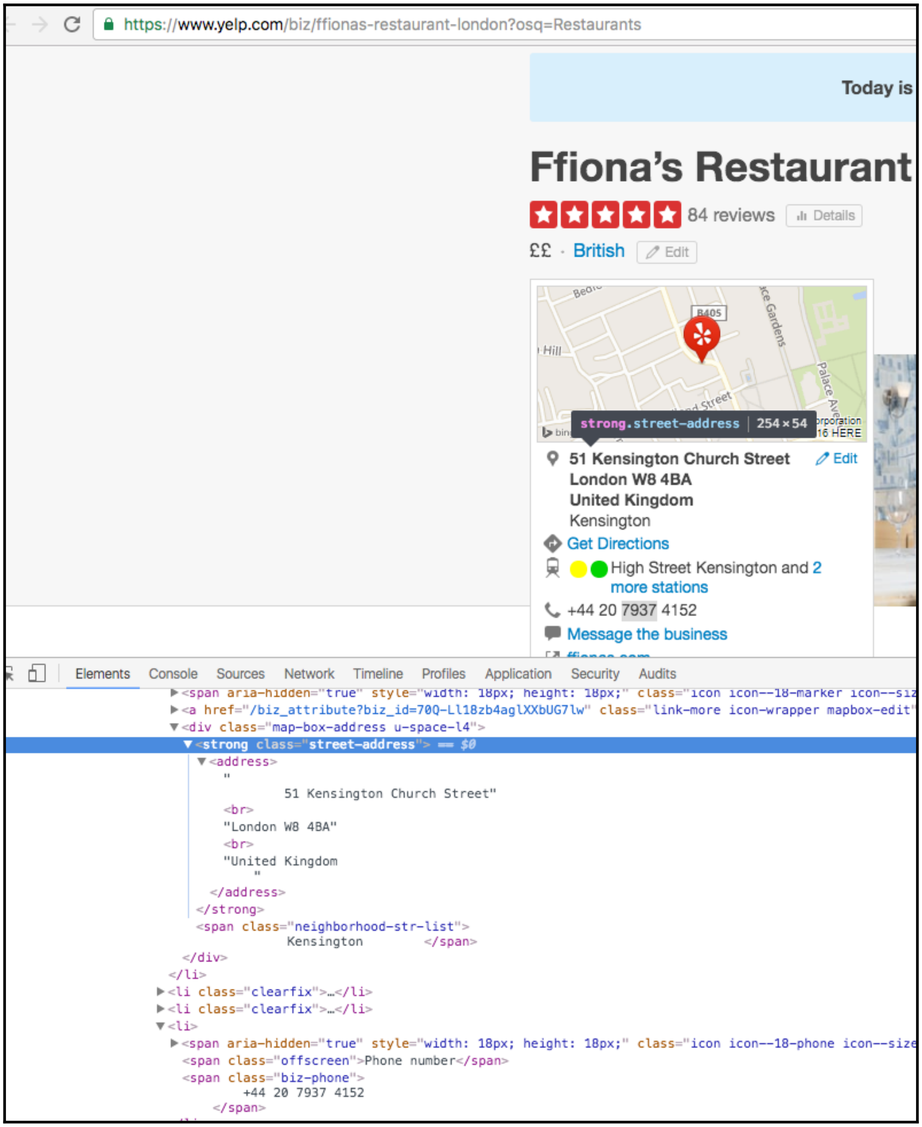
They say British Cusine can be mediocre but Ffiona proved that wrong. The menu here can suit anyone's needs be it carnivores, vegetarians, and vegans. Ffiona herself was incredibly… read more

2. Barrafina
227 reviews
£££ · Spanish, Tapas Bars

Soho
54 Frith Street
London W1D 4SL
United Kingdom
020 7440 1456

```
▼<div class="media-block media-block--12">
  ▶<div class="media-avatar">…</div>
  ▼<div class="media-story">
    ▼<h3 class="search-result-title">
      ▼<span class="indexed-biz-name">
         "1.        "
        ▼<a class="biz-name js-analytics-click" data-analytics-label="biz-name" href="/biz/ffionas-restaurant-london?osq=Restaurants" data-hovercard-id="Zm-ShZaLaBbzfe-Wci90aA">
            <span>Ffiona's Restaurant</span> == $0
          </a>
      </span>
```

5. Now, let's look at the dedicated page of the restaurant to see how we collect the street address and the contact number of the restaurant with the HTML elements. We perform the same operation, right-click, and **Inspect** to get the HTML elements of street address and contact number. See the following screenshot for reference. Note that for the street address, we have a separate CSS class, `street-address`, and the contact number is available under a span with the class name, **biz-phone**.



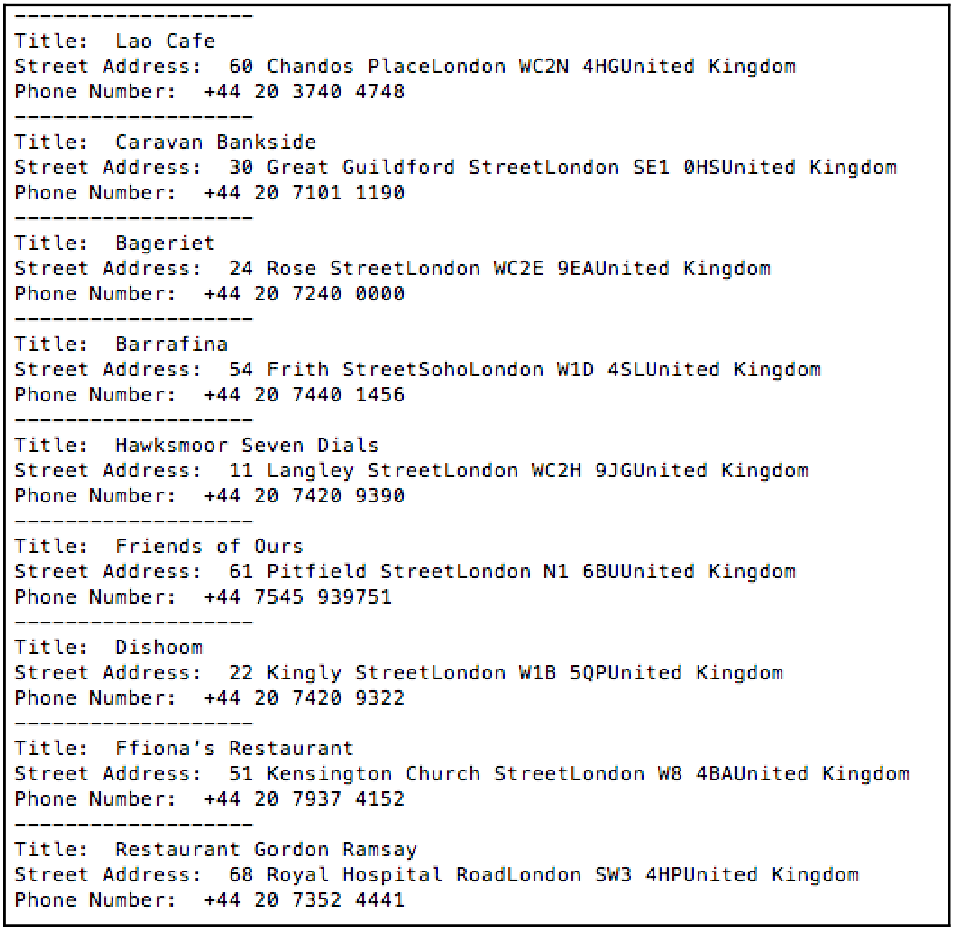
```
▶<span aria-hidden="true" style="width: 18px; height: 18px;" class="icon icon--18-marker icon--siz
▶<a href="/biz_attribute?biz_id=70Q-Ll18zb4aglXXbUG7lw" class="link-more icon-wrapper mapbox-edit"
▼<div class="map-box-address u-space-l4">
  ▼<strong class="street-address"> == $0
    ▼<address>
       "
               51 Kensington Church Street"
       <br>
       "London W8 4BA"
       <br>
       "United Kingdom
           "
     </address>
   </strong>
   <span class="neighborhood-str-list">
             Kensington          </span>
 </div>
</li>
▶<li class="clearfix">…</li>
▶<li class="clearfix">…</li>
▼<li>
  ▶<span aria-hidden="true" style="width: 18px; height: 18px;" class="icon icon--18-phone icon--size
   <span class="offscreen">Phone number</span>
   <span class="biz-phone">
         +44 20 7937 4152
     </span>
```

6. Awesome! So, we now have all the HTML elements that can be used to scrape the data in an automated way. Let's now look at the implementation. The following Python code performs these operations in an automated way:

```
from bs4 import BeautifulSoup
from threading import Thread
import urllib

#Location of restaurants
home_url = "https://www.yelp.com"
find_what = "Restaurants"
location = "London"

#Get all restaurants that match the search criteria
search_url = "https://www.yelp.com/search?find_desc=" +
find_what + "&find_loc=" + location
s_html = urllib.urlopen(search_url).read()
soup_s = BeautifulSoup(s_html, "lxml")

#Get URLs of top 10 Restaurants in London
s_urls = soup_s.select('.biz-name')[:10]
url = []
for u in range(len(s_urls)):
url.append(home_url + s_urls[u]['href'])


#Function that will do actual scraping job
def scrape(ur):
        html = urllib.urlopen(ur).read()
        soup = BeautifulSoup(html, "lxml")

        title = soup.select('.biz-page-title')
        saddress = soup.select('.street-address')
        phone = soup.select('.biz-phone')

        if title:
             print "Title: ", title[0].getText().strip()
        if saddress:
             print "Street Address: ",
saddress[0].getText().strip()
        if phone:
             print "Phone Number: ", phone[0].getText().strip()
        print "-------------------"

threadlist = []
i=0
#Making threads to perform scraping
while i<len(url):
```

```
              t = Thread(target=scrape,args=(url[i],))
              t.start()
              threadlist.append(t)
              i=i+1

    for t in threadlist:
              t.join()
```

7. OK, great! Now, if we run the preceding Python code, we get the details of the top 10 restaurants in *London,* along with their names, street addresses and contact numbers. Refer to the following screenshot:



```
-------------------
Title:  Lao Cafe
Street Address:  60 Chandos PlaceLondon WC2N 4HGUnited Kingdom
Phone Number:  +44 20 3740 4748
-------------------
Title:  Caravan Bankside
Street Address:  30 Great Guildford StreetLondon SE1 0HSUnited Kingdom
Phone Number:  +44 20 7101 1190
-------------------
Title:  Bageriet
Street Address:  24 Rose StreetLondon WC2E 9EAUnited Kingdom
Phone Number:  +44 20 7240 0000
-------------------
Title:  Barrafina
Street Address:  54 Frith StreetSohoLondon W1D 4SLUnited Kingdom
Phone Number:  +44 20 7440 1456
-------------------
Title:  Hawksmoor Seven Dials
Street Address:  11 Langley StreetLondon WC2H 9JGUnited Kingdom
Phone Number:  +44 20 7420 9390
-------------------
Title:  Friends of Ours
Street Address:  61 Pitfield StreetLondon N1 6BUUnited Kingdom
Phone Number:  +44 7545 939751
-------------------
Title:  Dishoom
Street Address:  22 Kingly StreetLondon W1B 5QPUnited Kingdom
Phone Number:  +44 20 7420 9322
-------------------
Title:  Ffiona's Restaurant
Street Address:  51 Kensington Church StreetLondon W8 4BAUnited Kingdom
Phone Number:  +44 20 7937 4152
-------------------
Title:  Restaurant Gordon Ramsay
Street Address:  68 Royal Hospital RoadLondon SW3 4HPUnited Kingdom
Phone Number:  +44 20 7352 4441
```

8. In the preceding screenshot, we get the records of 10 restaurants in London provided by Yelp. **Title** is the name of the restaurant and **Street Address** and **Phone Number** are self-explanatory. Awesome! We did it for Ryan.

How it works…

In the preceding code snippet, we built the search criteria. We searched on `https://yelp.com` and looked for restaurants in *London*. With these details, we got the search URL on Yelp.

We then created a `urllib` object and used the `urlopen()` method on this search URL to `read()` the list of all the restaurants provided by Yelp matching the search criteria. The list of all the restaurants is stored as an HTML page, which is stored in the variable, `s_html`.

Using the `BeautifulSoup` module, we created a soup instance on the HTML content so that we could start extracting the required data using the CSS elements.

Initially, we browsed the top 10 results of the search on Yelp and got the URLs of the restaurants. We stored these URLs in the URL Python list. To get the URL, we selected the CSS class name `biz-name` using the code `soup_s.select(.biz-name)[:10]`.

We also defined a method, `scrape()`, which takes the restaurant URL as a parameter. In this method, we read the details of the restaurant, such as name, street address, and contact number, using the CSS class names `biz-page-title`, `street-address`, and `biz-phone`, respectively. To get the exact data, we selected the HTML elements using `title=soup.select`(`.biz-page-title`) and got the data with `title[0].getText().strip()`. Note that the `select()` method returns the found element as an array, so we need to look for index `0` to get the actual text.

We iterated through all the restaurant URLs in a `while` loop and scraped the URL using the `scrape()` method to get the details for each restaurant. It prints the name, street address, and contact number for each restaurant on your console, as we saw in the preceding screenshot.

To improve on the performance of our screaping program, we performed data extraction for every restaurant in an independent thread. We created a new thread with `t = Thread(target=scrape,args=(url[i],))` and got the results from each of them with the `t.join()` call.

That’s it, folks! Ryan is extremely happy with this effort. In this example, we helped Ryan and automated a critical business task for him. Throughout this book we'll look at various use cases where Python can be leveraged to automate business processes and make them efficient. Interested in more? Well, see you in the next chapter.

2

Working with CSV and Excel Worksheets

Imagine a world where your important documents were stored in files and managed on work desks. Thanks to the advent of computers and software such as Excel sheets we can manage our data in an organized way. In fact, you can even manage worksheets in an automated way, and that too with Python.

In this chapter, we will cover the following recipes:

- Reading CSV files with reader objects
- Writing data into CSV files
- Developing your own CSV dialects
- Managing employee information in an automated way
- Reading Excel sheets
- Writing data into worksheets
- Formatting Excel cells
- Playing with Excel formulae
- Building charts within Excel sheets
- Automating the comparison of company financials

Introduction

Until computers became a part of our daily lives, office records were created on paper and stored in cabinets. Today, thanks to the ever-growing computational field, we store these records using computer applications in a text file. Text (`.txt`) files were great at saving large amounts of data; it was also easy to search for information within a text file, but the data was never stored in an organized way. Over time as the information grew, the need for storing information also increased and resulted in the advent of CSV and Excel sheets, where data not only could be stored in a structured format but also read and processed easily.

CSV files contain data separated by commas; hence, they are referred to as **comma-separated values** (**CSV**) files. CSV allows for storing data in a tabular format. CSV files are easier to import on any storage system, independent of the software being used. Since CSV files are plain text files, they can be modified easily and are hence used for the quick exchange of data.

On the other hand, Excel sheets contain data separated by tabs or other delimiters. Excel sheets store and retrieve data in a grid format of columns and rows. They allow the formatting of data, working with formulas, and have the capability of hosting multiple sheets in a file. Excel is ideal for entering, calculating, and analyzing company data, such as sales figures or commissions.

While CSV files are text files used to store and retrieve data from programs, Excel files are binary files and are used for more advanced operations like charting, calculations and often for storing reports.

Python has a useful set of modules to work with both CSV and Excel files. You can read/write CSV and Excel files, format Excel cells, prepare charts, and perform calculations on data with formulae.

The recipes in this chapter will focus on the Python modules that help us performing the preceding operations on CSV and Excel sheets. Specifically, we will focus on the following Python modules in this chapter:

- `csv` (`https://docs.python.org/2/library/csv.html`)
- `openpyxl` (`https://pypi.python.org/pypi/openpyxl`)
- `XlsxWriter` (`https://pypi.python.org/pypi/XlsxWriter`)

Reading CSV files with reader objects

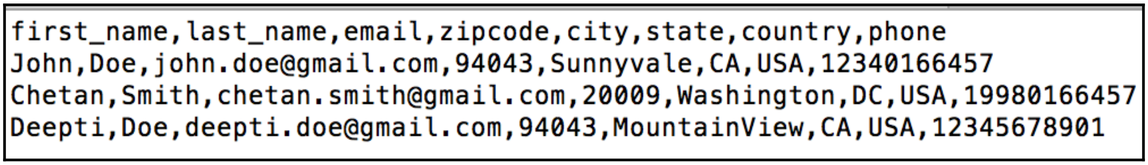
This recipe will show you how to read CSV files, specifically how to create and use the reader object.

Getting ready

To step through this recipe, you will need to install Python v2.7. To work with the CSV files, we have a nice module, `csv`, which is packaged with the default Python installation. So, let's get started!

How to do it…

1. On your Linux/Mac computer, go to Terminal and use Vim, or choose your favorite editor.
2. We start by creating a CSV file. As we know, a CSV file has a structured format where data is separated by commas, so creating one should be trivial. The following screenshot is of a CSV file that contains details of contacts in different parts of the country. We name it as `mylist.csv`:

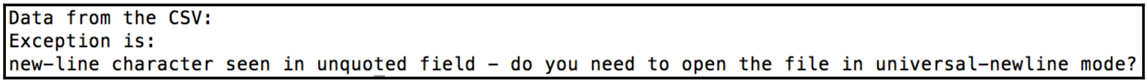


```
first_name,last_name,email,zipcode,city,state,country,phone
John,Doe,john.doe@gmail.com,94043,Sunnyvale,CA,USA,12340166457
Chetan,Smith,chetan.smith@gmail.com,20009,Washington,DC,USA,19980166457
Deepti,Doe,deepti.doe@gmail.com,94043,MountainView,CA,USA,12345678901
```

3. Now, let's write the Python code to read this CSV file and print data from it:

```
import csv
fh = open("mylist.csv", 'rt')
try:
    reader = csv.reader(fh)
    print "Data from the CSV:", list(reader)
    except Exception as e:
    print "Exception is:", e
finally:
    fh.close()
```

The output of the preceding code snippet is as follows:

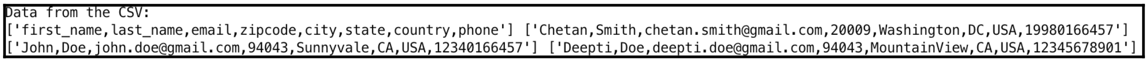


```
Data from the CSV:
Exception is:
new-line character seen in unquoted field - do you need to open the file in universal-newline mode?
```

4. Oh! What happened? We seem to have hit an error. The error suggests that the CSV reader could not find the new line character. This happens in CSV files written on a Mac platform. This is because Mac OS uses **carriage return** (**CR**) as the end of line character.
5. Python has a simple solution for this issue; we open the file in **rU** mode (which is **universal newline** mode). The following program runs perfectly fine and we can read the file contents appropriately:

```
try:
    reader = csv.reader(open("mylist.csv", 'rU'),
                        dialect=csv.excel_tab)
    print "Data from the CSV:"
    d = list(reader)
    print "\n".join("%-20s %s"%(d[i],d[i+len(d)/2]) for i in
                    range(len(d)/2))

    except Exception as e:
    print "Exception is:", e
finally:
    fh.close()
```

The output of the preceding program is as follows:

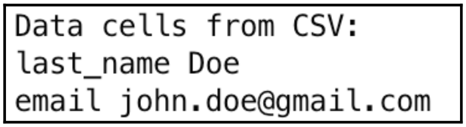


```
Data from the CSV:
['first_name,last_name,email,zipcode,city,state,country,phone'] ['Chetan,Smith,chetan.smith@gmail.com,20009,Washington,DC,USA,19980166457']
['John,Doe,john.doe@gmail.com,94043,Sunnyvale,CA,USA,12340166457'] ['Deepti,Doe,deepti.doe@gmail.com,94043,MountainView,CA,USA,12345678901']
```

6. That's great! There's another simple fix for the issue we observed in the preceding code snippet. What we could do is, simply change the file format from Mac CSV to Windows CSV. We can do this by performing the **Open** and **Save As** operations on the file. In the following example, I have saved `mylist.csv` as `mylist_wincsv.csv` (Windows CSV format), and reading the file contents is not an issue anymore:

```
fh = open("mylist_wincsv.csv", 'rt')
reader = csv.reader(fh)
data = list(reader)
print "Data cells from CSV:"
print data[0][1], data[1][1]
print data[0][2], data[1][2]
print data[0][3], data[1][3]
```

In the preceding code example, we print a part of the data from the CSV file. If you realize, a CSV file can also be read as a 2D list in Python with the first index as row and the second index as column. Here we print the second, third, and fourth columns from `row1` and `row2`:

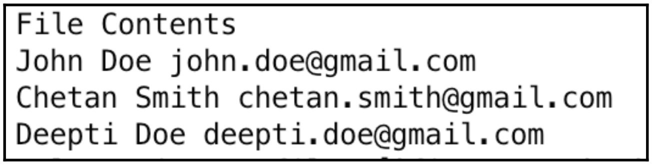


```
Data cells from CSV:
last_name Doe
email john.doe@gmail.com
```

7. With Python, it's also very convenient to read the contents of a CSV file in a dictionary with a helpful `DictReader(f)` method:

```
import csv
f = open("mylist_wincsv.csv", 'rt')
print "File Contents"
try:
    reader = csv.DictReader(f)
    for row in reader:
        print row['first_name'], row['last_name'], row['email']
finally:
    f.close()
```

In the preceding code snippet, we open the file with the file handle, `f`. This file handle is then used as an argument to `DictReader()`, which will treat the first row values as column names. These column names act as a key in the dictionary where the data gets stored. So, in the preceding program, we can selectively print the data from three columns: **first_name**, **last_name**, and **e-mail** and print them like in the following screenshot:



```
File Contents
John Doe john.doe@gmail.com
Chetan Smith chetan.smith@gmail.com
Deepti Doe deepti.doe@gmail.com
```

8. The `csv` module's `DictReader()` has a few helper methods and attributes that make it easy to read CSV files. These are also known as **reader objects**:

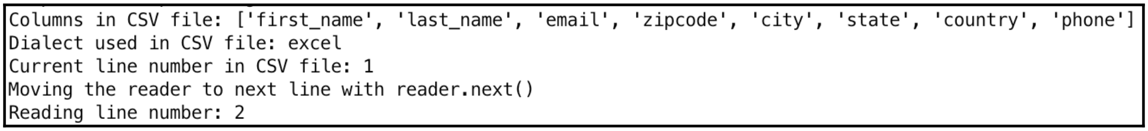
```
import csv
f = open("mylist_wincsv.csv", 'rt')
reader = csv.DictReader(f)
print "Columns in CSV file:", reader.fieldnames
print "Dialect used in CSV file:", reader.dialect
print "Current line number in CSV file:", reader.line_num
print "Moving the reader to next line with reader.next()"
```

```
        reader.next()
        print "Reading line number:", reader.line_num f.close()
```

In this code example, we used the following attributes and method:

- `fieldnames`: Gives list of column names
- `dialect`: CSV file format (we 'll read more about it)
- `line_num`: Current line number being read
- `next()`: Takes you to the next line

In the following screenshot, the first line contains all the column names from our CSV file. In the second line, we print the dialect used to read the CSV file. The third line prints the line number we're currently reading, and the last line of the screenshot depicts the next line that the reader object will move to while reading:



```
Columns in CSV file: ['first_name', 'last_name', 'email', 'zipcode', 'city', 'state', 'country', 'phone']
Dialect used in CSV file: excel
Current line number in CSV file: 1
Moving the reader to next line with reader.next()
Reading line number: 2
```

There's more…

The Python module, `csv`, is a helper and it is perfectly possible to process a CSV file by opening the file with the `open()` method and reading the file contents with the `readline()` method. You can then perform the `split()` operation on every line of the file to get the file contents.

Reading is great, but you'll read something only when something is written in a CSV file, isn't it? :) Let's look at the methods available for writing data into CSV files in the next recipe.

Writing data into CSV files

Again, for the recipes in this section, we don't need any new modules apart from the ones that are bundled with the Python installation, that is, the `csv` module.

How to do it…

1. First, lets open a file in write mode and in text format. We create two Python lists that contain the data to be written into the CSV file. The following code will perform these operations:

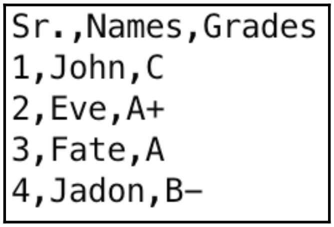
```
import csv
names = ["John", "Eve", "Fate", "Jadon"]
grades = ["C", "A+", "A", "B-"]
f = open("newlist.csv", 'wt')
```

2. Let's now add the data into the CSV file with the `write()` method, as follows:

```
try:
    writer = csv.writer(f)
    writer.writerow( ('Sr.', 'Names', 'Grades') )
    for i in range(4):
        writer.writerow( (i+1, names[i], grades[i]) )
finally:
    f.close()
```

In the preceding code, we initialized the CSV file with a header; the column names used are: **Sr**., **Names**, and **Grades**. Next, we start a Python `for` loop to run four times and write four rows of data into the CSV file. Remember, we have the data in the Python lists, `Names` and `Grades`. The `writerow()` method actually adds the content in the CSV file, adding a row one by one inside the `for` loop.

The output of the preceding code snippet can be seen in the following screenshot:

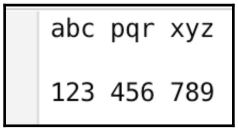


```
Sr.,Names,Grades
1,John,C
2,Eve,A+
3,Fate,A
4,Jadon,B-
```

3. Cool, that was simple and straightforward. It is interesting to note that, by default, when we write into a CSV file, the file contents in a row are separated by commas. But what if we want to change the behavior to make it separated by tabs (`\t`)? The `writer()` method has this facility of changing not only the delimiter but also the line terminator. (Note: Delimiter is the character that is used to separate the data within a row in a CSV file. The terminator character is used to mark the end of a row in a CSV file. You will relate to this in the following example):

```
import csv
f=open("write.csv", 'wt')
csvWriter = csv.writer(f, delimiter='\t',
                          lineterminator='\n\n')
csvWriter.writerow(['abc', 'pqr', 'xyz'])
csvWriter.writerow(['123', '456', '789'])
f.close()
```

When the preceding code snippet is run, a new file, `write.csv`, gets created on the filesystem. File contents can be viewed in the following screenshot. If you look at the contents in a given row, you will see them separated by tabs and not by commas. The new line delimeter is the return key (pressed twice), which is also evident in the following screenshot. Note that there's an extra newline character between both the rows:



```
abc pqr xyz

123 456 789
```

Developing your own CSV dialects

To make it easier to read and write into CSV files, we can specify the formatting parameters that are a part of the `Dialect` class of the `csv` module. Here, we look at some of the dialects available and learn how to write our own.

Getting ready

For this recipe, we will use the same `csv` module that is present in the default installation of Python, so there is no need to install anything explicitly.

How to do it…

1. Let's first look at some of the attributes that are present in the `Dialect` class:
    - `Dialect.delimeter`: We used this in the previous recipe where we changed the way the contents are written in the row of a CSV file. It is used to separate two fields.
    - `Dialect.lineterminator`: This is used to signify the termination of a line added in a CSV file. We also used this in our previous section.
    - `Dialect.skipinitialspace`: This will skip all the leading spaces after a delimiter. It helps avoid accidental human error.

   We can get a list of the available dialects with the following code:

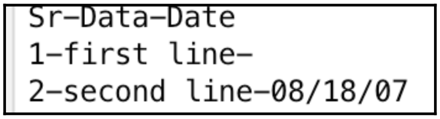
   ```
   print "Available Dialects:", csv.list_dialects()
   ```

   The two main dialects available are `excel` and `excel-tab`. The `excel` dialect is for working with data in the default export format for Microsoft Excel, and also works with OpenOffice or NeoOffice.

2. Let's now create a dialect of our choice. For instance, we choose the `-` symbol to demarcate columns in a CSV file:

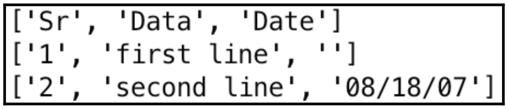
   ```
   import csv
   csv.register_dialect('pipes', delimiter='-')
   with open('pipes.csv', 'r') as f:
       reader = csv.reader(f, dialect='pipes')
       for row in reader:
           print row
   ```

3. We create a file, `pipes.csv`, which looks as follows:



```
Sr-Data-Date
1-first line-
2-second line-08/18/07
```

If we run the preceding Python code on the `pipes.csv` file it returns every line as an array with all the elements split by the `-` character. The following screenshot shows the output of our program:



```
['Sr', 'Data', 'Date']
['1', 'first line', '']
['2', 'second line', '08/18/07']
```

How it works…

In the second code snippet, we register our own dialect with the `register_dialect()` method. We have named our dialect as `pipes` and the delimiter associated with `pipes` is the symbol `-`, as we intended to do.

We now read the `pipes.csv` file with our very own `read()` method, and use the reader object to get the contents of the CSV file. But wait, did you see the use of `dialect='pipes'`? This will make sure that the reader expects the columns to be separated by `-` and reads the data accordingly.

If you observe, the `reader` object has split rows based on `-`, which is defined by the dialect, `pipes`.

You learned about reading and writing your own data into CSV files. You also understood the usage of dialects. It's time to get a feel of how to use the preceding concepts with a real-world use case.

Managing employee information in an automated way

Mike is the HR Manager of his organization and is trying to gather the contact information of all the employees from the state of California. He wants to segregate this information so that he can conduct a survey on all employees from the CA State. He not only wants to collect this information but also persist it to another CSV file so that it is easy to work on it at a later point in time.

Can we help Mike here? How do you apply the concepts you learned so far? Will you learn something more while helping Mike? Let's look at the implementation.

Getting ready

We don't need any special modules for this example. All the modules that have been installed as part of the previous recipes are enough for us. For this example, we use the same mylist.csv file that contained the employee information.

How to do it…

1. Let's directly get to the code and open the two files. One file handle is used for reading the file contents (to read the employee data) and the other one is used for writing into the CA_Employees.csv file. Note the differences in the mode in which the files are opened ('rt' and 'wt'). Of course, the employee CSV file is opened in the read mode and the CA_Employees.csv file is opened in the write mode.

```
import csv
f = open("mylist.csv", 'rt')
fw = open("CA_Employees.csv", 'wt')
```

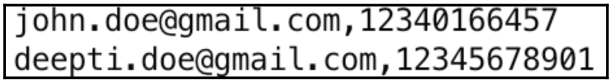
2. Next, we read the employee information from the CSV file as a dictionary with the DictReader() method. We also create a csvWriter object, using which we will write the data into the CA_Employees.csv file.

3. You would imagine when we start reading the rows of the CSV file, we'd also read the first row. We should skip this row as this just contains the column names, right? Yes, we skip the header using the line_num attribute of the reader object (Remember, we learned about attributes earlier in this chapter). Once the header is skipped, we iterate over all the rows and filter out employees who belong to the CA State and get the e-mail and phone information for these employees. The filtered data is then written into the CA_Employees.csv file with the csvWriter object. Note that once the file operations are complete, it is important to close the file handles as this may result in memory leaks or data inconsistencies:

```
try:
    reader = csv.DictReader(f)
    csvWriter = csv.writer(fw)
    for row in reader:
        if reader.line_num == 1:
           continue
        if row['state'] == 'CA':
            csvWriter.writerow([row['email'], row['phone']])
finally:
    f.close()
    fw.close()
```

How it works…

When we run the preceding program in its entirety, we will get a `CA_Employees.csv` file that looks like the following screenshot:



```
john.doe@gmail.com,12340166457
deepti.doe@gmail.com,12345678901
```

If you look at the code implementation, we use the `line_num` attribute to skip the header row, which is the first row of the `mylist.csv` file. We also write the filtered data into the newly created `CA_Employees.csv` file with the `writerow()` method. Nice work, I think Mike is already happy with you. His problems are solved. :)

We come to an end of this section on working with CSV files. CSV files essentially store data in pure text format. We can't achieve many things with these files, hence the advent of Excel sheets. In the next recipe, we start working with Excel sheets and appreciate what they can offer!

Reading Excel sheets

As you might be aware, Microsoft office has started providing a new extension to Microsoft Excel sheets, which is `.xlsx`, from Office 2007. With this change, Excel sheets moved to a XML based file format (Office Open XML) with ZIP compression. Microsoft made this change when the business community asked for an open file format that can help transferring data across applications that pushed them. Let's gets started and see how we can work with Excel sheets using Python!

Getting ready

In this recipe, we use the `openpyxl` module to read Excel sheets. The `openpyxl` module is a comprehensive module that performs both read and write operations on Excel sheets. Another alternative to `openpyxl` is the `xlrd` module. While `xlrd` has been good at supporting Excel formats since way back in 1995, the module can only be used to read data from Excel sheets. The `openpyxl` module helps in performing more operations, such as modifying data, writing data into files, and copying, which are imperative to working with Excel files.

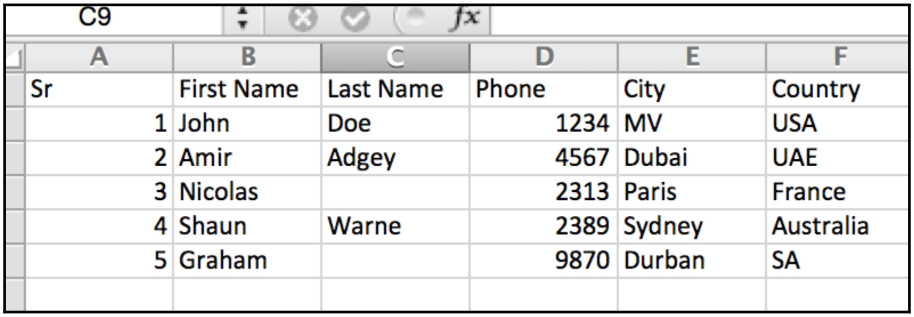
Let's install the `openpyxl` module with our favorite tool, `pip`:

```
pip install openpyxl
```

How to do it…

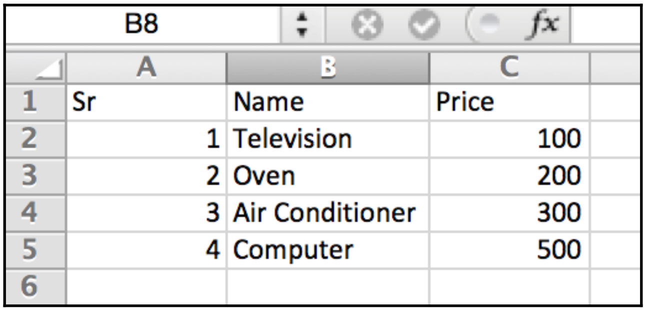
1. We start by creating our own Excel sheet with the content as shown in the following screenshot. As you must be aware, Excel files are called **workbooks** and contain one or more worksheets, and hence Excel files are also known as **spreadsheets**. We save the file as `myxlsx.xlsx` in two sheets, **People** and **Items**:

   Let's look at the data from the **People** sheet:



Sr	First Name	Last Name	Phone	City	Country
1	John	Doe	1234	MV	USA
2	Amir	Adgey	4567	Dubai	UAE
3	Nicolas		2313	Paris	France
4	Shaun	Warne	2389	Sydney	Australia
5	Graham		9870	Durban	SA

Now, let's look at the data from the **Items** sheet:



	A	B	C
1	Sr	Name	Price
2	1	Television	100
3	2	Oven	200
4	3	Air Conditioner	300
5	4	Computer	500
6			

2. Let's now go ahead and read the XLSX file. The following code will help us in getting the names of all the worksheets present in the Excel workbook:

   ```
   import openpyxl
   workbook = openpyxl.load_workbook('myxls.xlsx')
   print "Workbook Object:", workbook.get_sheet_names()
   ```

3. Now, if you want to work with a given sheet, how do you get access to that object? The following code snippet, takes us to the **People** worksheet:

   ```
   people = workbook.get_sheet_by_name('People')
   print "People sheet object:", people
   ```

   Wow, that's cool!

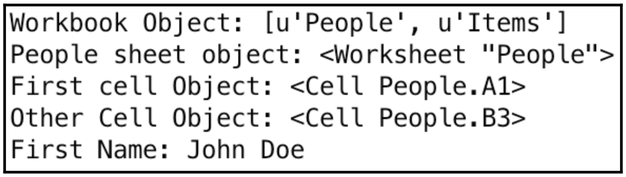
4. Let's now move ahead and read the cell objects. We can read the cells either by the name or based on the row/column location. The following code snippet demonstrates this:

   ```
   import openpyxl
   workbook = openpyxl.load_workbook('myxls.xlsx')
   people = workbook.get_sheet_by_name('People')
   print "First cell Object:", people['A1']
   print "Other Cell Object:", people.cell(row=3, column=2)
   ```

5. But how do I get the values in the cell? Simple enough, object.value returns you the value present in the cell:

   ```
   print "First Name:", people['B2'].value,
   people['C2'].value
   ```

If we run the Python code snippet, we will get the following output as seen in this screenshot:



```
Workbook Object: [u'People', u'Items']
People sheet object: <Worksheet "People">
First cell Object: <Cell People.A1>
Other Cell Object: <Cell People.B3>
First Name: John Doe
```

How it works…

In the preceding example, we import the `openpyxl` module. This module has a method with which you can access the worksheet objects and cells in it.
The `load_workbook()` method loads the complete Excel sheet in the memory. The `get_sheet_names()` and `get_sheet_by_name()` methods help in selecting the worksheets of the given workbook. Thus, we have the workbook and worksheet objects ready with us.

The cell objects can be accessed with the `cell()` method, and `cell().value` returns the actual value present in the cell of worksheet. Nice, see how trivial it is to read data from Excel sheets with Python. But again, reading is only helpful if we know how to write data into Excel sheets. So, what are we waiting for? Let's go ahead and learn that as well in the next recipe.

Writing data into worksheets

Reading files is a breeze with the `openpyxl` module. Now, let's shift our focus to writing Excel files. We'll perform multiple operations with Excel files in this section.

Getting ready

For this recipe, we will use another fantastic Python module, which is `xlsxwriter`. As the name suggests, this module help us perform multiple operations on Excel sheets. Interestingly, `xlsxwriter` doesn't support read operations on an Excel sheet (at the time of writing this book). We install the `xlsxwrite` module using `pip`, as follows:

```
pip install xlsxwriter
```

How to do it...

1. We start with a very basic operation of creating an XLSX file and adding a new sheet to it. The following code performs this operation:

```
import xlsxwriter
workbook = xlsxwriter.Workbook('add_sheet.xlsx')
worksheet = workbook.add_worksheet(name='New Sheet 2')
workbook.close()
```

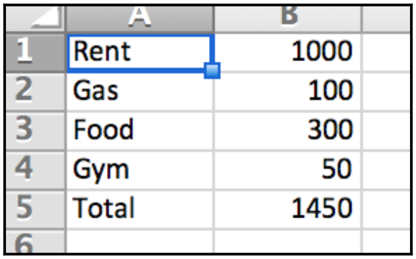
2. Let's move ahead and perform the `write` operations on the worksheet and store some useful information:

```
import xlsxwriter
workbook = xlsxwriter.Workbook('Expenses01.xlsx')
worksheet = workbook.add_worksheet()
expenses = (
    ['Rent', 1000],
    ['Gas',   100],
    ['Food',  300],
    ['Gym',    50],
)
row = 0
col = 0
for item, cost in (expenses):
    worksheet.write(row, col, item)
    worksheet.write(row, col + 1, cost)
    row += 1

worksheet.write(row, 0, 'Total')
worksheet.write(row, 1, '=SUM(B1:B4)')
workbook.close()
```

How it works…

The first code snippet of this recipe creates a `workbook` object with the `Workbook()` method under a new Excel file, `add_sheet.xlsx`. It then goes ahead and creates a `worksheet` object with the `add_worksheet()` method. A new sheet named `New Sheet 2` is created.

In the second code example, we create an XLSX file named `Expenses01.xlsx`. We add the expenses data to it from the `expenses` dictionary. For doing this, we iterate through the dictionary and use the keys as one column and the values as another column in the Excel sheet. Finally, we add one last row that sums up all the expenses. The contents of `Expenses01.xlsx` are shown in the following screenshot:



	A	B
1	Rent	1000
2	Gas	100
3	Food	300
4	Gym	50
5	Total	1450
6		

In the preceding code snippets, we performed simple write operations on an Excel sheet with the `xlsxwrite` module. We first created a workbook with the `Workbook()` method and added a new `sheet` object to this workbook with the `add_worksheet()` method. Using the `write()` method on the `worksheet` object, we added data to the Excel sheet. We also did a small formula operation to get the total of all expenses with `=SUM(B1:B4)`.

What we saw was a very basic example of writing Excel files. We could perform many more operations programmatically as we are used to doing manually on Excel sheets. Let's now learn how to format Excel cells in the next set of recipes.

Formatting Excel cells

Cells are formatted for various reasons. In the business world, they are used to group data based on a theme, or in the case of a software development process, cells are colored to indicate whether a feature is done or a bug is fixed.

Getting ready

For this recipe, we will use the same `xlsxwriter` module and format the cells. We will learn how to add and apply formats to the cells.

How to do it…

1. We continue with the expenses example to demonstrate the formatting of cells. But first let's understand how to create formats. Formats are added with the `add_format()` method. This method returns a `format` object. The following code example shows how to create a format:

   ```
   format = workbook.add_format()
   format.set_bold()
   format.set_font_color('green')
   ```

   In the preceding example, we created a cell format, wherein the data in the cell (to which the format is applied) is `bold` and color is set to `green`.

2. Coming back to the example of expenses sheet, how about highlighting the cells where the expenses have gone above 150? Yes, we can do that programmatically by creating a format to highlight the cells in red. But let's go in order. First, we create a sheet and add data to it, as in the following code:

   ```
   import xlsxwriter
   workbook = xlsxwriter.Workbook('cell_format.xlsx')
   worksheet = workbook.add_worksheet()
   expenses = (
       ['Rent', 1000],
       ['Gas',   100],
       ['Food',  300],
       ['Gym',    50],
   )

   row = 0
   col = 0
   for item, cost in (expenses):
       worksheet.write(row, col, item)
       worksheet.write(row, col + 1, cost)
       row += 1
   ```

The preceding code will create an Excel sheet named `cell_format.xlsx` and add expenses to it.

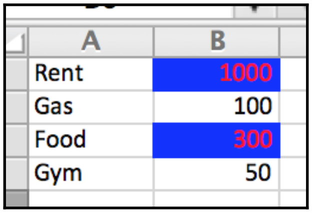
3. Now, let's create a format where the cells are colored with blue and the cell values will be in red. We could set the format with the `set_font_color()` method, but in the following example, we set the format through options like `'bg_color'` and `'font_color'`:

```
format1 = workbook.add_format({'bg_color': 'blue',
                               'font_color': 'red'})
```

4. Now, the only remaining step is to apply this format on the expenses that are above 150. The following code applies the format and respects the condition:

```
worksheet.conditional_format('B1:KB5',
          {'type': 'cell',
           'criteria': '>=',
           'value': 150,
           'format': format1}
)
workbook.close()
```

When we run this program, the contents of the `cell_format.xlsx` file looks as shown in the following screenshot:



A	B
Rent	1000
Gas	100
Food	300
Gym	50

There's more…

Cool, so now that we have the cell formatting done, how about moving on to working with formulas in Excel sheets?

Playing with Excel formulae

We take a very simple example to demonstrate the use of formulae in Excel sheets.

Getting ready

For this recipe, we will use the same `xlsxwriter` module and add formulae to the cells. There are numerous operations that are supported by Excel sheets, such as getting standard deviation of data, logarithm, getting trends among others, so it's worth spending time to understand the majority of available operations.

How to do it…

You need to perform the following step:

We work with a simple example, wherein we add a list of numbers with the `SUM()` formula and store the sum to the cell **A1**:

```
import xlsxwriter
workbook = xlsxwriter.Workbook('formula.xlsx')
worksheet = workbook.add_worksheet()
worksheet.write_formula('A1', '=SUM(1, 2, 3)')
workbook.close()
```

How it works…

When we run the preceding code, a new Excel file, `formula.xlsx` gets created with cell **A1** containing the number **6** (addition of `1`, `2`, and `3`).

As in the preceding section, we can perform more complex mathematical operations using Excel formulae. For instance, you can plan the yearly IT budget for your team in an Excel sheet.

There's more…

There's no fun if we don't discuss about charts and finish a chapter on Excel sheets. Yes, in the next section we will talk about working with Excel charts.

Building charts within Excel sheets

Excel sheets are capable of building a variety of charts, including line chart, bar chart, and pie charts among others that help us depict trends and visualize data.

Getting ready

For this recipe, we will use the same `xlsxwriter` module and use methods defined in the module to build charts.

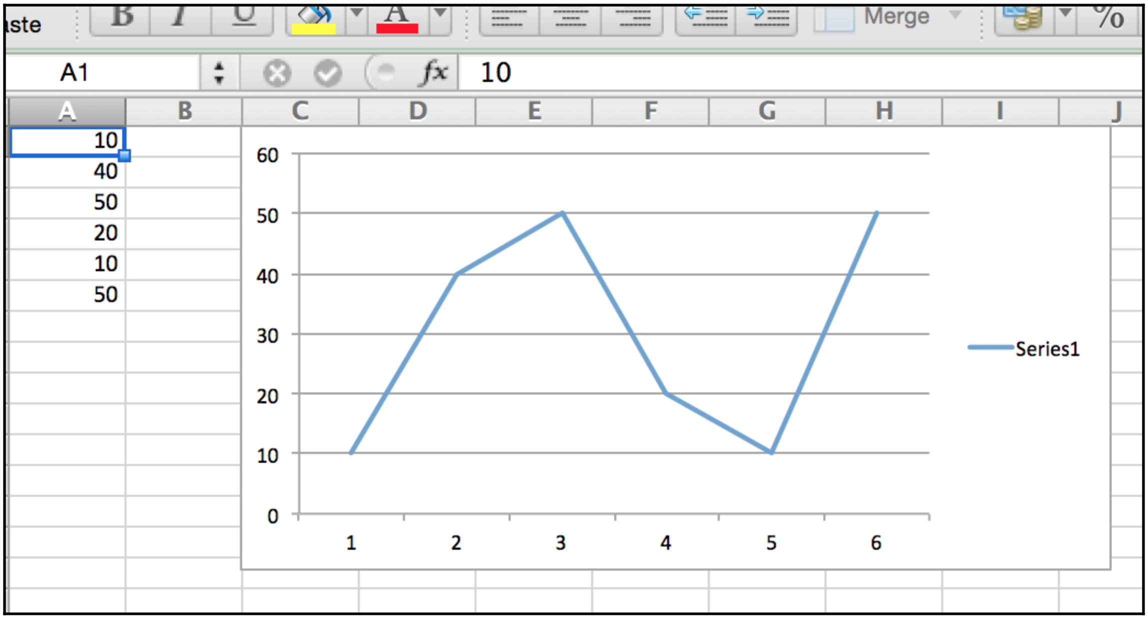
How to do it…

1. In this example, we will write a column in an Excel file that is filled with numbers. We can take the values in all the cells and construct a line chart:

```
import xlsxwriter

workbook = xlsxwriter.Workbook('chart_line.xlsx')
worksheet = workbook.add_worksheet()

data = [10, 40, 50, 20, 10, 50]
worksheet.write_column('A1', data)
chart = workbook.add_chart({'type': 'line'})
chart.add_series({'values': '=Sheet1!$A$1:$A$6'})
worksheet.insert_chart('C1', chart)

workbook.close()
```

In the preceding code snippet, we have a list of data that has integer values ranging from `10` to `50`. As usual, we create a workbook with the `Workbook()` method and add a default **Sheet1** worksheet. We then write a new column with all the numbers present in the list data.

2. The `add_chart()` method then defines the type of chart. In this case, it's a line chart. The `add_chart()` method returns an object of type chart. But simply creating an object doesn't help. How will the chart know the data points to be plotted? This happens with the `add_series()` method that takes the cell values to plot the graph. In this case, the cell ranges from **A1** to **A6** (remember we have added all the numbers from the `data` list to column **A** beginning at **A1**).
3. Once the chart is ready, it should also be added on to the Excel sheet. This is achieved with the `insert_chart()` method that takes the cell name and chart object as arguments. In this example, the chart is inserted at cell **C1**.
4. When we run this program, a new file `chart_line.xlsx` gets created with the line graph inserted in it. The following screenshot shows the line graph and plotted data:



A1 fx 10

A	B
10	
40	
50	
20	
10	
50	

Automating the comparison of company financials

Our recipes on Excel sheets covered multiple aspects like reading/writing files, formatting cells, working with formulae, and charts. Let's solve a nice business case with the knowledge we gained in this chapter.

Monica is a Finance manager at Xtel Inc and is responsible for the company's earnings. Xtel Inc is looking for funding, and Monica is tasked with comparing the company financials based on the income statements of the last three years. This data will go to the investors so that they can make an appropriate decision about investing in Xtel Inc. Getting this data for three years will be easy, but the CFO of Xtel has asked Monica to get this data on a month-on-month basis for the last 5 years. Monica is worried about comparing the company financials for 60 months manually!
With the knowledge gained in this chapter, do you think you can help Monica?

Getting ready

Lets solve Monica's problem with a Python recipe. For this recipe, we will compare the financials of the last three years for Xtel Inc and plot the comparison in an Excel sheet using Python. We will do that with the help of the factors influencing the company's income statement, that is, the revenue, costs, and gross profits.

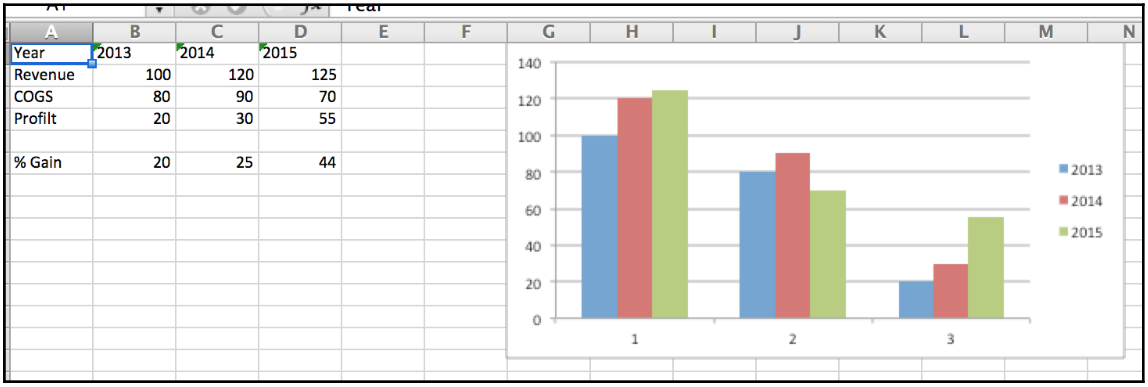
How to do it…

1. In the following code, we first add the information on the company financials, such as revenue, the cost of goods sold, and the gross profit. Assume that we have this data in a Python list `data`.
2. We then plot these values in a column chart and also calculate the net gain in percentages using Excel formulae. The following code snippet does exactly what Monica needs:

```
import xlsxwriter

workbook = xlsxwriter.Workbook('chart_column.xlsx')

worksheet = workbook.add_worksheet()
chart = workbook.add_chart({'type': 'column'})
```

```
        data = [
                ['Year', '2013', '2014', '2015'],
                ['Revenue', 100, 120, 125],
                ['COGS', 80, 90, 70],
                ['Profit', 20, 30, 55],
        ]

        worksheet.write_row('A1', data[0])
        worksheet.write_row('A2', data[1])
        worksheet.write_row('A3', data[2])
        worksheet.write_row('A4', data[3])

        chart.add_series({'values': '=Sheet1!$B$2:$B$4', 'name':'2013'})
        chart.add_series({'values': '=Sheet1!$C$2:$C$4', 'name':'2014'})
        chart.add_series({'values': '=Sheet1!$D$2:$D$4', 'name':'2015'})
        worksheet.insert_chart('G1', chart)

        worksheet.write(5, 0, '% Gain')
        worksheet.write(5, 1, '=(B4/B2)*100')
        worksheet.write(5, 2, '=(C4/C2)*100')
        worksheet.write(5, 3, '=(D4/D2)*100')

        workbook.close()
```

3. When we run this Python program, a new Excel sheet is generated that compares the company's financial performance across three years, as shown in the following screenshot. Exactly what Monica wanted! :)



A	B	C	D
Year	2013	2014	2015
Revenue	100	120	125
COGS	80	90	70
Profilt	20	30	55
% Gain	20	25	44

How it works…

In the preceding code snippet, we collect the data of the company financials in a Python list, `data`.

Using the `xlsxwriter` module, we create a workbook object and then add a worksheet to it with the `add_worksheet()` method.

Once we have the worksheet and the data, we start writing the data into the worksheet with the `write_row()` method.

We also add a chart object to our worksheet. This will help us add bar charts for comparing the company financials of the last three years easily. We add the chart object using the `add_chart()` method.

Since we have the data already populated in our sheet, we use this data to create bar charts for all three years using the `add_series()` method. The `add_series()` method takes the excel cells as parameters and plots the bar chart for the data in these cells. Finally, we insert the chart object (and the bar charts) in the worksheet with the `insert_chart()` method.

At last, we add the gain figures for all the years with excel formula using the `write()` method.

Cool! That was easy, you did it for Monica! She can modify this Python code snippet to compare the company financials for all the data points she needs and that too in a very short time. Indeed, the CEO of Xtel Inc will be very happy with her work!

There's more…

Well guys, that's it for this chapter. The fun with CSV and Excel files never ceases. There are many more operations that you can perform with these files, and they can be used in different ways in the business and software development world. So, I highly recommend you to try out the modules we discussed in this chapter and build on them for your own use case. See you in the next chapter!

3
Getting Creative with PDF Files and Documents

Word documents and PDF files are some of the most used file formats by business professionals. Want to send an invoice to you customer or send a set of requirements to your vendor, businesses often end up using PDF files and documents for their needs. Let's see how we can work with these file formats in Python.

In this chapter, we will cover the following recipes:

- Extracting data from PDF files
- Creating and copying PDF documents
- Manipulating PDFs (adding header/footer, merge, split, delete)
- Automating generation of payslips for finance department
- Reading Word documents
- Writing data into Word documents (adding headings, images, tables)
- Generating personalized new hire orientation for HR team in an automated way

Introduction

In the previous few chapters, we looked at working with CSV files and then extended our scope to learn about working with Excel worksheets. While CSV files are in a simple text format, Excel files are available in binary format.

In this chapter, we will discuss two more binary file formats: `.pdf` and `.docx`. You'll build knowledge on generating and reading PDF files, copying them and even manipulating them to build your own header and footer formats. Do you know you could merge many PDF files with a simple Python recipe?

This chapter also takes you on a journey of working with Word documents. It helps you build knowledge on reading and writing data into Word files. Adding tables, images, charts, you name it and this chapter covers it. Sounds interesting? Then this chapter is definitely for you!

Specifically, we will focus on the following Python modules in this chapter:

- `PyPDF2` (`https://pythonhosted.org/PyPDF2/`)
- `fpdf` (`https://pyfpdf.readthedocs.io/`)
- `python-docx` (`http://python-docx.readthedocs.io/en/latest/`)

While you'll learn about the majority of operations supported by `.pdf` and `.docx` files in this chapter, we won't be able to cover them in their entirety. I'd recommend you to try out the remaining APIs from the libraries discussed in this chapter.

Extracting data from PDF files

PDF (**Portable Document Format**) is a file format used to store data in documents agnostic to application software, hardware, and operating systems (hence the name, portable). PDF documents are fixed-layout flat files that include text and graphics and contain information needed to display the content. This recipe will show you how to extract information from PDF files and use the reader object.

Getting ready

To step through this recipe, you will need to install Python v2.7. To work with PDF files, we have `PyPDF2`, a nice module that can be installed with the following command:

```
sudo pip install PyPDF2
```

Already installed the module? So, let's get started!

How to do it…

1. On your Linux/Mac computer, go to Terminal and use Vim or choose your favorite editor.

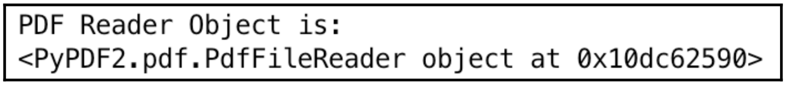
2. We start by downloading an existing PDF file from the Internet. Let's download the `diveintopython.pdf` file.

> You can search for this file on the Internet and easily obtain it. You'll also get the file if you download the code samples for this book.

3. Now, let's write the Python code for creating a PDF file reader object:

```
import PyPDF2
from PyPDF2 import PdfFileReader
pdf = open("diveintopython.pdf", 'rb')
readerObj = PdfFileReader(pdf)
print "PDF Reader Object is:", readerObj
```

The output of the preceding code snippet is as follows:

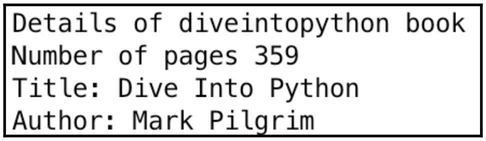


```
PDF Reader Object is:
<PyPDF2.pdf.PdfFileReader object at 0x10dc62590>
```

4. That's good; we now have a reader object of the PDF file. Let's move on to see what we can achieve with this object, based on the following Python code:

```
print "Details of diveintopython book"
print "Number of pages:", readerObj.getNumPages()
print "Title:", readerObj.getDocumentInfo().title
print "Author:", readerObj.getDocumentInfo().author
```

The output of the preceding code snippet is shown in the following screenshot. See how we use the `PdfFileReader` object to get the metadata of the file:

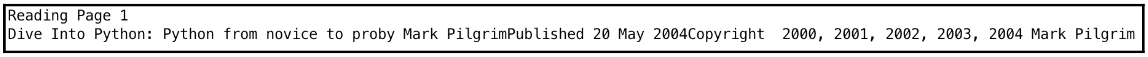


```
Details of diveintopython book
Number of pages 359
Title: Dive Into Python
Author: Mark Pilgrim
```

5. OK, that's neat! But we'd like to extract the contents of the file, wouldn't we? Let's go ahead and look at how to achieve this with a simple code snippet:

```
print "Reading Page 1"
page = readerObj.getPage(1)
print page.extractText()
```

So, what are we doing in the preceding code? I guess, the `print` statement is obvious. Yes, we read the first page of the `diveintopython` book. The following screenshot shows the contents of the first page of the `diveintopython` book:



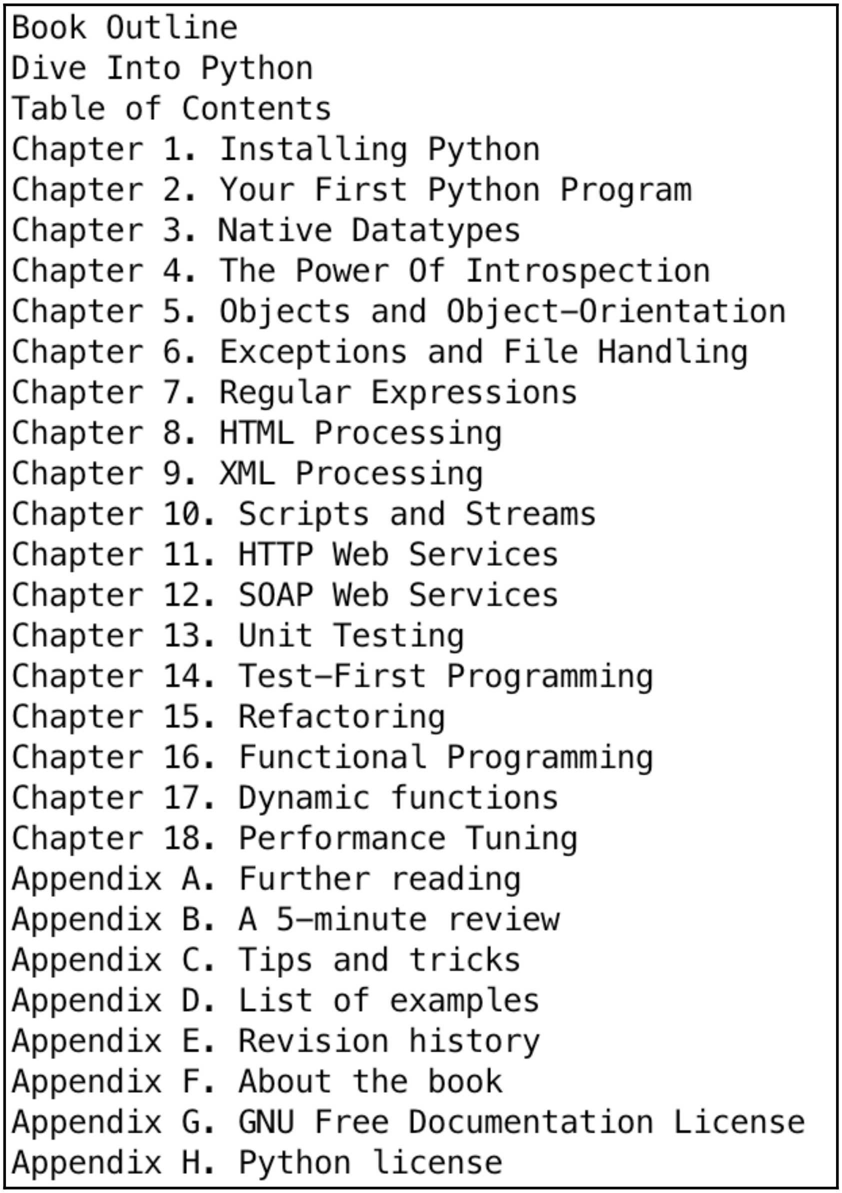
```
Reading Page 1
Dive Into Python: Python from novice to proby Mark PilgrimPublished 20 May 2004Copyright  2000, 2001, 2002, 2003, 2004 Mark Pilgrim
```

The contents are partial (as I couldn't fit the entire page in a screenshot), but as you can see, the contents are not in the same format as in the PDF file. This is a shortcoming with the text extract of the PDF file. Even though not 100%, we can still get the PDF file contents with decent accuracy.

6. Let's do one more interesting operation with the `PdfFileReader` object. How about getting the book outline with it? Yes, this is easily achievable in Python:

```
print "Book Outline"
for heading in readerObj.getOutlines():
    if type(heading) is not list:
        print dict(heading).get('/Title')
```

The output of the preceding code example can be seen in the following screenshot. As you can see, we get the complete outline of the book. In the beginning, we see the introduction of `Dive Into Python` and `Table of Contents`. Then we get the names of all the chapters, starting from `Chapter 1` to `Chapter 18`, and also the appendix, from `Appendix A` to `Appendix H`:



```
Book Outline
Dive Into Python
Table of Contents
Chapter 1. Installing Python
Chapter 2. Your First Python Program
Chapter 3. Native Datatypes
Chapter 4. The Power Of Introspection
Chapter 5. Objects and Object-Orientation
Chapter 6. Exceptions and File Handling
Chapter 7. Regular Expressions
Chapter 8. HTML Processing
Chapter 9. XML Processing
Chapter 10. Scripts and Streams
Chapter 11. HTTP Web Services
Chapter 12. SOAP Web Services
Chapter 13. Unit Testing
Chapter 14. Test-First Programming
Chapter 15. Refactoring
Chapter 16. Functional Programming
Chapter 17. Dynamic functions
Chapter 18. Performance Tuning
Appendix A. Further reading
Appendix B. A 5-minute review
Appendix C. Tips and tricks
Appendix D. List of examples
Appendix E. Revision history
Appendix F. About the book
Appendix G. GNU Free Documentation License
Appendix H. Python license
```

How it works…

In the first code snippet, we used the PdfFileReader class from the PyPDF2 module to generate an object. This object opens up the possibilities of reading and extracting information from the PDF file.

In the next code snippet, we used the PdfFileReader object to get the metadata of the file. We got the book details, such as the number of pages in the book, the book's title, and also the name of the author.

In the third example, we used the reader object that was created from the PdfFileReader class and pointed to the first page of the diveintopython book. This creates a page object represented by the page variable. We then used the page object and read the contents of the page with the extractText() method.

Finally, in the last code snippet of this recipe, we used the getOutlines() method to retrieve the book's outline as an array. The outline not only returns the topic's titles but also returns the subtopics under the main topic. In our example though, we filtered the subtopics and just printed the main outline of the book as seen in the screenshot.

There's more…

Cool, so we looked at multiple things that we could achieve with PdfFileReader. You learnt about reading the file metadata, reading the outline, browsing to a given page in a PDF file, and extracting the text information. All this is great, but hey, we'd like to create new PDF files, right?

Creating and copying PDF documents

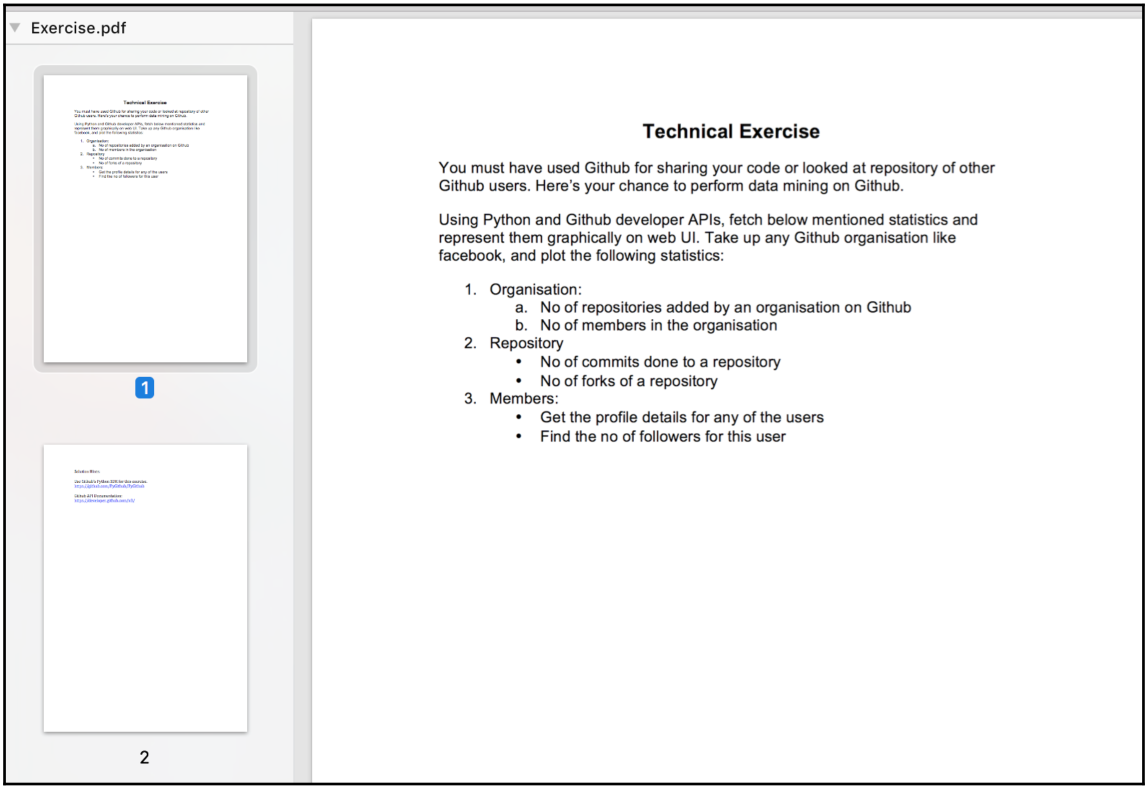
Working with PDFs adds more value when you can create them from scratch programmatically. Let's see how we can create our own PDF files in this section.

Getting ready

We will continue to use the PyPDF2 module for this recipe and will deal with its PdfFileWriter and PdfFileMerger classes. We will also use another module, fpdf, to demonstrate adding content to PDF files. We will talk about this later in the recipe.

How to do it…

1. We can create a PDF file in multiple ways; in this example, we copy the contents of an old file to generate a new PDF file. We start by taking an existing PDF–`Exercise.pdf`. The following screenshot shows the contents of this file. It contains two pages; the first page is a technical exercise and the second page gives possible hints for the solution of the exercise, as shown in the following screenshot:



Technical Exercise

You must have used Github for sharing your code or looked at repository of other Github users. Here's your chance to perform data mining on Github.

Using Python and Github developer APIs, fetch below mentioned statistics and represent them graphically on web UI. Take up any Github organisation like facebook, and plot the following statistics:

1. Organisation:
    a. No of repositories added by an organisation on Github
    b. No of members in the organisation
2. Repository
    - No of commits done to a repository
    - No of forks of a repository
3. Members:
    - Get the profile details for any of the users
    - Find the no of followers for this user

2. We will create a new PDF file by reading `Exercise.pdf` and writing the contents of the first page of the exercise into the new file. We will also add a blank page to the newly created PDF file. Let's start by writing some code:

```
from PyPDF2 import PdfFileReader, PdfFileWriter
infile = PdfFileReader(open('Exercise.pdf', 'rb'))
outfile = PdfFileWriter()
```

In the preceding code, we import the appropriate classes from the PyPDF2 module. As we need to read the Exercise.pdf file and then write the contents into a new PDF file, we will need both the PdfFileReader and PdfFileWriter classes. We then go ahead and open the exercise file in read mode with the open() method and create a reader object, infile. Later, we instantiate PdfFileWriter and create an object, outfile, which will be used to write the contents to the new file.

3. Let's move on and add a blank page to the outfile object using the addBlankPage() method. The page dimensions are typically 8.5 x 11 inches, but in this case, we need to convert them into units, which is 612 x 792 point.

Point is a desktop publishing point also known as PostScript point. 100 point=1.38 inch.

4. Next, we read the contents of the first page of Exercise.pdf with the getPage() method. Once we have the page object p, we pass this object to the writer object. The writer object uses the addPage() method to add the contents to the new file:

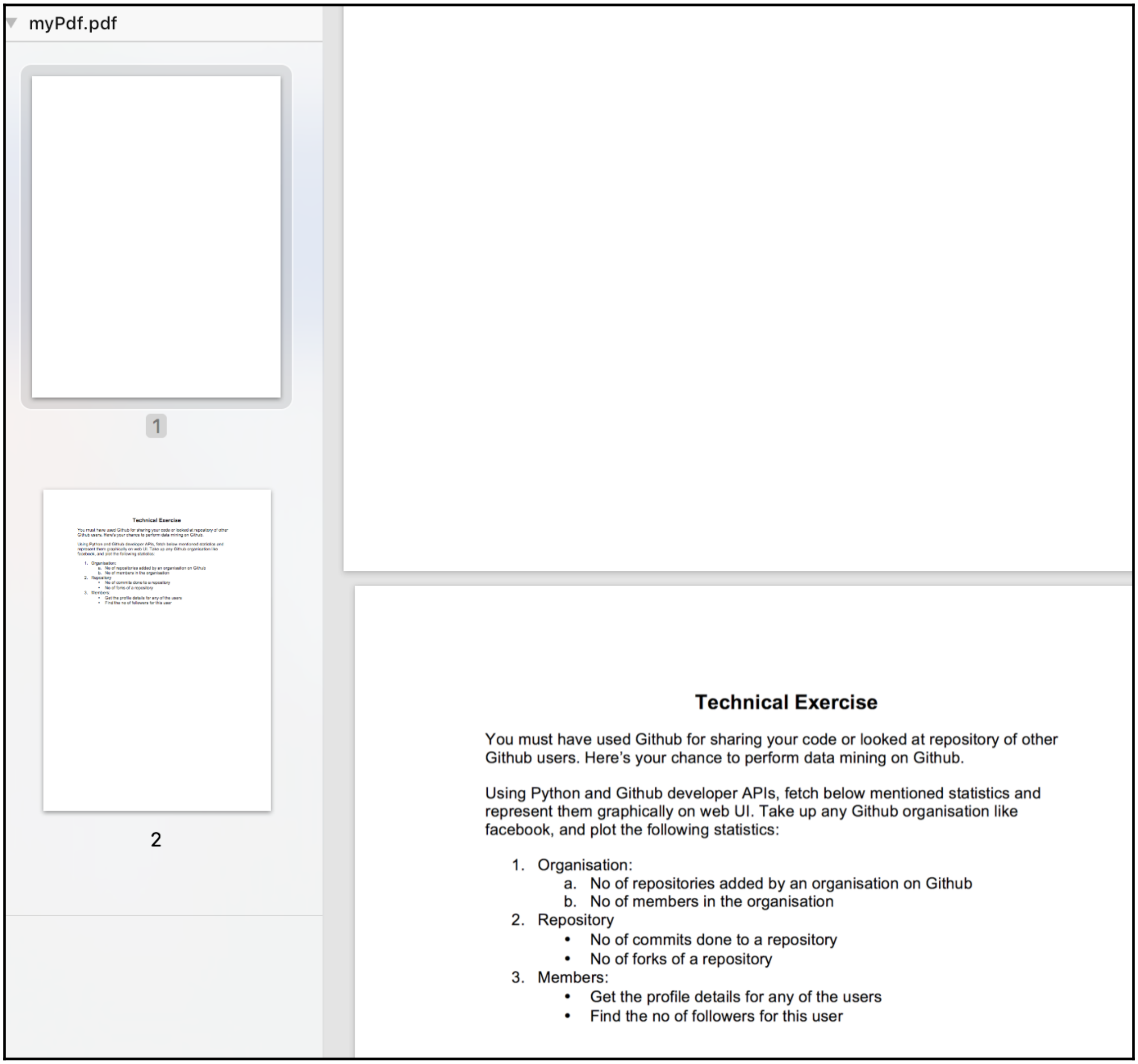
```
outfile.addBlankPage(612, 792)
p = infile.getPage(0)
outfile.addPage(p)
```

So far, we have created an output PDF file object outfile, but haven't yet created the file.

5. OK, cool! Now we have the writer object and the contents to be written into the new PDF file. So, we create a new PDF file with the open() method and use the writer object to write the contents and generate the new PDF, myPdf.pdf (this is where the PDF file is available on the file system for us to view). The following code achieves this. Here, f is the file handle of the newly created PDF file:

```
with open('myPdf.pdf', 'wb') as f:
    outfile.write(f)
f.close()
```

The following screenshot shows the contents of the newly created PDF file. As you can see, the first page is the blank page and the second page contains the contents of the first page of the Exercise.pdf file. Sweet, isn't it!



Technical Exercise

You must have used Github for sharing your code or looked at repository of other Github users. Here's your chance to perform data mining on Github.

Using Python and Github developer APIs, fetch below mentioned statistics and represent them graphically on web UI. Take up any Github organisation like facebook, and plot the following statistics:

1. Organisation:
   a. No of repositories added by an organisation on Github
   b. No of members in the organisation
2. Repository
   - No of commits done to a repository
   - No of forks of a repository
3. Members:
   - Get the profile details for any of the users
   - Find the no of followers for this user

6. But hey, we always need to create a PDF file from scratch! Yes, there is another way to create a PDF file. For this we will install a new module `fpdf` using the following command:

   ```
   pip install fpdf
   ```

7. Let's look at a very basic example, as given in the following code snippet:

   ```
   import fpdf
   from fpdf import FPDF
   pdf = FPDF(format='letter')
   ```

   In this example, we instantiate the `FPDF()` class from the `fpdf` module and create an object, `pdf`, which essentially represents the PDF file. While creating the object, we also define the default format of the PDF file, which is `letter`. The `fpdf` module supports multiple formats, such as `A3`, `A4`, `A5`, `Letter`, and `Legal`.

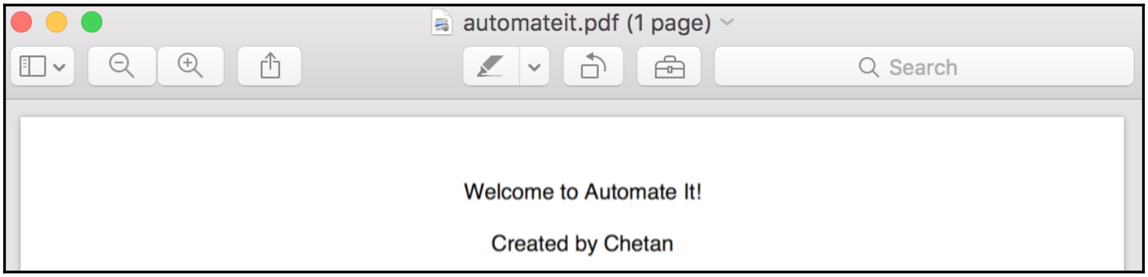
8. Next, we start inserting content into the file. But hey, the file is still empty, so before we write content, we use the `add_page()` method to insert a new page and also set the font using the `set_font()` method. We have set the font to `Arial` and its size to `12`:

   ```
   pdf.add_page()
   pdf.set_font("Arial", size=12)
   ```

9. Now we actually start writing content to the file with the `cell()` method. Cell is a rectangular area that contains some text. So, as you can see in the following code, we add a new line, `Welcome to Automate It!`, and follow it up with another line, `Created by Chetan`. There are a few things that you must have observed. 200 x 10 is the height x width of the cell. The `ln=1` designates a new line and `align=C` aligns the text to the center of the page. You may get into issues when adding long text to a cell, but the `fpdf` module has a `multi_cell()` method, which automatically breaks long lines of text with the available effective page width. You can always calculate the page width:

   ```
   pdf.cell(200, 10, txt="Welcome to Automate It!", ln=1,
   align="C")
   pdf.cell(200,10,'Created by Chetan',0,1,'C')
   pdf.output("automateit.pdf")
   ```

The output of the preceding code is a PDF file with contents as shown in the following screenshot:



Welcome to Automate It!

Created by Chetan

Manipulating PDFs (adding header/footer, merge, split, delete)

Ever wondered if you could merge PDF files programatically in a few seconds? Or could update header and footer of many PDF files in a jiffy? In this recipe, lets move on to do some interesting and most frequently performed operations on PDF files in this recipe.

Getting ready

For this recipe, we will use the `PyPDF2` and `fpdf` modules that were installed for the earlier recipes.

How to do it...

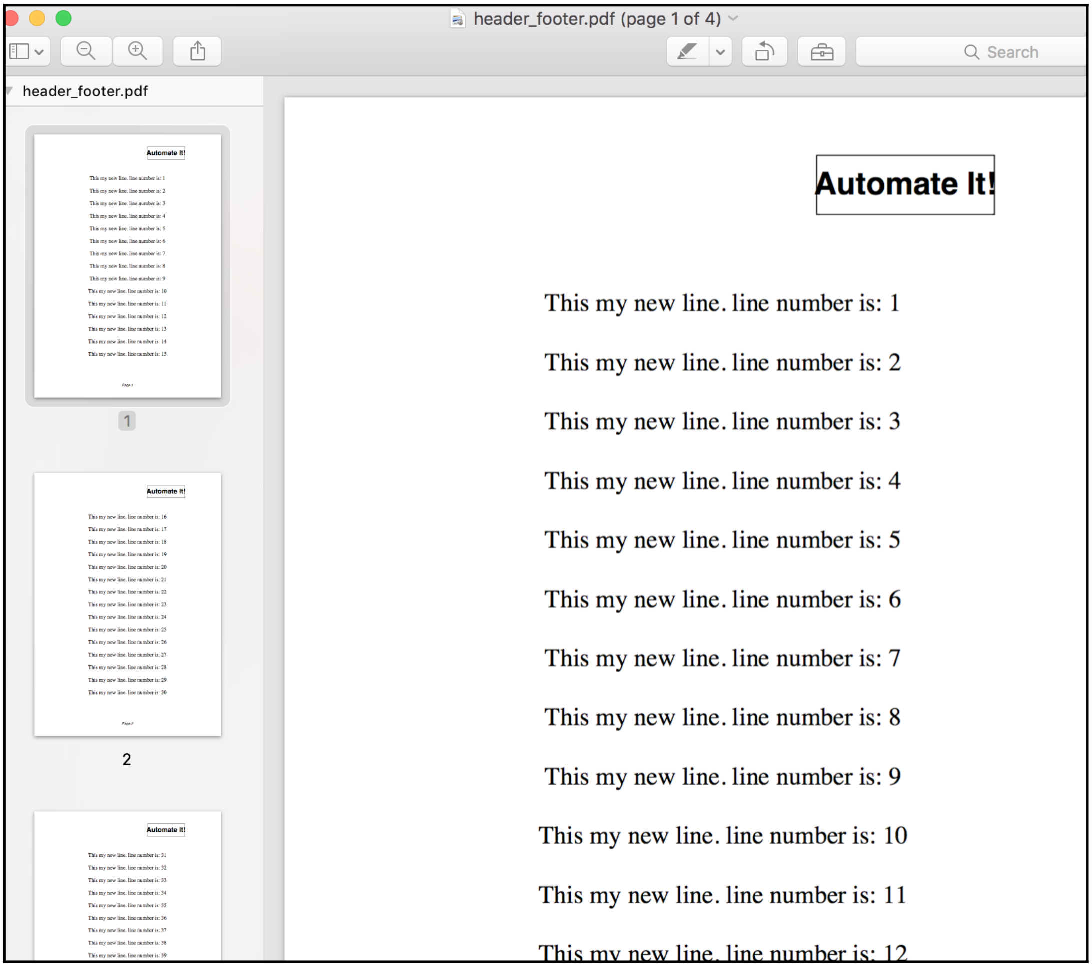
1. Let's start by working with the `PdfFileMerge` class of the `PyPDF2`. We use this class to merge multiple PDF files. The following code example does exactly the same:

```
from PyPDF2 import PdfFileReader, PdfFileMerger
import os
merger = PdfFileMerger()
files = [x for x in os.listdir('.') if
x.endswith('.pdf')]
for fname in sorted(files):
    merger.append(PdfFileReader(open(
                    os.path.join('.', fname), 'rb')))
merger.write("output.pdf")
```

2. If you run the preceding piece of code, it will generate a new file, `output.pdf`, which would have merged multiple PDF files. Open the `output.pdf` file and see for yourself.
3. That was cool! Now, how about adding a header and footer to a PDF file. Let's invoke the example from the previous recipe where we used the `fpdf` module to generate a PDF file (`automateit.pdf`). Now, what if we have to create a similar file with header and footer information? The following code does exactly that:

```
from fpdf import FPDF
class PDF(FPDF):
    def footer(self):
        self.set_y(-15)
        self.set_font('Arial', 'I', 8)
        self.cell(0, 10, 'Page %s' % self.page_no(), 0, 0, 'C')
    def header(self):
        self.set_font('Arial', 'B', 15)
        self.cell(80)
        self.cell(30, 10, 'Automate It!', 1, 0, 'C')
        self.ln(20)
pdf = PDF(format='A5')
pdf.add_page()
pdf.set_font("Times", size=12)
for i in range(1, 50):
    pdf.cell(0, 10, "This my new line. line number is:
             %s" % i, ln=1, align='C')
pdf.output("header_footer.pdf")
```

The output of the preceding code snippet can be viewed in the following screenshot. Look how we're able to manipulate the header and footer of our PDF document:



Automate It!

This my new line. line number is: 1

This my new line. line number is: 2

This my new line. line number is: 3

This my new line. line number is: 4

This my new line. line number is: 5

This my new line. line number is: 6

This my new line. line number is: 7

This my new line. line number is: 8

This my new line. line number is: 9

This my new line. line number is: 10

This my new line. line number is: 11

This my new line. line number is: 12

4. Wow! That was nice; now, let's quickly cover some more operations. Remember we added a blank page to the `myPdf.pdf` file in the previous recipe? What if I want to remove blank pages from PDF files?

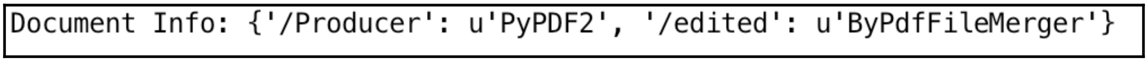
```
infile = PdfFileReader('myPdf.pdf', 'rb')
output = PdfFileWriter()

for i in xrange(infile.getNumPages()):
    p = infile.getPage(i)
    if p.getContents():
        output.addPage(p)
with open('myPdf_wo_blank.pdf', 'wb') as f:
    output.write(f)
```

If you run the Python code and look at the contents of `myPdf_wo_blank.pdf`, you will just see one page and the blank page would be removed.

5. Now, what if we want to add specific meta information to our file? We should be able to easily edit the metadata for a PDF file with the following code in Python:

```
from PyPDF2 import PdfFileMerger, PdfFileReader
mergerObj = PdfFileMerger()
fp = open('myPdf.pdf', 'wb')
metadata = {u'/edited':u'ByPdfFileMerger',}
mergerObj.addMetadata(metadata)
mergerObj.write(fp)
fp.close()
pdf = open("myPdf.pdf", 'rb')
readerObj = PdfFileReader(pdf)
print "Document Info:", readerObj.getDocumentInfo()
pdf.close()
```

The output of the preceding code can be seen in the following screenshot. See how we were successful in adding edited metadata to our PDF file.

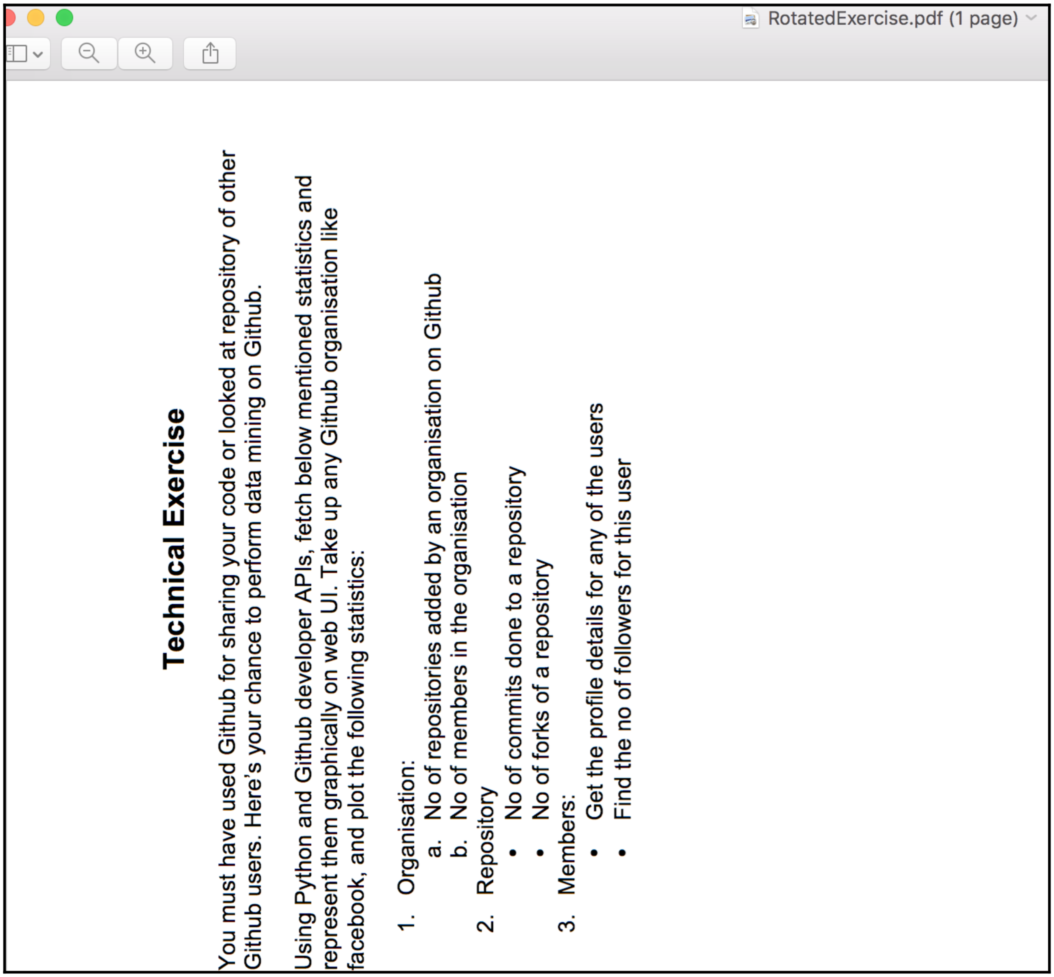


```
Document Info: {'/Producer': u'PyPDF2', '/edited': u'ByPdfFileMerger'}
```

6. Another good option to have from a development perspective is the ability to rotate a page in a PDF file. Yes, we could do that as well using the `PyPDF2` module. The following code rotates the first page of `Exercise.pdf` anticlockwise by `90` degrees:

```
from PyPDF2 import PdfFileReader
fp = open('Exercise.pdf', 'rb')
readerObj = PdfFileReader(fp)
page = readerObj.getPage(0)
page.rotateCounterClockwise(90)
writer = PdfFileWriter()
writer.addPage(page)
fw = open('RotatedExercise.pdf', 'wb')
writer.write(fw)
fw.close()
fp.close()
```

The following screenshot shows how the file looks when rotated anticlockwise:



Technical Exercise

You must have used Github for sharing your code or looked at repository of other Github users. Here's your chance to perform data mining on Github.

Using Python and Github developer APIs, fetch below mentioned statistics and represent them graphically on web UI. Take up any Github organisation like facebook, and plot the following statistics:

1. Organisation:
   a. No of repositories added by an organisation on Github
   b. No of members in the organisation
2. Repository
   - No of commits done to a repository
   - No of forks of a repository
3. Members:
   - Get the profile details for any of the users
   - Find the no of followers for this user

How it works…

In the first code snippet, we created an object of the `PdfFileMerger` class, called `merger`. We then went through all the files in the current working directory and selected all the files with the `.pdf` extension using Python's list comprehension.

We then sorted the files and ran a loop to pick up one file at a time, read it, and append it to the `merger` object.

Once all the files were merged, we used the `write()` method of the `merger` object to generate a single merged file: `output.pdf`.

In this example, we didn't have to create a file handle for the `output.pdf` file. The merger internally handles it and generates a nice PDF file.

In the second code snippet, we performed multiple operations:

1. We inherited the standard `FPDF` class and wrote our own class, `PDF`.
2. We overrode two methods–`header()` and `footer()`–to define how the header and footer should look when we create a new PDF file with our PDF class.
3. In the `footer()` method, we added page numbers to every page. The page numbers are in `Italics` with font size `8` and in `Arial` font. We also centered them and set them to appear 15 mm above the bottom of the page.
4. In the `header()` method, we created the header cell and positioned it to the extreme right. The title of the header is `Automate It` and it is in `Arial` and `Bold` with font size `15`. The title is also centered in the context of the cell. Lastly, we added a line break of 20 pixels below the header.
5. We then created our own PDF with the page format set to `A5`.
6. The contents of the PDF will be `This is my new line. Line number is <line_no>` with the font set to `Times` and the font size being `12`.
7. The generated PDF looks like the following screenshot. Note that the page is of `A5` size; hence, only 15 lines could be added to the page. If it had been of letter size, it would have accommodated at least 20 lines on a page.

In the third code example of this recipe, `getContents()` does the critical task of checking whether a given page has any content. So, when we start reading the old PDF file, we check the content of the page. If there is no content, the page is ignored and is not added to the new PDF file.

In the fourth snippet, we added the metadata information to our PDF file using the `addMetadata()` method. The `addMetadata()` method takes a key value pair as arguments, where we can pass the attributes that need to be modified for the PDF file. In our example, we used the method to add the `/edited` metadata field to the PDF file.

For the final example, I think the rest of the code is self-explanatory, except the use of `rotateCounterClockwise()`, which actually rotates the page. We could use `rotateClockwise()` to rotate the page in a clockwise direction as well.

There's more…

You learned about reading and writing PDF files and also understood many ways of manipulating PDF files. It's now time to put things into perspective with a real-life example.

Automating generation of payslips for finance department

Let's take an example of an organization use case, where the Finance Manager of the company wants to make the payslip generation process quicker. He realizes that not only is the task mundane but also consumes a lot of time. With more employees expected to join the company, it's going to get harder. He chooses to automate the process and approaches you. How can you help?

Well, with what you learnt throughout the chapter, I bet this will be a piece of cake for you! Let's work on it.

Getting ready

We don't need any special modules for this example. All the modules that have been installed as part of the previous recipes are enough for us, don't you think?

How to do it…

Let's first think of a payslip template. What does a payslip contain?

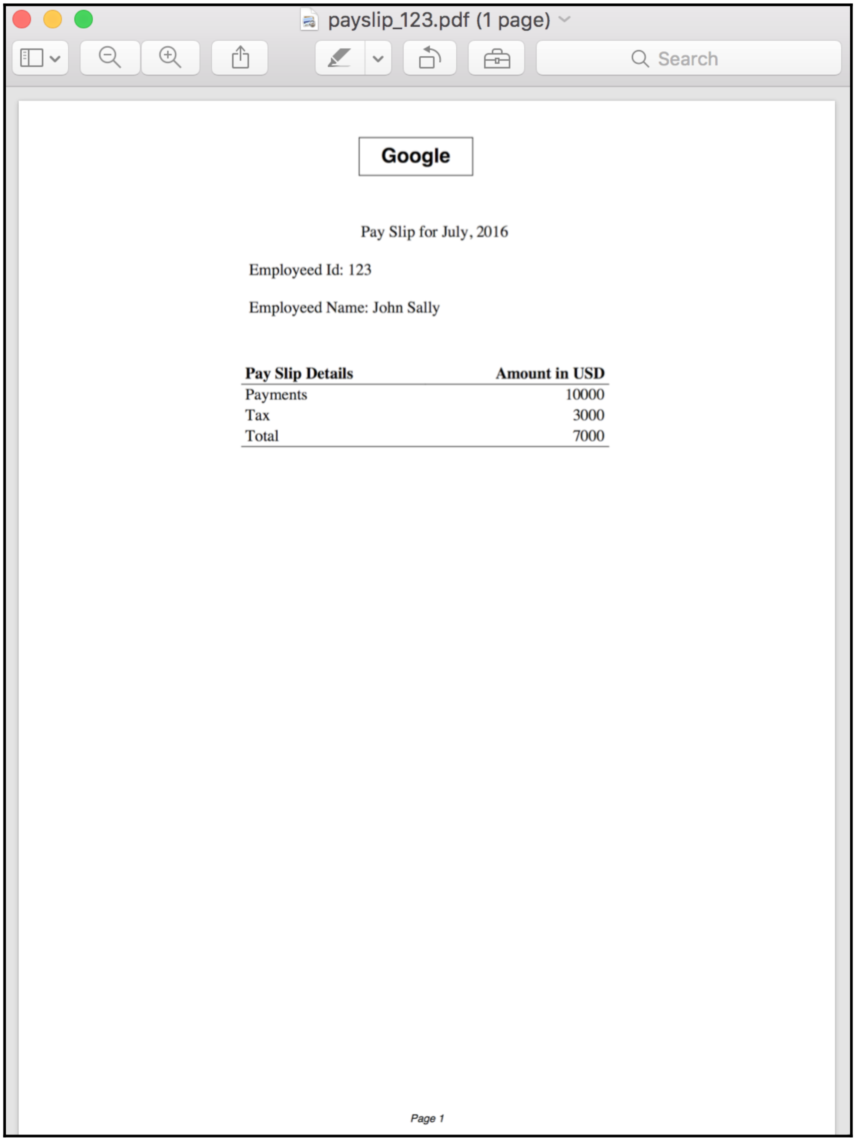
- Employee information
- Payments from the company
- Deductions (tax paid to government)
- Total payment made

So we need to get the employee information, add tables for payments and deductions, and add an entry for the total salary paid for the month.

The code implementation for this scenario could be as follows:

```
from datetime import datetime

employee_data = [
     { 'id': 123, 'name': 'John Sally', 'payment': 10000,
       'tax': 3000, 'total': 7000 },
     { 'id': 245, 'name': 'Robert Langford', 'payment': 12000,
       'tax': 4000, 'total': 8000 },
]
from fpdf import FPDF, HTMLMixin
class PaySlip(FPDF, HTMLMixin):
      def footer(self):
          self.set_y(-15)
          self.set_font('Arial', 'I', 8)
          self.cell(0, 10, 'Page %s' % self.page_no(), 0, 0, 'C')
      def header(self):
          self.set_font('Arial', 'B', 15)
          self.cell(80)
          self.cell(30, 10, 'Google', 1, 0, 'C')
          self.ln(20)
def generate_payslip(data):
     month = datetime.now().strftime("%B")
     year = datetime.now().strftime("%Y")
     pdf = PaySlip(format='letter')
     pdf.add_page()
     pdf.set_font("Times", size=12)
     pdf.cell(200, 10, txt="Pay Slip for %s, %s" %
              (month, year),  ln=3, align="C")
     pdf.cell(50)
     pdf.cell(100, 10, txt="Employeed Id: %s" % data['id'],
              ln=1, align='L')
     pdf.cell(50)
     pdf.cell(100, 10, txt="Employeed Name: %s" %
              data['name'], ln=3, align='L')
     html = """
         <table border="0" align="center" width="50%">
         <thead><tr><th align="left" width="50%">
          Pay Slip Details</th><th align="right" width="50%">
          Amount in USD</th></tr></thead>
         <tbody>
             <tr><td>Payments</td><td align="right">""" +
                     str(data['payment']) + """</td></tr>
             <tr><td>Tax</td><td align="right">""" +
                     str(data['tax']) + """</td></tr>
             <tr><td>Total</td><td align="right">""" +
                     str(data['total']) + """</td></tr>
         </tbody>
         </table>
         """
     pdf.write_html(html)
     pdf.output('payslip_%s.pdf' % data['id'])
for emp in employee_data:
     generate_payslip(emp)
```

This is how the payslip looks with the header, footer, and payslip details:



Google

Pay Slip for July, 2016

Employeed Id: 123

Employeed Name: John Sally

Pay Slip Details	Amount in USD
Payments	10000
Tax	3000
Total	7000

Page 1

How it works…

We first got the employee data in the dictionary `employee_data`. Now, in a real scenario, it could be from the employee table and would be retrieved with an SQL query. We wrote our own `PaySlip` class, which is inherited from the `FPDF` class, and defined our own header and footer.

We then wrote our own method to generate the payslip. This includes the header at the top with the company name (in this case, say **Google**) and the period for which the payslip is. We also added the **Employee Id** and **Employee Name**.

Now, this is interesting. We created an HTML document that generates a table and adds the payment, tax, and total salary information to the payslip with the `add_html()` method.

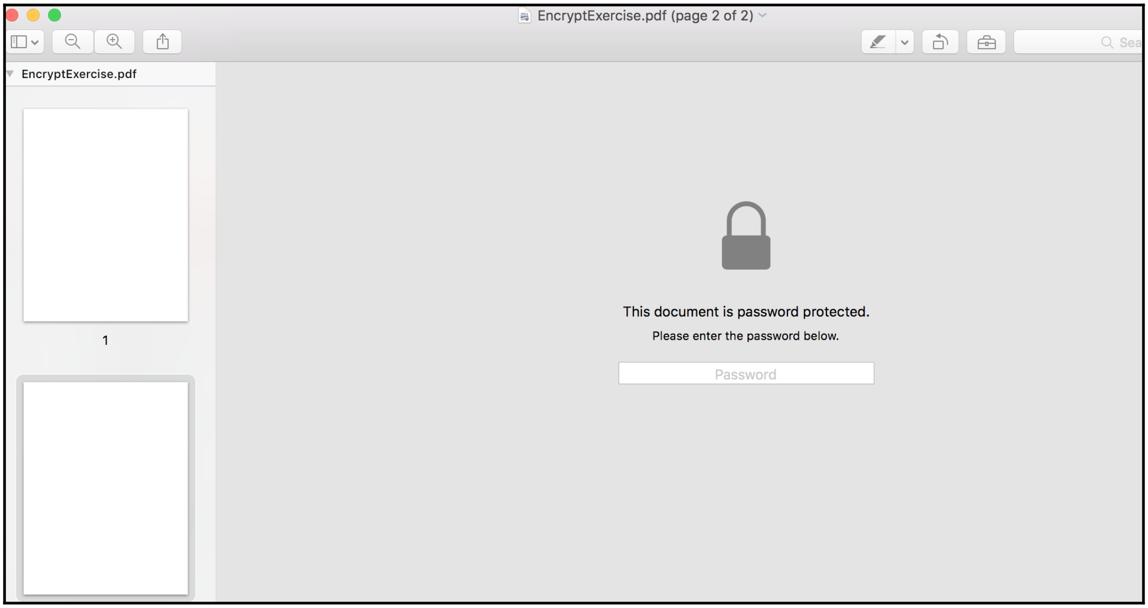
In the end, we added all this information to the PDF file with the `output()` method and named the payslip as `payslip_<employee_id>`.

There's more…

Even though we coded the example alright, do you think something is missing? Yeah, we didn't encrypt the PDF. It's always a great idea to secure the payslip with a password, so that nobody apart from the employee is able to view it. The following code will help us encrypt the file. In this example, we encrypt `Exercise.pdf`, secure it with the password, `P@$$w0rd`, and rename it to `EncryptExercise.pdf`:

```
from PyPDF2 import PdfFileWriter, PdfFileReader
fp = open('Exercise.pdf', 'rb')
readerObj = PdfFileReader(fp)

writer = PdfFileWriter()

for page in range(readerObj.numPages):
     writer.addPage(readerObj.getPage(page))
writer.encrypt('P@$$w0rd')

newfp = open('EncryptExercise.pdf', 'wb')
writer.write(newfp)
newfp.close()
fp.close()
```

If we open the secured file, it will ask you for the password:



This document is password protected.
Please enter the password below.

Well, that's an awesome solution! Your finance manager will be happy I must say! Wondering how do I decrypt the secure file? I'll leave that to you; it's fairly straightforward. Read the documentation.

We have come to the end of this section on working with PDF files. PDF files essentially store data in a binary format and support multiple operations as we discussed. In the next section we start working with the documents (`.docx`) and appreciate what they can offer!

Reading Word documents

As you might be aware, Microsoft Office started providing a new extension to Word documents, which is `.docx`, from Office 2007 onwards. With this change, documents moved to XML-based file formats (Office Open XML) with ZIP compression. Microsoft made this change when the business community asked for an open file format that could help with transferring data across applications. So, let's begin our journey with DOCX files!

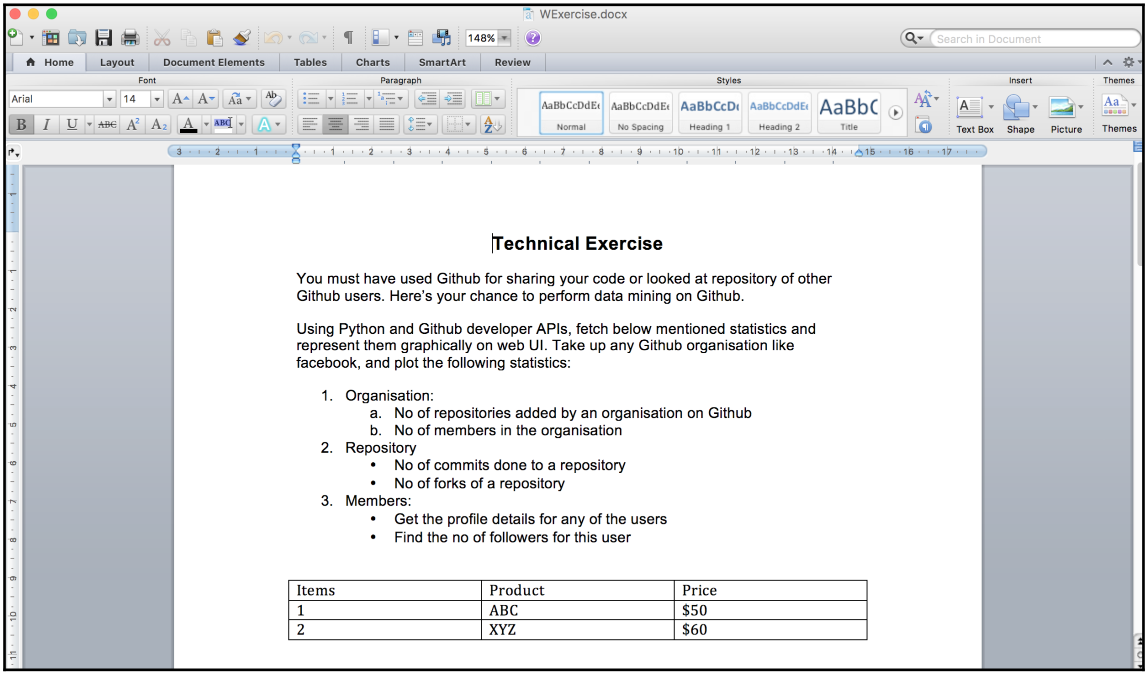
Getting ready

In this recipe, we will use the `python-docx` module to read Word documents. The `python-docx` is a comprehensive module that performs both read and write operations on Word documents. Let's install this module with our favorite tool, `pip`:

```
pip install python-docx
```

How to do it...

1. We start by creating our own Word document. It's the same exercise as we saw in the previous section while working with PDF files. Except for the fact that we have added a table to it and stored it as `WExercise.docx`. It looks as follows:



Technical Exercise

You must have used Github for sharing your code or looked at repository of other Github users. Here's your chance to perform data mining on Github.

Using Python and Github developer APIs, fetch below mentioned statistics and represent them graphically on web UI. Take up any Github organisation like facebook, and plot the following statistics:

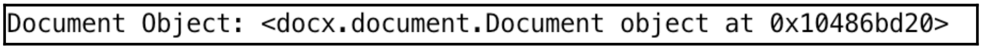
1. Organisation:
   a. No of repositories added by an organisation on Github
   b. No of members in the organisation
2. Repository
   - No of commits done to a repository
   - No of forks of a repository
3. Members:
   - Get the profile details for any of the users
   - Find the no of followers for this user

Items	Product	Price
1	ABC	$50
2	XYZ	$60

2. Let's now go ahead and read the document `WExercise.docx` file. The following code will help us in getting the object that points to the `WExercise.docx` file:

   ```
   import docx
   doc = docx.Document('WExercise.docx')
   print "Document Object:", doc
   ```

   The output of the preceding code is shown in the following screenshot. Reading a Word document is conceptually pretty similar to reading a file in Python. Just like how we created a file handle using the `open()` method, we create a document handle in this code snippet:

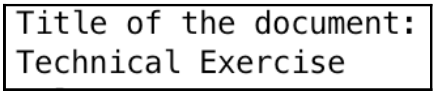


```
Document Object: <docx.document.Document object at 0x10486bd20>
```

3. Now, if we want to get the basic information about a document, we can use the document object, `doc`, from the preceding code. For instance, if we want to retrieve the title of a document, we can do that with the following code. If you look at the code carefully, we use the `paragraphs` object to get the text. Paragraphs are the lines present in the document. Assuming that the title of the document is the first line of the document, we get the `0` index for the paragraphs in the document and call the `text` attribute to get the text of the title:

   ```
   import docx
   doc = docx.Document('WExercise.docx')
   print "Document Object:", doc
   print "Title of the document:"
   print doc.paragraphs[0].text
   ```

   Notice in the following screenshot how we print the title of our exercise document:

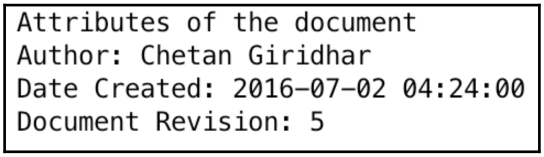


```
Title of the document:
Technical Exercise
```

4. Wow, that's cool! Let's go ahead and read other attributes of the Word document that we care about. Let's use the same `doc` object:

   ```
   print "Attributes of the document"
   print "Author:", doc.core_properties.author
   print "Date Created:", doc.core_properties.created
   print "Document Revision:", doc.core_properties.revision
   ```

The output of the preceding code is shown in the following screenshot. The author of the document is `Chetan Giridhar`. As you may have observed, it was created on July 2nd at 4:24 am. Also, note that the document has been changed five times, as this is the fifth revision of the document:



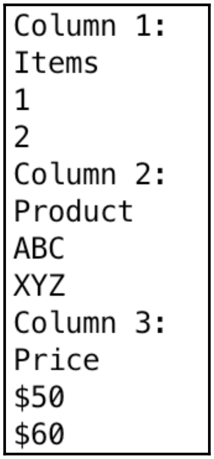
```
Attributes of the document
Author: Chetan Giridhar
Date Created: 2016-07-02 04:24:00
Document Revision: 5
```

5. Well, I will get more audacious now and read the table down there in the document. The `python-docx` module is awesome for reading tables. Look at the following code snippet:

```
table = doc.tables[0]

print "Column 1:"
for i in range(len(table.rows)):
    print table.rows[i].cells[0].paragraphs[0].text

print "Column 2:"
for i in range(len(table.rows)):
    print table.rows[i].cells[1].paragraphs[0].text

print "Column 3:"
for i in range(len(table.rows)):
    print table.rows[i].cells[2].paragraphs[0].text
```

6. In the preceding example, we used the `tables` object to read the tables in the document. Since we have only one table throughout the document, we get the first index with `tables[0]` and store the object in the `table` variable.
7. Each table contains rows and columns and they can be accessed with `table.rows` or `table.columns`. We used the `table.rows` to get the number of rows in the table.
8. Next, we iterated over all the rows and read the text in the cell with `table.rows[index].cells[index].paragraphs[0].text`. We needed the `paragraphs` object as this contains the actual text of the cell. (We again used the 0th index, as the assumption is that every cell has only one line of data.)
9. From the first `for` loop, you'd identify that we're reading all three rows but reading the first cell in every row. Essentially, we're reading the column values.

10. The output of the preceding code snippet shows all the columns with their values:



```
Column 1:
Items
1
2
Column 2:
Product
ABC
XYZ
Column 3:
Price
$50
$60
```

11. Nice! So, we're now experts in reading a Word document. But what's the use if we cant write data into a Word document? Let's look at how to write or create a `.docx` document in the next recipe.

Writing data into Word documents (adding headings, images, tables)

Reading files is a breeze with the `python-docx` module. Now, let's shift our focus to writing Word documents. We'll perform multiple operations with documents in this section.

Getting ready

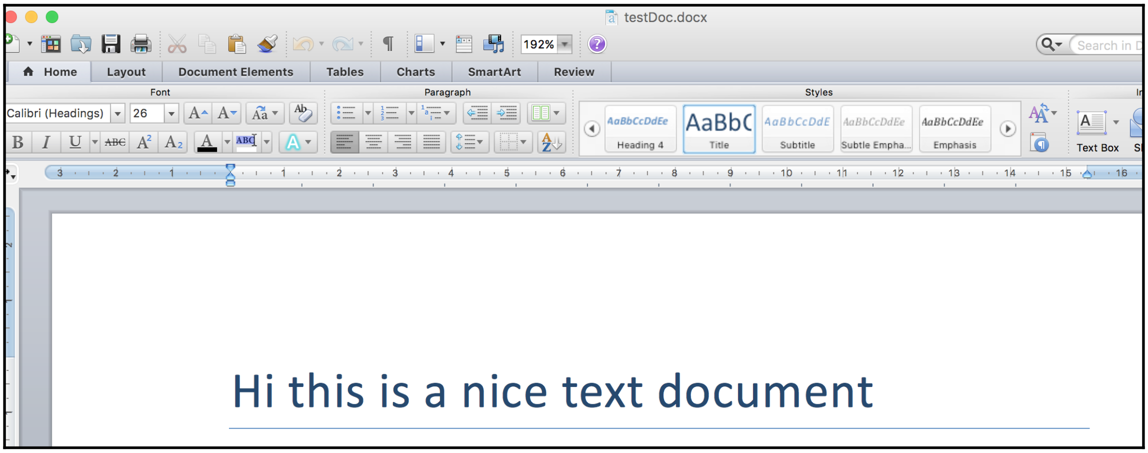
For this recipe, we will use the same fantastic Python module, `python-docx`. We don't need to spend much time on the setup. Let's get started!

How to do it...

1. We start with a very basic operation of creating a `.docx` file and then to add a heading to it. The following code performs this operation:

```
from docx import Document
document = Document()
document.add_heading('Test Document from Docx', 0)
document.save('testDoc.docx')
```

   This is how the document looks:

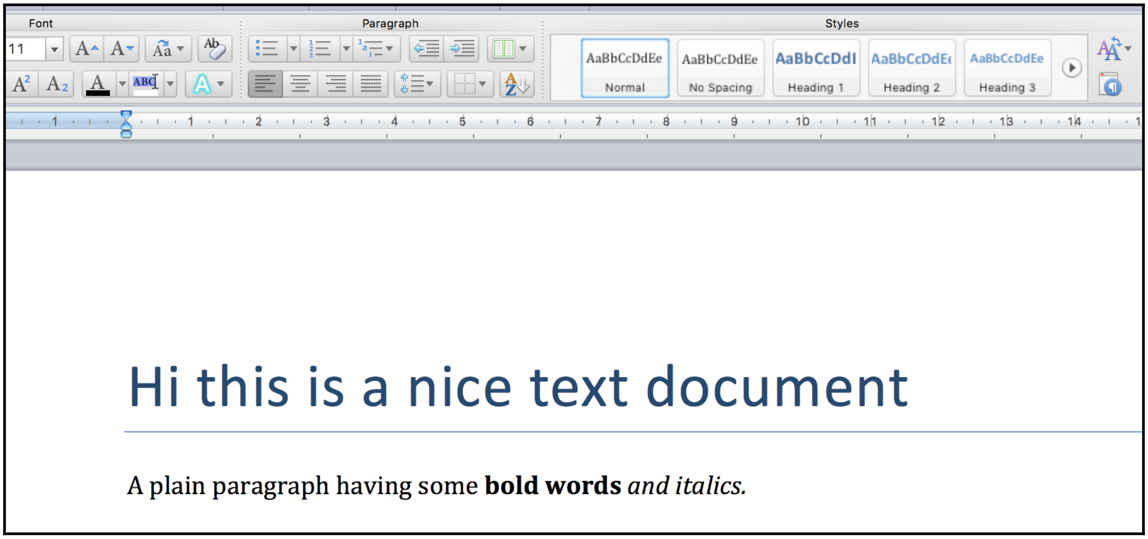


Hi this is a nice text document

2. If you look at the screenshot, you will see a new document being created with a string in it. Observe how the screenshot indicates it is styled as a **Title** text. How did we achieve this? Do you see `0` in the third line of our Python code? It talks about the heading type and styles the text accordingly. The `0` indicates title; `1` and `2` indicate text with **Heading 1** or **Heading 2**.
3. Let's move ahead and add a new line to the document. We decorate the string with some words in bold and some in italics:

```
document = Document('testDoc.docx')
p = document.add_paragraph('A plain paragraph
                            having some ')
p.add_run('bold words').bold = True
p.add_run(' and italics.').italic = True
document.save('testDoc.docx')
```

4. The document now looks as shown in the following screenshot. Observe the line added in the **Normal** style. Some words in the text are bold and few of them are in italics:



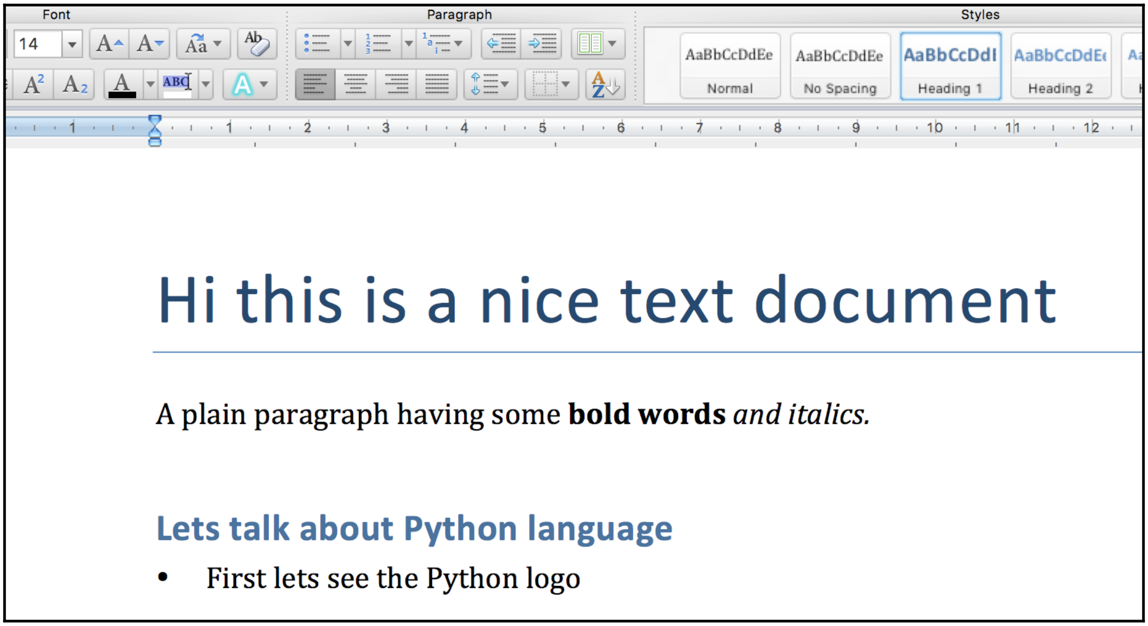
Hi this is a nice text document

A plain paragraph having some **bold words** *and italics.*

5. OK, good. Let's add another subtopic to our document. See the following code implementation. Here, we create a subtopic with style **Heading 1** and add a new line under this topic:

```
document = Document('testDoc.docx')
document.add_heading('Lets talk about Python
                      language', level=1)
document.add_paragraph('First lets see the Python
                        logo', style='ListBullet')
document.save('testDoc.docx')
```

6. The document now looks as shown in the following screenshots. When taking the screenshot, I had clicked on the line with **Heading 1** that shows up in the screenshot. Note how the subtopic is styled as a bullet point:



Hi this is a nice text document

A plain paragraph having some **bold words** *and italics.*

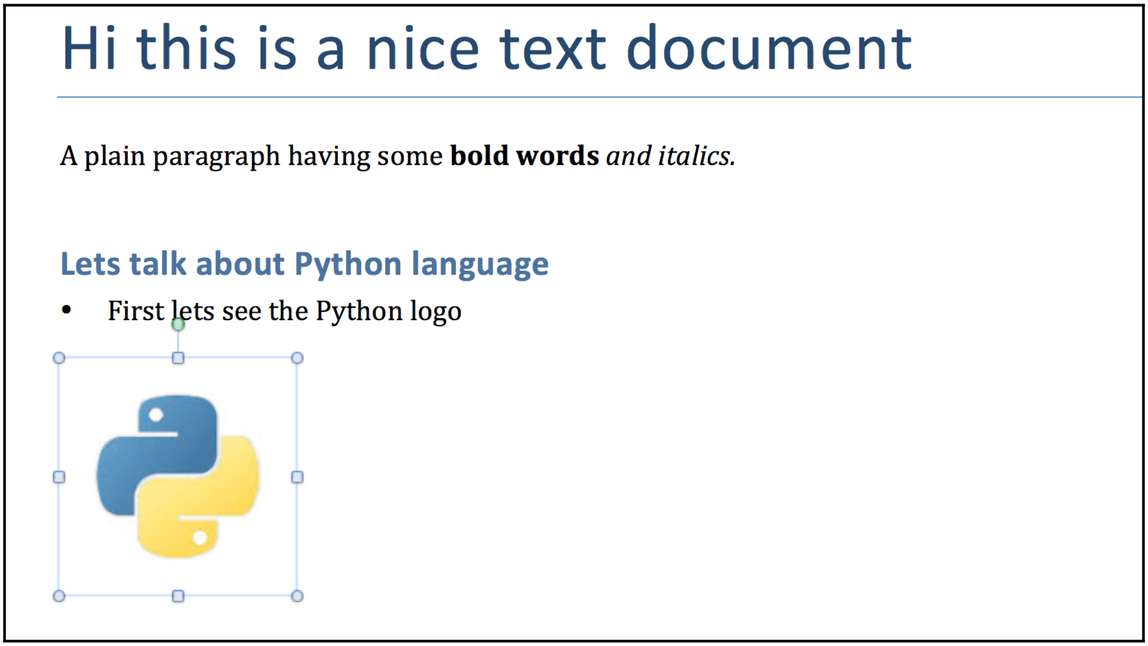
Lets talk about Python language

- First lets see the Python logo

7. There's often a need to include images in the documents. Now that's really, really, easy. Check out the following code for this step:

```
from docx.shared import Inches
document = Document('testDoc.docx')
document.add_picture('python.png',
                     width=Inches(1.25))
document.save('testDoc.docx')
```

If you run the Python code on your interpreter, you will see that the document now contains a nice Python logo. Please note that I had clicked on the image before taking the screenshot to catch your attention, so it's not done by the library:



Hi this is a nice text document

A plain paragraph having some **bold words** *and italics.*

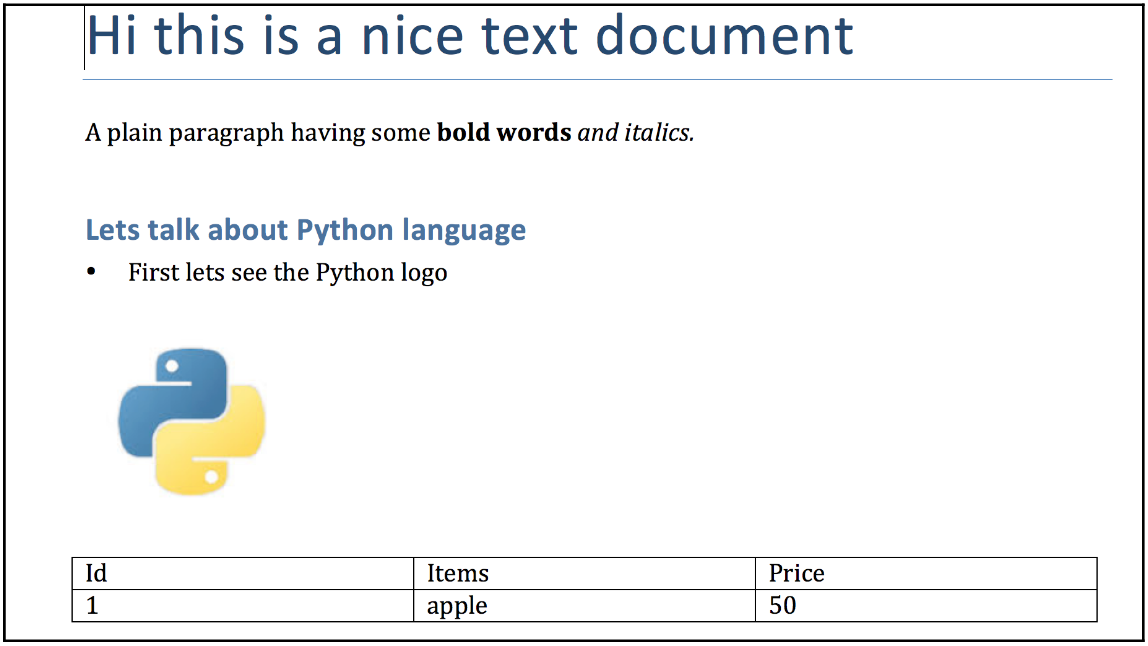
Lets talk about Python language

- First lets see the Python logo

8. Last but not least, we may also want to add tables to our document, right? Let's do that. The following code demonstrates adding tables to the DOCX file:

```
document = Document('testDoc.docx')
table = document.add_table(rows=1, cols=3)
table.style = 'TableGrid'

data = {'id':1, 'items':'apple', 'price':50}

headings = table.rows[0].cells
headings[0].text = 'Id'
headings[1].text = 'Items'
headings[2].text = 'Price'

row = table.add_row().cells
row[0].text = str(data.get('id'))
row[1].text = data.get('items')
row[2].text = str(data.get('price'))
document.save('testDoc.docx')
```

The following screenshot shows how the complete document looks along with the table. Nice!



Hi this is a nice text document

A plain paragraph having some **bold words** *and italics.*

Lets talk about Python language

- First lets see the Python logo

Id	Items	Price
1	apple	50

How it works…

In the first code snippet, we created the `document` object from the `Document` class. We then used this object to add a new heading, which contains the text `Hi this is a nice text document`. I know this is not a text document but just a string.

In the second example, adding a new line is done with the `add_paragraph()` method (remember, `paragraphs` was used to read the lines from the word document in the previous section). And how did we get the styling? That is possible with the `add_run()` method by setting the attributes `bold` and `italic` to `true`.

In the fourth example, we just used the `add_image()` method to add the picture to the document. We could also set the height and width of the image in inches. To do this, we imported a new class, `Inches`, and set the width of the image to 1.25 inches. Simple and neat!

In the final example, we added a table to the document by performing the following steps:

1. We started by creating a table object with the `add_table()` method. We configured the table to contain one row and three columns. We also styled the table to be a grid table.
2. As we saw in the previous section, the `table` object has the `rows` and `columns` objects. We used these to fill in the table with the dictionary `data`.
3. Then we added a heading to the table. Heading is the first row of the table and, hence, we used `table.rows[0]` to fill the data in it. We filled the first column by `Id`, the second by `Items`, and the third by `Price`.
4. After the heading, we added a new row and filled the cells of this row from the data dictionary.
5. If you look at the screenshot, the document now has one table added to it, where ID is `1`, item is `apple` and price is `50`.

There's more…

What you learnt in the preceding section were straightforward, frequently done, day-to-day operations of writing into DOCX files. We could perform many more operations programmatically as we are used to doing manually on word documents. Let's now bring the learning together in a business use case.

Generating personalized new hire orientation for HR team in an automated way

As the HR manager of your company, you are responsible for new hire orientation. You see that, every month, you have at least 15-20 new employees joining your organization. Once they complete a month in the company, you have to take them through your company policies in an orientation program.

For this, you need to send them a personalized document with new hire orientation details. Getting the details of employees one by one from the database is tedious; on top of that, you have to filter for employees who are due for orientation, based on different departments.

All this is time-consuming and you feel this process can be easily automated. Let's see how we can use the knowledge we have acquired so far in this chapter to automate this process.

Getting ready

For this recipe, we will use `python-docx`, which has been so helpful in our previous recipes. So, we need not install any new modules.

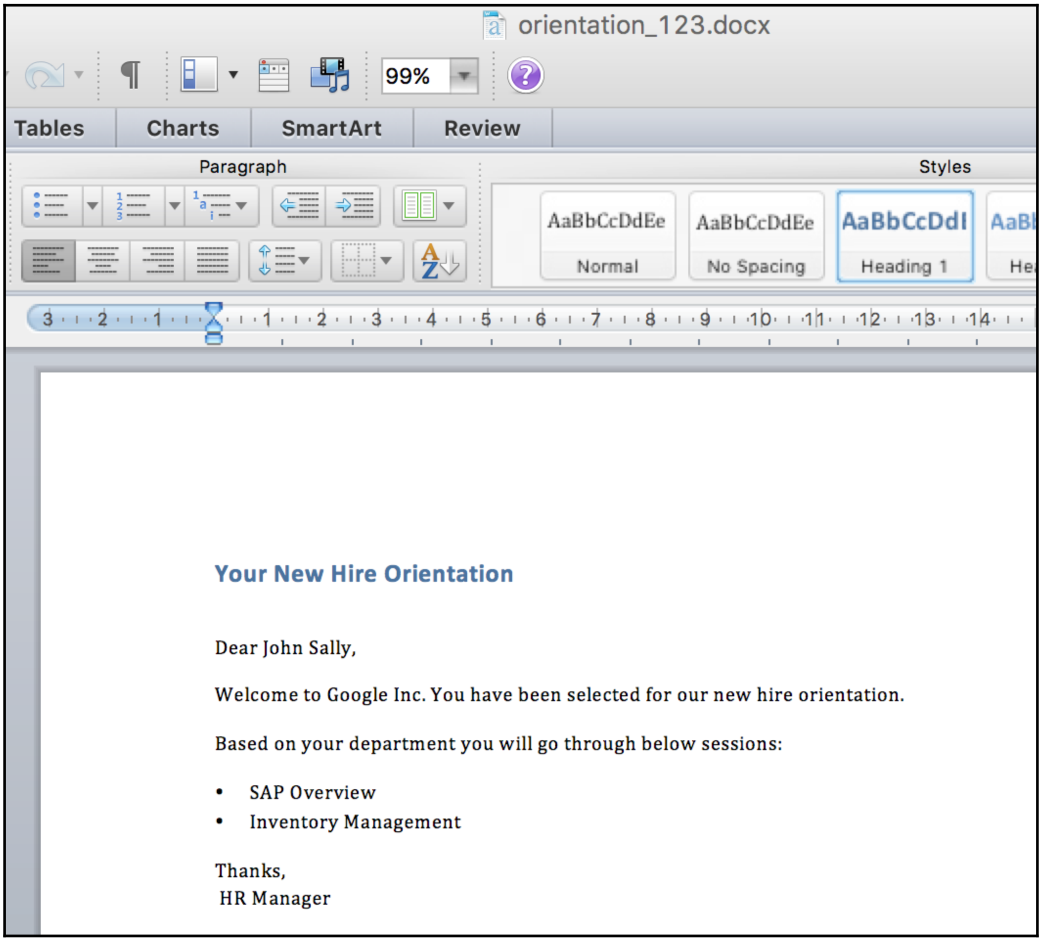
How to do it…

1. Let's first split the problem. First, we need to collect the employees who are due for orientation. Next, we need to know their department and look at the schedule template based on the department. Once these details are available, we need to put this together in a document.
2. Look at the code implementation for this scenario:

```
from docx import Document

employee_data = [
    {'id': 123, 'name': 'John Sally', 'department':
     'Operations', 'isDue': True},
    {'id': 245, 'name': 'Robert Langford',
    'department': 'Software', 'isDue': False},
]
agenda = {
    "Operations": ["SAP Overview", "Inventory Management"],
    "Software": ["C/C++ Overview", "Computer Architecture"],
    "Hardware": ["Computer Aided Tools", "Hardware Design"] }
def generate_document(employee_data, agenda):
    document = Document()
    for emp in employee_data:
        if emp['isDue']:
            name = emp['name']
            document.add_heading('Your New Hire
                            Orientationn', level=1)
            document.add_paragraph('Dear %s,' % name)
            document.add_paragraph('Welcome to Google
              Inc. You have been selected for our new
              hire orientation.')
           document.add_paragraph('Based on your
              department you will go through
              below sessions:')
           department = emp['department']
           for session in agenda[department]:
               document.add_paragraph(
                 session , style='ListBullet'
           )
```

```
document.add_paragraph('Thanks,n HR Manager')
document.save('orientation_%s.docx' % emp['id'])
generate_document(employee_data, agenda)
```

3. If you run this code snippet, this is how your document will look with all the relevant details about the orientation. Cool! But how did it work? We will see that in the *How it works* section.



Your New Hire Orientation

Dear John Sally,

Welcome to Google Inc. You have been selected for our new hire orientation.

Based on your department you will go through below sessions:

- SAP Overview
- Inventory Management

Thanks,
 HR Manager

How it works…

In the preceding code, we have a prefilled dictionary, employee_data, which contains employee information. This dictionary also contains information on whether an employee is due for orientation or not. We also have the agenda dictionary that acts as a template for different sessions based on the department. We have manually added all this data in Python dictionaries in this example, but in the real world, it needs to be pulled from your organization's database.

Next, we write a generate_document() method that takes employee_data and agenda. It iterates through all the employees and checks if a given employee is due for orientation and starts writing the document. First it adds a title, then follows up with a personalized address to the employee, and then shifts down to the sessions the employee needs to attend based on his or her department.

In the end, all the text is saved as a document with the name, orientation_<emp_id>.docx file.

That was cool! Imagine the time you saved. How happy are you as a HR manager? You acquired some new skills and quickly applied them for the benefit of your team. Awesome!

We have come to the end of this chapter on reading, writing, and manipulating PDF files and documents. Hope you enjoyed it and learnt many new things that you can apply to your work at office or at school! Of course you could do more; I highly encourage you to try out these modules and have fun with them.

4

Playing with SMS and Voice Notifications

Cloud telephony is the technology that moves your phone system to the cloud. This has made sure that we can now explore the possibilities of automation with SMS and voice notifications. This chapter begins with an introduction to cloud telephony and covers automation of business use cases with text and voice messages in Python.

In this chapter, we will cover the following recipes:

- Registering with a cloud telephony provider
- Sending text messages
- Receiving SMS messages
- SMS workflows for Domino's
- Sending voice messages
- Receiving voice calls
- Building your own customer service software

Introduction

In the previous few chapters, we looked at working with plain text and **comma-separated value** (**CSV**) files, and then we extended our scope to learn about working with Excel worksheets, Word documents, and PDF files. Excel, Word, and PDF files are available in the binary format and support mathematical operations, tables, charting, and many other operations. We also looked at interesting business use cases that can be automated with Python.

In this chapter, we take a look at an interesting world of cloud telephony. With the advent of the Internet, businesses have moved their communication systems to the cloud as well. Internet-based hosted telephony has replaced the conventional telephone equipment, such as PBX. This has opened up a world of possibilities of using cloud telephony to solve business needs, and that too, in Python. Using cloud telephony for your business allows you to make and receive multiple calls and SMS simultaneously. Services like call transfers, recording, bulk SMS are some of the awesome features cloud telephony can be leveraged for.
Cloud telephony ensures management of business needs without compromising on quality, cost, and without having to invest in any additional infrastructure.

The recipes in this chapter will focus on the Python modules that help us in sending/receiving SMS messages and voice calls. We'll learn how to register with a cloud telephony provider, use Python APIs, and automate interesting business flows. Specifically, we will work with the `Twilio` telephony provider and use the following Python modules in this chapter:

- `Flask` (`http://flask.pocoo.org/`)
- `twilio` (`https://pypi.python.org/pypi/twilio`)

While we'll learn about the Twilio cloud telephony provider, there are other providers as well. Each of these has a great API set and works with Python libraries. If you choose to implement your solution, you can have a look at a few of them at `http://www.capterra.com/telephony-software/`.

Registering with a cloud telephony provider

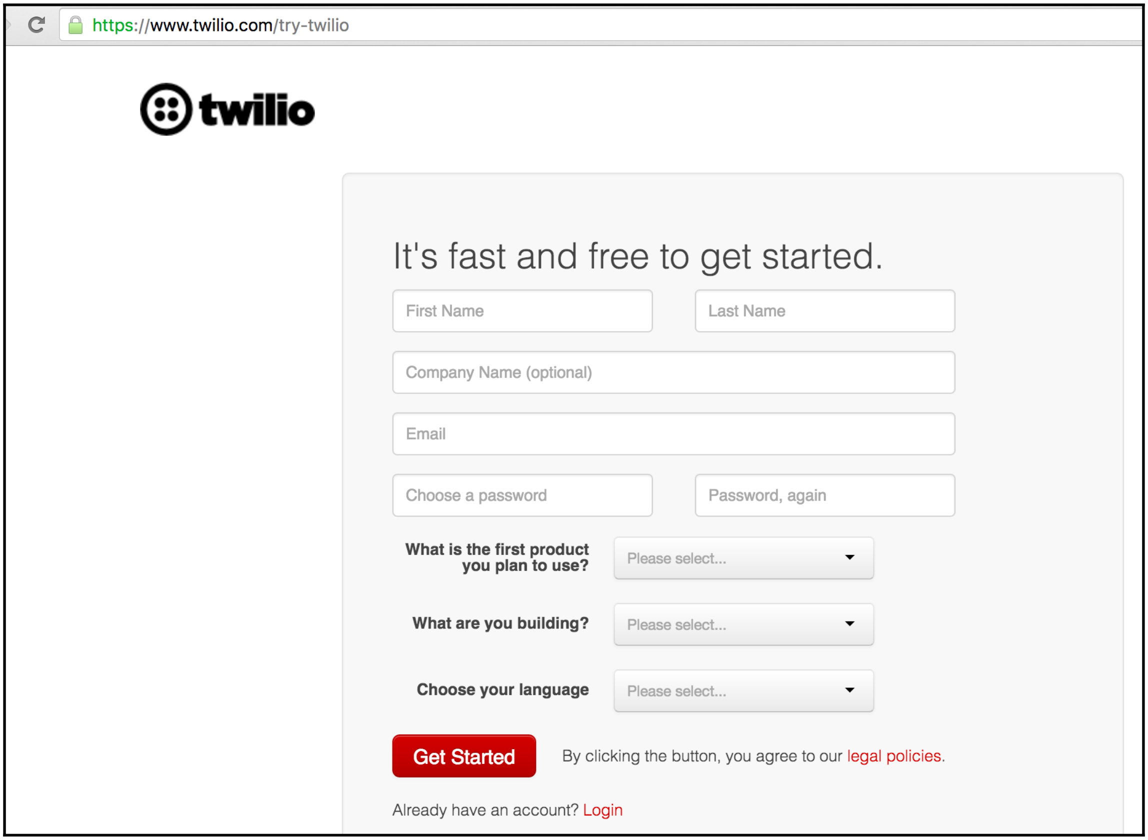
To work with cloud-based telephony services, we need to register for an account with a telephony provider. There are a couple of popular cloud providers that you can find when you search on the Internet. For this chapter, we use Twilio (`https://www.twilio.com/`). Let's see how to register for an account.

Getting ready

In order to use cloud telephony APIs, we need to register for an account with Twilio so that we can get the **AccountSID** and **AuthToken**. We'd also need to rent numbers for the recipes in the SMS and voice sections. Let's learn how to work with Twilio APIs in this recipe.

How to do it…

1. On your computer, open your favorite browser and browse to `https://www.twilio.com/try-twilio`:



It's fast and free to get started.

First Name

Last Name

Company Name (optional)

Email

Choose a password

Password, again

What is the first product you plan to use? Please select...

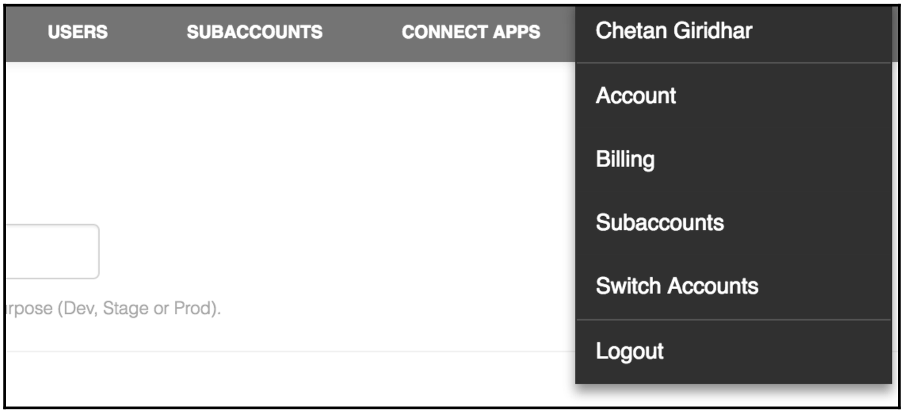
What are you building? Please select...

Choose your language Please select...

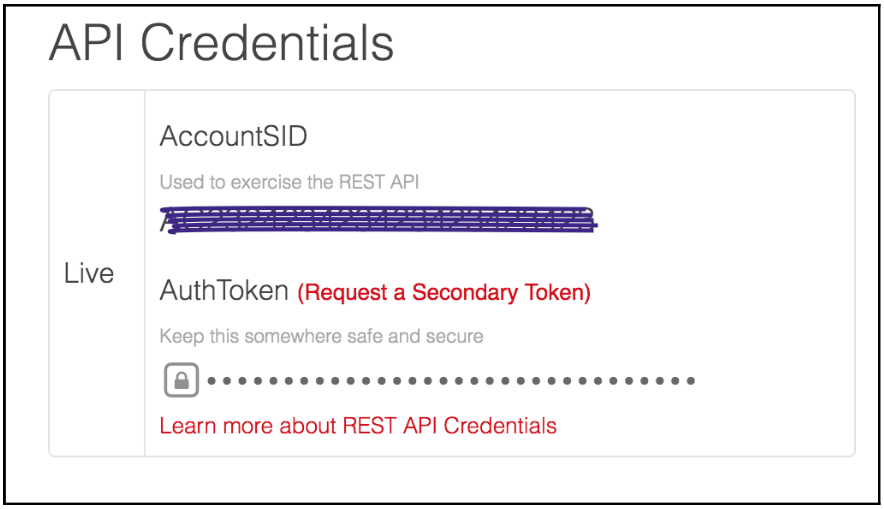
By clicking the button, you agree to our legal policies.

Already have an account? Login

2. Once you create an account, log in to it and add a few funds from the **Billing** page that is available as a dropdown option on your account dashboard. You can also directly browse to the **Billing** page at `https://www.twilio.com/user/billing`if you're already logged in:



3. To make Twilio API calls, we need **AccountSID** and **AuthToken**. We can get these details by clicking on the **Account** section from the dropdown or directly browsing to `https://www.twilio.com/user/account/settings`. On this page, you will get the API credentials as shown in the following screenshot. Don't worry about two-factor authentication for now, but make sure **SSL Certification Validation** is enabled for your account:



API Credentials

Live

AccountSID

Used to exercise the REST API

[redacted]

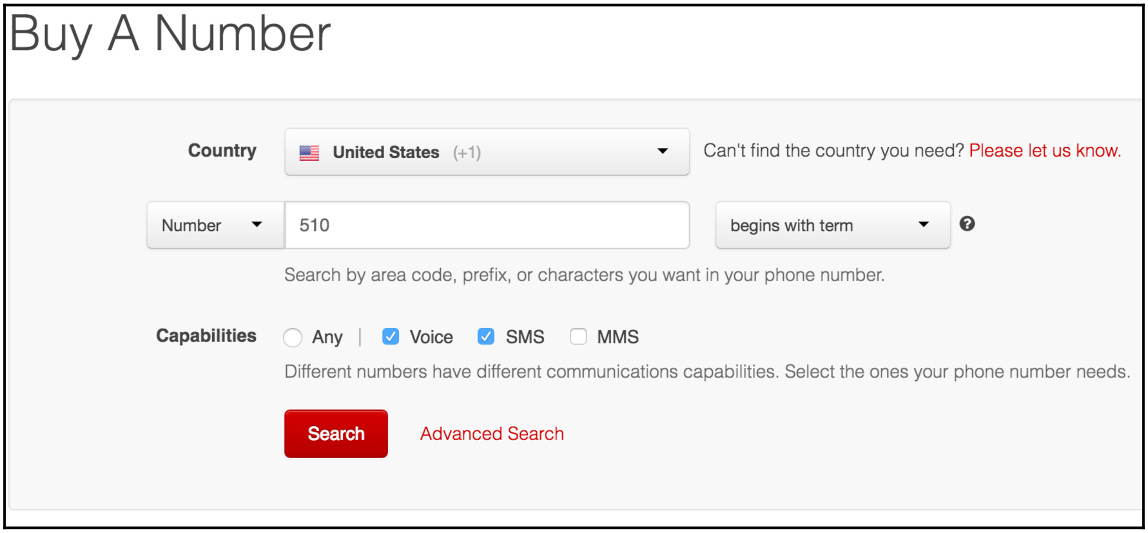
AuthToken (Request a Secondary Token)

Keep this somewhere safe and secure

••••••••••••••••••••••••••••••••

Learn more about REST API Credentials

4. Okay, that's good. Now, let's rent a phone number by directly browsing to `https://www.twilio.com/user/account/phone-numbers/search`.
5. Once you click on **Buy a Number**, you'll see a page where you can rent a number based on the country and prefix or area code. Here, I have chosen country as **United States** and prefix as `510`. I'm also renting a number that is capable of handling both **SMS** messages and **Voice** calls:



Buy A Number

Country: United States (+1)

Can't find the country you need? Please let us know.

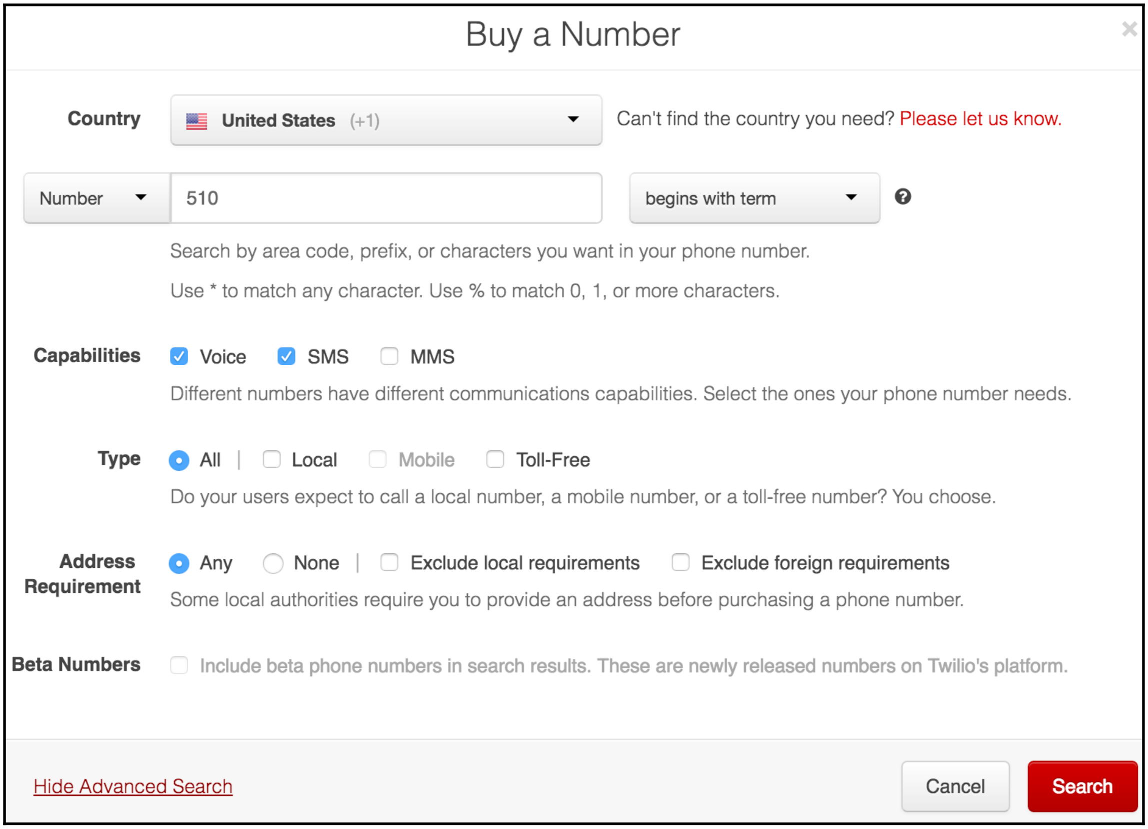
Number: 510 begins with term

Search by area code, prefix, or characters you want in your phone number.

Capabilities ○ Any | ☑ Voice ☑ SMS □ MMS

Different numbers have different communications capabilities. Select the ones your phone number needs.

6. Now, click on the **Advanced Search** option at the bottom of this page to get all the options, as shown in the following screenshot. You can go ahead with the default settings **All** or choose to rent **Local** or **Toll-Free** numbers. Based on the local rules of a region or country, renting numbers requires you to provide your address proof, but you don't need all this and could select the **Any** option. **Beta Numbers** are newly added numbers to Twilio from specific countries, added to support a list of countries. We don't have to care about this option in this recipe, so we keep it blank:



Buy a Number

Country United States (+1) ▾ Can't find the country you need? Please let us know.

Number ▾ 510 begins with term ▾ ?

Search by area code, prefix, or characters you want in your phone number.

Use * to match any character. Use % to match 0, 1, or more characters.

Capabilities ☑ Voice ☑ SMS □ MMS

Different numbers have different communications capabilities. Select the ones your phone number needs.

Type ● All | □ Local □ Mobile □ Toll-Free

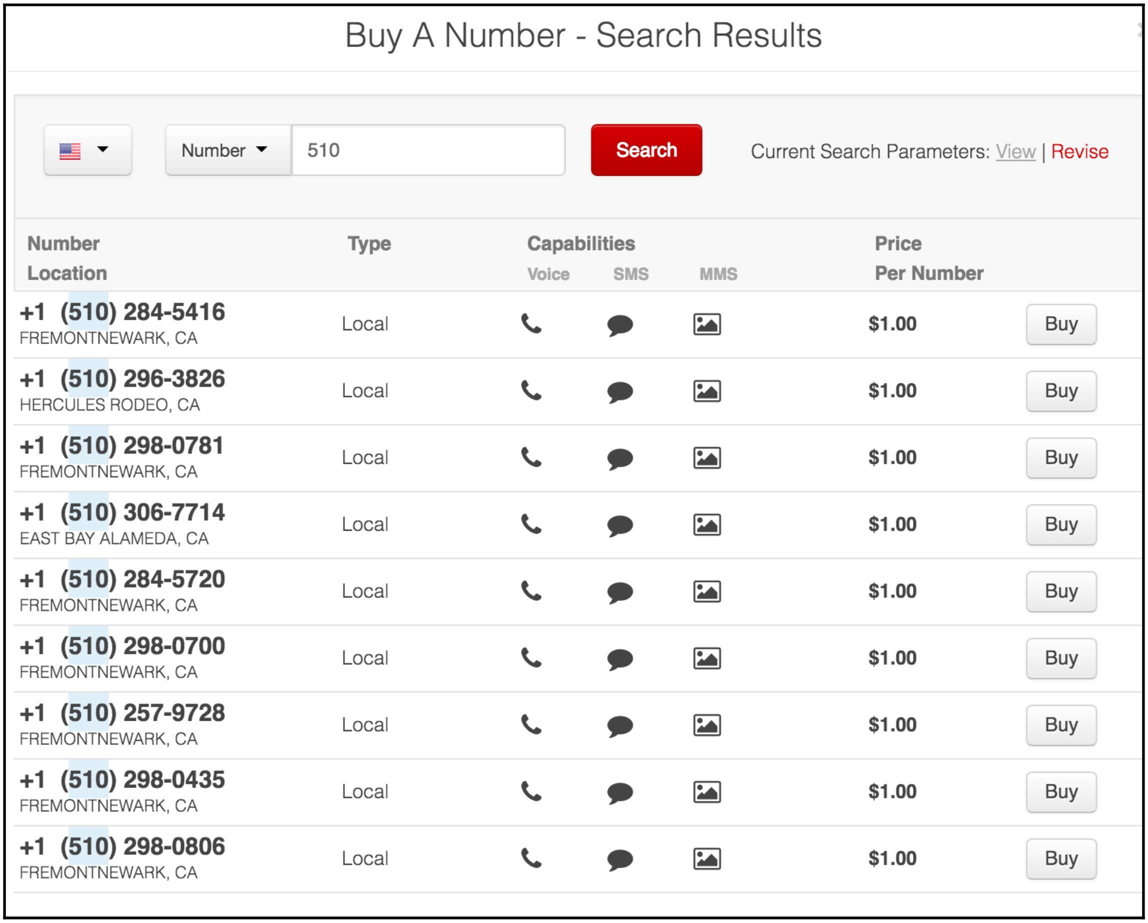
Do your users expect to call a local number, a mobile number, or a toll-free number? You choose.

Address Requirement ● Any ○ None | □ Exclude local requirements □ Exclude foreign requirements

Some local authorities require you to provide an address before purchasing a phone number.

Beta Numbers □ Include beta phone numbers in search results. These are newly released numbers on Twilio's platform.

7. When you click on **Search**, this page will take you to the results screen where you can choose to buy any of the available numbers. Make sure you buy a number that has both **Voice** and **SMS** enabled. Toll-free numbers are costly, and it's best to buy a **Local** number for this exercise:



Buy A Number - Search Results

Number	Type	Capabilities			Price	
Location		Voice	SMS	MMS	Per Number	
+1 (510) 284-5416 FREMONTNEWARK, CA	Local	☎	💬	🖼	$1.00	Buy
+1 (510) 296-3826 HERCULES RODEO, CA	Local	☎	💬	🖼	$1.00	Buy
+1 (510) 298-0781 FREMONTNEWARK, CA	Local	☎	💬	🖼	$1.00	Buy
+1 (510) 306-7714 EAST BAY ALAMEDA, CA	Local	☎	💬	🖼	$1.00	Buy
+1 (510) 284-5720 FREMONTNEWARK, CA	Local	☎	💬	🖼	$1.00	Buy
+1 (510) 298-0700 FREMONTNEWARK, CA	Local	☎	💬	🖼	$1.00	Buy
+1 (510) 257-9728 FREMONTNEWARK, CA	Local	☎	💬	🖼	$1.00	Buy
+1 (510) 298-0435 FREMONTNEWARK, CA	Local	☎	💬	🖼	$1.00	Buy
+1 (510) 298-0806 FREMONTNEWARK, CA	Local	☎	💬	🖼	$1.00	Buy

8. Cool! If you're already through this, then congratulations! You're now all set to start learning about using SMS messages and voice calls with Python APIs.

How it works…

As stated earlier, to work with Twilio APIs, we need to register for an account. Twilio creates an account for us and provides us with a unique **AccountSID** and **AuthToken**, which will validate our requests and make sure that our account gets billed for the API calls we make.

Phone numbers in Twilio are used as caller IDs to send text messages or voice calls. Caller ID (also known as calling line identification) is a caller's number that is flashed on the called party's equipment (landline or mobile). In this case, the number we rented from Twilio will be used as the caller ID.

There's more…

We looked at how to create an account, get **AccountSID** and **AuthToken**, and generate a phone number with Twilio. Let's now make use of these in the next recipe.

Sending text messages

Let's look at our first recipe on working with text messages. In this recipe, we will send a message to the recipient via SMS. Please note that you may also now have to charge your account for performing the next set of operations.

Getting ready

We start by sending SMS messages with the Twilio APIs. Let's look at how to do it in this section. Before doing that, let's create a Python virtual environment and install `flask` and `twilio` modules with the following steps.

Note that we will use `flask` to host a simple web service, which will be called by the telephony provider Twilio. The *flask* app will then perform the required business operation based on the callback from Twilio. We will know more about this when we look at the recipes.

Setting up a virtual environment and installing modules needs to be done from the command line of your computer. We use Python `pip` to install the `flask` and `twilio` modules:

```
    virtualenv ~/book/ch05/
    source ~/book/ch05/
    pip install flask
    Collecting flask==0.10.1
      Downloading Flask-0.10.1.tar.gz (544kB)
        100% |████████████████████████████████| 544kB
774kB/s
    Collecting Werkzeug>=0.7 (from flask==0.10.1)
      Downloading Werkzeug-0.11.10-py2.py3-none-any.whl (306kB)
        100% |████████████████████████████████| 307kB
1.5MB/s
    Collecting Jinja2>=2.4 (from flask==0.10.1)
      Downloading Jinja2-2.8-py2.py3-none-any.whl (263kB)
        100% |████████████████████████████████| 266kB
2.4MB/s
    Collecting itsdangerous>=0.21 (from flask==0.10.1)
      Downloading itsdangerous-0.24.tar.gz (46kB)
        100% |████████████████████████████████| 49kB
6.2MB/s
    Collecting MarkupSafe (from Jinja2>=2.4->flask==0.10.1)
      Downloading MarkupSafe-0.23.tar.gz
    Building wheels for collected packages: flask, itsdangerous, MarkupSafe
      Running setup.py bdist_wheel for flask
      Stored in directory:
/Users/chetan/Library/Caches/pip/wheels/b6/09/65/5fcf16f74f334a215447c26769
e291c41883862fe0dc7c1430
      Running setup.py bdist_wheel for itsdangerous
      Stored in directory:
/Users/chetan/Library/Caches/pip/wheels/fc/a8/66/24d655233c757e178d45dea2de
22a04c6d92766abfb741129a
      Running setup.py bdist_wheel for MarkupSafe
      Stored in directory:
/Users/chetan/Library/Caches/pip/wheels/a3/fa/dc/0198eed9ad95489b8a4f45d14d
d5d2aee3f8984e46862c5748
    Successfully built flask itsdangerous MarkupSafe
    Installing collected packages: Werkzeug, MarkupSafe, Jinja2,
itsdangerous, flask
    Successfully installed Jinja2-2.8 MarkupSafe-0.23 Werkzeug-0.11.10
flask-0.10.1 itsdangerous-0.24
```

Next, install `twilio` using the following command:

```
    pip install Twilio
    Collecting twilio
      Downloading twilio-5.4.0.tar.gz (193kB)
        100% |████████████████████████████████| 196kB
2.2MB/s
    Collecting httplib2>=0.7 (from twilio)
      Downloading httplib2-0.9.2.zip (210kB)
        100% |████████████████████████████████| 212kB
2.0MB/s
    Collecting six (from twilio)
      Downloading six-1.10.0-py2.py3-none-any.whl
    Collecting pytz (from twilio)
      Downloading pytz-2016.6.1-py2.py3-none-any.whl (481kB)
        100% |████████████████████████████████| 483kB
1.0MB/s
    Building wheels for collected packages: twilio, httplib2
      Running setup.py bdist_wheel for twilio
      Stored in directory:
/Users/chetan/Library/Caches/pip/wheels/91/16/85/2ea21326cf1aad3e32f88d9e81
723088e1e43ceb9eac935a9b
      Running setup.py bdist_wheel for httplib2
      Stored in directory:
/Users/chetan/Library/Caches/pip/wheels/c7/67/60/e0be8ccfc1e08f8ff1f50d99ea
5378e204580ea77b0169fb55
    Successfully built twilio httplib2
    Installing collected packages: httplib2, six, pytz, twilio
    Successfully installed httplib2-0.9.2 pytz-2016.6.1 six-1.10.0
twilio-5.4.0
```

How to do it…

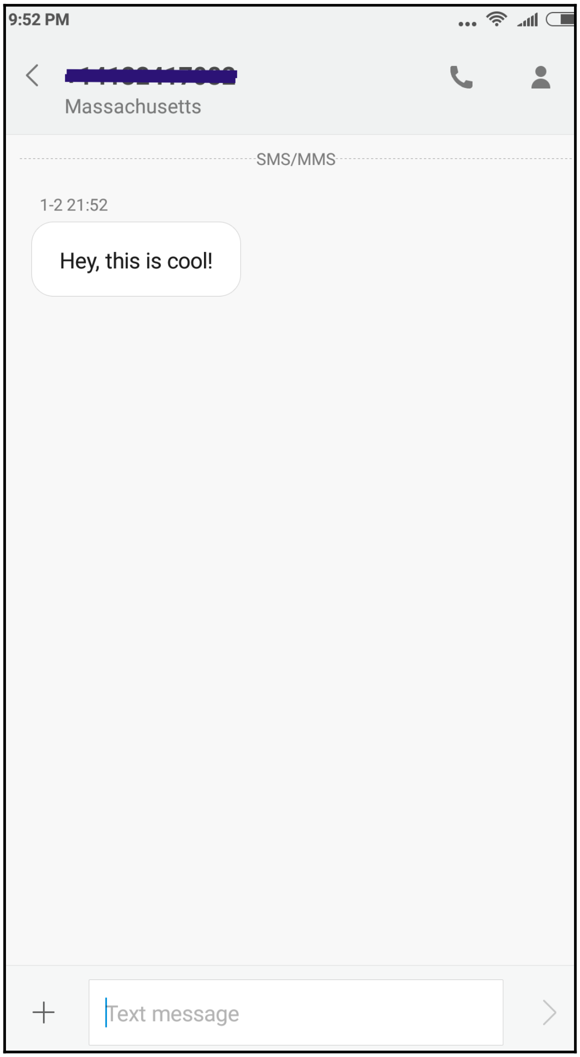
1. Let's start by creating a configuration that has our Twilio **AccountSID** and **AuthToken**. We also add our rented number as caller ID to the configuration and also add the custom number to whom the message will be sent. You can add your own number in this field to see if it works for you.
2. Our configuration file looks as follows, and we store it as `config.py`:

```
TWILIO_ACCOUNT_SID = 'Account SID'
TWILIO_AUTH_TOKEN = 'Auth Token'
CALLERID = '+Rented Number'
MYNUMBER = '+Your Number'
```

3. Now, let's write our application that will actually send out this message. We store this as `send_sms.py` and import the configuration from `config.py`:

```
import config
from flask import Flask
from twilio.rest import TwilioRestClient

app = Flask(__name__)
client = TwilioRestClient(config.TWILIO_ACCOUNT_SID,
                          config.TWILIO_AUTH_TOKEN)

message = client.messages.create(
    to=config.MYNUMBER,
    from_=config.CALLERID,
    body="Hey, this is cool!")
```

4. We run this code by going to the Terminal or from your favorite editor. Check your mobile and you should have a message from your rented number saying, **Hey, this is cool!** Indeed, this is cool! Your message can take 2-3 minutes to arrive depending on your carrier and network congestion, so be patient. See the following screenshot for the received message:



9:52 PM

[redacted]
Massachusetts

SMS/MMS

1-2 21:52

Hey, this is cool!

How it works…

First, we created a configuration file and filled it with all the required fields. In the `send_sms.py` file, we imported this configuration file by importing `config.py` and also imported the required modules such as `flask` and the `twilio` Python module.

We then created the `twilio` object, `client`, by using the class, `TwilioRestClient`, which is imported from the `twilio.rest` module. An object is created with **AccountSID** and **AuthToken**.

We then use the `create()` method of the `client.messages` class and send a message. Here `to` is the number to which the message is sent, `from_` is the caller ID, and `body` is the text to be sent. Wondering why `from_` instead of `from`? Well, that's because `from` will conflict with Python's `from` keyword, which is used to import modules.

There's more…

Trivial, isn't it?! Sending a text message to our number was a breeze. Now, can you write your own application? Let's see what you can think of. Probably, send me a message to invite me to your birthday party!

But hey, what's the fun if we don't know how to receive an incoming message? Let's learn how to do this in the next section.

Receiving SMS messages

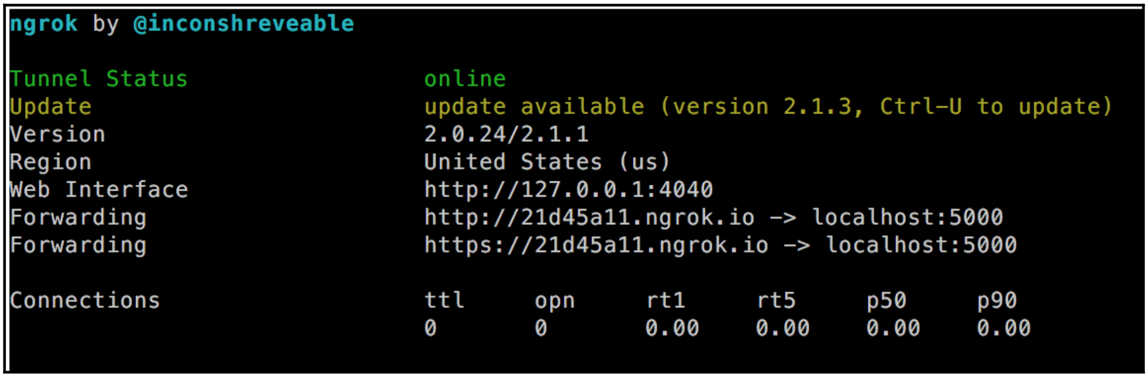
While you were thinking of the various use cases of sending SMS messages, you must have felt that the capability of receiving a text message is equally important. So, let's look at it with an auto-response-messaging app.

Getting ready

For this recipe, we will use the **ngrok** software. The ngrok (`https://ngrok.com/`) software helps you tunnel your local machine to the Internet. It means that you can expose your local server behind NAT or Firewall to the Internet. Really powerful utility! For the next recipe, download ngrok (`https://ngrok.com/download`) and run it on port 5000 using the command from your Terminal. If you're connected to the Internet you should see an instance up and running for you:

```
./ngrok http 5000
```

The following screenshot shows how ngrok has made your app visible to the Internet and on the URL, `https://<uuid>.ngrok.io/`:



```
ngrok by @inconshreveable

Tunnel Status                 online
Update                        update available (version 2.1.3, Ctrl-U to update)
Version                       2.0.24/2.1.1
Region                        United States (us)
Web Interface                 http://127.0.0.1:4040
Forwarding                    http://21d45a11.ngrok.io -> localhost:5000
Forwarding                    https://21d45a11.ngrok.io -> localhost:5000

Connections                   ttl     opn     rt1     rt5     p50     p90
                              0       0       0.00    0.00    0.00    0.00
```

Don't start ngrok yet, we start ngrok during our recipe.

How to do it...

1. Let's start by writing our server to receive SMSes. We'll call this `recv_sms.py`. The code for the server looks as follows:

```
from flask import Flask
import twilio.twiml
app = Flask(__name__)
@app.route("/insms", methods=['GET', 'POST'])
def respond_sms():
    resp = twilio.twiml.Response()
    resp.message("Thanks for your query. We will
                  get back to you soon")
return str(resp)
if __name__ == "__main__":
app.run(debug=True)
```

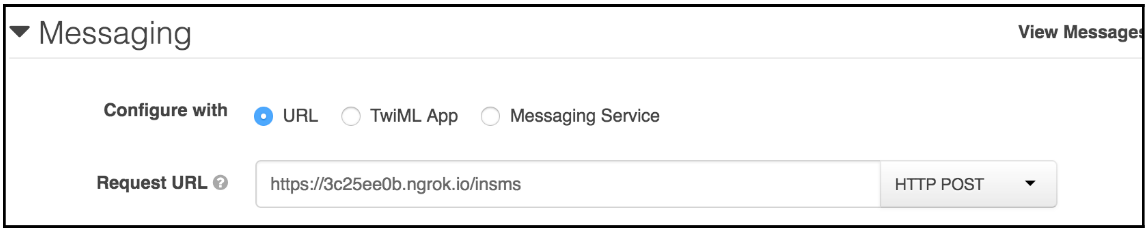
2. Run the server with Python from your Terminal with the following command:

```
python recv_sms.py
```

3. Start ngrok with the following command:

```
./ngrok http 5000
```

4. Nice, we got our server running. Let's configure our Twilio number by adding a **Request URL**. For doing this, log in to Twilio, go to the **Phone Numbers** section, and click on your rented number. Here, go to the messaging section and enter your **Request URL** as shown in the following screenshot. Make sure that the URL points to the ngrok instance that was fired up in the previous step:



Configure with ● URL ○ TwiML App ○ Messaging Service

Request URL https://3c25ee0b.ngrok.io/insms HTTP POST

And we're done! So, when someone sends a message to your rented number, they will receive an auto response saying **Thanks for your query. We will get back to you soon**.

Awesome! That's great, but hey, how did it work?

How it works…

The server that accepts the incoming message is written in `Flask` and runs on port 5000. This makes the server run on a local machine. To make it available on the Internet, we fire `ngrok` and make it run on the same port as the Flask server, which is port 5000.

We configure the Twilio phone number to route the incoming messages to our Flask server. To achieve this, we add the request URL to Twilio. So, whenever a message is sent to the rented number, it gets routed to our Flask server via ngrok. In our app, we have routed it to `https://<ngrokId>.ngrok.io/insms`.

If you look at our Flask server, we already a have a route configured with the URL, `/insms`. This route gets a `POST` call from the Twilio server (thanks to the **Request URL** setting), which then responds back with the message, **Thanks for your query. We will get back to you soon**.

There's more…

You learnt about sending and receiving text messages using Twilio. I know you're already thinking about your use cases and how you can leverage text messages to resolve them. Let's look at an example from the food retail industry.

SMS workflows for Domino's

John, the owner of a Domino's Pizza outlet in the US, is looking for ways to improve his pizza sales. One way to improve the sales would be to focus on making the process of ordering pizzas easier. He also wants to automate the workflows so that he can keep his customers updated on the order and delivery status. He also feels that while the Internet is great, customers may also want to order pizzas when they are in areas where the network reception is weak. What do you think he must do?

Getting ready

Let's think about the use case and write down what all we'd need? Here are a few things I can think of:

- Ability to receive incoming messages
- Maintain and query the status of order
- Send outgoing status message

How to do it…

Let's look at the solution and then understand how it works for the Domino's shop.

The following code snippet is divided into three main aspects: the ability to receive incoming messages with the `flask` route, maintaining the status of the order for customers who query their orders, and the ability to send outbound messages from the *Flask* app:

```
from flask import Flask, request
import twilio.twiml

class Pizza:
    def __init__(self):
        self.status = None

    def setStatus(self, status):
        self.status = status

    def getStatus(self):
        return self.status

 app = Flask(__name__)
@app.route("/insms", methods=['GET', 'POST'])
def respond_sms():
    content = request.POST['Body']
    resp = twilio.twiml.Response()
    if content == 'ORDER':
        resp.message("Thanks! We're already on your order!")
        pizza = Pizza()
        pizza.setStatus('complete')
        return str(resp)
    if content == 'STATUS':
        pizza = Pizza()
        status = pizza.getStatus()
        if status == 'complete':
            resp.message("Your order is ready!")
            return str(resp)
        else:
            resp.message("Sorry! could not locate your order!")
            return str(resp)
        else:
            resp.message("Sorry! Wrong selection")
            return str(resp)

if __name__ == "__main__":
        app.run(debug=True)
```

How it works…

We already have an app to receive the incoming messages. So, we use this app and extend it for our needs. Our rented number becomes a Domino's number, which is flashed on their advertising boards:

- In our use case, we have decided to use two keywords, `ORDER` and `STATUS`, which Domino's customers can use.
- When customers send the message, `ORDER`, to Domino's, they can place an order for a pizza of their choice. The pizza shop positively responds to the order by saying they're already working on it.
- When the customers want to know the status of their order, they can check it with a text saying, `STATUS`. In our case, when the customer queries for his order, he gets a response from the pizza shop saying that his order is ready. Imagine the satisfaction the customer gets when he reads this response.
- Customers feel it's so easy to place an order and know its status. They'll certainly start ordering more! Well, at least I will.
- John is so happy that he decides to give a pay rise to his IT manager, and incidentally, that happens to be you! Cool, isn't it?

There's more…

Now, if you note the messages sent by or to the pizza shop, they are sent to a rented number. How about sending the messages to a custom code like **DOMP** (abbreviation for Domino's Pizza)? Yes, you can achieve that using SMS Short codes; they're not free and you need to buy them at a hefty price.

See also

- We have come to the end of all the recipes for SMS messages. Hope you learnt a few things and will implement a few of them for your benefit. Can you try sending MMS messages, such as an offer to your esteemed customers with the Domino's logo? That's something for you to explore.
- In the next recipe, we will start working on voice calls and appreciate what they can offer. Can we do some awesome stuff with voice messages? Let's look at them in the next set of recipes.

Sending voice messages

Heard of VoIP? Yes, **Voice over Internet Protocol**. **VoIP** (an abbreviation) is a group of technologies used for the delivery of voice and multimedia over Internet Protocol networks, such as the Internet itself. VoIP has opened up a whole new world for communication on the Internet with products such as Skype and Google Talk offering communication solutions in the consumer and enterprise domain.

Telephony API providers such as Twilio also use the VoIP protocol to send voice messages. You will learn how to make or receive voice calls using the Twilio APIs in this section. So, let's jump in and start using the APIs!

Getting ready

In this recipe, we use the `twilio` and `flask` modules like we used in the previous recipe for SMS. So, no new installations are needed for this section.

How to do it…

1. We start by creating our configuration file. Sending a voice message is as trivial as sending a text message. We need the Twilio **AccountSID** and **AuthToken** here as well. We'd need the caller ID and the number to send the voice message to. Here's how our configuration looks:

```
TWILIO_ACCOUNT_SID = '<Account SID>'
TWILIO_AUTH_TOKEN = '<Auth Token>'
CALLERID = '<Rented Number>'
MYNUMBER = '<Number to call>'
```

2. Let's now go ahead and write the code for our Flask server. The following code helps us in making voice calls using the Twilio Python APIs. We save the file as `voice_outbound.py`:

```
import config
from flask import Flask, Response, request
from twilio import twiml
from twilio.rest import TwilioRestClient
app = Flask(__name__)
client = TwilioRestClient(config.TWILIO_ACCOUNT_SID,
                          config.TWILIO_AUTH_TOKEN)
@app.route('/call', methods=['POST'])
```

```
def outbound_call():
    response = twiml.Response()
    call = client.calls.create(
        to=config.MYNUMBER,
        from_=config.CALLERID,
        record='true',
    )
    return Response(str(response), 200,
mimetype="application/xml")

if __name__ == '__main__':
app.run(debug=True)
```

3. Run the Flask server with the following command. This will run our Flask server on the default port 5000:

   ```
   python voice_outbound.py
   ```

4. Start ngrok with the following command on port 5000. This will make sure that our server is available on the Internet with the tunneling facility provided by ngrok. Copy the URL on which ngrok is running. It will be in the format, `https://<ngrokid>.ngrok.io/`, like we saw in previous section:

   ```
   ./ngrok http 5000
   ```

   Our server is now ready to make calls, so just go ahead and make a `POST` request to `https://<ngrokid>.ngrok.io/call`, and you should receive a call to your number as added in the configuration file.

   Wow, that's cool! But what happens when you pick up the call? Your call gets disconnected. Why? That's because in Twilio, every voice call is accompanied with a callback URL, which will execute the next set of instructions once the call gets picked. This is not defined in our code, and hence, the call gets disconnected with an error. Let's fix this.

5. So, let's add the answer callback URL and complete our server code:

   ```
   import config
   from flask import Flask, Response, request
   from twilio import twiml
   from twilio.rest import TwilioRestClient
   app = Flask(__name__)
   client = TwilioRestClient(config.TWILIO_ACCOUNT_SID,
                             config.TWILIO_AUTH_TOKEN)
   @app.route('/call', methods=['POST'])
   ```

```
def outbound_call():
    response = twiml.Response()
    call = client.calls.create(
        to=config.MYNUMBER,
        from_=config.CALLERID,
        record='true',
        url=config.BASE_URL + '/answer_url',
    )
    return Response(str(response), 200,
                    mimetype="application/xml")


@app.route('/answer_url', methods=['POST'])
def answer_url():
    response = twiml.Response()
    response.addSay("Hey! You are awesome. Thanks for answering.")
    return Response(str(response), 200,
                    mimetype="application/xml")

if __name__ == '__main__':
    app.run(debug=True)
```

6. If you look at the `url` parameter in the `outbound_call()` method, it points to `BASE_URL`. This is the same as the ngrok URL suffixed with `/answer_url`. Now, if you make a `POST` request to `https://<ngrokid>.ngrok.io/call`, your number will receive a call. Once you pick up the call, a callback `POST` request is made to `https://<ngrokid>.ngrok.io/answer_url` and you'll hear the message "*Hey! You are awesome. Thanks for answering*". Wow!
7. Here's how the server logs would be:

```
* Detected change in '/Users/chetan/book/ch05/app.py',
  reloading
* Restarting with stat
* Debugger is active!
* Debugger pin code: 608-057-122

127.0.0.1 - - [16/Jul/2016 21:35:14] "POST
               /call HTTP/1.1" 200 -
127.0.0.1 - - [16/Jul/2016 21:35:22] "POST
               /answer_url HTTP/1.1" 200 -
```

How it works…

So, how does it work? After doing the SMS section, this should be pretty straightforward to understand. We will go through the code step by step:

1. We first create a `twilio` object, `client`, with the `TwilioRestClient` class from the `twilio.rest` module.
2. We define a route in our Flask app, `/call`, which accepts the `POST` method calls. This route makes a voice call to our number.
3. The actual call is made in the `outbound_call()` route method with the help of the `create()` method of the `client.calls` class.
4. In the `create()` call, we define the parameters, such as:
   - `to`: This is the mobile/landline number that is called
   - `from_`: This is the rented number from Twilio
   - `record`: This will decide if the call should be recorded
   - `url`: This is the callback answer `url` that gets called when the voice call is answered
5. In the Flask app, we have also defined a new route, `/answer_url`, which gets called when the call is picked up. Now, this is interesting to understand. Twilio works on the philosophy of TwiML, also known as the Twilio markup language. If you look at the markup, it is pretty much an XML markup. TwiML is a set of instructions that can be used to tell Twilio what it needs to do with an incoming SMS or voice call. So, the `addSay()` method is the same as:

```
<?xml version="1.0" encoding="UTF-8"?> <Response>
<Say>Hey! You are awesome. Thanks for answering</Say>
</Response>
```

There's more…

So, we learnt how to send voice calls to a given number and how the answer callback URL gets called when the call is answered. Now, let's learn how to handle an incoming voice call.

Receiving voice calls

Receiving voice calls is an important facet to developing apps using cloud telephony. A lot of business cases (as you'd have imagined) are dependent on incoming voice calls. Let's look at handling incoming voice calls using the Twilio APIs.

Getting ready

In this recipe, we use the `twilio` and `flask` modules like we used in the previous recipe for SMS. So, no new installations are needed for this section.

How to do it…

1. We start by creating our configuration file. We need the Twilio **AccountSID** and **AuthToken** here as well. We don't need any caller ID in this case as the rented number itself is the caller ID.
2. Now, let's look at our Flask server, the code for which is given in the following. We call this as `voice_inbound.py`:

```
from flask import Flask, Response, request
from twilio import twiml
from twilio.rest import TwilioRestClient

app = Flask(__name__)
client = TwilioRestClient(config.TWILIO_ACCOUNT_SID,
                          config.TWILIO_AUTH_TOKEN)

@app.route('/incall', methods=['POST'])
def inbound_call():
    response = twiml.Response()
    response.addSay("Thanks for calling our customer
                     service." "Please hold while we
                     connect you to our advisors.")
    return Response(str(response), 200,
                    mimetype="application/xml")

if __name__ == '__main__':
    app.run(debug=True)
```

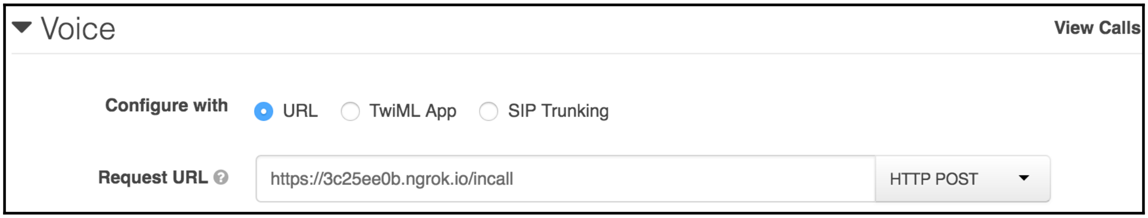
3. Run the Flask server with the following command. This will run our Flask server on the default port 5000:

   ```
   python voice_outbound.py
   ```

4. Start ngrok with the following command on port 5000. This will make sure that our server is available on the Internet with the tunneling facility provided by ngrok. Copy the URL on which ngrok is running. It will be in the format, `https://<ngrokid>.ngrok.io/`, like we saw in the previous section:

   ```
   ./ngrok http 5000
   ```

5. Now, log in to Twilio and configure the rented number for incoming voice calls. We configure the **Request URL** to point to the ngrok URL. See the following screenshot to know how to add the **Request URL** to your rented number:



Configure with ● URL ○ TwiML App ○ SIP Trunking

Request URL: https://3c25ee0b.ngrok.io/incall HTTP POST

6. Once you have the servers running and the settings configured on Twilio, make a call to your rented number through Skype or Google Talk. This will make a `POST` call to our Flask server, which in turn will respond with a TwiML response saying, *Thanks for calling our customer service. Please hold while we connect you to our advisors*.

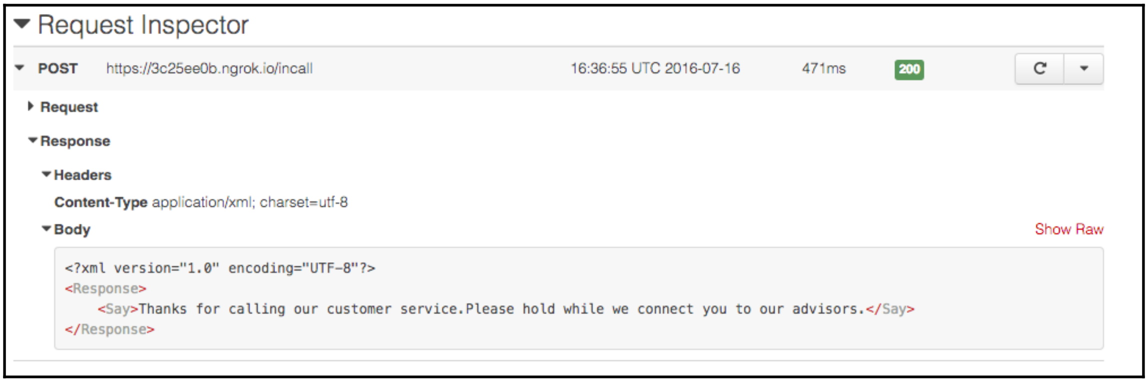
How it works…

The server that accepts the incoming message is written in Flask and runs on port 5000. This is local to our machine, and to make it available on the Internet, we create a tunnel with ngrok.

Now, when the rented number is called by anyone, Twilio looks up for the **Request URL** and makes a request to this URL, suggesting that there is an incoming call to the rented number.

Our flask server defines a route, `/incall` (to match with the **Request URL**), which gets called when our rented number receives an incoming call. The `/incall` route, in turn, creates a TwiML response that adds `<Say>` to the `<Response>` markup, and the caller gets to the message added in the `<Say>` XML.

The following screenshot shows how the TwiML response looks in Twilio. By the way, every call or SMS received or sent can be seen from the Twilio interface:



Content-Type application/xml; charset=utf-8

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
    <Say>Thanks for calling our customer service.Please hold while we connect you to our advisors.</Say>
</Response>
```

Building your own customer service software

Paul is responsible for customer service at his company. The company has a nice looking fancy website that has a facility to receive customer grievances or questions over chat. Paul often receives feedback from his customers that the chat system is not useful as they would like to get in touch with someone from the company when they hit product issues and would like to get these resolved quickly. Can you make Paul's life easier?

Getting ready

Let's think about the use case and write down what all we'd need? Here are a few things I can think of:

- Ability to receive incoming calls
- Transfer the call to a customer support engineer

How to do it…

Let's look at the solution and then understand how it would work for Paul.

In this code snippet, we will add the ability to receive incoming calls to the rented numbers and also add the functionality of call transfer:

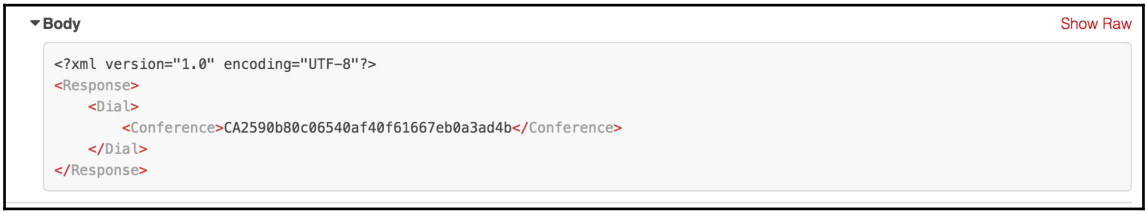
```
import config
from flask import Flask, Response, request
from twilio import twiml
from twilio.rest import TwilioRestClient

app = Flask(__name__)
client = TwilioRestClient(config.TWILIO_ACCOUNT_SID,
                          config.TWILIO_AUTH_TOKEN)

@app.route('/call', methods=['POST'])
def inbound_call():
    call_sid = request.form['CallSid']
    response = twiml.Response()
    response.dial().conference(call_sid)
    call = client.calls.create(to=config.MYNUMBER,
                               from_=config.CALLERID,
                               url=config.BASE_URL +
                               '/conference/' + call_sid)
    return Response(str(response), 200,
                    mimetype="application/xml")


@app.route('/conference/<conference_name>',
            methods=['GET', 'POST'])
def conference_line(conference_name):
    response = twiml.Response()
response.dial(hangupOnStar=True).conference(
                  conference_name)
    return Response(str(response), 200,
                    mimetype="application/xml")

if __name__ == '__main__':
    app.run(debug=True)
```

How it works…

We have already created an app that receives incoming calls. We extend this app to our use case so that when a customer calls the rented number, a `POST` call is made to the `/call` route as defined by the `inbound_call()` method.

Our `flask` route takes the incoming call and adds it to a conference with the help of the TwiML instruction set. Conference, as you know, is a group of calls bridged with each other in one conference.

The `response.dial().conference(conference_name)` method is the method that helps us in adding a call leg to the conference. This is how the TwiML looks; you can see the `<Response>` tag under which we have the `<Dial>` and `<Conference>` tags:



```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
    <Dial>
        <Conference>CA2590b80c06540af40f61667eb0a3ad4b</Conference>
    </Dial>
</Response>
```

The `flask` route makes sure it makes an outgoing call to the customer support engineer (identified by `MYNUMBER`). The outgoing call to the customer support engineer is configured with the `url` parameter (the answer URL like we saw in our outgoing voice call section). So, when the support engineer picks up the call, the callback answer URL gets called and the engineer call leg is also added to the same conference as the incoming call leg.

Both the call legs, the incoming call from the customer and the outgoing call made to the support engineer, are now in one conference and can have a conversation. The customer gets his questions resolved in a jiffy and Paul is happy. Cool!

There's more…

You learned how to build your own SMS and voice applications by using the cloud telephony APIs. However, if you're really interested in leveraging the already built solutions for your needs, you can look up to some standard software applications, such as CallHub (`https://callhub.io/`), which will help you automate your use cases efficiently at reasonable costs. You can also build your own call center solution with their APIs. So, what are you building next?

I'm sure you enjoyed the chapter; let's have some more fun in the next chapter. Let's see what we have in store!

5
Fun with E-mails

E-mail communication has become a primary mode of information exchange over last couple of decades. You work with e-mails on a daily basis and for multiple reasons. But did it ever strike that you could manipulate your inbox with Python?

In this chapter, we will cover the following recipes:

- Sending e-mail messages
- E-mail encryption
- Beautifying e-mail messages with MIME messages
- E-mail messages with attachments
- Connecting to your inbox
- Fetching and reading e-mail messages
- Marking e-mail messages
- Clearing up e-mail messages from your inbox
- Automating customer support flows with e-mail responses

Introduction

Hello, folks! Hope you're having a great day. In this chapter, we will talk about e-mails and the numerous operations we can achieve with e-mails using Python. We will also understand ways of automating business processes with e-mails with the help of real world business use case.

So, what are we waiting for? Let's get started and understand a bit about the history of e-mails and its technical implementation.

E-mails don't need any introduction actually; of course, they're a method of exchanging digital messages between computer users. E-mails operate on computer networks available over Internet for information exchange. You can log in to your favorite e-mail client and start working on your messages stored in the e-mail servers. The most widely used web client is Gmail.

E-mails have a very interesting history. In the past, e-mails needed the sender and the recipient to be online for the communication to succeed. That didn't make much sense, right? Gradually with time, e-mail servers became intelligent and started to work on a store-and-forward philosophy. Today, e-mail messages are stored asynchronously on the servers so that recipients can view them later at their convenience. E-mail servers are thus able to provide facilities such as accepting, forwarding, and marking messages.

E-mail messages started with ASCII-only characters, which were later extended by **Multipurpose Internet Mail Extensions** (**MIME**) for rich text and attachments. From a protocols standpoint, e-mails initially worked with the **File Transfer Protocol** (**FTP**) to send messages across computers, but as you might be aware, **Simple Mail Transfer Protocol** (**SMTP**) is the most widely used protocol for working with e-mails.

Please note that setting up of e-mail servers is not covered in this book. If you look on the Internet, you'll find many more resources that can help you get started. The scope of this chapter is to make you aware of things you can do with e-mail using Python programs. We take examples that apply to the Gmail web client so that you can try out code examples quickly and get to appreciate the power of using Python to automate e-mail tasks without having to set up your own e-mail server. While we use Gmail as an example, these snippets will work with any other e-mail servers that support SMTP for sending e-mails and IMAP for retrieving e-mails.

In this chapter, we'll learn how to work with e-mails using Python. We'll also use multiple Python modules, listed below, to perform various operations on e-mail messages:

- `smtplib` (`https://docs.python.org/2/library/smtplib.html`)
- `email` (`https://docs.python.org/2/library/email.html`)
- `imaplib` (`https://docs.python.org/2/library/imaplib.html`)
- `gmail` (`https://github.com/charlierguo/gmail`)

When it comes to working with e-mails using Python , what you need is a module that helps you construct messages, a module that can send e-mails, and a module that helps retrieve and update messages.

Sending e-mail messages

The first and foremost thing that you may want to achieve with an e-mail client is to send a message to your friend's or colleague's e-mail address. Let's go ahead and see how we can achieve this in Python.

Getting ready

In order to send e-mail messages, we will need to have Python's `smtplib` module. This library, as the name suggests, uses the SMTP protocol to send e-mail messages. We can install `smtplib` with our favorite `pip` tool with the following command. But Python's default installation should also have this module already:

```
pip install smtplib
```

How to do it…

1. On your computer, open your favorite editor and add the following code snippet. Let this be known as `config.py`. The configuration file has login details, such as e-mail address, password, and the e-mail address to which the e-mail needs to be sent:

   ```
   fromaddr = "abc@gmail.com"
   password = "xyz@123"
   toaddr = "abc@gmail.com"
   ```

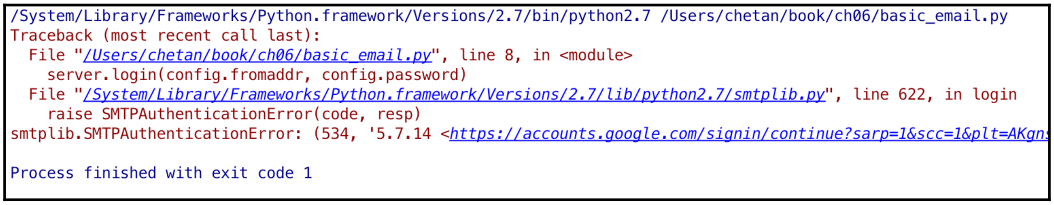
2. Now, let's write the code to send an e-mail using this configuration file:

   ```
   import smtplib
   import config server = smtplib.SMTP('smtp.gmail.com', 587)
   server.starttls()
   server.login(config.fromaddr, config.password)
   msg = "Some nice msg"
   server.sendmail(config.fromaddr, config.toaddr, msg)
   server.quit()
   ```

3. Store the preceding code as `basic_email.py`, and run the code using the following command:

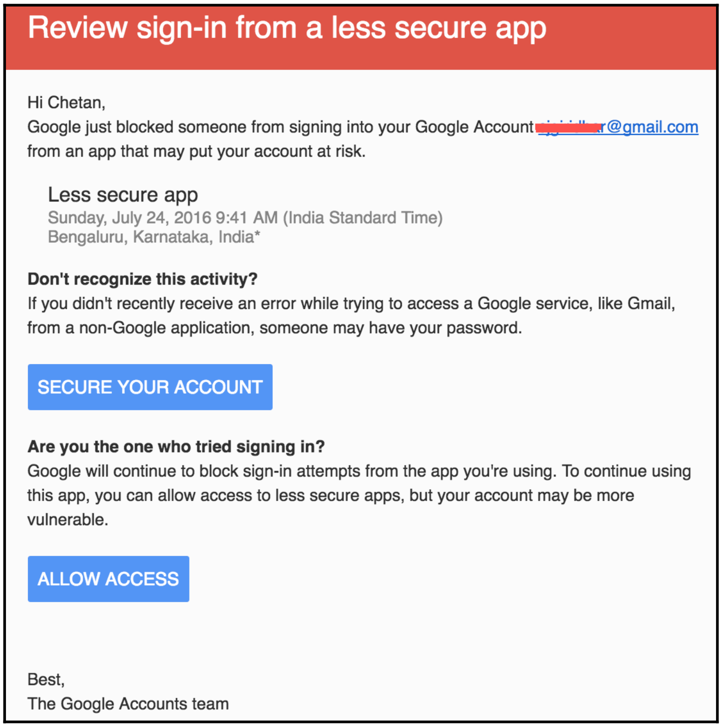
   ```
   python basic_email.py
   ```

4. If you run the preceding code, you'll see exceptions with `SMTPAuthenticationError`, and your program will fail with exit code `1`. This is how your exception will look like:



```
/System/Library/Frameworks/Python.framework/Versions/2.7/bin/python2.7 /Users/chetan/book/ch06/basic_email.py
Traceback (most recent call last):
  File "/Users/chetan/book/ch06/basic_email.py", line 8, in <module>
    server.login(config.fromaddr, config.password)
  File "/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/smtplib.py", line 622, in login
    raise SMTPAuthenticationError(code, resp)
smtplib.SMTPAuthenticationError: (534, '5.7.14 <https://accounts.google.com/signin/continue?sarp=1&scc=1&plt=AKgn

Process finished with exit code 1
```

5. Okay, that's bad, but on the contrary, it's good as well! Exception suggests that the login to the server was fine, but Gmail blocked you from sending the message. Now, if you log in to Gmail, you should see an e-mail suggesting there was a signup detected from a less secure app. Really?! Yes, that's because we tried to access the Gmail account from our Python program. This is why we get an e-mail from Google suggesting a possible security breach in case there was a malicious activity on our account. The e-mail message from Google can be viewed in the following screenshot:



Review sign-in from a less secure app

Hi Chetan,
Google just blocked someone from signing into your Google Account [redacted]r@gmail.com from an app that may put your account at risk.

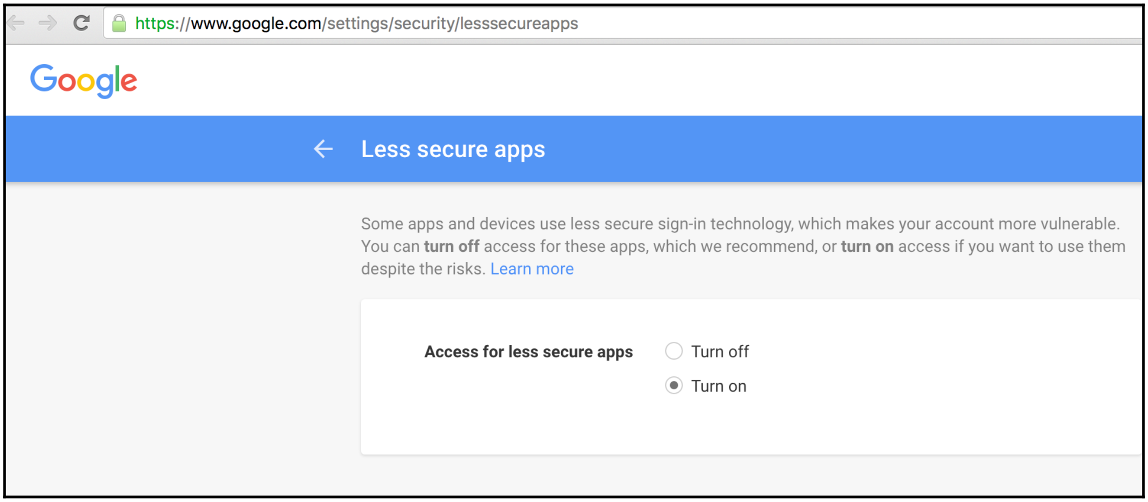
Less secure app
Sunday, July 24, 2016 9:41 AM (India Standard Time)
Bengaluru, Karnataka, India*

Don't recognize this activity?
If you didn't recently receive an error while trying to access a Google service, like Gmail, from a non-Google application, someone may have your password.

Are you the one who tried signing in?
Google will continue to block sign-in attempts from the app you're using. To continue using this app, you can allow access to less secure apps, but your account may be more vulnerable.

Best,
The Google Accounts team

6. But obviously, this was a legitimate attempt to use the Gmail account, so let's confirm that to Google. Open the e-mail message from Google and click on **ALLOW ACCESS.** You'll be taken to the **Less secure apps** page where you can turn this setting on, as shown in the following screenshot:



https://www.google.com/settings/security/lesssecureapps

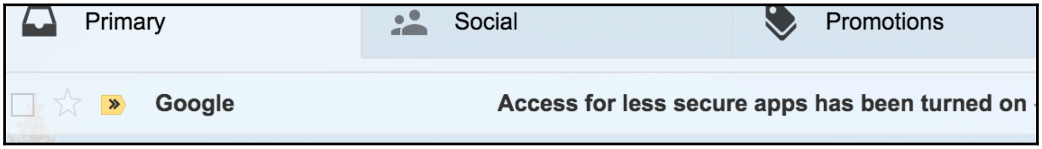
Google

Less secure apps

Some apps and devices use less secure sign-in technology, which makes your account more vulnerable. You can **turn off** access for these apps, which we recommend, or **turn on** access if you want to use them despite the risks. Learn more

Access for less secure apps ○ Turn off ● Turn on

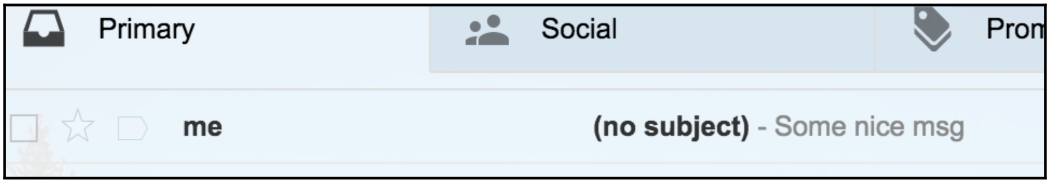
7. Now, log out of the Gmail web client and log in again to let the settings take effect on your account. If this went through fine, you will receive an e-mail from Google that the **Access for less secure apps is turned on.** The confirmation e-mail from Google will look similar to this screenshot:



Primary | Social | Promotions

□ ☆ **Google** **Access for less secure apps has been turned on**

Now, if you run the Python program again, it should run successfully and you will receive an e-mail in your inbox:



Primary | Social | Prom

□ ☆ **me** **(no subject)** - Some nice msg

8. Cool! Notice, the message contents are the same as we added in our code snippet. Also, since the *from* and *to* addresses were the same, the e-mail came from you, but it doesn't have any subject, which is not great. We will do something about this in the next recipe.

How it works…

As stated earlier, SMTP is used to send e-mail messages. We use Python module, `smtplib`, for this purpose.

If you look at the preceding code snippet, we use the constructor, `smtplib.SMTP()` to configure Gmail's SMTP settings and get access to the e-mail server. Gmail's SMTP server runs on `smtp.gmail.com` and on port 587.

Once we have the server object, `server`, we use this to log in to Gmail with our username and password. Note that we have another line in the preceding code: `server.starttls()`; we will come to this later in this chapter.

We have created a test message and stored it in the variable `msg`, which is then sent using the `sendmail` method `('fromaddr', 'toddr', msg)`.

Finally, we close the connection to the e-mail server with `server.quit()`.

There's more…

We looked at how to sign in to Gmail and send a basic e-mail using the SMTP protocol and Python's `smptlib` library. While this recipe gets us started, there are more details that we'll delve into in the next recipe. Let's look at them.

E-mail encryption

E-mail is prone to disclosure of information. Most e-mails are currently transmitted in clear text format. E-mail encryption involves encrypting or disguising content of the e-mail so that the content is read by the intended recipients. Always remember that security is of prime importance when dealing with e-mails. Let's see how we can encrypt e-mails with Python.

Getting ready

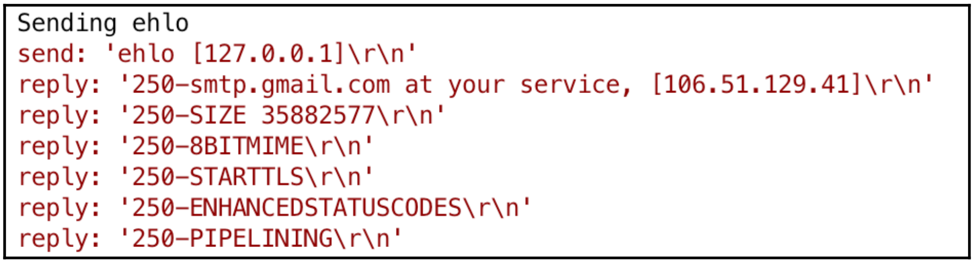
We looked at sending a basic e-mail in the previous recipe, but what's the `starttls()` method? How does e-mail encryption work? We will get answers to these questions in this section.

How to do it…

1. Let's start by opening our favorite editor and typing in the following code snippet:

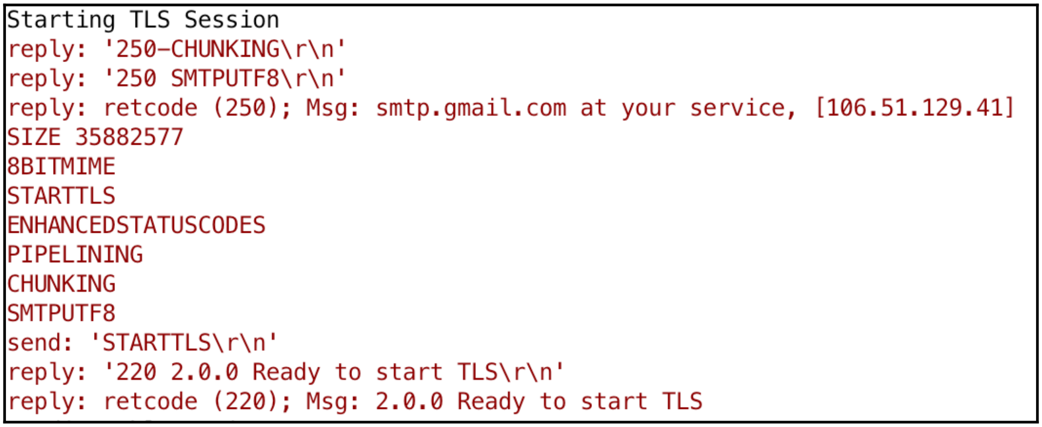
```
import smtplib
server = smtplib.SMTP('smtp.gmail.com', 587)
try:
    server.set_debuglevel(True)
    print "Sending ehlo"
    server.ehlo()
    if server.has_extn('STARTTLS'):
       print "Starting TLS Session"
       server.starttls()
       print "Sending ehlo again"
       server.ehlo()
finally:
    server.quit()
```

2. Now, let's run the Python code and see what it prints. We have outputs in three different segments. The first one is when we send an `ehlo()` message to our e-mail server:



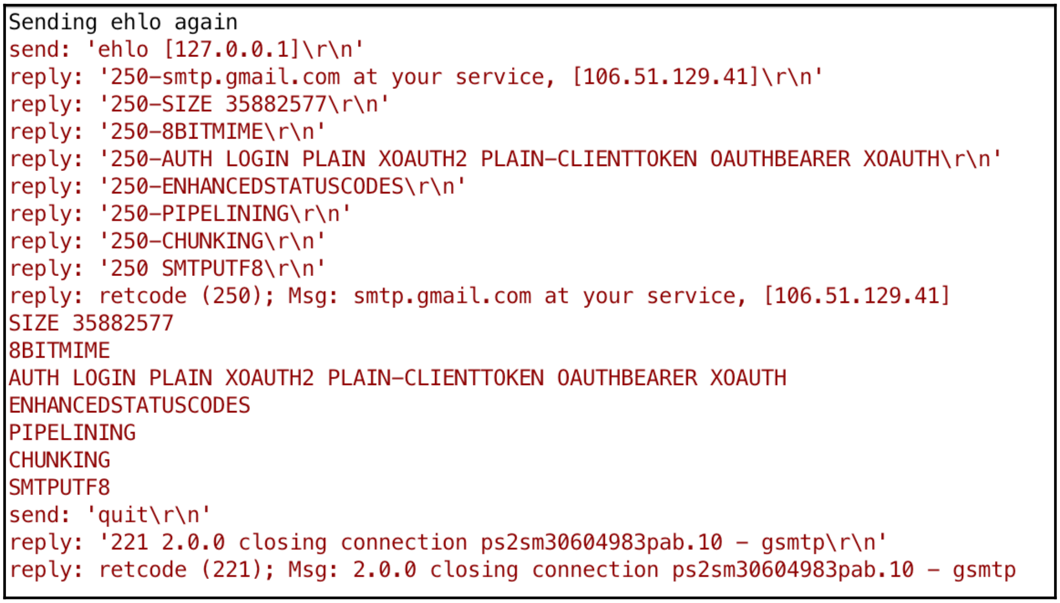
```
Sending ehlo
send: 'ehlo [127.0.0.1]\r\n'
reply: '250-smtp.gmail.com at your service, [106.51.129.41]\r\n'
reply: '250-SIZE 35882577\r\n'
reply: '250-8BITMIME\r\n'
reply: '250-STARTTLS\r\n'
reply: '250-ENHANCEDSTATUSCODES\r\n'
reply: '250-PIPELINING\r\n'
```

3. The second one is when we call `starttls()` method on the server object. Check out the following screenshot:



```
Starting TLS Session
reply: '250-CHUNKING\r\n'
reply: '250 SMTPUTF8\r\n'
reply: retcode (250); Msg: smtp.gmail.com at your service, [106.51.129.41]
SIZE 35882577
8BITMIME
STARTTLS
ENHANCEDSTATUSCODES
PIPELINING
CHUNKING
SMTPUTF8
send: 'STARTTLS\r\n'
reply: '220 2.0.0 Ready to start TLS\r\n'
reply: retcode (220); Msg: 2.0.0 Ready to start TLS
```

4. The third one is when we connect to the e-mail server with `ehlo()` again:



```
Sending ehlo again
send: 'ehlo [127.0.0.1]\r\n'
reply: '250-smtp.gmail.com at your service, [106.51.129.41]\r\n'
reply: '250-SIZE 35882577\r\n'
reply: '250-8BITMIME\r\n'
reply: '250-AUTH LOGIN PLAIN XOAUTH2 PLAIN-CLIENTTOKEN OAUTHBEARER XOAUTH\r\n'
reply: '250-ENHANCEDSTATUSCODES\r\n'
reply: '250-PIPELINING\r\n'
reply: '250-CHUNKING\r\n'
reply: '250 SMTPUTF8\r\n'
reply: retcode (250); Msg: smtp.gmail.com at your service, [106.51.129.41]
SIZE 35882577
8BITMIME
AUTH LOGIN PLAIN XOAUTH2 PLAIN-CLIENTTOKEN OAUTHBEARER XOAUTH
ENHANCEDSTATUSCODES
PIPELINING
CHUNKING
SMTPUTF8
send: 'quit\r\n'
reply: '221 2.0.0 closing connection ps2sm30604983pab.10 - gsmtp\r\n'
reply: retcode (221); Msg: 2.0.0 closing connection ps2sm30604983pab.10 - gsmtp
```

How it works…

Let's start with the basics. E-mail encryption means protecting e-mail messages from being read by someone other than the intended parties. E-mails are typically sent in clear text and can be sniffed by third parties. To avoid this, we encrypt e-mails at the protocol layer; this may include authentication as well.

SMTP servers typically send e-mails on port 25 using SSL/TLS protocols. However, with the advent of STARTTLS (layer on top of SMTP) and the usage of port 587 for message submissions, e-mail clients like Gmail use STARTTLS and port 587 for sending e-mails. Gmail also has authentication implemented; remember that we used `server.login (username, password)` to login to the Gmail server.

For STARTTLS to be used across the server and client, the client needs to first know if the server supports this protocol. When we issue `server.ehlo()`, the program sends an `EHLO` message to the SMTP server to establish communication. The server responds with the message and allowed extensions, as observed in the first screenshot.

Now, from the code, we check whether the server supports the `STARTTLS` extension with `server.has_extn('STARTTLS')`. As we saw in the first screenshot, the SMTP server responded with the `STARTTLS` extension; this confirms that gmail supports the `STARTTLS` protocol layer, which is awesome.

Now, we communicate with the server using `server.starttls()`. The server responds to this by sending a message, `Ready to start TLS`. This way, we have encrypted our session. If you now look at the third screenshot, when we send `server.ehlo()`, we re-identify ourselves with the server over the TLS session. It also suggests that the server implements an authentication extension.

Finally, we quit our SMTP session using `server.quit()` and the server responds with `closing connection`, as shown in the third screenshot.

There's more…

Well, that was pretty detailed. Take a few moments to understand it. It's actually interesting what goes on behind the sending of a simple e-mail message. But don't worry about it too much; let's start the fun and get into many more examples.

Beautifying e-mail messages with MIME

In the first few recipes, we sent e-mail messages in a simple plain old text format. The MIME Internet standard helps us construct messages with non-ASCII characters, multipart messages and images. It also helps with attachments and many other tasks. This way, we can construct enriched e-mail messages. Let's look at how the MIME format is used in this recipe.

Getting ready

For this recipe, we will use the same module, `smtplib`, to send e-mail messages. We will also introduce another module `email`, which will help us construct better e-mail messages with the MIME format. The `email` module comes in with Python installation; hence, we don't need any new modules or installations to be carried out. In this section, we will look at how to use the MIME attributes to send better looking e-mails.

How to do it…

1. Let's start by importing all of the modules we need:

```
import smtplib
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
import config
import email.utils
```

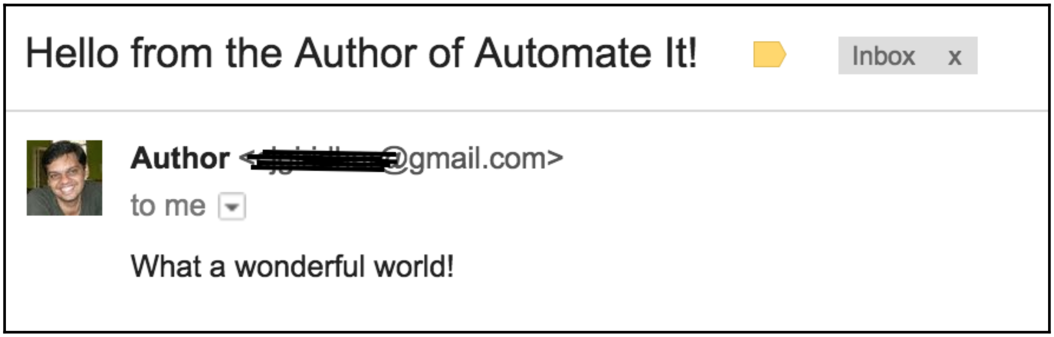
2. Now, let's construct our e-mail message using the MIME module. The following code creates the message:

```
fromaddr = config.fromaddr
toaddr = config.toaddr
msg = MIMEMultipart()
msg['Subject'] = "Hello from the Author of Automate It!"
msg['To'] = email.utils.formataddr(('Recipient', toaddr))
msg['From'] = email.utils.formataddr(('Author',
fromaddr))
body = "What a wonderful world!"
msgBody = MIMEText(body, 'plain')
msg.attach(msgBody)
```

3. So, now we have the details of whom to send the e-mail message to. We have also constructed the e-mail message in the MIME format. What are we waiting for? Let's send it using the following code:

```
server = smtplib.SMTP('smtp.gmail.com', 587)
server.starttls()
server.login(fromaddr, config.password)
text = msg.as_string()
print "Text is:", text
server.sendmail(fromaddr, toaddr, text)
server.quit()
```

The received e-mail looks as follows:



Hello from the Author of Automate It!

Author <[redacted]@gmail.com>

to me

What a wonderful world!

Awesome! That's great, but hey… how did it work?

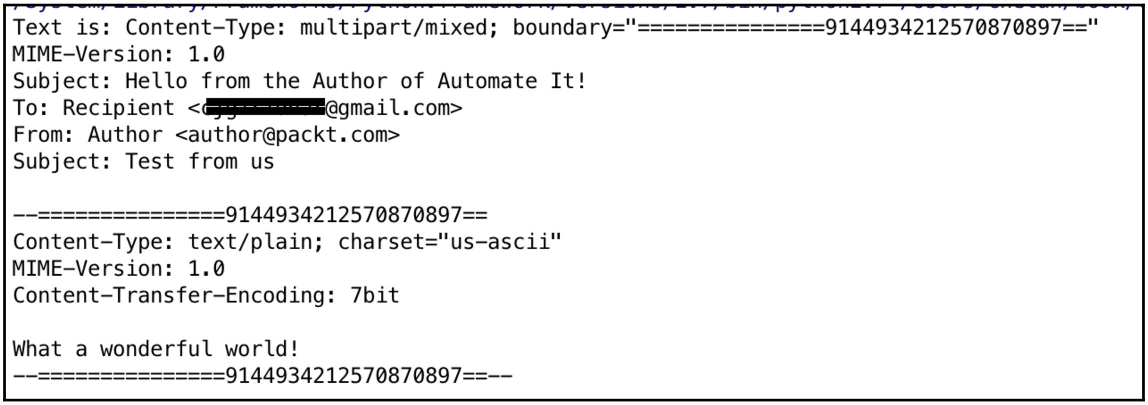
How it works…

In the preceding example, we imported the configuration file from where we got `fromaddress` and `password` to log in to the SMTP server and `toaddress` to whom the e-mail message will be sent.

Now, before sending the message, we construct a new MIME message object. We do that with `MIMEMultipart()` class from the `email.mime.multipart` module of Python. For those who are not aware, a MIME multipart message means both HTML and text content in a single e-mail. So, in this code, we create a new multipart MIME message and then add the text content to it.

The text content, which is the body of the e-mail, is created with the `MIMEText()` constructor from the `email.mime.text` module and is then attached to the multipart message with the `attach()` method.

The constructed MIME message is seen in the following screenshot, where the content-type is multipart and the MIME Version is 1.0, the **Subject**, **To**, and **From** details are as expected, and the e-mail body contains the expected text:



```
Text is: Content-Type: multipart/mixed; boundary="===============9144934212570870897=="
MIME-Version: 1.0
Subject: Hello from the Author of Automate It!
To: Recipient <[redacted]@gmail.com>
From: Author <author@packt.com>
Subject: Test from us

--===============9144934212570870897==
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit

What a wonderful world!
--===============9144934212570870897==--
```

Once we have the message and the recipient details, we send the e-mail as usual, using the `SMTP.sendmail()` method.

E-mail messages with attachments

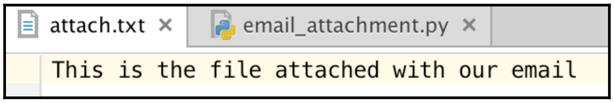
One of the most used and simple use cases with e-mail is the ability to add attachments to your e-mail messages. In this section, we will learn how to add attachments to our e-mails in Python.

Getting ready

We use the same `smtplib` and `email` modules for this example. So, don't bother about the modules to be installed. Let's get on with the recipe.

How to do it…

1. Let's begin by quickly creating a small text file. We will call it `attach.txt`, and its contents are shown in the following screenshot:

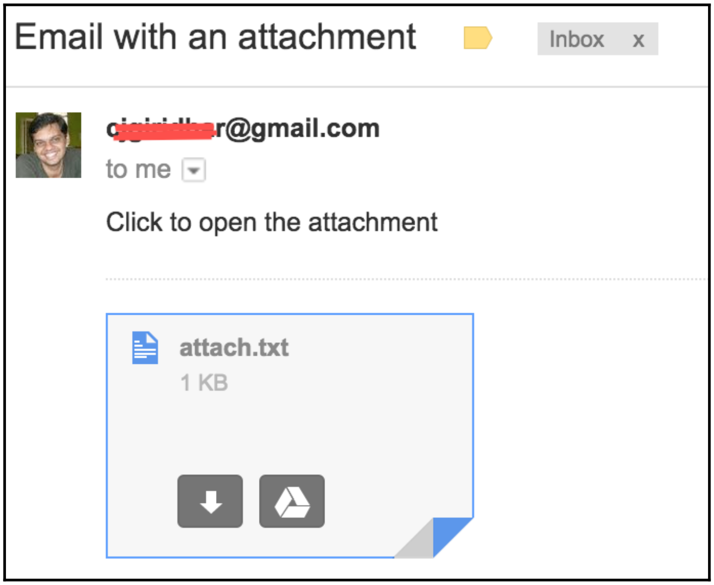


```
This is the file attached with our email
```

2. Let's look at the code that will help add an attachment to our e-mail:

```
import smtplib
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
from email.mime.base import MIMEBase
from email import encoders
import config

fromaddr = config.fromaddr
toaddr = config.toaddr

msg = MIMEMultipart()

msg['From'] = fromaddr
msg['To'] = toaddr
msg['Subject'] = "Email with an attachment"

body = "Click to open the attachment"

msg.attach(MIMEText(body, 'plain'))

filename = "attach.txt"
attachment = open(filename, "rb")

part = MIMEBase('application', 'octet-stream')
part.set_payload((attachment).read())
encoders.encode_base64(part)
part.add_header('Content-Disposition', "attachment;
                filename= %s" % filename)
msg.attach(part)
server = smtplib.SMTP('smtp.gmail.com', 587)
server.starttls()
server.login(fromaddr, config.password)
text = msg.as_string()
server.sendmail(fromaddr, toaddr, text)
server.quit()
```

3. When you run the preceding code, you will receive the e-mail in your inbox, which will look similar to the following screenshot:



Email with an attachment

[redacted]r@gmail.com

to me

Click to open the attachment

attach.txt

1 KB

How it works…

We're already familiar with creating MIME message objects. So, in this code, we create a multipart message object, `msg`. We then add a text message to it with `msg.attach()`. The text body says `Click to open the attachment`.

In this recipe, we use another new class, `MIMEBase` from the `email` module that will be used to attach the text file. Remember, we already have the `attach.txt` file created, we open the file using Python's `open()` method and get the file handle `attachment`. We then create a `MIMEBase` object, `part`, and assign the contents of the file as payload to this object. The contents of the file are obtained with `attachment.read()` and the payload is set with the `set_payload()` method.

To attach the file, the `MIMEBase` object has to be encoded to base64 and the `Content-Disposition` header needs to be added to the `part` object. Now that we have the `part` object, it can be attached to the multipart object, `msg`, with the `attach()` method, as we did for the body text.

Cool, so we have the complete MIME message and the details of whom to send the message to. So, we go ahead and send the e-mail with the attachment. Just what we intended to achieve.

Connecting to your inbox

Throughout the chapter, we have been talking about sending e-mails with Python. However, at some point, you might also want to scan through your inbox and read the incoming messages. So, how do you do that? Let's understand and learn it in this recipe.

Getting ready

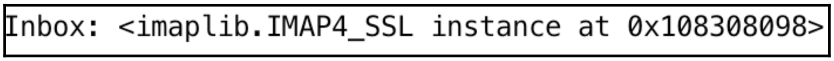
In this recipe, we use a new Python module that will help us retrieve messages from the inbox. We use the Python module, `imaplib`, which is available in the default Python installation. Cool, so let's get started.

How to do it…

1. We start by using the configuration file, which we have already created to store our e-mail and password, to log in to the server. We then add our code to create a handle or an object to work with our inbox. This is how the code looks:

```
import config, imaplib
M = imaplib.IMAP4_SSL("imap.gmail.com", 993)
M.login(config.fromaddr, config.password)
print "Inbox:", M
M.logout()
```

If you run the preceding piece of code, you will get the following output:

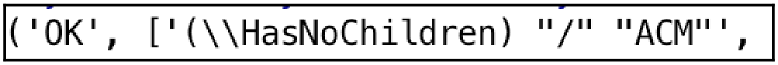


```
Inbox: <imaplib.IMAP4_SSL instance at 0x108308098>
```

2. By default, when we login to Gmail, default inbox gets selected, but if we have created other inboxes as well, we can get the list by adding a small line of code. Now, from all the labels, if we specifically want to select `Inbox`, even that can be achieved. Look at the following code example:

```
import config, imaplib
M = imaplib.IMAP4_SSL("imap.gmail.com", 993)
M.login(config.fromaddr, config.password)
print M.list()
M.select('INBOX')
print "Inbox:", M
M.logout()
```

The output of the preceding code snippet is shown in the following screenshot. Though I have many labels created, I have taken a screenshot with fewer labels:



```
('OK', ['(\\HasNoChildren) "/" "ACM"',
```

How it works…

As discussed in the beginning of this chapter, we have three main protocols to work with e-mails. We used SMTP heavily for sending e-mails, but while reading e-mails, we can use either POP or IMAP to retrieve messages from the e-mail server. We will go through the code step by step.

Python's `imaplib` library helps us connect to our mailbox using the **Internet Message Access Protocol** (**IMAP**). The Gmail server is configured to IMAP, with the server running on `imap.gmail.com` and on port '993'.

In our code example, we create an object of type `imaplib` with the constructor `IMAP4_SSL("imap.gmail.com", 993);` we call this object `M`.

Regarding the encryption, we use `IMAP4_SSL` for connecting to the server because it uses encrypted communications over SSL sockets. We avoid the usage of `IMAP4` class, which internally uses clear text sockets.

Now, with object `M`, we can log in to Gmail with our username and password and get connected to our inbox.

When we call the `list()` method on object `M`, it returns all the labels you have already created. Now, in my case, I have created the `ACM` label (for my work with ACM), and hence it shows up in my list of labels.

If you look at the code example, we can explicitly connect to the `INBOX` using the `select()` method. Once connected to the inbox, we can start fetching the e-mail messages from the inbox.

Finally, we close our connection with the inbox using the `M.logout()` method. Cool! That was nice and easy.

There's more…

So, we learnt how to connect to our inbox in this recipe, but we may also want to read the messages, mark them, and perform interesting actions on them. Let's look at how to perform operations on the messages in the next recipe.

Fetching and reading e-mail messages

Retrieving e-mail messages with `imaplib` is also easy to achieve. In this recipe, we will learn how to do that with Python code. In this recipe, we will search for e-mails with a particular subject line and fetch the latest message from the inbox that matches a predefined criteria.

Getting ready

We continue to use the `imaplib` module for reading e-mail messages, so no new installations are required for this recipe.

How to do it…

1. We utilize the configuration file and import `fromaddress`, `password`, and `toaddress` to log in to the server. Once we're logged in, we select the default inbox, fetch e-mail messages, and read them. Let's look at the complete code:

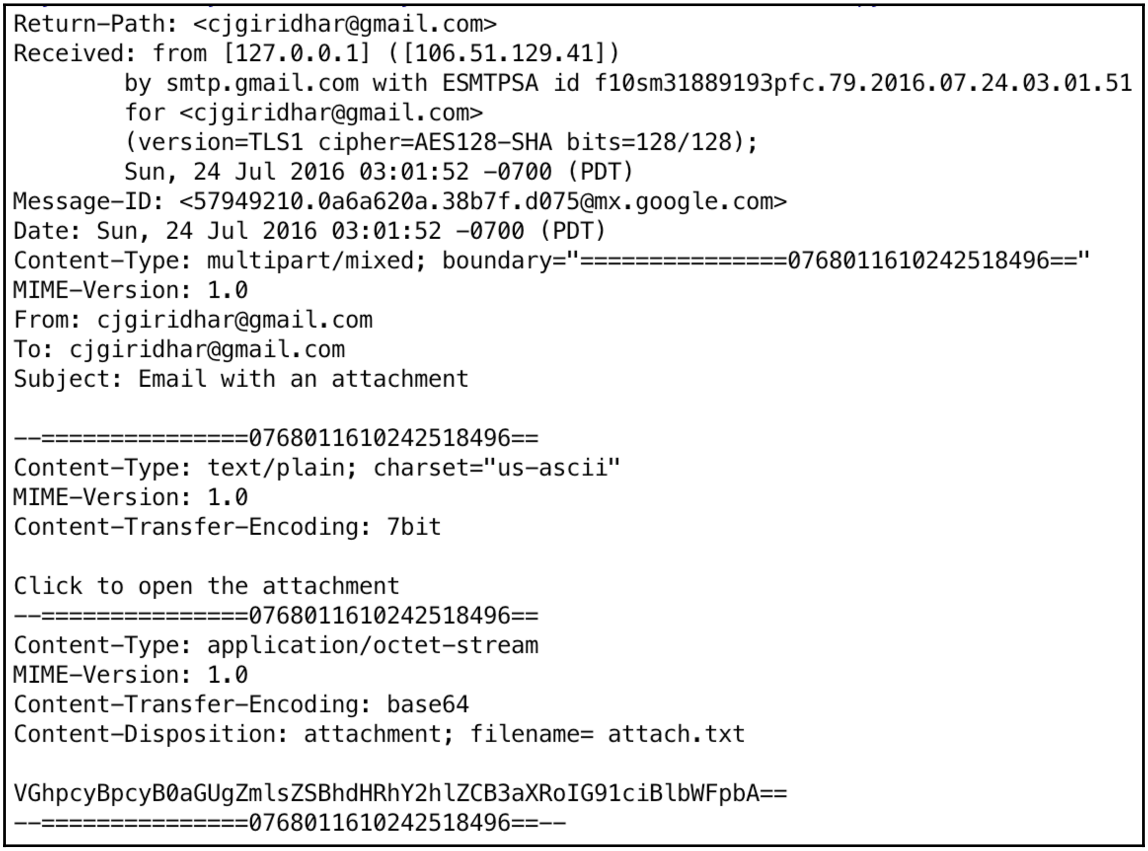
   ```
   import config, imaplib
   M = imaplib.IMAP4_SSL("imap.gmail.com", 993)
   ```

```
M.login(config.fromaddr, config.password)
M.select("INBOX")
typ, data = M.search(None, 'SUBJECT',
                     "Email with an attachment")
typs, msg = M.fetch(data[0].split()[-1], '(RFC822)')
print "Message is ", msg[0][1]
M.close()
M.logout()
```

2. Store the preceding file as `inbox_search.py` and run the code using the following command:

```
python inbox_search.py
```

3. The output of the preceding code snippet is shown in the following screenshot:



```
Return-Path: <cjgiridhar@gmail.com>
Received: from [127.0.0.1] ([106.51.129.41])
        by smtp.gmail.com with ESMTPSA id f10sm31889193pfc.79.2016.07.24.03.01.51
        for <cjgiridhar@gmail.com>
        (version=TLS1 cipher=AES128-SHA bits=128/128);
        Sun, 24 Jul 2016 03:01:52 -0700 (PDT)
Message-ID: <57949210.0a6a620a.38b7f.d075@mx.google.com>
Date: Sun, 24 Jul 2016 03:01:52 -0700 (PDT)
Content-Type: multipart/mixed; boundary="===============0768011610242518496=="
MIME-Version: 1.0
From: cjgiridhar@gmail.com
To: cjgiridhar@gmail.com
Subject: Email with an attachment

--===============0768011610242518496==
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit

Click to open the attachment
--===============0768011610242518496==
Content-Type: application/octet-stream
MIME-Version: 1.0
Content-Transfer-Encoding: base64
Content-Disposition: attachment; filename= attach.txt

VGhpcyBpcyB0aGUgZmlsZSBhdHRhY2hlZCB3aXRoIG91ciBlbWFpbA==
--===============0768011610242518496==--
```

How it works…

In the preceding code snippet, we first create an object of `IMAP_SSL4()` with the appropriate IMAP settings. We then log in to the client using the credentials from the configuration file with the help of the IMAP object. We then select our `INBOX` so that we can perform search operations on it.

The `M.search()` method called on the IMAP object helps us search for e-mails with the subject *Email with an attachment*. The `search()` method returns an array of messages that match the criteria we searched for.

Now, if we have to get to a particular message, and since we have to read the latest e-mail message that matched our criteria, we use the `M.fetch()` method. The `fetch()` method requires a given message object and the part of the message that you want to fetch. So, in this code example, we pass the latest e-mail object that matches the criteria and pass `RFC822`, which suggests that we want the e-mail body in the RFC 822 format.

When we print the message obtained from `fetch()`, we get the contents of the e-mail body for the latest e-mail that matched the search.

Now, do you remember seeing the contents of the e-mail? Well, it's the same e-mail message we had sent in one of our earlier recipes, which was used for demonstrating the e-mail attachment.

There's more…

Good! So, now we can search for a message and fetch it too. There are many more granular operations such as marking messages that we'd like to perform on our inbox. Let's look at them in the next recipe.

Marking e-mail messages

In the previous recipe, we looked at fetching and reading messages. Wasn't it too complicated? Do we have to take care of so many details while doing a simple operation like search or read? In this section, let's look at another library that can help us not only to search or read better, but also to perform various operations on our e-mails.

Getting ready

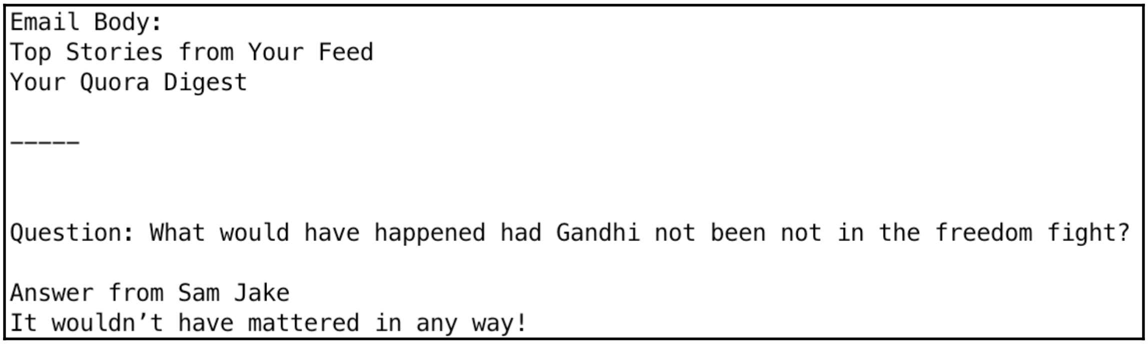
For this section, we will install the `gmail` module. You can install this module through your Terminal using the `pip` command, as shown here:

```
pip install gmail
```

Let's look at how to search for e-mails and read them using `gmail` APIs. This will get us started with the `gmail` module. The following code snippet searches for e-mails that were received after July 22, 2016. We then take the latest message and fetch it. Once we have the message fetched, we just go ahead and read the body of the e-mail:

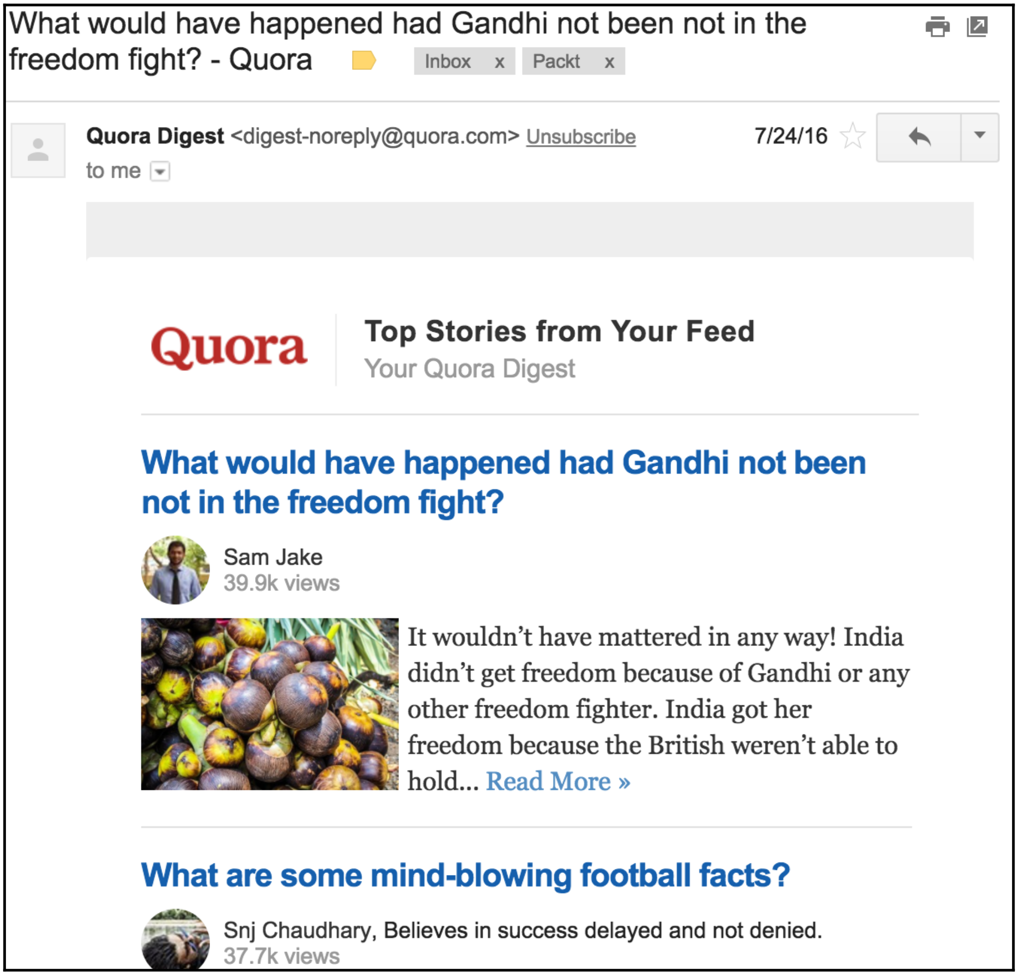
```
import gmail, config
from datetime import date
g = gmail.login(config.fromaddr, config.password)
mails = g.inbox().mail(after=date(2016, 7, 22))
mails[-1].fetch()
print "Email Body:\n", mails[-1].body
g.logout()
```

The output of the preceding code is shown in the following screenshot. Looks like I may have received an e-mail digest from Quora!



```
Email Body:
Top Stories from Your Feed
Your Quora Digest

-----



Question: What would have happened had Gandhi not been not in the freedom fight?

Answer from Sam Jake
It wouldn't have mattered in any way!
```

Here's a screenshot from my inbox:



What would have happened had Gandhi not been not in the freedom fight? - Quora

Quora Digest <digest-noreply@quora.com> Unsubscribe

to me

7/24/16

Quora

Top Stories from Your Feed

Your Quora Digest

What would have happened had Gandhi not been not in the freedom fight?

Sam Jake

39.9k views

It wouldn't have mattered in any way! India didn't get freedom because of Gandhi or any other freedom fighter. India got her freedom because the British weren't able to hold... Read More »

What are some mind-blowing football facts?

Snj Chaudhary, Believes in success delayed and not denied.

37.7k views

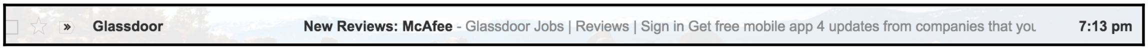
Wasn't that too easy? Incidentally, the `gmail` module is written on top of `imaplib` but has better APIs, so let's take advantage of this module and do some fantastic operations.

How to do it…

1. Let's open the inbox and look for an unread message that matches a certain criteria and mark the message as read. The following code easily does this:

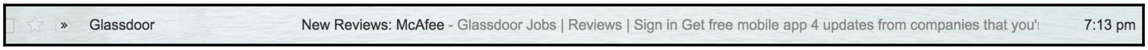
```
import gmail, config
g = gmail.login(config.fromaddr, config.password)
mails = g.inbox().mail(unread=True,
sender='noreply@glassdoor.com')
mails[-1].fetch()
mails[-1].read()
g.logout()
```

Before running this program, I had one e-mail in my inbox from `https://glassdoor.com` that I hadn't read. It looked like this in my inbox:



Glassdoor **New Reviews: McAfee** - Glassdoor Jobs | Reviews | Sign in Get free mobile app 4 updates from companies that you **7:13 pm**

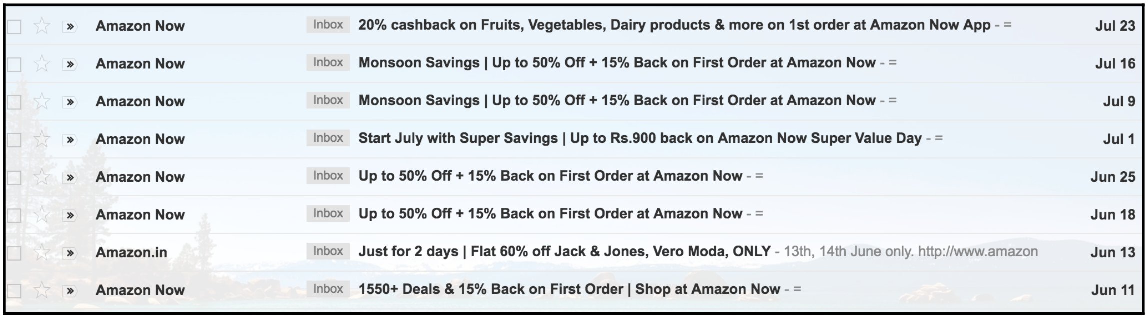
After running the code snippet, it identified this e-mail as matching my criteria of unread messages from `noreply@glassdoor.com` and marked my message as read. So, now it looks like this in my inbox. Gmail un-bolds the read messages and that's what happened in my inbox:



Glassdoor New Reviews: McAfee - Glassdoor Jobs | Reviews | Sign in Get free mobile app 4 updates from companies that you' 7:13 pm

Nice!

2. Let's look at another example. I have been receiving so many promotional e-mails from Amazon Now from Jan 2016. This is how my mailbox looks:

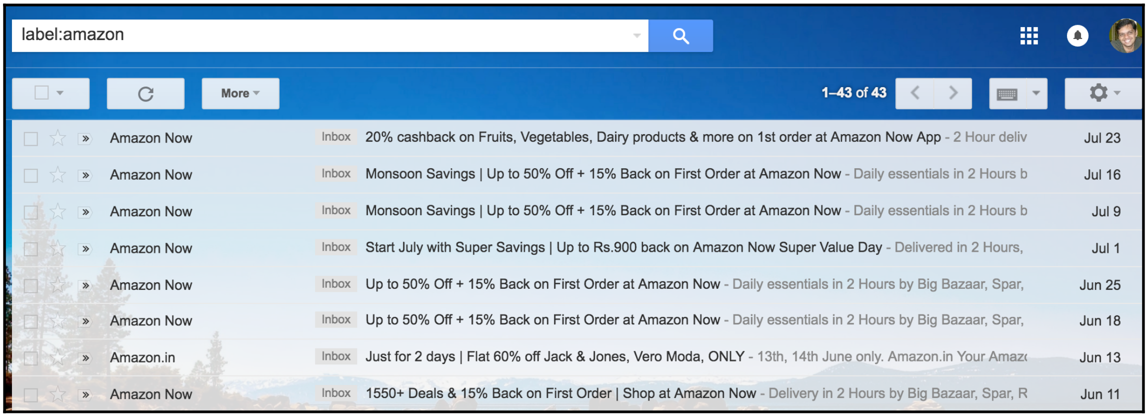


Amazon Now | Inbox | 20% cashback on Fruits, Vegetables, Dairy products & more on 1st order at Amazon Now App - = | Jul 23
Amazon Now | Inbox | Monsoon Savings | Up to 50% Off + 15% Back on First Order at Amazon Now - = | Jul 16
Amazon Now | Inbox | Monsoon Savings | Up to 50% Off + 15% Back on First Order at Amazon Now - = | Jul 9
Amazon Now | Inbox | Start July with Super Savings | Up to Rs.900 back on Amazon Now Super Value Day - = | Jul 1
Amazon Now | Inbox | Up to 50% Off + 15% Back on First Order at Amazon Now - = | Jun 25
Amazon Now | Inbox | Up to 50% Off + 15% Back on First Order at Amazon Now - = | Jun 18
Amazon.in | Inbox | Just for 2 days | Flat 60% off Jack & Jones, Vero Moda, ONLY - 13th, 14th June only. http://www.amazon | Jun 13
Amazon Now | Inbox | 1550+ Deals & 15% Back on First Order | Shop at Amazon Now - = | Jun 11

3. Now, I want to mark them all as `read` and assign them under one label, `AMAZON`. How can I do it? The following code does this operation:

```
 import gmail, config
 from datetime import date
 g = gmail.login(config.fromaddr, config.password)
 mails = g.inbox().mail(unread=True,
                        sender='store-news@amazon.in',
                        after=date(2016, 01, 01))
for email in mails:
    email.read()
    email.add_label("AMAZON")
    g.logout()
```

4. After running this code, a new label will appear in your inbox with the name, `AMAZON`. Now, if you search your inbox for all e-mails with the label `AMAZON`, you'll see that all these e-mail messages have been marked as read. Look at the following screenshot where I search for e-mails with the label `AMAZON`:



Amazon Now	Inbox	20% cashback on Fruits, Vegetables, Dairy products & more on 1st order at Amazon Now App - 2 Hour deliv	Jul 23
Amazon Now	Inbox	Monsoon Savings | Up to 50% Off + 15% Back on First Order at Amazon Now - Daily essentials in 2 Hours b	Jul 16
Amazon Now	Inbox	Monsoon Savings | Up to 50% Off + 15% Back on First Order at Amazon Now - Daily essentials in 2 Hours b	Jul 9
Amazon Now	Inbox	Start July with Super Savings | Up to Rs.900 back on Amazon Now Super Value Day - Delivered in 2 Hours,	Jul 1
Amazon Now	Inbox	Up to 50% Off + 15% Back on First Order at Amazon Now - Daily essentials in 2 Hours by Big Bazaar, Spar,	Jun 25
Amazon Now	Inbox	Up to 50% Off + 15% Back on First Order at Amazon Now - Daily essentials in 2 Hours by Big Bazaar, Spar,	Jun 18
Amazon.in	Inbox	Just for 2 days | Flat 60% off Jack & Jones, Vero Moda, ONLY - 13th, 14th June only. Amazon.in Your Amaz	Jun 13
Amazon Now	Inbox	1550+ Deals & 15% Back on First Order | Shop at Amazon Now - Delivery in 2 Hours by Big Bazaar, Spar, R	Jun 11

How it works…

In the first step, we created an object, `g` by logging into the Gmail server. Note that we didn't pass any parameters like IMAP settings or port to create the object. The `gmail` module internally handles this.

Now, using this object, we start searching our inbox for e-mails that are *unread* and sent by `noreply@glassdoor.in`', and all the mail objects matching this criteria are stored in *mails*.

Later, we fetch the latest record with the `fetch()` method and mark this mail as read with the `read()` method.

Similarly, in the second recipe, we iterate through all the e-mail messages that are *unread*, sent by `store-news@amazon.in`, and that were sent to me this year.

Each mail is then marked as read with the `read()` method and added to the label, `AMAZON`. Works like a breeze, awesome!

There's more…

We looked at some of the operations that we can perform on our e-mail messages. There are many more. With the `gmail` module, you can mark messages as unread or even make them important with a star. Let's look at an example with which we can clear up our inbox.

Clearing up e-mail messages from your inbox

Last but not least, this recipe will take you through the steps with which you can delete e-mail messages from your inbox. As you'd expect, it's pretty straightforward to delete your e-mails programmatically.

> Even if you delete the messages from your e-mail client, the e-mail server can still choose to store them. So, when you delete your messages, you're merely marking them to be hidden from your inbox while they can continue to stay on your e-mail server, based on the server implementation.

Getting ready

We continue to use the `imaplib` module for deleting e-mail messages, so no new installations are required for this recipe.

How to do it…

1. Let's utilize the configuration file and import the `fromaddress`, `password`, and `toaddress` to log in to the server.
2. This is how the complete code looks like:

```
import gmail, config
g = gmail.login(config.fromaddr, config.password)
emails = g.inbox().mail(sender='junk@xyz.com')
if emails:
   for mail in emails:
       mail.delete()
```

3. Store the preceding file as `inbox_delete.py` and run the code using the following command:

```
python inbox_delete.py
```

How it works…

Similar to what we saw in the previous examples, we first log in to Gmail with our login credentials from the configuration file.

We then connect to our inbox and search for e-mails coming from `junk@xyz.com`. If we find any e-mails that match this criteria, we want to delete them.

So, we loop over the mail objects and perform the `delete()` operation on them, and that's it! Our inbox is now free from all messages that we deem as junk. :)

There's more…

Excellent! So, now we know how to send e-mail messages, add attachments, and fetch and read them. We also learnt how to mark our message as read, add appropriate labels, and delete these messages if needed. Armed with this knowledge, can we do something for Kelly who has a few problems?

Automating customer support flows with with e-mail responses

Kelly, Director of Customer Support, has a problem at hand. Most of her support engineers end up responding to Level 1 support requests where customers are looking for information that is already available on the website. Customers just end up sending e-mails to support without trying to search for themselves.

This sequence of events is non-productive for the customer and support engineers. Customers simply wait for information instead of getting it on the website directly, and support engineers manually send a pointer to the **Frequently Asked Questions** (**FAQ**) section from the website to the customers. Kelly sees it as an opportunity to improve and wants to reduce the time spent on support by automating this flow. Can we do something to help her?

Getting ready

Of course, this is a bigger problem to solve but at the very least, we can do something that will help automate the flow. Whenever the support team receives a new ticket via e-mail, we can auto respond to the ticket acknowledging the receipt of the ticket and also send the link to the FAQ section from the company's website. This way, the customers can browse and look up the information they need from the FAQ section. It also reduces the load on support engineers as the auto-response e-mail would have already resolved the customer query and that too in a quick time.

So now, what do we actually need? We need to monitor our support inbox, look at any new customer queries, and then auto-respond with our template e-mail.

How to do it…

1. Let's directly jump to our solution. Create a Python file and copy the following code snippet into it. It does exactly what we need to automate the support flow:

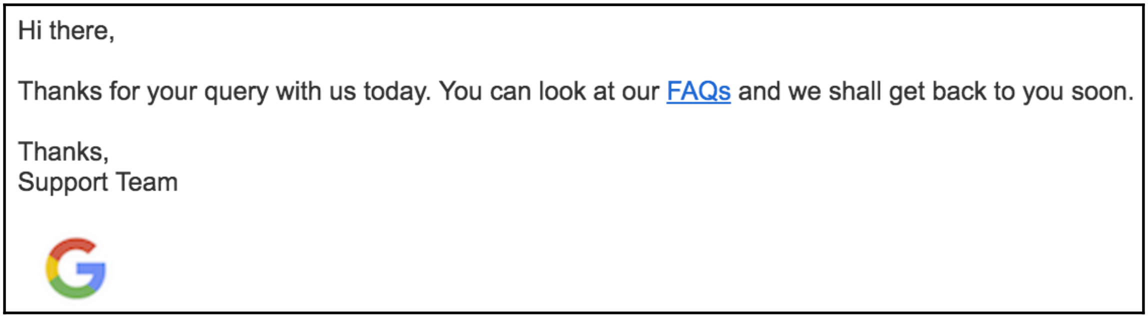
```
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
from email.mime.image import MIMEImage
import config, time, gmail
```

```
def send_email(strTo):
    strFrom = config.fromaddr
    msgRoot = MIMEMultipart('related')
    msgRoot['Subject'] = 'Thanks for your ticket'
    msgRoot['From'] = strFrom
    msgRoot['To'] = strTo

    msgRoot.preamble = 'This is a multi-part message
                        in MIME format.'
    msgAlternative = MIMEMultipart('alternative')
    msgRoot.attach(msgAlternative)
    msgText = MIMEText('This is the alternative plain
                        text message.')
    msgAlternative.attach(msgText)
    msgText = MIMEText('Hi there, <br><br>Thanks for your
                        query with us today.'
                       ' You can look at our
                       <a href="https://google.com">FAQs</a>'
                       ' and we shall get back to you
                         soon.<br><br>'
                       'Thanks,<br>Support Team<br><br>
                       <img src="cid:image1">', 'html')
    msgAlternative.attach(msgText)
    fp = open('google.png', 'rb')
    msgImage = MIMEImage(fp.read())
    fp.close()
    msgImage.add_header('Content-ID', '<image1>')
    msgRoot.attach(msgImage)

    import smtplib
    server = smtplib.SMTP('smtp.gmail.com', 587)
    server.starttls()
    server.login(config.fromaddr, config.password)
    server.sendmail(config.fromaddr, config.toaddr,
                    msgRoot.as_string())
    server.quit()

    while True:
          g = gmail.login(config.fromaddr, config.password)
          mails = g.inbox().mail(unread=True)
          mails[-1].fetch()
          from_ = mails[-1].fr
          send_email(from_)
          time.sleep(60)
```

2. Run the preceding code with Python, and you will observe that the program is still running. It is actually waiting for new e-mails, in this context, the customer requests for the support engineers.
3. If you now send an e-mail to customer support, you will receive an auto reply from our Python program.

In this case the support inbox is my e-mail address, but you can easily set up an e-mail account for your company so that the customer requests are directed to this account.

This is how the auto response e-mail looks:



Hi there,

Thanks for your query with us today. You can look at our FAQs and we shall get back to you soon.

Thanks,
Support Team

How it works…

We start with a `while` loop that runs every one min (60 seconds). Every iteration reads the support inbox and searches for unread e-mails.

If the `while` loop finds any unread e-mail, it will fetch the e-mail and get the `fr` attribute. The `fr` attribute gets you the `from` field of the e-mail message. The `from` field is the e-mail address of the customer asking for information from the support engineer.

Once we have the customer's e-mail address, we send an automated response to the customer from our inbox. The preceding screenshot shows exactly what the auto response looks like.
Cool, so now when the customers asks a question to the customer support engineer by sending an e-mail, they will get an automated e-mail response with the link to the Frequently Asked Questions section.

This way, the customer gets the required information quickly from the FAQ link. Also, the load on the support engineers is reduced, as they don't have to respond to the mundane support requests manually.

I'm sure Kelly feels happy about this improvement. She understands that the customer support flow is automated to a certain extent and hopes to see productivity gains soon!

There's more…

Awesome! There are many other things that you can do with e-mails. Tried downloading e-mail attachments? Want to give it a shot? We'll let you try that for yourself. See you in the next chapter!

6
Presentations and More

Oh, one more presentation to the boss today! Can I simply not run a program and generate a presentation instead of doing it manually all over again? Worry no more; this chapter might just resolve all your worries. We look at various ways you can create presentations of your own in an automated way using Python.

In this chapter, we will cover the following recipes:

- Reading PowerPoint presentations
- Creating and updating presentations, and adding slides
- Playing with layouts, placeholders, and textboxes
- Working with different shapes and adding tables
- Visual treat with pictures and charts
- Automating weekly sales reports

Introduction

When it comes to reporting data or work management status or presenting an idea, PowerPoint presentations are one of your best bets. PowerPoint allows you to make interactive multimedia slides to present information. Ask a few of your friends from the business world; professionals almost think in presentations, meaning that the thought process gets structured around the slides of **PowerPoint** presentation (in short, **PPT**).

In this chapter, we discuss the generation of custom PowerPoint presentations with Python. You learn about reading, writing, and manipulating PPT files. PPTs offer capabilities of adding tables, hosting images, and presenting charts, among others; we will learn to work with all these interesting features using Python.

Throughout the chapter, we will work with a binary file format: `.pptx`. PPTX presentations are different from PPT versions, in that they use Microsoft Open XML format that was introduced in Microsoft Office 2007.

The recipes in this chapter will focus on Python modules that help us perform multiple operations on PPTX files; specifically, we will focus on the following Python modules in this chapter:

- `python-pptx` (`https://python-pptx.readthedocs.io/en/latest/index.html`)
- `pandas` (`http://pandas.pydata.org/`)

> Note that while we try to cover all the major operations that can be done with respect to PowerPoint presentations, there is always the possibility to do more. So, I highly encourage playing around and learning multiple tricks with Python.
> You could also use Win32 COM APIs from Windows to work on PPTs in the Windows operating system instead of using the `python-pptx` module (which we will use in this chapter).

Reading PowerPoint presentations

Based on our experience with PowerPoint presentations, we know a PPT file contains slides, with each slide containing details that need to be presented to the audience. This recipe will show you how to extract information from PPTX files with the `python-pptx` module.

Getting ready

To step through this recipe, we will need to install the `python-pptx` module. Let's install the module with Python `pip`:

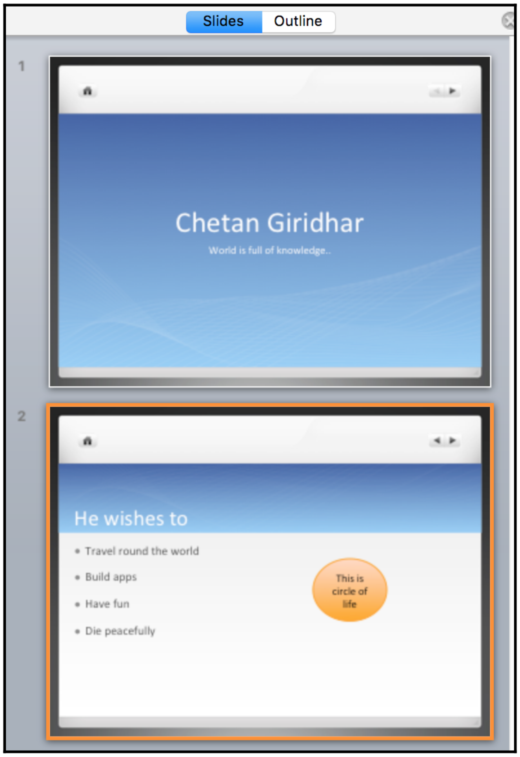
```
chetans-MacBookPro:ch08 Chetan$ sudo pip install python-pptx
Password:
Downloading/unpacking python-pptx
  Downloading python-pptx-0.6.0.tar.gz (6.3MB): 6.3MB downloaded
  Running setup.py (path:/private/tmp/pip_build_root/python-pptx/setup.py)
egg_info for package python-pptx
Requirement already satisfied (use --upgrade to upgrade): lxml>=3.1.0 in
/Library/Python/2.7/site-packages (from python-pptx)
Requirement already satisfied (use --upgrade to upgrade): Pillow>=2.6.1 in
/Library/Python/2.7/site-packages (from python-pptx)
```

```
Requirement already satisfied (use --upgrade to upgrade): XlsxWriter>=0.5.7
in /Library/Python/2.7/site-packages (from python-pptx)
Installing collected packages: python-pptx
  Running setup.py install for python-pptx
Successfully installed python-pptx
Cleaning up...
```

Installed the module, already? Let's get started!

How to do it…

1. We start by creating a PPTX file with Microsoft PowerPoint 2013. We'll use this file as a sample to learn how to read and extract data from the presentations. You'll also get the file if you download the code samples for this book. We call this file as `myprofile.pptx`; it contains information about the author of the book with two slides in it. The following screenshot shows the file contents:



Chetan Giridhar

World is full of knowledge..

He wishes to

- Travel round the world
- Build apps
- Have fun
- Die peacefully

This is circle of life

2. If you look at the presentation, the first slide has two text items: the headline item is **Chetan Giridhar** and the subtitle is **World is full of knowledge..**.. The second slide has more data; it has a different layout, and the title of the slide is **He wishes to**, containing all the four wishes of the author and a circular shape with the content, **This is circle of life**. Interesting!
3. On your computer, go to Terminal and use `vim` or choose your favorite editor. To read the PPT file, let's first use Python code to create a presentation object for `myprofile.pptx`:

   ```
   from pptx import Presentation
   path_to_presentation = 'myprofile.pptx'
   prs = Presentation(path_to_presentation)
   print "Presentation object for myprofile file: ", prs
   ```

   The output of the preceding code snippet is as follows:

   ```
   Presentation object for myprofile file:
   <pptx.presentation.Presentation object at 0x10e56e550>
   ```

4. That's good; now that we have the presentation object, let's use it to get the slide objects. We know the presentation has two slides. The following code gets us the slide objects. Slides are represented as a Python list in the `slide` object and can be iterated over in a for loop:

   ```
   print "Slides are:"
   for slide in prs.slides:
       print "Slide object:", slide
   ```

5. The preceding code snippet uses the presentation object, `prs`, to retrieve the slide objects. The output of the code snippet is as follows:

   ```
   Slides are:
   Slide object: <pptx.slide.Slide object at 0x10e59f500>
   Slide object: <pptx.slide.Slide object at 0x10e59f460>
   ```

6. OK, that's neat! Let's go one level deeper and look at some of the attributes of the `slides` object. The following code prints a few attributes of the `slide` object:

   ```
   print "Slide has following objects:"
   slide1, slide2 = prs.slides[0], prs.slides[1]
   print "Slide Ids: \n", slide1.slide_id, ",", slide2.slide_id
   print "Slide Open XML elements: \n", slide1.element, ",",
          slide2.element
   print "Slide layouts: \n", slide1.slide_layout.name, ",",
          slide2.slide_layout.name
   ```

7. Observe that the first slide's layout is **Title Slide** and the next slide is **Title and Content**, which is indeed the case. We also print the `Slide Ids` and the `Open XML elements`:

   ```
   Slide has following objects:
   Slide Ids:
   256, 257
   Slide Open XML elements:
   <Element {http://schemas.openxmlformats.org/
             presentationml/2006/main}sld at 0x109fc2d60>,
   <Element {http://schemas.openxmlformats.org/
             presentationml/2006/main}sld at 0x109fc23c0>
   Slide layouts:
   Title Slide, Title and Content
   ```

8. Now, every slide contains a few shapes. For instance, the first slide has two text placeholders, the title and the subtitle. In the second slide, we have two placeholders but have a circular shape as well. The following code prints this information:

   ```
   print "Shapes in the slides"
   i=1
   for slide in prs.slides:
       print 'Slide', i
       for shape in slide.shapes:
           print "Shape: ", shape.shape_type
       i +=1
   ```

   The output of the preceding code snippet is as follows. You can observe that the first slide contains text frames, but the second slide also has an auto shape:

   ```
   Shapes in the slides
   Slide 1
   Shape:  PLACEHOLDER (14)
   Shape:  PLACEHOLDER (14)

   Slide 2
   Shape:  PLACEHOLDER (14)
   Shape:  PLACEHOLDER (14)
   Shape:  AUTO_SHAPE (1)
   ```

9. OK, so now we have the slides, the slide layout, and the slide shapes. Let's try to get the text content from both the slides and all the shapes. The following code does exactly what we need:

```
text_runs = []
for slide in prs.slides:

    for shape in slide.shapes:
        if not shape.has_text_frame:
        continue
    for paragraph in shape.text_frame.paragraphs:
        for run in paragraph.runs:
            text_runs.append(run.text)

print "Text is: ", text_runs
```

The output of the code example is as follows. It contains all the text from both the slides. These are termed as **text runs** in the `python-pptx` world:

```
Text is:  [u'Chetan Giridhar', u'World is full of knowledge..', u'He wishes
to', u'Travel round the world', u'Build apps', u'Have fun', u'Die
peacefully', u'This is circle of life']
```

How it works…

In this recipe, we read the complete presentation and got the contents of both the slides.

We started by manually creating a PPTX file with Microsoft PowerPoint 2013 and used the `Presentation` class from the PPTX module to create an object of the `myprofilepptx` file, `prs`. Using this object, we got access to both the slides with the `prs.slides` method of the `presentation` object.

Next, we used the `slides` object to get all the available shapes in both the slides with `slides.shapes`. Iterating over this object helped us to get the shapes such as `PLACEHOLDER` and `AUTO_SHAPE` from the slides. We will learn more about slides and shapes in the latter part of the chapter.

We then used the `shape.has_text_frame` attribute to check whether the shape has a text frame and, if available, has got the `paragraphs` object from the text frame. The paragraph object's `runs` attribute contained the list of the actual text data, which was then stored in the array, `text_runs`.

There's more…

Cool! So, we learnt many things in one recipe: presentation object, slides, layouts, shapes, text frames, and paragraphs. With this, we're in a great position to read the PPTX files.

All this is great, but hey, we'd like to create new PPTX files, right? And hopefully automate creating presentations? So, let's go ahead and see how that can be achieved in Python.

Creating and updating presentations, and adding slides

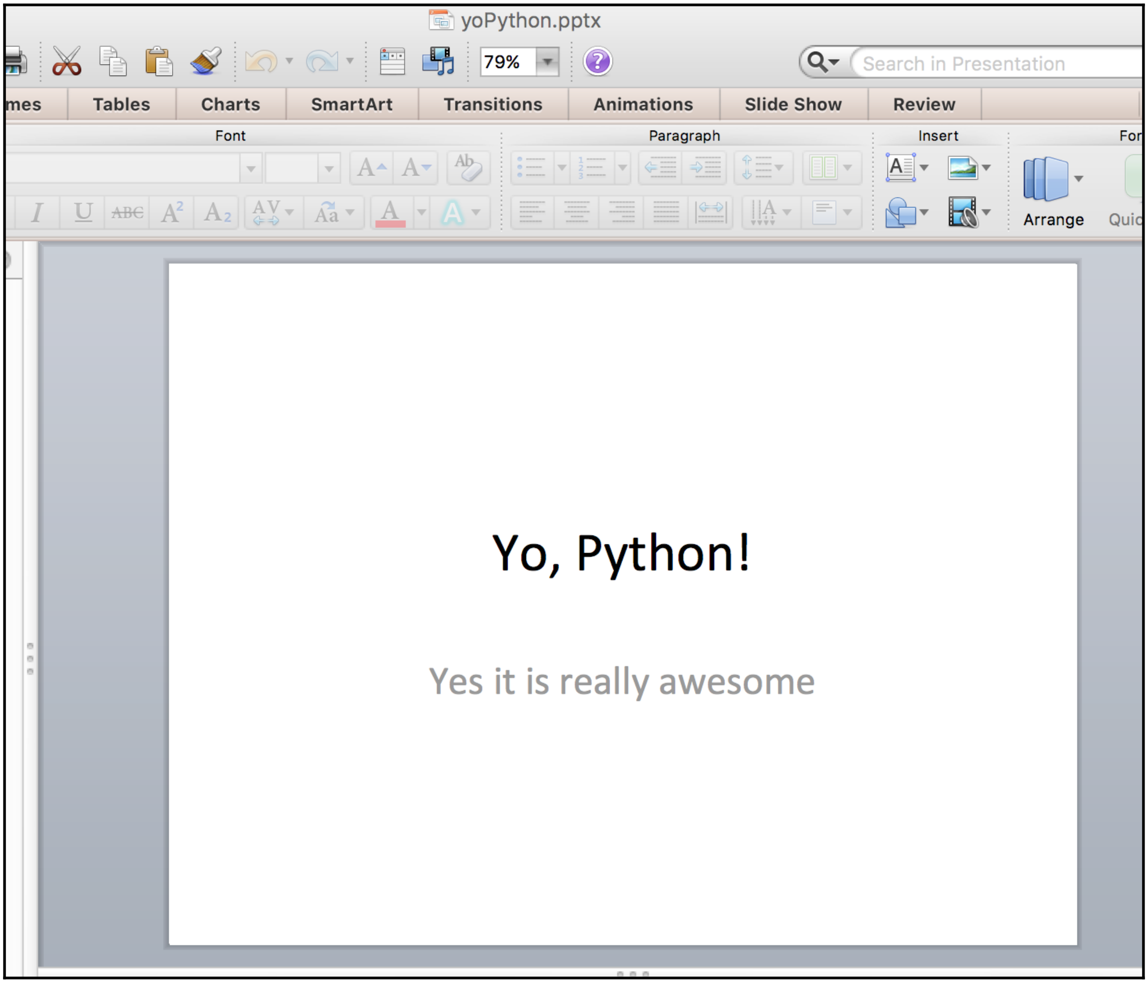
We continue using the `python-pptx` module for the recipes in this section. So, we don't have to install any new modules. We will learn how to create a blank presentation and add slides to it and, of course, some content on it.

How to do it…

1. Let's start with a very simple example of creating a new PPT with **Yo! Python** written on it. The following code helps us create the presentation:

```
from pptx import Presentation

prs = Presentation()
slide = prs.slides.add_slide(prs.slide_layouts[0])
slide.shapes.title.text = "Yo, Python!"
slide.placeholders[1].text = "Yes it is really awesome"

prs.save('yoPython.pptx')
```

If we run the preceding piece of code, it will create a PPTX file with the title as **Yo, Python!** and sub title as **Yes it is really awesome**. The following screenshot shows the contents of the slide:

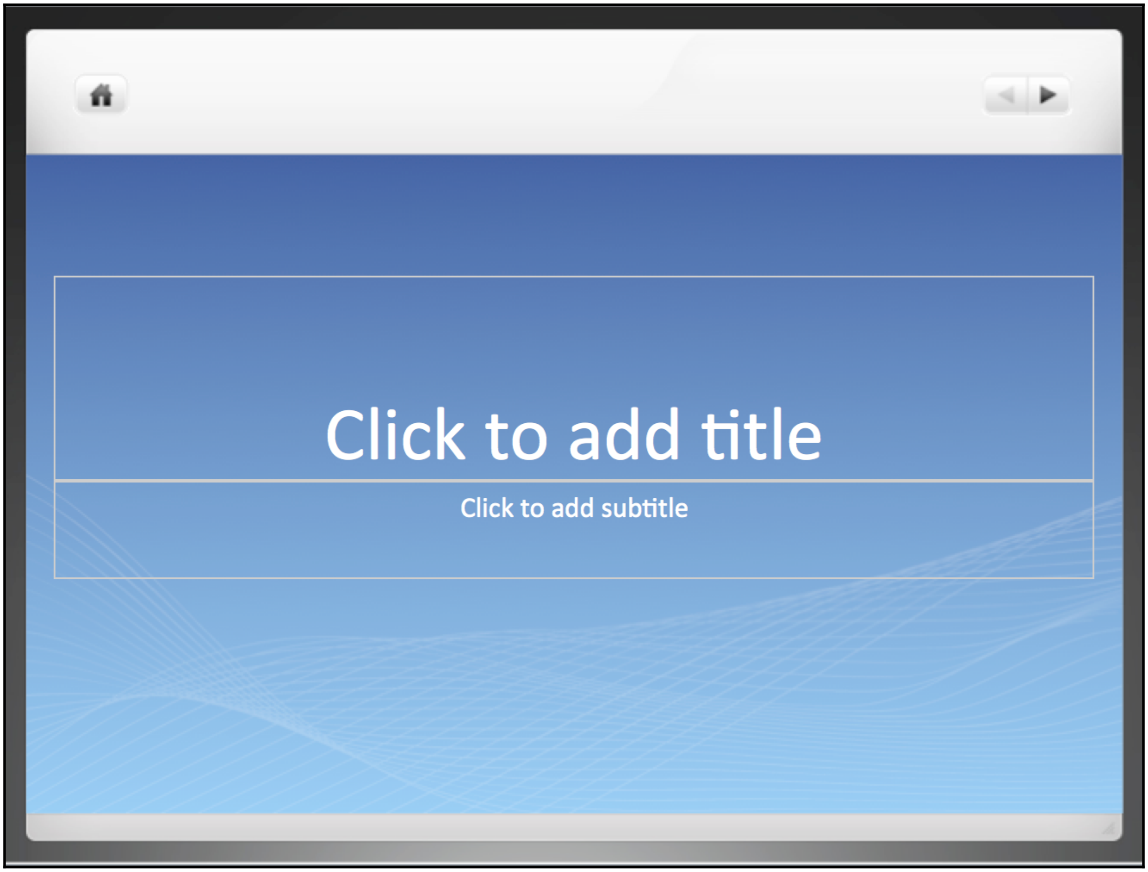


Yo, Python!

Yes it is really awesome

Also, observe that the slide layout is that of the **Title Slide**.

2. We can also create a new presentation from an existing presentation. In the next code example, we take a PowerPoint template, create a new PPT, and add a slide to it with text content. We use the following template for this example:



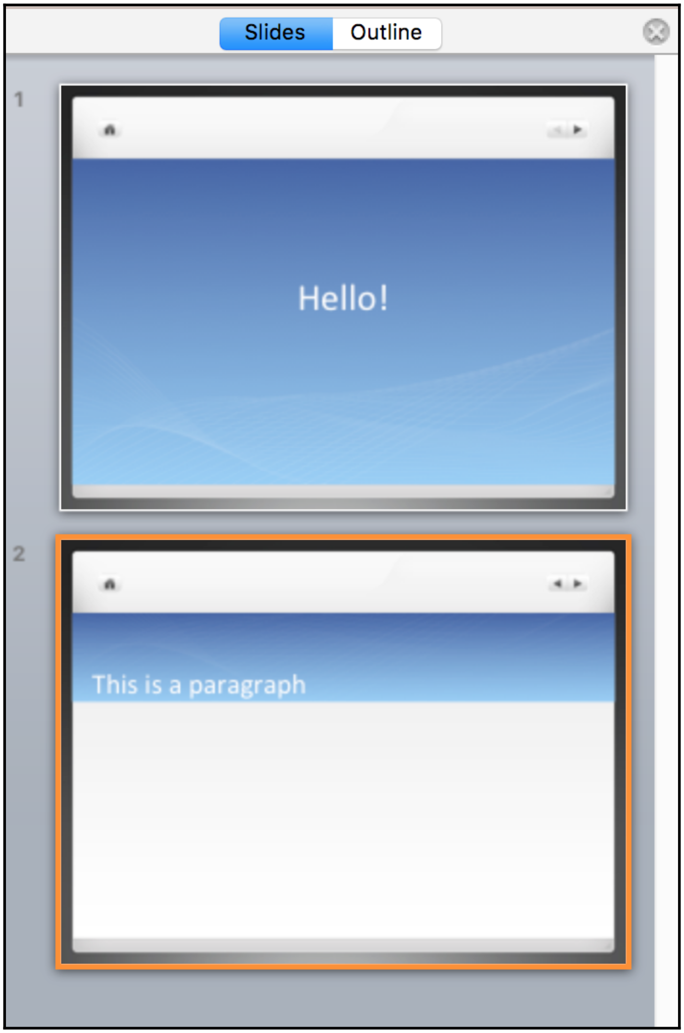
Click to add title

Click to add subtitle

If we run the following program on the template presentation, we get a new PPT, as shown in the next screenshot:

```
from pptx import Presentation
prs = Presentation('sample_ppt.pptx')

first_slide = prs.slides[0]
first_slide.shapes[0].text_frame.paragraphs[0]
                     .text = "Hello!"

slide = prs.slides.add_slide(prs.slide_layouts[1])
text_frame = slide.shapes[0].text_frame
p = text_frame.paragraphs[0]
p.text = "This is a paragraph"

prs.save('new_ppt.pptx')
```

3. Slide one is updated with a title text, **Hello!**, and a new slide with the layout, title, and content is added with the text, **This is a paragraph**. The following screenshot shows the newly created presentation, `new_ppt.pptx`:



How it works…

In this section, we learnt how to create presentations with Python. We achieved three different things in the preceding code snippets.

First, we used the default template to create a **Title Slide** and added title text and subtitle text to it. We achieved this with the following steps:

1. Created a presentation object, `prs`, using the `Presentation` class of the `pptx` module.
2. The `prs` object was then utilized to add a new slide with the help of the `add_slide()` method. Layout `0` was passed as an argument to `add_slide()`, which indicated that the new slide was of type `Title` and was referenced by variable `slide`.
3. Title layout typically contains a title and subtitle. The contents of the title text were added with the `slide.shape.title.text` attribute, while the contents of the subtitle were added with the help of `slide.placeholders.text` attribute.

Next, we created a new presentation from an existing PPT template. The template was stored in the `sample_ppt.pptx` file and already contained a blank layout slide. This is what we achieved in this recipe:

1. We created a presentation object `prs` from the template PPT. We then used the presentation object to reference the first slide, `prs.slides[0]`, which was stored in the variable, `first_slide`.
2. The `first_slide` object was then used to access the first shape, which is the title text. Title text was then updated to content **Hello!**
3. Later, we added a new slide with **Layout** one (**Title and Content**) and referenced it with the `slide` variable. The newly created slide's first shape was a text frame to which the content **This is a paragraph** was added.
4. In the end, we saved the newly created presentation under the name `new_ppt.pptx`.

There's more…

Cool! So, we learnt how to create new presentations from scratch, update an existing template, add new content to it and create a presentation, and lastly, create presentations with different types of layout and bullet data. In the next recipe, let's look at what else we can do with presentations using Python.

Playing with layouts, placeholders, and textboxes

Now, let's move on to do some interesting operations on PPTs. Importantly, we'll discuss operations that are most frequently used.

Getting ready

No specific modules are required for this recipe; we would use the `python-pptx` module that was installed for the earlier recipes. In this recipe, we will work with different slide layouts and play with shapes and text.

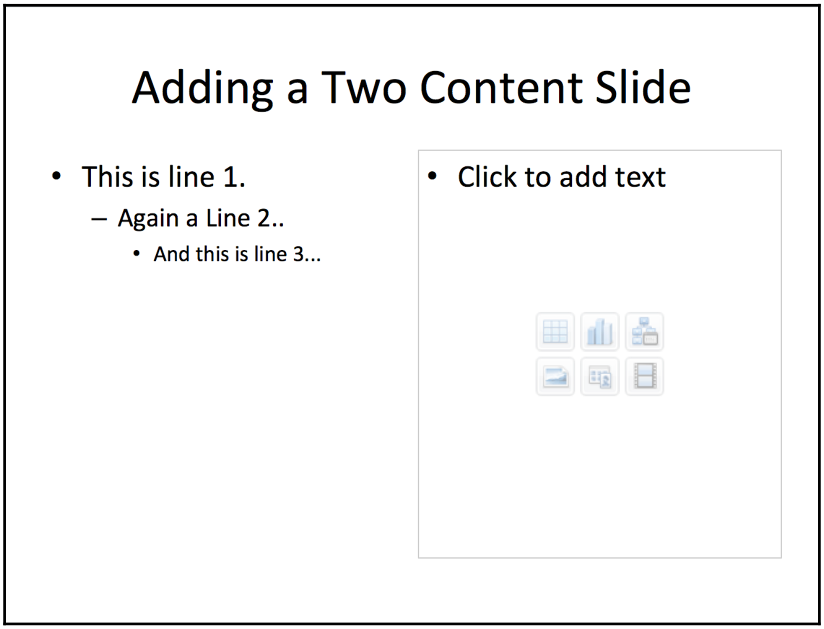
How to do it…

1. Let's go further and use a different type of slide layout, along with bulleted content in it. The following code snippet does what we need:

```
from pptx import Presentation

prs = Presentation()
two_content_slide_layout = prs.slide_layouts[3]
slide = prs.slides.add_slide(two_content_slide_layout)
shapes = slide.shapes

title_shape = shapes.title
title_shape.text = 'Adding a Two Content Slide'

body_shape = shapes.placeholders[1]
tf = body_shape.text_frame
tf.text = 'This is line 1.'

p = tf.add_paragraph()
```

```
p.text = 'Again a Line 2..'
p.level = 1

p = tf.add_paragraph()
p.text = 'And this is line 3...'
p.level = 2

prs.save('two_content.pptx')
```

When we run the preceding Python code, we get a new PPT with a *two content* slide with bulleted content added to it. The following screenshot shows the output of the new presentation that was created. Nice, right? We can add bulleted content to the placeholders on the left-hand side of the slide:



Adding a Two Content Slide

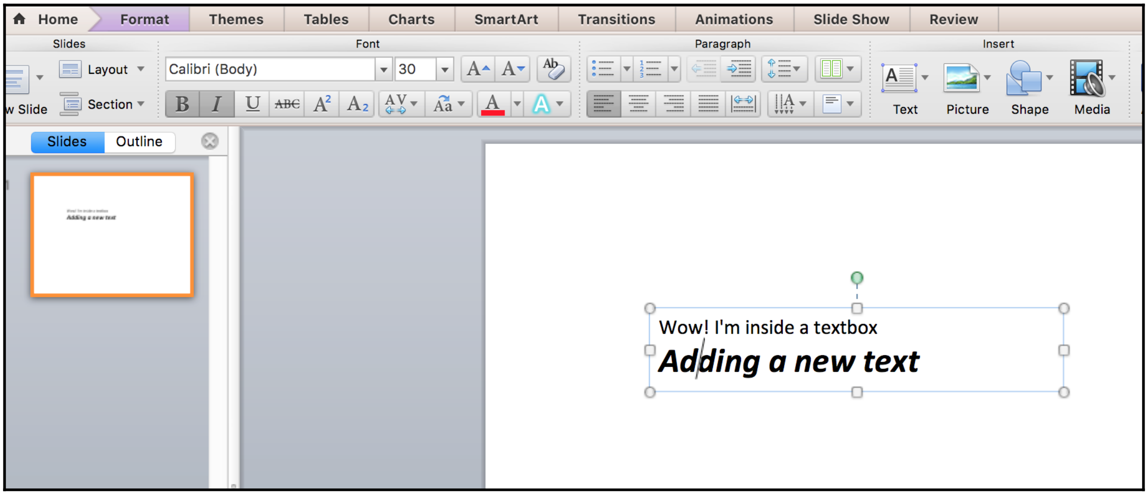
- This is line 1.
  - – Again a Line 2..
    - And this is line 3...

- Click to add text

2. Now, let's get creative and add another type of shape to our slide, that is, the textbox. The following code snippet adds a textbox to our slide:

```
from pptx import Presentation
from pptx.util import Inches, Pt
prs = Presentation()
blank_slide_layout = prs.slide_layouts[6]
slide = prs.slides.add_slide(blank_slide_layout)

txBox = slide.shapes.add_textbox(Inches(2),
        Inches(2), Inches(5), Inches(1))
tf = txBox.text_frame
tf.text = "Wow! I'm inside a textbox"

p = tf.add_paragraph()
p.text = "Adding a new text"
p.font.bold = True
p.font.italic = True
p.font.size = Pt(30)

prs.save('textBox.pptx')
```

The following screenshot shows how the newly created PPTX file looks. If you carefully look at it, you'll notice that we added a text box and the second text inside the textbox is in bold, italics, and the font size is 30.



Wow! I'm inside a textbox

Adding a new text

How it works…

In this recipe, we used the blank template to add a two-content layout slide. Let's see what else we did on top of it:

1. In the code, we used `prs.slide_layouts[3]` to convey the presentation object, `prs`, to add a slide with the two content layout. The two-content layout also has a title text, which was updated to **Adding a Two Content Slide** by using the `shapes.title` attribute.

2. Next, we looked at the *placeholders*. A placeholder is a preformatted container to which content can be added. Placeholders are categories of shape, which means multiple shapes can have placeholders. For instance, an autoshape (circle, as seen in the first recipe) is a placeholder; a picture or graphic frame can be a placeholder. We have two placeholders, one on the left and one on the right, in a two-content slide. We targeted the left one with `shapes.placeholders[1]` and added a first line **This is line 1.** to the text frame referenced by `shapes.placeholders[1].text_frame`.
3. We then added bulleted lines by adding a paragraph to `text_frame` with the `add_paragraph()` method and added the text **Again a line 2...** at level one and **And this is line 3...** at level two.
4. Basically, not all shapes contain text, but finding if a shape supports text can be done using the `shape.has_text_frame` attribute. In this recipe, we know our shape contains a placeholder that can handle text content. So, we used the `text_frame` attribute to add the first line of text. Similarly, we used the `add_paragraph()` method to add the subsequent lines in bulleted fashion using the `level` attribute.
5. In the end, we saved the new presentation under the name, `two_content.pptx`. If you look at the screenshot, we see the bulleted look on the text added to the left text frame on the slide.

Next, we added a textbox to our presentation. Textboxes are very commonly used in presentations. People use textboxes to highlight points and use the resizing and moving capabilities of textboxes to good effect. Here's what we did in our recipe:

1. We first created a blank slide with layout six and added it to the presentation with the `add_slide()` method.
2. Next, we created a textbox with the appropriate dimensions. We used `Inches(2)` for the left and top coordinates and then managed the width and height with `Inches(5)` and `Inches(1)` respectively. Inches here map to the same real world entity where *1 inch = 2.54 cm*. We added this textbox to the slide with the `add_textbox()` method.
3. Using the textbox object, we added a text frame object `tf` with the `text_frame` attribute.
4. As seen in the previous recipe also, we added the text **Wow! I'm inside a textbox** to the text frame.

5. We followed it up by adding a paragraph with the `add_paragraph()` method and adding the text **Adding a new text** to this paragraph and making the text bold, italics, and increasing its font to `30`.
6. In the end, we saved the file as `textBox.pptx`.

You learned about placeholders and text frames. You learnt how to add text to our slides using text frames and paragraphs. You also learnt about adding a textbox of the required dimensions to our slide deck.

Working with different shapes and adding tables

OK, let's go ahead and make our presentations more interesting by adding different shapes, tables, or even pictures! Why wait? Let's quickly get into action.

Getting ready

No specific modules are required for this recipe; we will use the `python-pptx` module that was installed for the earlier recipes.

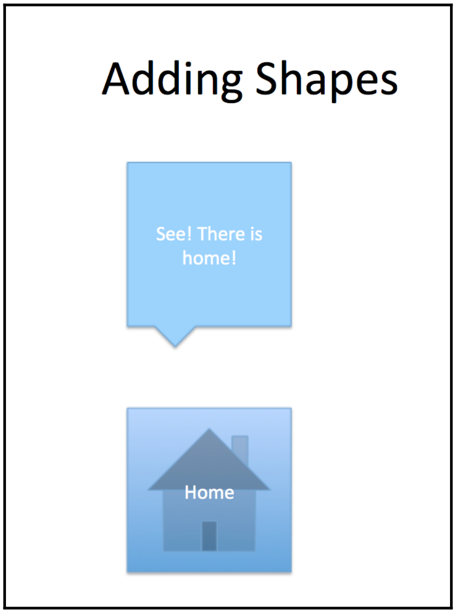
How to do it…

1. In this recipe, we will add a few shapes to our presentation. Shapes are very useful in presentations as they can represent real world objects, can indicate relations, and provide great visual feedback to the audience (who are listening to the presentation). In this recipe, we add a **Home** button and then a **Rectangular Callout** to show where our **Home** button is. We'll also fill custom color into our `Callout` element. The following code adds the shapes to the presentation:

```
from pptx import Presentation
from pptx.enum.shapes import MSO_SHAPE
from pptx.util import Inches
from pptx.dml.color import RGBColor

prs = Presentation()
title_only_slide_layout = prs.slide_layouts[5]
slide = prs.slides.add_slide(title_only_slide_layout)
shapes = slide.shapes
```

```
shapes.title.text = 'Adding Shapes'

shape1 = shapes.add_shape(MSO_SHAPE.RECTANGULAR_CALLOUT,
Inches(3.5), Inches(2), Inches(2), Inches(2))
shape1.fill.solid()
shape1.fill.fore_color.rgb = RGBColor(0x1E, 0x90, 0xFF)
shape1.fill.fore_color.brightness = 0.4

shape1.text = 'See! There is home!'

shape2 = shapes.add_shape(MSO_SHAPE.ACTION_BUTTON_HOME,
Inches(3.5), Inches(5), Inches(2), Inches(2))
shape2.text = 'Home'

prs.save('shapes.pptx')
```

After running the preceding code, we get a new presentation, `shapes.pptx`, which looks like the following screenshot:

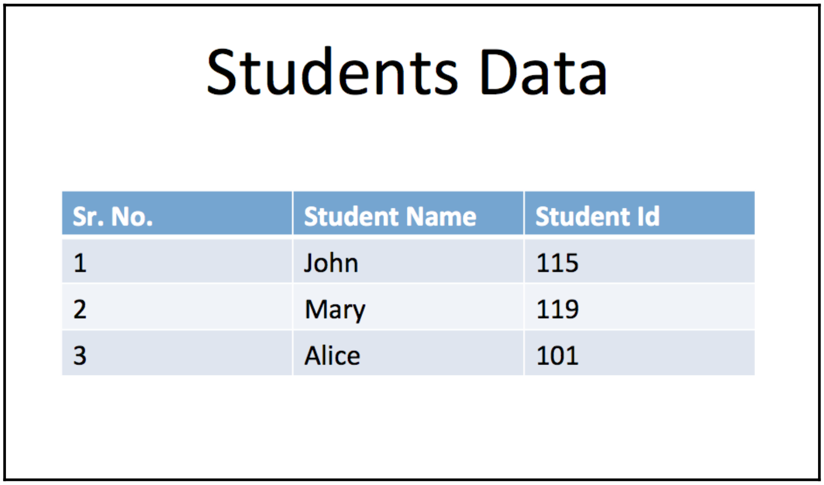


Adding Shapes

2. That was neat! Now let's see if we can add a table to our presentation. Again, tables are used in presentations to manifest data and make informed decisions. Speakers (persons responsible for making the presentation) often bring out facts about certain projects through tables and solicit discussions or feedback from audiences. Adding tables in presentations is trivial with Python; refer to the following code. In this, we add a table containing the student information for three students:

```
from pptx import Presentation
from pptx.util import Inches

prs = Presentation()
title_only_slide_layout = prs.slide_layouts[5]
slide = prs.slides.add_slide(title_only_slide_layout)
shapes = slide.shapes
shapes.title.text = 'Students Data'

rows = 4; cols = 3
left = top = Inches(2.0)
width = Inches(6.0)
height = Inches(1.2)

table = shapes.add_table(rows, cols, left, top,
                         width, height).table
table.columns[0].width = Inches(2.0)
table.columns[1].width = Inches(2.0)
table.columns[2].width = Inches(2.0)

table.cell(0, 0).text = 'Sr. No.'
table.cell(0, 1).text = 'Student Name'
table.cell(0, 2).text = 'Student Id'

students = {
    1: ["John", 115],
    2: ["Mary", 119],
    3: ["Alice", 101]
}

for i in range(len(students)):
    table.cell(i+1, 0).text = str(i+1)
    table.cell(i+1, 1).text = str(students[i+1][0])
    table.cell(i+1, 2).text = str(students[i+1][1])

prs.save('table.pptx')
```

If we run this code snippet, you'll see a table generated in the presentation containing all the student data. Refer to the following screenshot:



Students Data

Sr. No.	Student Name	Student Id
1	John	115
2	Mary	119
3	Alice	101

How it works...

In the first code snippet, we added a couple of shapes to our presentation by performing the following operations:

1. We added shapes with the help of the `add_shapes()` method, which takes the type of shape as input.
2. In our code, we take the help of the `MSO_SHAPE` enumeration (this has all the shapes listed) and pick up two shapes, namely, `MSO_SHAPE.RECTANGULAR_CALLOUT` and `MSO_SHAPE.ACTION_BUTTON_HOME`. Just like in the case of a textbox, the `add_shapes()` method also needs the size of the shape defined with the `Inches()` method.
3. We also managed to define the custom color for the callout shape with the help of the `fill` method from the `SlideShape` class. The `shape.fill.fore_color.rgb` attribute was used to set the color of the callout shape. The RGB color used was `1E90FF`, which is light blue color, as seen in the screenshot. We also set the color brightness with the help of the `shape.fill.fore_color.brightness` attribute.
4. Of course, we added text to both the shapes by setting the `shape.text` attribute. In the end, we saved the file as `shapes.pptx`.

In the second example, we added a nice table to our presentation with the help of Python code. This is how we did it:

1. We created a presentation with **Title only** layout and added a single slide to it using the `add_slide()` method. We also defined the title of the slide as **Students Data**.
2. Adding the table is as simple as adding shapes. We used the method `add_table()` to add a table to the presentation. As expected, the `add_table()` method expects a number of rows and columns as input, and at the same time, expects the size of the table. In our example, we have set the rows to `4` and columns to be `3` and the size of the table with coordinates `Inches(2)`, `Inches(2)`, `Inches(6)`, and `Inches(8)`, which means the table is located 2 inches from the left, 2 inches below the top of slide, the width of the table is 6 inches, and the height is 1.2 inches (15.3cm x 3.1cm).
3. We defined our table to have three columns; the width of each of those is set to 2 inches. We set that using the `table.columns.width` attribute. We also set the text of the column headings to **Sr. No**, **Student Name**, and **Student Id**, with the help of `table.cell(row, column).text` attribute. Note that here, the row value is always `0`, which indicates the first row or the title row, and the column varies from `0` to `2`, indicating three columns.
4. For the purpose of this example, we used a predefined dictionary, `students`, which has information such as student name and student ID. We iterate through all the student information and update the cells of the table to fill in the table with the appropriate information, hence the table with all the student data, as observed in the screenshot.
5. In the end, we saved the presentation as `table.pptx`.

There's more…

Nice! What else can we do with presentations using Python, you'd ask? Or a few of you are already expecting me to get graphical and talk about charts or pictures, aren't you? Oh yes, we'll get that covered as well. Let's get graphical, so to speak!

Visual treat with pictures and charts

Yes, it's time. In this section, we will look at adding pictures and charts to your presentations. They say that a picture speaks a thousand words, and truly, you must have seen presentations with a lot of pictures and graphs in them. They are there for a reason. You can convey as much information as possible in one slide. Both charts and pictures have this power, and without learning about them, this chapter remains incomplete. So, let's jump in!

Getting ready

No specific modules are required for this recipe; we will use the `python-pptx` module that was installed for the earlier recipes.

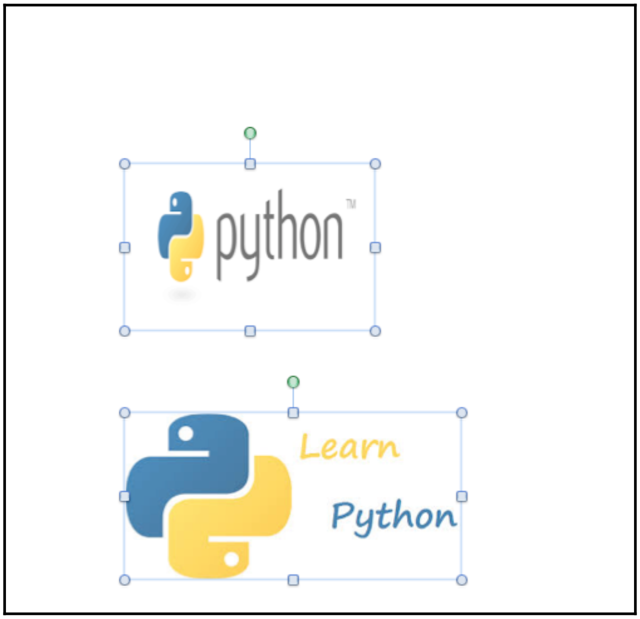
How to do it…

We divide this recipe into two parts. First, we will cover the adding of pictures to our slide, and in the next, we'll deal with charts.

1. The following code snippet helps us add pictures to our slide:

```
from pptx import Presentation
from pptx.util import Inches

img_path = 'python.png'
img_path2 = 'learn_python.jpeg'
prs = Presentation()
blank_slide_layout = prs.slide_layouts[6]
slide = prs.slides.add_slide(blank_slide_layout)

left = top = Inches(2)
pic = slide.shapes.add_picture(img_path, left,
      top, height=Inches(2), width=Inches(3))

left = Inches(2)
top = Inches(5)
height = Inches(2)
pic = slide.shapes.add_picture(img_path2, left,
                               top, height=height)

prs.save('picture.pptx')
```

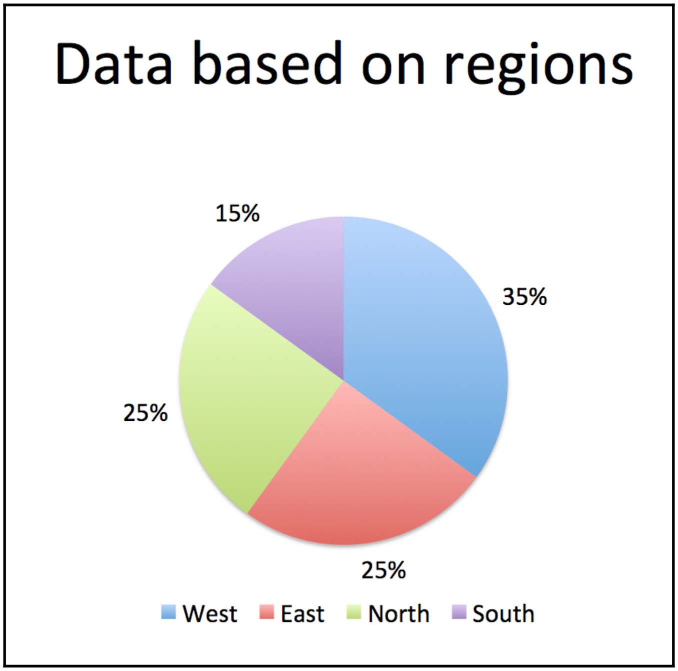
When we run the preceding program, we get a slide with two pictures on it, as shown in the following screenshot:



2. Now let's see if we can write Python code to add a chart to our slide. The `python-pptx` module supports multiple types of charts, such as line chart, bar chart, and bubble chart, but my favorite chart always has been a pie chart. How about adding one in our recipe? Yes, the following code adds a pie chart to the presentation:

```
from pptx import Presentation
from pptx.chart.data import ChartData
from pptx.enum.chart import XL_CHART_TYPE
from pptx.enum.chart import XL_LABEL_POSITION,
                            XL_LEGEND_POSITION
from pptx.util import Inches

prs = Presentation()
slide = prs.slides.add_slide(prs.slide_layouts[5])
slide.shapes.title.text = 'Data based on regions'

chart_data = ChartData()
chart_data.categories = ['West', 'East',
                         'North', 'South']
chart_data.add_series('Series 1', (0.35,
                         0.25, 0.25, 0.15))

x, y, cx, cy = Inches(2), Inches(2), Inches(6),
               Inches(4.5)
chart = slide.shapes.add_chart(
    XL_CHART_TYPE.PIE, x, y, cx, cy, chart_data
).chart

chart.has_legend = True
chart.legend.position = XL_LEGEND_POSITION.BOTTOM
chart.legend.include_in_layout = False

chart.plots[0].has_data_labels = True
data_labels = chart.plots[0].data_labels
data_labels.number_format = '0%'
data_labels.position = XL_LABEL_POSITION.OUTSIDE_END

prs.save('chart.pptx')
```

The output of the preceding program is as follows:



How it works…

In the first code snippet, we added two images to the slide. And how did we do it? As logical as it sounds, we used the `add_picture()` method. Isn't the library nice? Add a textbox with `add_textbox()`, add a slide with `add_slide()` and now add a picture with `add_picture()`. Let's take a deeper look at what we did in the first part of the recipe:

1. As expected, `add_picture()` expects the path from where the image needs to be added to the presentation, and just like other methods, the coordinates and size of the picture. In our example, we added two pictures. The first picture is `python.org` and we configured it to show up 2 inches from the left and 2 inches from the top. We also configured the size of the picture to have a width of 3 inches and height of 2 inches.
2. The second picture we added was `learn_python.jpeg`; it was configured to be 2 inches from left, 5 inches from the top, with a height of 2 inches and width to be same as the width of the image.
3. In our example, we created a new slide with a blank slide layout and added both the pictures, and in the end, we saved the file as `picture.pptx`.

In the second part, we added a pie chart to our slide deck. We did this in the following way:

1. We added a slide and set the title text to **Data based on regions**.
2. We then created an object of the class, `ChartData()`, and named it `chart_data`. We defined the categories for our pie chart with the `chart_data.categories` attribute and set it to an array of regions `['West', 'East', 'North', 'South']`. We also configured the `chart_data` object with data for all the regions; we did that with the `add_series()` method.
3. How did we add this chart on the presentation slide? Yes, you guessed it: the `add_chart()` method does that for us. The `add_chart()` method expects the type of chart as one of the arguments, and like other methods, it expects the dimensions. In our code, we also set the attributes `has_legend`, `number_format`, and data labels to make the pie chart look great!

There's more…

Cool! So we learnt many interesting things in this chapter. But what better fun than applying this knowledge to solve a real world use case. Did you hear Alex has some issues with his weekly sales report?

Automating weekly sales reports

Alex is the director of sales at Innova 8 Inc that sells laptops and business software. He has a bunch of sales managers reporting to him. He is responsible for measuring the success of his subordinates and reporting this to the VP of sales on a weekly basis. Alex's boss is majorly interested in two things: the revenue generated from business accounts and the performance of his sales managers. Alex needs to report these numbers in the weekly staff meeting. He uses PowerPoint as a tool to collate and represent the data to the VP of sales on a weekly basis.

However, Alex has a few problems. The data that he gets from his sales managers is often in an Excel sheet. Also, the data is so dynamic that it changes till the last moment based on whether the customer paid just before the meeting. It becomes difficult for Alex to create the presentation in advance for the meeting due to this variability. Also, it is tedious for Alex to analyze the data and come up with charts–a completely manual process.

Can you help Alex with what you learnt in this chapter?

Getting ready

If you analyze the problems, we can automate the complete process for Alex. Alex's data is in an Excel sheet; we can easily read this data with the `pandas` module of Python. Also, we can create a new presentation with the `python-pptx` module.

The following steps can help solve Alex's problem:

1. Read the contents of the Excel sheet and get the required data.
2. Create a new PowerPoint presentation and add two slides to it.
3. On the first slide, create a pie chart that shows the revenue figures for different accounts and compare them based on percentages.
4. On the second slide, add a bar chart comparing the performance of sales managers based on revenue.

For this recipe, let's install the `pandas` module. We do that with our favorite utility Python `pip`. We use the following command for installing `pandas`:

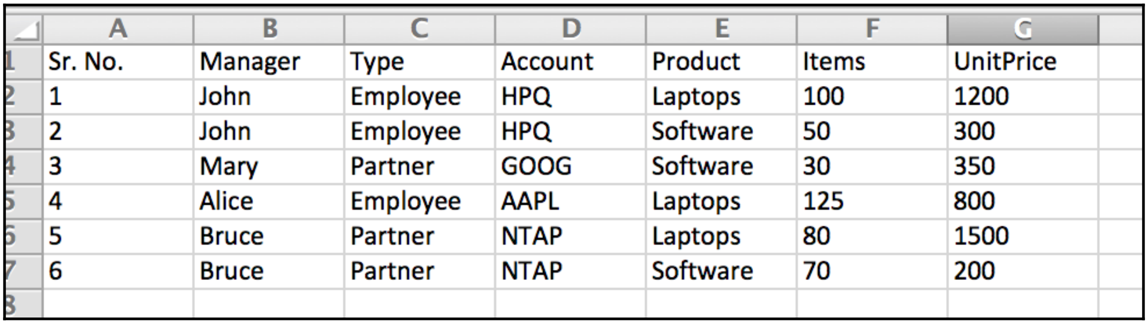
```
pip install pandas
```

This is a trivial example of working with Excel sheets using pandas. The pandas module has comprehensive set of APIs that can be used for data analysis, filtering, and aggregation. We talk about all this and much more in the chapter that deals with data analysis and visualizations.

Now that we're ready, let's look at the code that will help Alex automate this process.

How to do it…

1. Let's start by looking at the Excel sheet that has the weekly sales data. We call this file as Sales_Data.xlsx and it looks like the following screenshot:

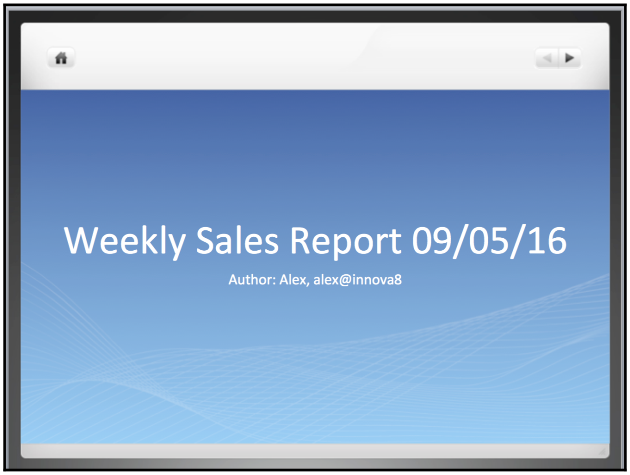


	A	B	C	D	E	F	G
1	Sr. No.	Manager	Type	Account	Product	Items	UnitPrice
2	1	John	Employee	HPQ	Laptops	100	1200
3	2	John	Employee	HPQ	Software	50	300
4	3	Mary	Partner	GOOG	Software	30	350
5	4	Alice	Employee	AAPL	Laptops	125	800
6	5	Bruce	Partner	NTAP	Laptops	80	1500
7	6	Bruce	Partner	NTAP	Software	70	200
8							

2. Now, let's look at the code snippet that will help Alex read this data and generate the exact presentation he needs:

```
from pptx import Presentation
from pptx.chart.data import ChartData
from pptx.enum.chart import XL_CHART_TYPE
from pptx.enum.chart import XL_LABEL_POSITION,
                            XL_LEGEND_POSITION
from pptx.util import Inches
from datetime import datetime
import pandas as pd

xls_file = pd.ExcelFile('Sales_Data.xlsx')


prs = Presentation('sample_ppt.pptx')

first_slide = prs.slides[0]
first_slide.shapes[0].text_frame.paragraphs[0]
           .text = "Weekly Sales Report %s" \
                  % datetime.now().strftime('%D')
first_slide.placeholders[1].text =
                   "Author: Alex, alex@innova8"
blank_slide_layout = prs.slide_layouts[6]
slide = prs.slides.add_slide(blank_slide_layout)
slide.shapes.title.text = '% Revenue for Accounts'
df = xls_file.parse('Sales')
df['total'] = df['Items'] * df['UnitPrice']
plot = df.groupby('Account')['total']
         .sum().plot(kind='pie', \
         autopct='%.2f', fontsize=20)
f=plot.get_figure()
f.savefig("result.png", bbox_inches='tight',
                                     dpi=400)
left = Inches(2.5);  top = Inches(3)
pic = slide.shapes.add_picture("result.png", left, top,
height=Inches(4), width=Inches(5))

slide = prs.slides.add_slide(prs.slide_layouts[6])
slide.shapes.title.text = 'Sales Manager Performance'
df = xls_file.parse('Sales')
df['total'] = df['Items'] * df['UnitPrice']
mgr_data = df.groupby(['Manager'])['total'].sum()
managers = mgr_data.index.values.tolist()
sales = []
for mgr in managers:
    sales.append(mgr_data.loc[mgr])

chart_data = ChartData()
chart_data.categories = managers
chart_data.add_series('Series 1', tuple(sales))
x, y, cx, cy = Inches(2), Inches(3), Inches(6),
               Inches(4)
chart = slide.shapes.add_chart(
XL_CHART_TYPE.COLUMN_CLUSTERED, x, y, cx, cy,
                                   chart_data
).chart

chart.has_legend = True
chart.legend.position = XL_LEGEND_POSITION.BOTTOM
chart.legend.include_in_layout = False

prs.save('sales.pptx')
```

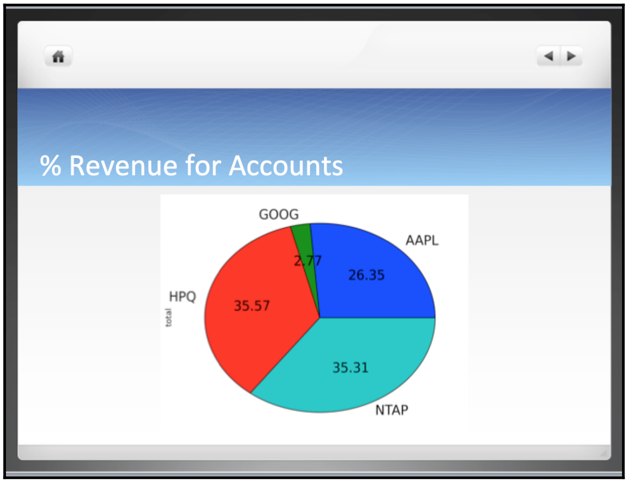
3. Now, if we run the preceding piece of code, we get a PowerPoint presentation with all the necessary charts and data points that Alex needs in his weekly sales report. Look at the screenshots for all the slides of the presentation. Exactly what the doctor ordered for Alex! The first slide is the title slide of the presentation and is titled as **Weekly Sales Report <Date>**. It also mentions Alex's name and e-mail address as the author of this presentation:



Weekly Sales Report 09/05/16

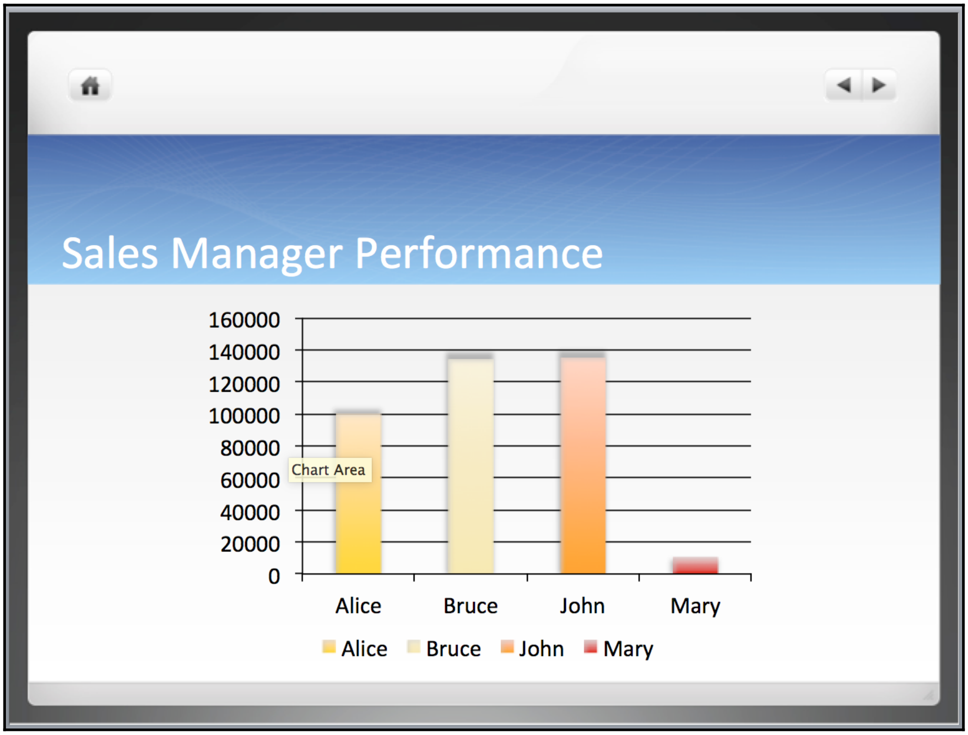
Author: Alex, alex@innova8

The second slide shows the revenue distribution across accounts with the help of a pie chart:



% Revenue for Accounts

And finally, the last slide compares the performance of all sales managers with the help of a bar chart. Cool, huh?



Sales Manager Performance

How it works…

In the preceding example, we started by creating a reader object for the Excel sheet, `Sales_Data.xlsx` containing all the sales data. We achieved this by using the `ExcelFile()` method of the `pandas` module.

Next, we created a new presentation by utilizing the `sample_ppt.pptx` file. If you recollect, our sample presentation has a title slide without any text on it. So, in our code snippet, we updated this title slide by setting the title to **Weekly Status Report <YYYY–MM–YY>**. We also added a subtitle that contained the name of the author, in this case **Author: Alex alex@innova8**. We set these titles with the help of placeholders on the text frame.

Next, we added a new blank slide with layout six to our new presentation. We used this slide to add the revenue data for the accounts. We achieved this with the help of the `prs.slides.add_slide()` method. However, our data is in an Excel sheet, and hence, we use the `reader` object of the Excel sheet to read the **Sales** worksheet. The **Sales** worksheet has all the data Alex uses for his analysis. The pandas module reads Excel data in the form of a data frame (data stored in the matrix format). If you look at the Excel sheet screenshot, we have two columns, `Price` and `Quantity`, which indicates the quantity of laptops or software licenses sold and the price per unit. So, in our code, we first multiply these values to get the revenue for each entry in the Excel record and store it in the data frame with the column name `total`. Now, we don't just need the total revenue figure; we also need to categorize it based on the account. With data frames, getting this information is really trivial; it's like running an SQL query. If you look at the code, we have grouped the data frame by `Account` and summed all the `total` data (obtained from quantity * price) and used these data points to plot a pie chart so that its easier for Alex to compare the revenue for every account as a percentage of the total revenue. For grouping the data, we used the `groupby()` method and then totaled the revenue with the `sum()` method and plotted the pie chart with the `plot()` method. Again, it's not useful if the chart is available in `pandas` and not in PowerPoint, so we save the chart as a PNG file in `result.png`; we did that with the `savefig()` method of `pandas`. In the end, we added this picture file to our presentation using the `add_picture()` method and managed the coordinates and size of the picture so that it is visible and looks great.

Alex also needs to plot the performance of all his sales managers. For this, we used the same method to read the Excel sheet data and stored it in the form of data frames. For this problem though, we grouped the data by sales managers and got the total revenue attributed to each of them. Here too, we used the `groupby()` method but on the `Manager` column of the Excel data. We stored the names of all the sales managers in the array `managers`, iterated through all the records for each sales manager, got the sales figures for each of them, and added it to a list, `sales`. We then converted this list to a tuple to be used later. Just like we saw in the previous recipe, we created a chart data object using the `ChartData()` method, created a clustered bar chart with the 'sales' tuple as input, and added the chart to the second slide of the presentation with the `add_chart()` method.

In the end, we saved this newly created presentation as `sales.pptx`, which acts as a weekly sales report for Alex. That's it!

I hope you enjoyed the recipes in this chapter, the examples and the use cases we discussed. I'm sure your hands are itching to automate your presentations as well. When do I drop by your desk to appreciate your work?

7
Power of APIs

This chapter takes you on a journey to the interesting world of APIs. APIs are a critical part of the business world today. Tasks like querying data, exchanging information across services amongst others rely on web APIs and Webhooks.

In this chapter, we will cover the following topics:

- Designing your own REST APIs
- Automating social media marketing with Twitter APIs
- An introduction to Webhooks
- Implementing Webhooks
- Automating lead management with Webhooks

Introduction

APIs have become absolutely indispensable in the world driven by the Internet. Every web application that you must have interacted with uses an API on the backend to implement its core functionality–Amazon, Google, Twitter, you name it! What's more, you see all these applications thrive on APIs. Amazon uses it to drive its payment transactions and Google for showing you all the fancy maps. APIs are so essential to business that you hear the word API right from CEOs to managers, all the way to software developers. In general, using an API is a fundamental way of enabling different software to talk to each other. Operating system operations are also performed with APIs. They have been absolutely critical from the very beginning of computing.

But what are APIs and how are they useful? How to develop APIs of our own? How do they find their way into business process automation? We'll find answers to all these questions in this chapter.

We start with a more familiar and older web term: web service. Web services are essentially the key points of integration between different applications hosted across platforms and built on different languages using independent systems. Web services communicate with each other via WWW and typically involve two parties: one that exposes a set of APIs, also known as the server, and another that calls or consumes the server APIs, also known as consumers or clients. Web services are independent of the client implementation and thus work well with browsers, mobile phones, or just any software that can make an API call.

They use different protocols for communication, with different messaging and URI contracts. The most common implementations of web services are:

- HTTP-based **REST** (**Representation State Transfer**) web service
- SOAP-based (**Simple Object Access Protocol**) web service
- XML **RPC** (**Remote Procedure Call**)

Messaging formats that have been commonly used for these services are:

- **JSON** (**JavaScript Object Notation**)
- **XML** (**eXtensible Markup Language**)

Web services are at the very core of web applications today and hence need to provide good performance; they need to be scalable and reliable.

Right, so in this chapter, we'll cover HTTP-based REST APIs. You'll understand how to develop RESTful web services with Python in detail. You'll also learn how clients automate their business processes using RESTful web services.

Note that there are different terminologies available for referring to APIs, for example, the HTTP API, Web API, and so on. I suggest you read about them for better clarity. However, essentially, at their core APIs are integration points between two services in an application or between multiple servers/services across applications.

In this chapter, we'll look at multiple Python modules, mentioned in the following list:

- `flask` (`http://flask.pocoo.org/`)
- `twython` (`https://twython.readthedocs.io/en/latest/`)
- `pytz` (`https://pypi.python.org/pypi/pytz`)
- `django` (`https://www.djangoproject.com/`)
- `django-rest-hooks` (`https://github.com/zapier/django-rest-hooks`)

Designing your own REST APIs

Representational State Transfer (**REST**) has gained lot of preference and popularity in the community and is virtually the default architectural style for designing and implementing RESTful web services.

Note that there are other possible implementations of web services for which you can follow the SOAP and XML-RPC way, these are not within the scope of this chapter.

In this recipe, we'll learn how to implement a simple RESTful web service using the Python flask micro framework. We'll implement a user service for user management, which is an imperative aspect of any web application.

The REST architecture is designed to fit with the HTTP protocol and has the notion of resources, that is, **Uniform Resource Identifiers** (**URIs**). Clients send requests to these URIs with different HTTP request methods and get back the state of the affected resources as a response.

So what are we waiting for? Let's design and implement the user web service.

How to do it…

1. Let's begin by defining our model–**user**. Our user resource would be typically identified by these attributes:
    - `id`: A unique ID to identify the user
    - `username`: The name of the user used in the application
    - `email`: The e-mail address of the user for e-mail notifications
    - `status`: To check whether the user is active or verified
2. Designing REST APIs involves identifying resources (URIs) and verbs (HTTP methods) that take action on the user model. We need to perform actions such as creating a new user, updating certain attributes of the user, getting a user or list of users, or if needed deleting a user. We also need to relate our actions to HTTP verbs and have to define CRUD operations for our service. **CRUD** means performing the **Create, Read, Update, and Delete** operations on a user model. The following table shows exactly what we need:

URI	Method	Action
`http://v1/users/`	`GET`	To get a list of available users
`http://v1/users/`	`POST`	To create a new user
`http://v1/users/1/`	`GET`	To get details of an existing user with ID equal to 1
`http:///v1/users/1/`	`PUT/DELETE`	To update or remove the user with ID equal to 1

3. Now let's write the code to implement the RESTful user service. We start by creating a virtual environment. I hope we all know about `virtualenv`, but for beginners virtual environment is a tool that isolates Python modules. This helps in resolving issues related to permissions; it also helps to avoid polluting the global Python installation and to manage versions of the same module used across applications. If you don't have `virtualenv` on your system, you can install it using Python `pip` or download it from `https://pypi.python.org/pypi/virtualenv`.

```
chetans-MacBookPro:ch07 Chetan$ pip install virtualenv

chetans-MacBookPro:ch07 Chetan$ virtualenv user
New python executable in user/bin/python2.7
Also creating executable in user/bin/python
Installing setuptools, pip, wheel...done.
```

4. Once we have `virtualenv` installed, we need to activate it using a simple command. As you can see in the second line in the following, the `virtualenv` user has been activated:

```
chetans-MacBookPro:ch07 Chetan$ source user/bin/activate
 (user)chetans-MacBookPro:ch07 Chetan$
```

5. Let's go ahead and install `flask` in our virtual environment. We do this with the help of Python `pip` using the `pip install flask` command:

```
(user)chetans-MacBookPro:ch07 Chetan$ pip install flask
Collecting flask
  Using cached Flask-0.11.1-py2.py3-none-any.whl
Collecting click>=2.0 (from flask)
  Using cached click-6.6.tar.gz
Collecting its dangerous>=0.21 (from flask)
Collecting Werkzeug>=0.7 (from flask)
  Using cached Werkzeug-0.11.10-py2.py3-none-any.whl
Collecting Jinja2>=2.4 (from flask)
  Using cached Jinja2-2.8-py2.py3-none-any.whl
Collecting MarkupSafe (from Jinja2>=2.4->flask)
Building wheels for collected packages: click
  Running setup.py bdist_wheel for click
  Stored in directory: /Users/chetan/Library/
  Caches/pip/wheels/b0/6d/8c/
  cf5ca1146e48bc7914748bfb1dbf3a40a440b8b4f4f0d952dd
Successfully built click
Installing collected packages: click, itsdangerous,
  Werkzeug, MarkupSafe, Jinja2, flask
Successfully installed Jinja2-2.8 MarkupSafe-0.23
  Werkzeug-0.11.10 click-6.6 flask-0.11.1
  itsdangerous-0.24
```

6. If you look at the installation logs, we seem to have installed `flask` along with the template engines `Jinja2` and `Werkzeug`, a WSGI utility that supports multiple operations, such as cookie handling, file uploads, and request/response objects.

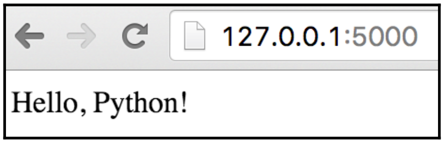
7. Okay good! We have `flask` installed and the environment is nicely set up. Let's write a minimalistic web application and name it `app.py`. The code for our web service looks like this:

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return "Hello, Python!"

if __name__ == '__main__':
    app.run(debug=True)
```

8. If you run the app, you will see the Flask server running on port 5000:

```
(user)chetans-MacBookPro:ch07 Chetan$ python app.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger pin code: 272-029-183
```

9. If you try to access the server on port 5000, you will see what we intended:



Hello, Python!

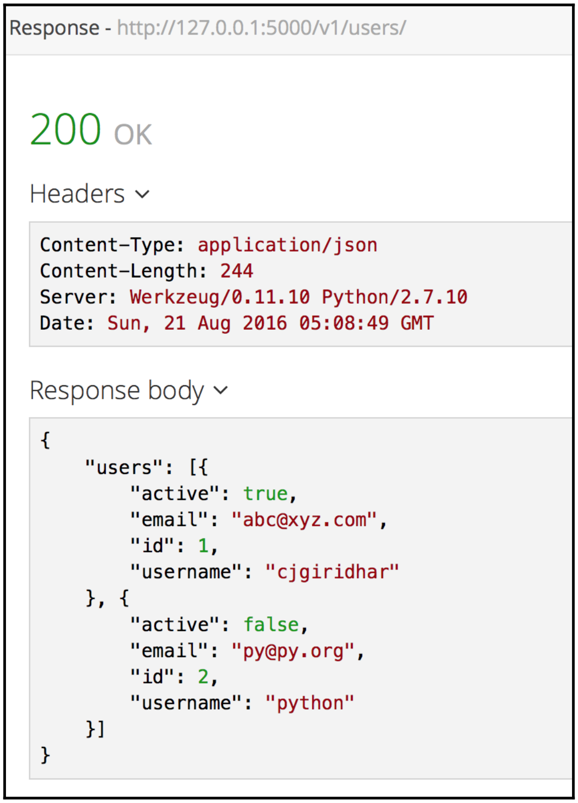
10. Great! Now let's improve on this to implement the user REST APIs. We first implement HTTP's `GET` on the `/v1/users/` resource. In the following code, we implement a `flask` route with the `get_users()` API to return all the users in JSON format:

```
from flask import Flask, jsonify
app = Flask(__name__)

users = [
    {
        'id': 1,
        'username': u'cjgiridhar',
        'email': u'abc@xyz.com',
        'active': True
    },
    {
            'id': 2,
            'username': u'python',
            'email': u'py@py.org',
            'active': False
        }
]

@app.route('/v1/users/', methods=['GET'])
def get_users():
    return jsonify({'users': users})

if __name__ == '__main__':
    app.run(debug=True)
```

11. If we rerun the app (if you're using an editor such as PyCharm, it will itself reload the app for you every time you save your code), our `flask` route gets loaded; we can now make an HTTP `GET` request on the `/v1/users/` API. The output of the request will yield the response, as seen in the next screenshot. Hey, cool! We wrote our first resource for the RESTful user service.
12. Notice the header section of the response:
    - `Content-Type` is application/JSON (we'll talk about message formats later in the chapter)
    - Server is Werkzeug on which Flask is based
    - Date refers to when the server responded to the request
    - The response body has the following:
    - An output with the users key containing information for all the users
    - Information about a user consists of desired attributes such as ID, username, e-mail, and account status

The response format is JSON, as indicated in the headers

Note that we use the RESTED plugin of Firefox to make these requests.



Response - http://127.0.0.1:5000/v1/users/

200 OK

Headers

```
Content-Type: application/json
Content-Length: 244
Server: Werkzeug/0.11.10 Python/2.7.10
Date: Sun, 21 Aug 2016 05:08:49 GMT
```

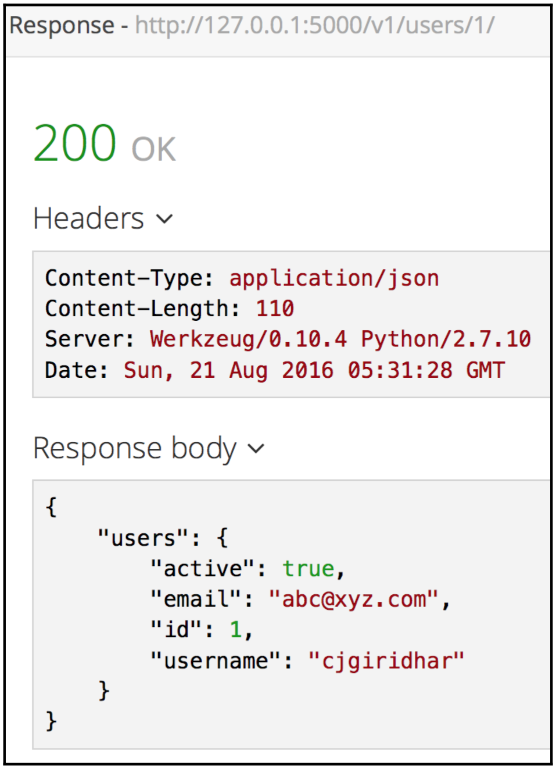
Response body

```
{
    "users": [{
        "active": true,
        "email": "abc@xyz.com",
        "id": 1,
        "username": "cjgiridhar"
    }, {
        "active": false,
        "email": "py@py.org",
        "id": 2,
        "username": "python"
    }]
}
```

13. Nice! We have the first URI implemented as part of user service. Now let's quickly go ahead and implement the next resource. Here, we need to get the user based on the ID. The following `flask` route will do the job for us:

```
@app.errorhandler(404)
def not_found(error):
    return make_response(jsonify({'error': 'Not found'}), 404)

@app.route('/v1/users/<int:id>/', methods=['GET'])
def get_user(id):
    for user in users:
        if user.get("id") == id:
            return jsonify({'users': user})
    abort(404)
```

14. In the preceding code, we defined a `flask` route with the API `get_user(id)`, which takes the user ID as an argument. When we make an HTTP `GET` request on this URI, the `get_user()` API gets invoked; it internally looks up all the available users to locate the user with the desired ID. If the user is found, the user record is returned in JSON format; if not, the server sends an HTTP 404 response. Here's a screenshot illustrating this:



Response - http://127.0.0.1:5000/v1/users/1/

200 OK

Headers

```
Content-Type: application/json
Content-Length: 110
Server: Werkzeug/0.10.4 Python/2.7.10
Date: Sun, 21 Aug 2016 05:31:28 GMT
```

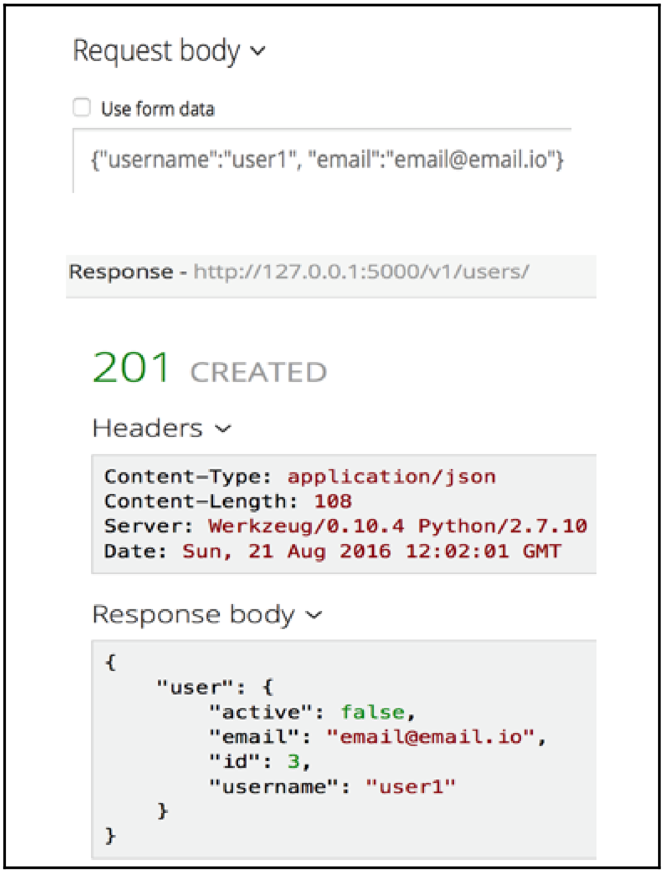
Response body

```
{
    "users": {
        "active": true,
        "email": "abc@xyz.com",
        "id": 1,
        "username": "cjgiridhar"
    }
}
```

15. You want users to sign up on your web application, right? So let's write a `flask` route that will help create a new user. The following code performs this operation:

```
@app.route('/v1/users/', methods=['POST'])
def create_user():
    if not request.json or not 'email' in request.json:
        abort(404)
    user_id = users[-1].get("id") + 1
    username = request.json.get('username')
    email = request.json.get('email')
    status = False
    user = {"id": user_id, "email": email,
            "username": username, "active": status}
    users.append(user)
    return jsonify({'user':user}), 201
```

16. Now if you make an HTTP `POST` request on the `/v1/users/` resource and pass user information to the body, you'll be able to create a new user. By default, the status of the user will be inactive (`'active': False`); you can make it `'active': False` when the user verifies her/his e-mail address:



```
{"username":"user1", "email":"email@email.io"}
```

Response - http://127.0.0.1:5000/v1/users/

201 CREATED

Headers

```
Content-Type: application/json
Content-Length: 108
Server: Werkzeug/0.10.4 Python/2.7.10
Date: Sun, 21 Aug 2016 12:02:01 GMT
```

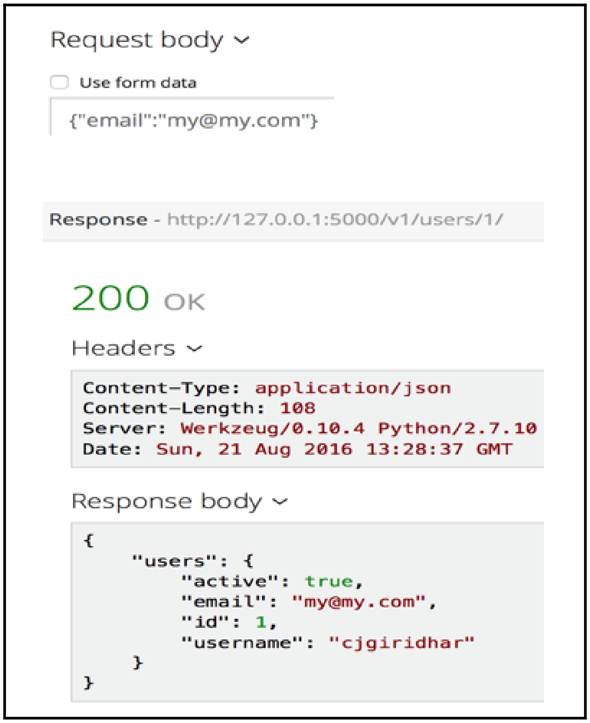
Response body

```
{
    "user": {
        "active": false,
        "email": "email@email.io",
        "id": 3,
        "username": "user1"
    }
}
```

17. OK! Now let's quickly look at the REST APIs that will edit the user details and delete the user if needed. The following flask route will edit the user details:

```
@app.route('/v1/users/<int:id>/', methods=['PUT'])
def update_user(id):
    user = [user for user in users if user['id'] == id]
    user[0]['username'] = request.json.get(
            'username', user[0]['username'])
    user[0]['email'] = request.json.get(
'email', user[0]['email'])
    user[0]['active'] = request.json.get(
'active', user[0]['active'])
    return jsonify({'users': user[0]})
```

18. Now if we perform the HTTP `PUT` operation on the `/v1/users/:id/` resource with the changed data, we should be able to update the user information. In the following screenshot, the request body contains the new e-mail address that needs to be updated for user ID equal to `1`. When we make an HTTP `PUT` request, the information gets updated and we have the new e-mail address for the user:



```
{"email":"my@my.com"}
```

Response - http://127.0.0.1:5000/v1/users/1/

200 OK

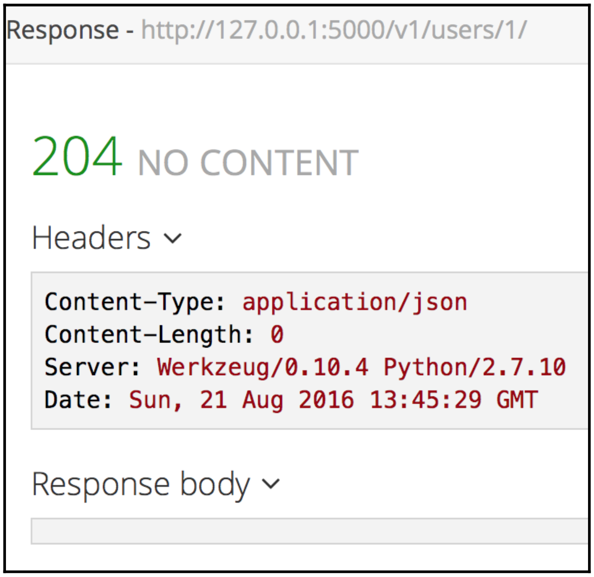
```
Content-Type: application/json
Content-Length: 108
Server: Werkzeug/0.10.4 Python/2.7.10
Date: Sun, 21 Aug 2016 13:28:37 GMT
```

```
{
    "users": {
        "active": true,
        "email": "my@my.com",
        "id": 1,
        "username": "cjgiridhar"
    }
}
```

19. The only operation pending now is to implement the `DELETE` operation. We can use this operation to delete a user. But you might ask, "Why would I delete the user?" So you can have your own implementation of `DELETE`; possibly, you can make the user inactive (by setting the `active` attribute to `False`). But for the sake of this discussion, let's delete the user for the heck of it.
20. The following code deletes the user based on the user ID:

```
@app.route('/v1/users/<int:id>/', methods=['DELETE'])
def delete_user(id):
    user = [user for user in users if user['id'] == id]
    users.remove(user[0])
    return jsonify({}), 204
```

21. The `DELETE` operation typically returns the status code **204 NO CONTENT**, which is shown in the following screenshot:



Response - http://127.0.0.1:5000/v1/users/1/

204 NO CONTENT

Headers

```
Content-Type: application/json
Content-Length: 0
Server: Werkzeug/0.10.4 Python/2.7.10
Date: Sun, 21 Aug 2016 13:45:29 GMT
```

Response body

22. Cool! So we have our RESTful user service completely implemented and up-and-running. Great!

How it works…

REST is an architectural style of the WWW. It consists of a coordinated set of components where the focus is on component roles and interactions between data elements rather than the implementation. Its purpose is to make the Web more scalable, portable, and reliable, and improve performance.

The REST architecture works according to the following constraints:

- **Client-server**: A **Uniform Resource Locator** (**URL**) separates the REST API from the client. Servers are not concerned about the user interface or the state; as a result, REST APIs are more scalable.
- **Stateless**: This means every request is independent and has no connection to the previous request or client. The client must contain all the necessary information to complete the request, and the session state remains with the client and hence is not stored on the server.
- **Cacheable**: RESTful web service can cache or not cache responses. Services must let the clients know whether the response is cached. This helps improve the performance of the system as some requests may no longer be needed based on the cache expiry time.
- **Layered system**: A client may or may not directly interact with the server; they can always have intermediary servers such as caches or load balancers.
- **Uniform resource**: Each REST resource should be independent; this allows you to have a separation of concerns, and it decouples the architecture.
- **Code on demand**: The server can provide code for clients to execute in their context. This is an optional requirement though.

The `GET` and `HEAD` methods are examples of safe methods as they don't change the state of the resource. The `PUT`/`DELETE` methods are idempotent. This means clients can make multiple similar calls to the resource and the resource will behave in the exact same way; of course, the response itself will have been changed as well.

Cool! So now we're in a position to create our own RESTful APIs. We can now host these on the Internet for our customers to use or implement functionalities in our web applications. Nice work!

There's more…

We looked at the fundamentals of the REST architecture and learned how to design a RESTful web service. We were helped by the Flask micro framework and learned how to write our own REST APIs.

In the next recipe, we'll see how the clients, due to their needs, consume REST APIs. We will also learn about using REST APIs to automate business processes.

Automating social media marketing with Twitter APIs

Joy is a marketing manager of a world-known consumer brand. She handles the content marketing portfolio of the company and heavily relies on blogs and social media to showcase the company's product line and create a buzz in the market.

Without doubt, she has a few problems! Some of the products that she markets have been designed differently for different markets, so she has to work across time zones to make sure her content gets published at the right time. She also feels the need to repeat her posts just to make sure she reaches the majority of her customers; this would also help in improving brand recognition.

Getting ready

If you carefully analyze her situation, Joy has two issues. One, she has to ensure her social media content is posted at the right time, based on her customer market. So if her product is being sold in Australia, she needs to make sure that her tweets are posted as per the Australian time when her customers are most likely to view them. Second, to get more traction for her product announcements, such as a weekend offer, she may want to repeat a few tweets at a later point in time.

OK! Now that we understand her problem, let's try to devise a solution. Looks like we need to take care of the following points:

- We should provide her with the capability of posting tweets in an automated fashion
- Her tweets should go at a desired time, even if Joy is asleep
- We should also provide the capability of scheduling repeat tweets

How to do it…

REST APIs to the rescue! Twitter has a fabulous set of REST APIs that can be used by the users for playing with Twitter data, user information, and of course posting tweets. You can also perform multiple operations such as uploading images, querying timelines, and sending direct messages. Wow! That's cool! But lets not get distracted and instead move on to the problem at hand:

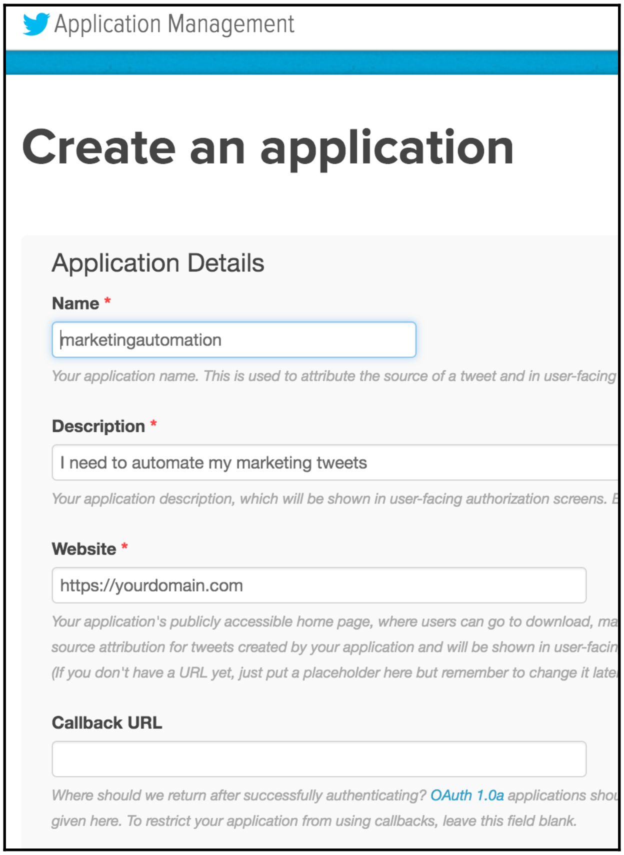
1. First, let's see how we can post tweets using Python even without logging in to Twitter. To post tweets, we will use a Python library called `twython`. So let's install `twython` using our friend, the Python `pip`:

```
        (user)chetans-MacBookPro:ch07 Chetan$ pip install twython
        Collecting twython
          Downloading twython-3.4.0.tar.gz
        Collecting requests>=2.1.0 (from twython)
          Downloading requests-2.11.1-py2.py3-none-any.whl (514kB)
            100% |████████████████████████████████|
516kB 495kB/s
        Collecting requests-oauthlib>=0.4.0 (from twython)
          Downloading requests_oauthlib-0.6.2-py2.py3-none-any.whl
        Collecting oauthlib>=0.6.2 (from requests-oauthlib>=0.4.0-
        >twython)
          Downloading oauthlib-1.1.2.tar.gz (111kB)
            100% |████████████████████████████████|
114kB 80kB/s
        Building wheels for collected packages: twython, oauthlib
          Running setup.py bdist_wheel for twython
          Stored in directory: /Users/chetan/Library/Caches/pip/
          wheels/48/e9/f5/a4c968725948c73f71df51a3c6
          759425358c1eda2dcf2031f4
          Running setup.py bdist_wheel for oauthlib
          Stored in directory: /Users/chetan/Library/Caches/pip/
          wheels/e6/be/43/e4a2ca8cb9c78fbd9b5b14b9
          6cb7a5cc43f36bc11af5dfac5b
        Successfully built twython oauthlib
        Installing collected packages: requests, oauthlib,
                               requests-oauthlib, twython
        Successfully installed oauthlib-1.1.2
          requests-2.11.1 requests-oauthlib-0.6.2 twython-3.4.0
         (user)chetans-MacBookPro:ch07 Chetan$
```

2. But before we can start playing with our Twitter account, we'd need to register an app with Twitter. This makes sure that Twitter is aware of our API calls and considers them legitimate. We can register an app by navigating to `https://apps.twitter.com/` and clicking on **Create New App**. You can fill in the details as shown in the following screenshot and create your app.

   Please note a few things about the details you need to fill in:

- The application name is unique across all the users on Twitter, so try to make it really unique to you, but at the same time keep it simple
- Make a description that exactly defines your use case so you remember it later
- Fill in your website name; keep it short
- A callback URL is only needed if you want Twitter to send data to you about your authentication, which is not needed for this exercise



Create an application

Application Details

Name *

marketingautomation

Your application name. This is used to attribute the source of a tweet and in user-facing

Description *

I need to automate my marketing tweets

Your application description, which will be shown in user-facing authorization screens. E

Website *

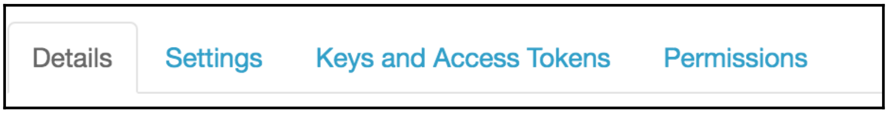
https://yourdomain.com

Your application's publicly accessible home page, where users can go to download, ma
source attribution for tweets created by your application and will be shown in user-facin
(If you don't have a URL yet, just put a placeholder here but remember to change it late

Callback URL

Where should we return after successfully authenticating? OAuth 1.0a applications shou
given here. To restrict your application from using callbacks, leave this field blank.

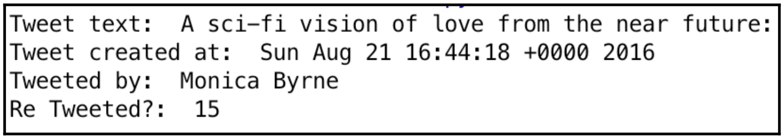
3. You also need to get your App Key and App Secret, and for that you need OAuth Token and OAuth Token Secret. These are needed essentially to authenticate your API calls with Twitter, or else Twitter will reject your REST API calls as malicious. You will get these details by clicking on your newly created app and browsing to the **Keys and Access Tokens** tab at the top of your page. You can also navigate to `https://apps.twitter.com/app/<app_id>/keys/` to get these details:



4. OK! Let's write some of the code and check whether we're good to work with Twitter's REST APIs. The following code makes a call to the Twitter timeline REST API and pulls the details of the topmost tweet on your timeline. Here we perform an HTTP `GET` operation on the `https://dev.twitter.com/rest/reference/get/statuses/home_timeline`REST API:

```
from twython import Twython

APP_KEY = ''
APP_SECRET = ''
OAUTH_TOKEN =''
OAUTH_TOKEN_SECRET = ''
twitter = Twython(APP_KEY, APP_SECRET,
                  OAUTH_TOKEN, OAUTH_TOKEN_SECRET)

tweet = twitter.get_home_timeline()[1]
print "Tweet text: ", tweet["text"]
print "Tweet created at: ", tweet["created_at"]
print "Tweeted by: ",
tweet["entities"]["user_mentions"][0]["name"]
print "Re Tweeted?: ", tweet["retweet_count"]
```

The output of the preceding code snippet is as follows:



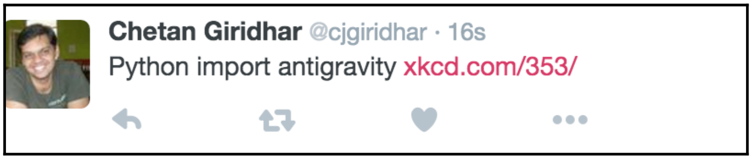
```
Tweet text:  A sci-fi vision of love from the near future:
Tweet created at:  Sun Aug 21 16:44:18 +0000 2016
Tweeted by:  Monica Byrne
Re Tweeted?:  15
```

That's cool! We get all the essential details from the post on our timeline. Looks like we're all set then with respect to the Twitter App and Keys.

5. Now let's try to tweet with the status's REST API by posting data to it. The REST API used here is `https://dev.twitter.com/rest/reference/post/statuses/update`. The following Python code will make a `POST` request on this REST resource and create a tweet on behalf of my account on Twitter:

```
from twython import Twython

APP_KEY = ''
APP_SECRET = ''
OAUTH_TOKEN =''
OAUTH_TOKEN_SECRET = ''
twitter = Twython(APP_KEY, APP_SECRET,
                  OAUTH_TOKEN, OAUTH_TOKEN_SECRET)
twitter.update_status(status='Python import antigravity
https://xkcd.com/353/')
```

After running the preceding code, I looked at Twitter, and voilà! I had a tweet under my name in an automated manner. Here's a screenshot of the tweet:



Chetan Giridhar @cjgiridhar · 16s

Python import antigravity xkcd.com/353/

So we have solved the first problem for Joy. The tweet can be posted on her behalf even when she is unavailable or can't log in to the Internet, and this can be done using the preceding Python code snippet. But she can't schedule her tweet as per the Australian time zone yet. Hmm, let's look at resolving the scheduling problem now.

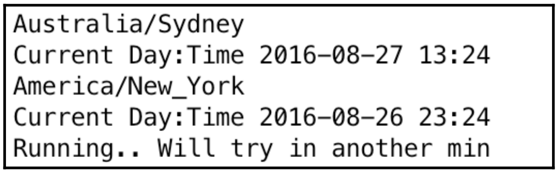
6. Before we look at how to schedule tweets, we will install a module that will be very useful for the next recipe. We will install `pytz`. It helps us work with time zones and will be helpful in solving Joy's problem:

```
(user)chetans-MacBookPro:ch07 Chetan$ pip install pytz
Collecting pytz
  Using cached pytz-2016.6.1-py2.py3-none-any.whl
Installing collected packages: pytz
Successfully installed pytz-2016.6.1
```

7. To solve the scheduling problem, we need two things. First, we need a configuration that can be used to decide the content, time, and time zone of the tweet. Second, we need a runner program that will use this configuration to post a tweet on Twitter. Now let's look at the following code, which does exactly what we need:

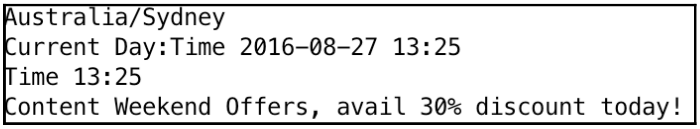
```
scheduled_tweets.py

from twython import Twython

APP_KEY = ''
APP_SECRET = ''
OAUTH_TOKEN =''
OAUTH_TOKEN_SECRET = ''
twitter = Twython(APP_KEY, APP_SECRET,
          OAUTH_TOKEN, OAUTH_TOKEN_SECRET)

from datetime import datetime
import pytz, time
from pytz import timezone
import tweet_config as config

while True:

    for msg in config.scheduled_messages:
        print msg["timezone"]
        tz = timezone(msg["timezone"])
        utc = pytz.utc
        utc_dt = datetime.utcnow().replace(tzinfo=utc)
        au_dt = utc_dt.astimezone(tz)
        sday = au_dt.strftime('%Y-%m-%d')
        stime = au_dt.strftime('%H:%M')
        print "Current Day:Time", sday, stime

        if sday == msg["day"]:
            if stime == msg["time"]:
                print "Time", stime
                print "Content", msg["content"]
                twitter.update_status(status='%s' %
                msg["content"] )


    print "Running.. Will try in another min"
    time.sleep(60)
    tweet_config.py
offers_sydney = {
    "content":"Weekend Offers, avail 30% discount today!",
    "day":"2016-08-27",
    "time":"13:25",
    "timezone":"Australia/Sydney"
}

post_newyork = {
    "content":"Introducing sun glasses at your favorite stores
    in NY!",
    "day":"2016-08-27",
    "time":"12:41",
    "timezone":"America/New_York"
}

scheduled_messages = [offers_sydney, post_newyork]
```

The output of the preceding code is as follows. Here's the first iteration:



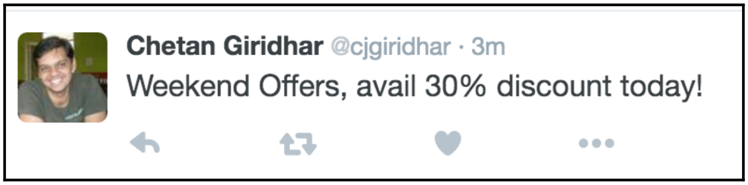
```
Australia/Sydney
Current Day:Time 2016-08-27 13:24
America/New_York
Current Day:Time 2016-08-26 23:24
Running.. Will try in another min
```

Here's the second iteration:



```
Australia/Sydney
Current Day:Time 2016-08-27 13:25
Time 13:25
Content Weekend Offers, avail 30% discount today!
```

Here's the actual tweet:



Chetan Giridhar @cjgiridhar · 3m

Weekend Offers, avail 30% discount today!

8. Cool! So we have what Joy needs. An automated tweet with the right content at the right time for the right audiences across the globe. Woohoo!

How it works…

In the preceding code snippets, we have two files. We have `tweet_config.py`, and it contains a configuration dictionary that is used to specify the content and schedule of the tweets. It also mentions the time zone in which the tweets need to be posted.

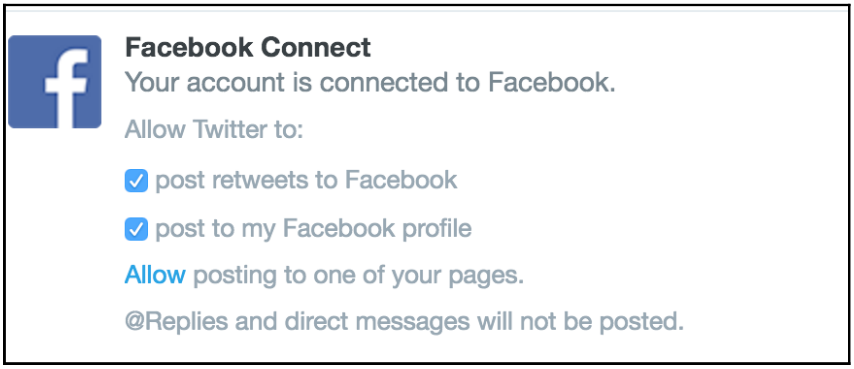
The second file `scheduled_tweets.py` is a runner program. It looks at the configuration every minute and checks whether there is any tweet scheduled for the given minute of the day.

When the runner program, `scheduled_tweets.py`, runs, it checks whether there are any messages scheduled. In iteration 1, the runner program doesn't find anything that it needs to work on; it just returns the current day and time of the time zone.

In iteration 2, it does find that there is a tweet scheduled in the Australian time zone, Sydney to be precise, at 13:25 hours on August 27; since the time matched, it posted a tweet. Of course, the example taken here is a very crude one. We might want to schedule Cron jobs instead of an endless while loop. But hey, this was an example to bring home the point about automatically scheduled tweets.

In this section, we automated the marketing process for Joy. Now she can not only tweet when she is is asleep, but also schedule tweets for different time zones and with different content. Now that's the power of automation.

“But hey, this is just one social media platform; what about Facebook?” you might ask. Yes, we have a trick up our sleeves. Twitter provides apps for connecting you to multiple services, including Facebook. So configure an app for your account so that every tweet that you post also gets posted on Facebook. This is how the configuration looks. It posts your original tweets and retweets them on your Facebook profile:



Facebook Connect
Your account is connected to Facebook.

Allow Twitter to:

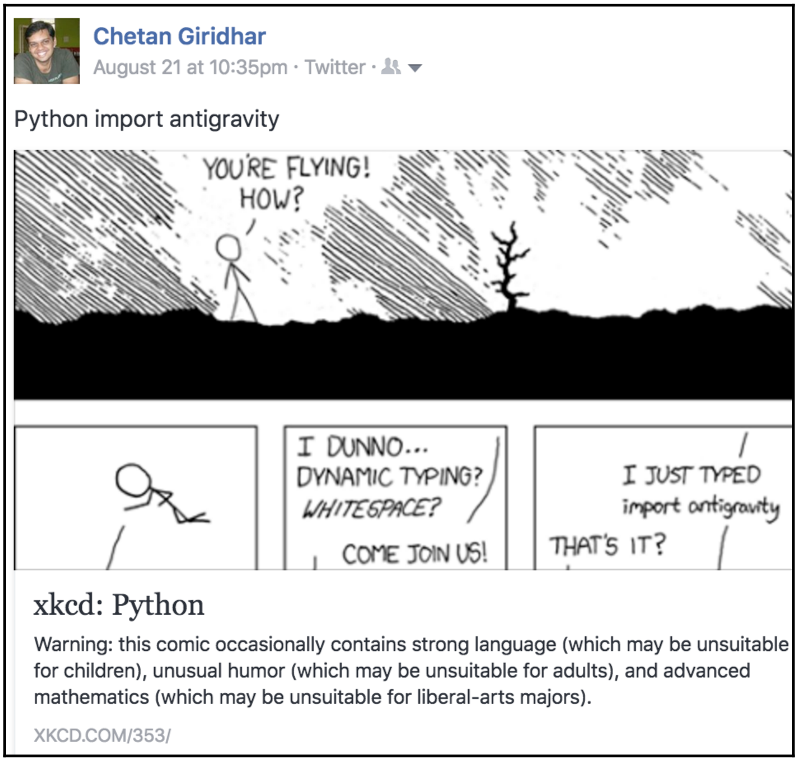
☑ post retweets to Facebook

☑ post to my Facebook profile

Allow posting to one of your pages.

@Replies and direct messages will not be posted.

Remember the first message that we posted about Python antigravity? Yes, it actually got posted on the Facebook wall as well. Look at the source next to the date and time of the tweet; yes it's Twitter! Perhaps, Twitter uses Facebook APIs to automate this:



Chetan Giridhar
August 21 at 10:35pm · Twitter ·

Python import antigravity

xkcd: Python

Warning: this comic occasionally contains strong language (which may be unsuitable for children), unusual humor (which may be unsuitable for adults), and advanced mathematics (which may be unsuitable for liberal-arts majors).

XKCD.COM/353/

An introduction to Webhooks

In the last section, we understood how to design and develop REST APIs and how to leverage REST APIs to our benefit by taking an example of Twitter/Facebook automation. Let's look at another amazing piece: Webhooks. A Webhook is an HTTP callback–an HTTP `POST` request to a user-defined URL (implemented as an HTTP API) when a favorable event occurs. Webhooks are often referred to as reverse APIs and are used for real-time communication or integration across services. But before we go deeper, let's understand a bit about polling.

You might have seen applications polling for long hours to check whether an event has occurred so that they can perform some follow-up action for the event. Take a real-world example. You go to a self-service restaurant and order your favorite pizza for lunch. The guy at the counter gives you an order number and tells you to watch the token machine for your order number so that you can collect your pizza. While everybody around is busy eating, you are hungry and tend to watch this token machine every 5 seconds hoping to see your order number flash on it. Now this is polling. You are polling the token machine. In the API world, the client would be polling the pizza place API to check the status of the order.

Wouldn't it be simple enough for the guy at the service counter to shout the order number when its ready? So after placing your order, you could get busy checking your official e-mails. When the service guy calls out your order number, you can collect your pizza from the delivery counter. This makes sure that your time is better utilized. Now this is a Webhook. When a favorable event occurs (your order is ready), you get a callback (the service guy shouts your order number) on your URL (in this case, your ears) that is literally listening and responding to callbacks. In the API world, you'd register your URL (the HTTP API) that gets called by the pizza place when your order is ready.

Webhooks can be used for three main purposes:

- Receiving data in real time
- Receiving data and pushing it on to another service
- Receiving the data and then processing and returning it

You can think of many different ways of using Webhooks in the preceding three scenarios.

If you think of polling and Webhooks, they both use APIs for integration needs. While polling is a client-driven integration technique, Webhooks are server-driven. Polling is very inefficient in the sense that the client keeps making server API calls to check the state of the resource (in our example, an order resource) with the help of a time stamp. Polling can happen every *x* minutes, *x* hours, or even *x* seconds to become more real time, but I think you get the inefficiencies associated with polling. On the other hand, Webhooks post data back to the callback URIs in the case of a favorable event. This is much more efficient than constant polling, but the flip side is you end up developing APIs on the client side, so your client tends to behave like a server itself.

Implementing Webhooks

With this knowledge, let's get started and implement Webhooks in this recipe.

Getting ready

For this recipe, we will use a famous Python web framework called **Django**. It allows you to use multiple plugins that get simply plugged in. Here, we will use the `django-rest-hooks` plugin developed by Zapier for implementing Webhooks.

So let's gets started and install the required packages. We install `Django==1.10` and `django-rest-hooks==1.3.1` using our favorite tool, Python `pip`:

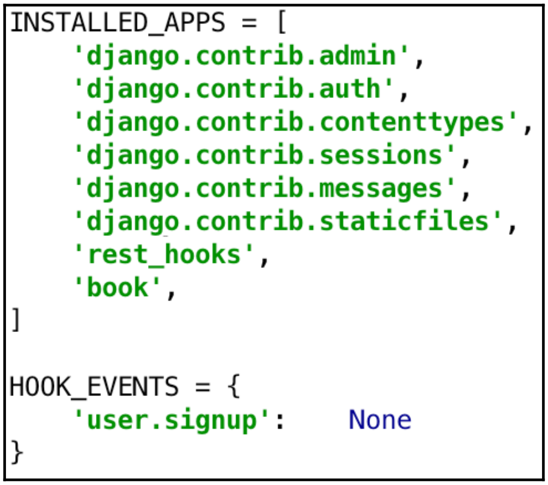
```
        (user)chetans-MacBookPro:ch07 Chetan$ pip install Django==1.10
        Collecting Django==1.10
          Downloading Django-1.10-py2.py3-none-any.whl (6.8MB)
            100% |████████████████████████████████|
6.8MB 71kB/s
        Installing collected packages: Django
        Successfully installed Django-1.10

         (user)chetans-MacBookPro:ch07 Chetan$ pip install django-rest-
hooks
        Collecting django-rest-hooks
          Downloading django-rest-hooks-1.3.1.tar.gz
        Requirement already satisfied (use --upgrade to upgrade):
Django>=1.4 in ./user/lib/python2.7/site-packages (from django-rest-hooks)
        Requirement already satisfied (use --upgrade to upgrade): requests
in ./user/lib/python2.7/site-packages (from django-rest-hooks)
        Building wheels for collected packages: django-rest-hooks
          Running setup.py bdist_wheel for django-rest-hooks
          Stored in directory:
/Users/chetan/Library/Caches/pip/wheels/96/93/12/3ec10693ee2b394a7d8594e893
9f7506d7231fab69c8e69550
        Successfully built django-rest-hooks
        Installing collected packages: django-rest-hooks
        Successfully installed django-rest-hooks-1.3.1
```

How to do it…

1. OK, let's create a Django app. We do this with the following commands:

```
python manage.py startproject bookstore
cd bookstore
python manage.py startapp book
```

2. Next, let's configure Django to use the `rest_hooks` module in the app. We do this by adding `rest_hooks` to `bookstore/settings.py` under `INSTALLED_APPS`. We also add our app book to this list. Add an event, namely `user.signup`, to `settings.py` by using the constant `HOOK_EVENTS`. We haven't tied the event `user.signup` to any action here, so it's none. This is how `settings.py` should look like:



```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_hooks',
    'book',
]

HOOK_EVENTS = {
    'user.signup':    None
}
```

3. Now, let's register this event to a callback URL. But before we go there, navigate to the root of your project and run this command to initialize your Django models:

```
(user)chetans-MacBookPro:bookstore Chetan$ python manage.py
migrate
Operations to perform:
  Apply all migrations: admin, contenttypes, rest_hooks, auth,
  sessions
Running migrations:
  Rendering model states... DONE
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
```

```
    Applying auth.0007_alter_validators_add_error_messages... OK
    Applying rest_hooks.0001_initial... OK
    Applying sessions.0001_initial... OK
```

4. Once the models are initialized, go to the database shell and run the following Python code snippets. They will create a user in the Django user table and register a Webhook for this user:

```
>>> from django.contrib.auth.models import User
>>> from rest_hooks.models import Hook
>>> usr=User.objects.create(username='chetan')
>>> hook = Hook(user=usr, event='user.signup',
target='http://localhost:8000/hook/')
>>> hook.save()
>>> hook
<Hook: user.signup => http://localhost:8000/hook/>
```

5. Now add a file called `urls.py` to the Book app and add this code:

```
from django.conf.urls import url
from . import views
urlpatterns = [
    url(r'event/$', views.event),
    url(r'hook/$', views.webhook),]
```

6. Add the following methods to `book/views.py` to create Django views:

```
from django.shortcuts import render
from django.views.decorators.csrf import csrf_exempt
from rest_hooks.signals import raw_hook_event
from django.contrib.auth.models import User
import datetime
from django.http.response import HttpResponse
# Create your views here.

@csrf_exempt
def webhook(request):
    print request.body
    return HttpResponse()

def event(request):
    user = User.objects.get(username='chetan')
    raw_hook_event.send(
        sender=None,
        event_name='user.signup',
        payload={
            'username': user.username,
            'email': user.email,
            'when': datetime.datetime.now().isoformat()
        },
        user=user # required: used to filter Hooks
    )
    return HttpResponse()
```

7. Also, include these URLs in the project under `bookstore.urls.py` like this:

```
from django.conf.urls import url, include
from django.contrib import admin


urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^', include('book.urls'))
]
```

8. Now run the Django server as follows:

```
(user)chetans-MacBookPro:bookstore Chetan$ python manage.py
runserver
Performing system checks...

System check identified no issues (0 silenced).
August 27, 2016 - 11:14:52
Django version 1.9, using settings 'bookstore.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

9. From your browser, go to `http://localhost:8000/event/` and look at your server logs. You will see that the registered Webhook got invoked, which means an HTTP `POST` request was sent to the target URL `http://localhost:8000/hook/` along with the payload, which contains all of the information we configured in the view. The server logs look like this:

```
[27/Aug/2016 10:53:29] "GET /event/ HTTP/1.1" 200 0

{"hook": {"target": "http://localhost:8000/hook/", "id": 1,
"event": "user.signup"}, "data": {"username": "chetan", "when":
"2016-08-27T10:53:29.301317", "email": ""}}

 [27/Aug/2016 10:53:29] "POST /hook/ HTTP/1.1" 200 0
```

Cool! Did you look at it? We invoked the /event URL, which in turn posted the required information to the target URL, which was registered for the event `user.signup` in our Webhook.

Similar to custom Webhooks, one could also develop RESTful Webhooks. RESTful Webhooks support subscription, notification, and publication actions through a RESTful interface. RESTful Webhooks must support four event types, namely `ACCESSED`, `CREATED`, `UPDATED`, and `DELETED`, which correspond to four HTTP verbs; notifications should be sent for the actions that get applied to the resource. For instance, when a resource gets created, an event gets generated; with this, the Webhook must be triggered and the target URL should be posted. In our example, we can define two more hook events, namely `book.added` and `book.deleted`, along with actions such as `book.Book.added` or `book.Book.deleted`. When we do a `book.save()` action on the model, the `book.added` event gets triggered, and if we have a hook defined for this event for the user, the HTTP `POST` request will be called on the target URL.

How it works…

In the preceding code snippets, we first defined an event in the `settings.py` file. The event was named `user.signup`. Since it was a custom event, it had no action defined.

We then created a new user `chetan` in the `auth_user` table using the default Django user model.

Later, we defined a Webhook for the the user `chetan`. This Webhook was configured for the event `user.signup`, and the target URL was set to `http://localhost:8000/hook/`.

We also defined two views in our Django app. The first view event was responsible for firing the Webhook for the corresponding user and event and for sending the payload information. The second view Webhook was defined for the target URL.

We then ran the Django development server and navigated to `http://localhost:8000/event/`, which posted the payload information to the target URL, namely `http://localhost:8000/hook/`. The target URL received all of the payload data, such as the username, e-mail, and the time when the signup happened.

There's more…

In this section, we looked at polling and Webhooks, which are other forms of integration on the Web that use APIs. We learned inefficiencies with polling and how Webhooks were much more useful. In the preceding recipe, we covered a custom event useful for user signup as I wanted it to explain the concept in a generic way. The coverage was brief, so I would like you to study more about RESTful Webhooks as they present a powerful use case for automation. With this understanding, let's see what problems Oliver has and how we can help him.

Automating lead management with Webhooks

Oliver is Joy's colleague and works in the marketing department. He is responsible for the user onboarding process. His primary responsibility includes:

- Sending a welcome e-mail to users who sign up on your website
- Adding a lead record for the new signee into the CRM

Previously, it was easy for him to perform these two tasks manually since the signups on the site were minimal. But with the growing popularity of the website, he has begun to see an upsurge in the number of signups on a daily basis. Without doubt, he sees this as a highly time-consuming activity that can be easily automated. Can you help Oliver?

If we carefully analyze the problem, Oliver's main issue is integration across services. The two services that he needs to integrate the signup are e-mail and CRM. He needs to track a signup event and take an action on this event. Webhooks are a perfect solution for this use case. Let's see how we can help Oliver automate his task.

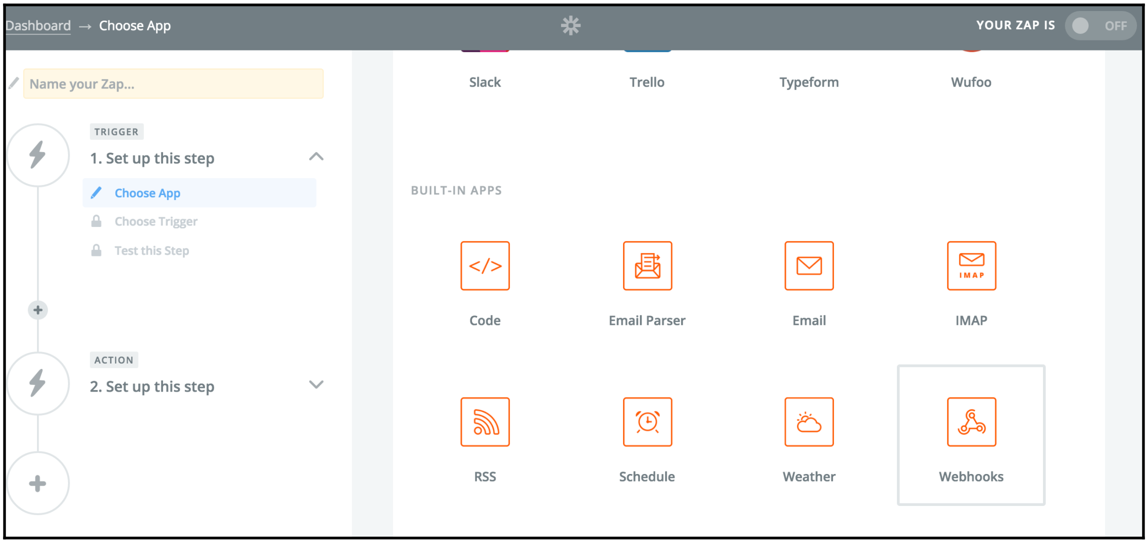
How to do it…

We will use the same Django project to solve this problem. We will also use an external service, Zapier, and see how it helps make things so much easier. Let's get started:

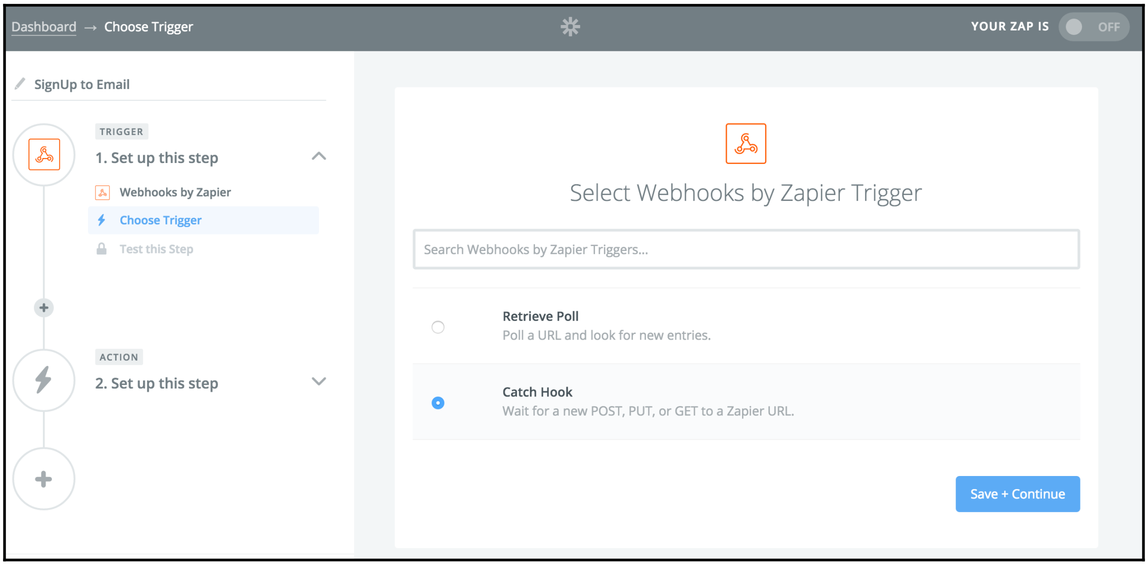
1. From the Terminal, go to the Django project's root directory and run the Python `manage.py` shell command to log in to the DB shell. Here, update our user `chetan` with the e-mail address. This can be achieved with the following set of commands:

```
>>> from django.contrib.auth.models import User
>>> from rest_hooks.models import Hook
>>> usr = User.objects.get(username='chetan')
>>> usr.email='chetan@email.io'
>>> usr.save()
```

2. Now create an account with the Zapier app by navigating to `https://zapier.com/`. Once you have the account created, click on **MAKE A ZAP!** to reach **Choose App** and click on **Webhooks** under the **BUILT-IN APPS** section:



3. Once you select a Webhook, you will get a screen to create a **TRIGGER** and **ACTION** on the left-hand pane. On the right-hand side, select the **Catch Hook** option. Click on **Save + Continue**. Refer to the following screenshot:



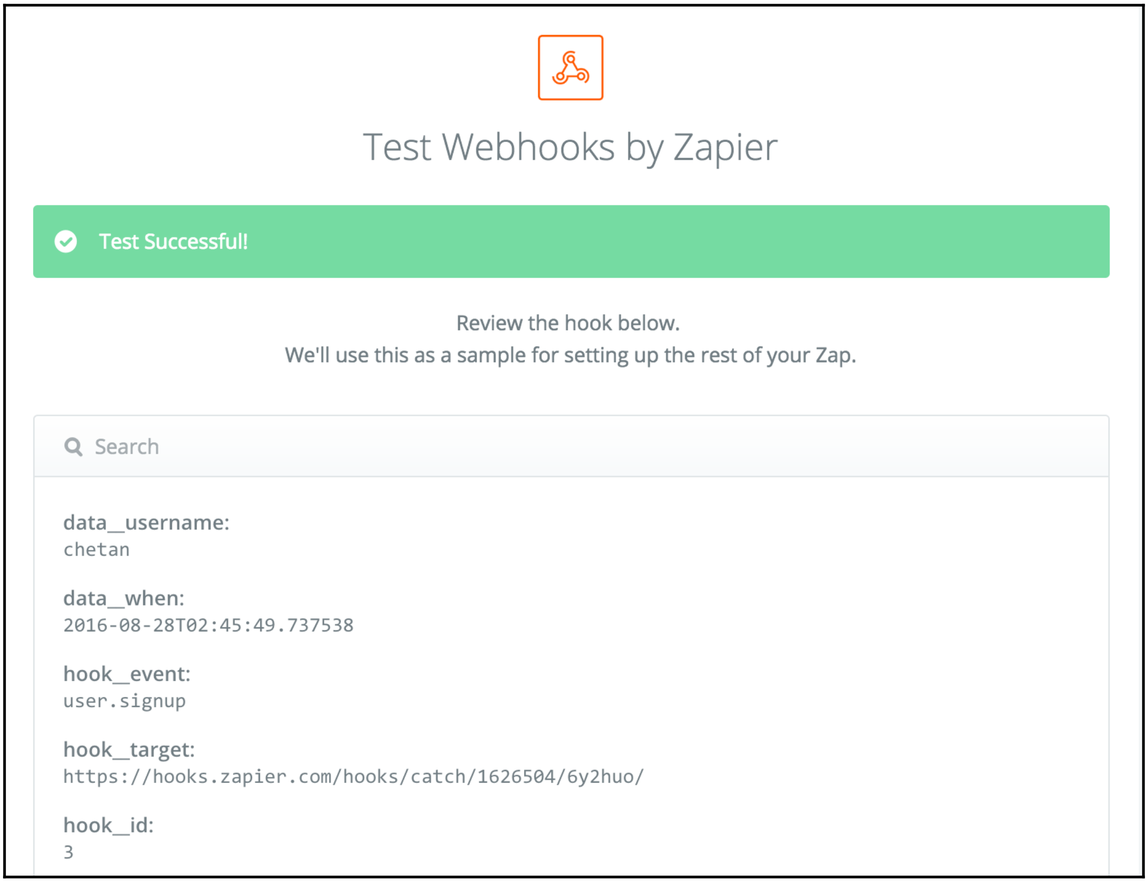
○ Retrieve Poll
Poll a URL and look for new entries.

● Catch Hook
Wait for a new POST, PUT, or GET to a Zapier URL.

4. Next, you will get the page to provide a JSON key that you want to select from the payload. This is an optional step and can be ignored. Click on **Continue** to reach the next step. Here, you'll get a custom Webhook URL. Copy this URL; it will act as a target URL.
5. Now go back to your Django project and navigate to the DB shell. Create a new hook with the same event `user.signup` and target the URL that you received from Zapier in the earlier step. The commands will look as follows:

```
>>> hook = Hook(user=usr, event='user.signup',
target=
'https://hooks.zapier.com/hooks/catch/<Id>/<Webhook_Id>/')
>>> hook.save()
>>> hook
<Hook: user.signup =>
https://hooks.zapier.com/hooks/catch/<Id>/<Webhook_Id>/>
```

6. Run the Django development server with the `runserver` command of Python `manage.py`. Once the server is running, go to `http://localhost:8000/event/`; this will make a callback request to the target URL obtained from Zapier. You can verify this by going to Zapier again and looking at **Test this Step** on the left pane, under the **Catch Hook** section:



Test Webhooks by Zapier

Test Successful!

Review the hook below.
We'll use this as a sample for setting up the rest of your Zap.

data__username:
chetan

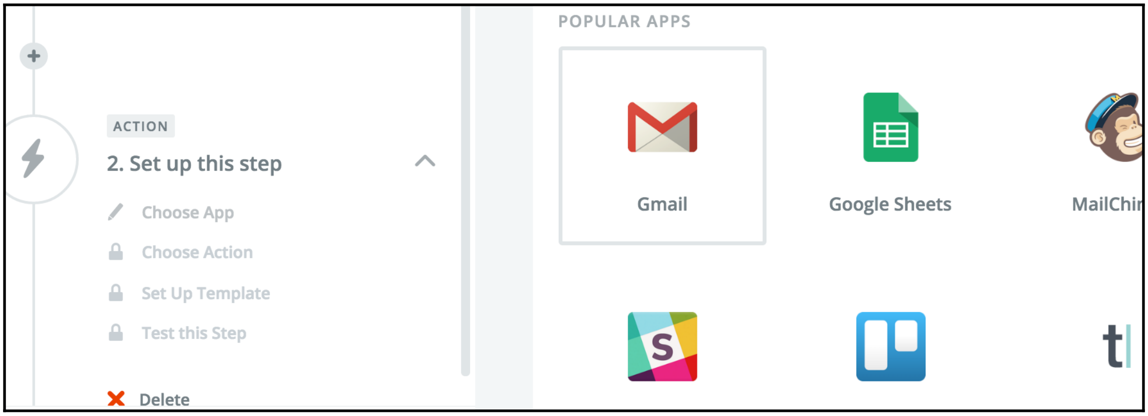
data__when:
2016-08-28T02:45:49.737538

hook__event:
user.signup

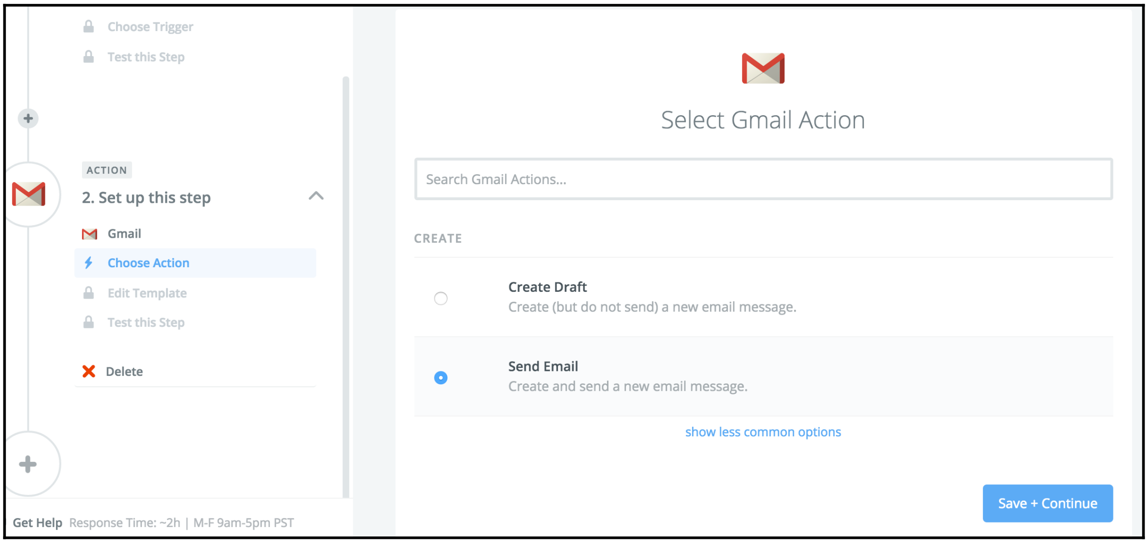
hook__target:
https://hooks.zapier.com/hooks/catch/1626504/6y2huo/

hook__id:
3

7. Cool! We now have the trigger set up. Let's set up the action. For this, go to your left-hand pane and click on **Set up this step** under **ACTION**. Choose Gmail from the list of apps that will show up on the right-hand side of the screen:



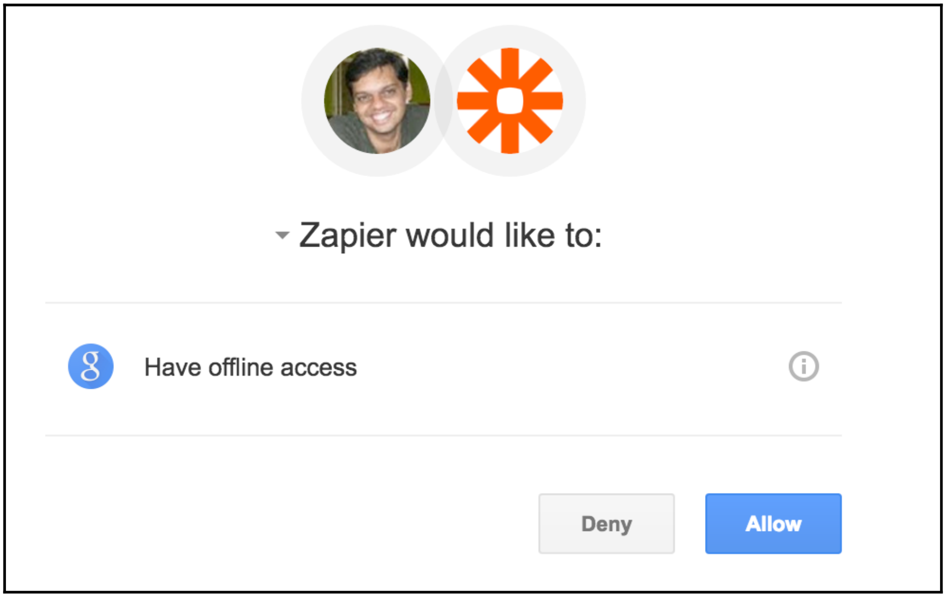
8. Once you click on **Gmail**, you get the next option to select an action, such as **Create Draft** or **Send Email**. Click on **Send Email** and activate your e-mail account by allowing Zapier to access it. The following screenshots will show you how to perform these steps:



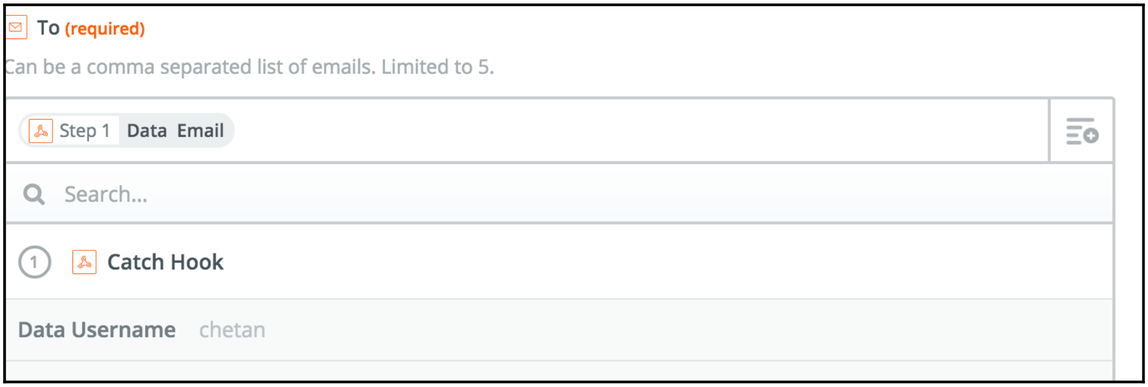
○ Create Draft
Create (but do not send) a new email message.

● Send Email
Create and send a new email message.

In the next screenshot, we allow Zapier to access the Gmail app:



9. OK! Now the only thing pending is to create the e-mail template. Our template contains the **To** e-mail address, subject, and body. Zapier gives you a nice option to configure your template. If you have already tested your trigger by posting data to the target URL, you will see a set of options on the extreme right of every field in the e-mail template. In the next two screenshots, I have **Data Email** in the **To** field, **Welcome Data Username!** in the **Subject** field, and **Your Signup made our day!** as the e-mail body.
10. The following screenshot shows a dropdown of all the available options from the payload that the target URL received in the **Test this Step** section of the trigger. I have just shown the username. Look how the field name **To** of the template can chose **Data Username** from the payload:



To (required)

Can be a comma separated list of emails. Limited to 5.

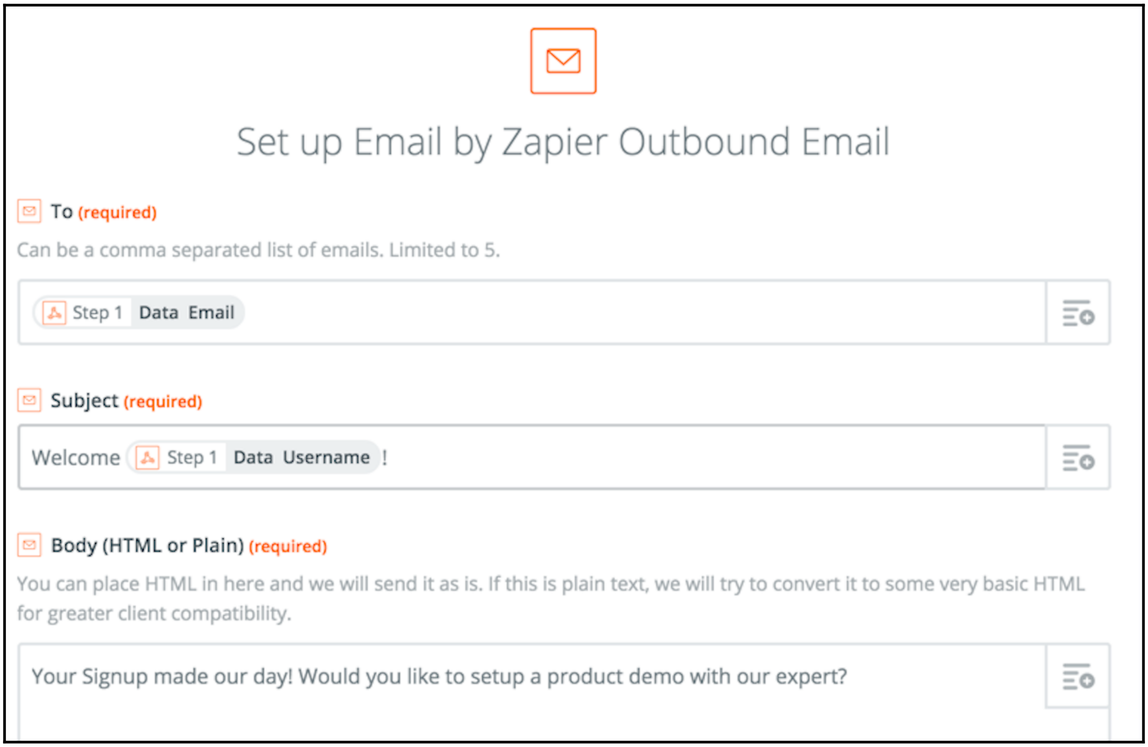
Step 1 Data Email

Search...

1 Catch Hook

Data Username chetan

The e-mail template with all the necessary fields configured can be seen in the following screenshot. We have configured the **To**, subject, and body parts of the e-mail in Zapier:



Set up Email by Zapier Outbound Email

To (required)

Can be a comma separated list of emails. Limited to 5.

Step 1 Data Email

Subject (required)

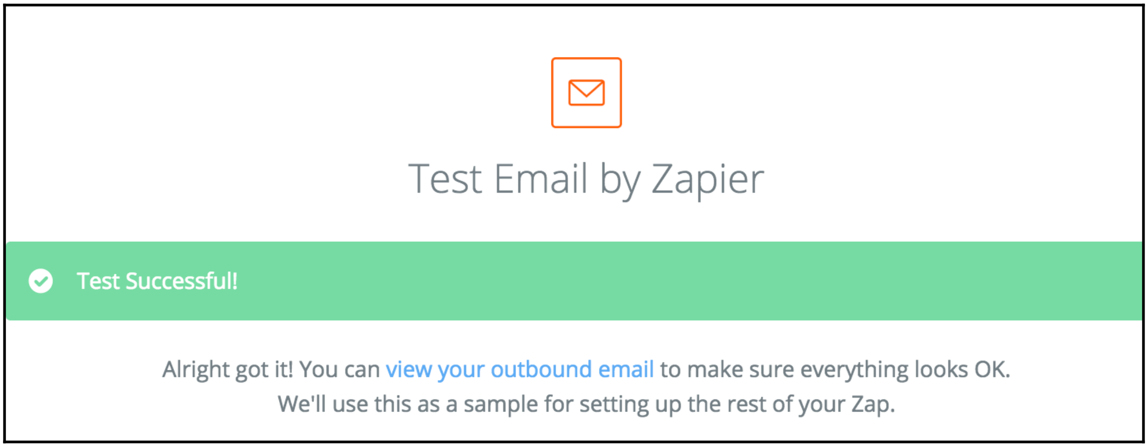
Welcome Step 1 Data Username !

Body (HTML or Plain) (required)

You can place HTML in here and we will send it as is. If this is plain text, we will try to convert it to some very basic HTML for greater client compatibility.

Your Signup made our day! Would you like to setup a product demo with our expert?

11. That's it! Click on **Continue** at the bottom of this screen; Zapier will test your action and you're done. The following screenshot shows the confirmation of success!

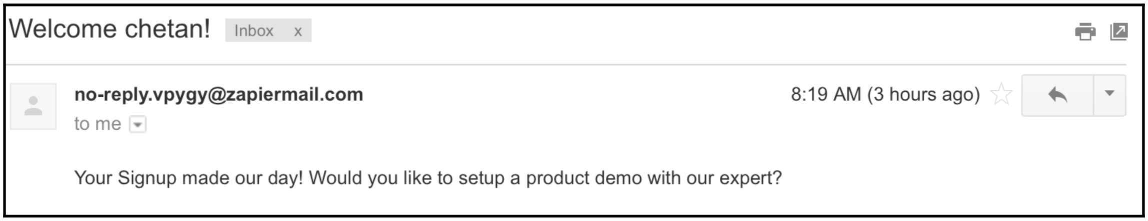


Test Email by Zapier

Test Successful!

Alright got it! You can view your outbound email to make sure everything looks OK.
We'll use this as a sample for setting up the rest of your Zap.

12. Now if you check your e-mail, you should have received a test e-mail from Zapier, which was used for testing the Zapier action. The contents of the e-mail are the way we wanted them to be. Pretty cool! So now when anyone signs up on Oliver's product website, the view will `POST` the signee's information as the payload to Zapier's Webhook (the target URL), and Zapier will automate the e-mail part.



Welcome chetan! Inbox x

no-reply.vpygy@zapiermail.com
to me

8:19 AM (3 hours ago)

Your Signup made our day! Would you like to setup a product demo with our expert?

How it works…

Zapier provides us with a feature to create custom Webhooks. It has integration with almost all the apps under the sun, such as Gmail, Trello, Slack, and so on. We just created a Webhook as a trigger and followed it up with an action from Gmail.

Whenever a user signs up (new user creation), the Django app will `POST` the user's data as the payload to the Zapier target URL, which we got when we created a trigger in Zapier.

Once Zapier receives the payload data to the target URL, it checks the action and finds that it has to send an e-mail to a Gmail account. Zapier is also intelligent enough to get the data from the payload and send the e-mail to the user's e-mail address; it also allows configuration of the e-mail's subject and body.

Fantastic! Oliver is happy! And what about step 2? Well, it's another Zapier trigger with either the Salesforce or Pipedrive CRM to create a lead record in the CRM. A walk in the park!

In this section, we looked at automating user onboarding with a user signup event. We took an example of Zapier as it's the best possible way to automate apps. If we had not done this, we would have ended up understanding the APIs provided by all these apps and writing code for all of them ourselves, an activity which may not be the core of your product or service.

Well that's it, folks! Hope you enjoyed this piece of automation and I'm sure you will definitely implement this in your organization.

8
Talking to Bots

Wow, bots?! Really? Will I learn to build bots for fun or my business use case? Yes, of course this chapter takes you to the brand new world of bots with Python.

In this chapter, we will cover the following recipes:

- Building a moody Telegram bot
- Different types of bots: stateless, stateful, and smart
- A smart bot with artificial intelligence
- Automating business processes with bots

Introduction

The last couple of decades have been an age of digital transformation and automation. Most businesses today prefer an online sales model rather than the traditional brick-and-mortar way of selling products.

Websites have not only helped companies increase their reach, but have also made it cheaper (no fixed costs such as rentals) for them to sell their products. A responsive **graphical user interface** (**GUI**), combined with the power of real-time technologies, has made the process of selling easier; now executives can just chat with potential customers and guide them to buy products, increasing conversions.

With advancements in **artificial intelligence** (**AI**) and language processing techniques, businesses are slowly but steadily adopting conversational interfaces to automate their processes. A conversational user interface refers to an interface that has free-form text for natural languages. With conversational interfaces and natural language processing techniques, businesses feel that a machine can respond to certain customer queries by analyzing the context. These machines, in today's world, are referred to as **chatbots**.

In this chapter, you will learn about the different types of bots, look at how to develop simple chatbots, and learn about how bots can be used to automate business processes. Also, note that when we refer to bots in this chapter, we're talking about chatbots or text-based bots.

What are bots?

OK, let's take a simple example. Say you want to order a pizza from **Pizza Hut** for an evening get-together with friends this coming weekend. Usually, you'd go to the Pizza Hut website, spend time looking for a certain type of pizza or that particular topping you like, and place an order. More often than not, you already know what you want to order; then the question really is, why take the pain to look for it on the Pizza Hut website?

Worry no more! Just log in to **Facebook** and use the Facebook Messenger chatbot to buy what you need from Pizza Hut. Not just this, the chatbot will also keep you posted on the latest offers and updates from Pizza Hut. So a chatbot can give you the same experience of visiting a website from your favorite social networking platform. Look at `http://blog.pizzahut.com/press-center/pizza-hut-announces-new-social-ordering-platform/` for the announcement `Pizza Hut` made regarding collaborating with Facebook Messenger.

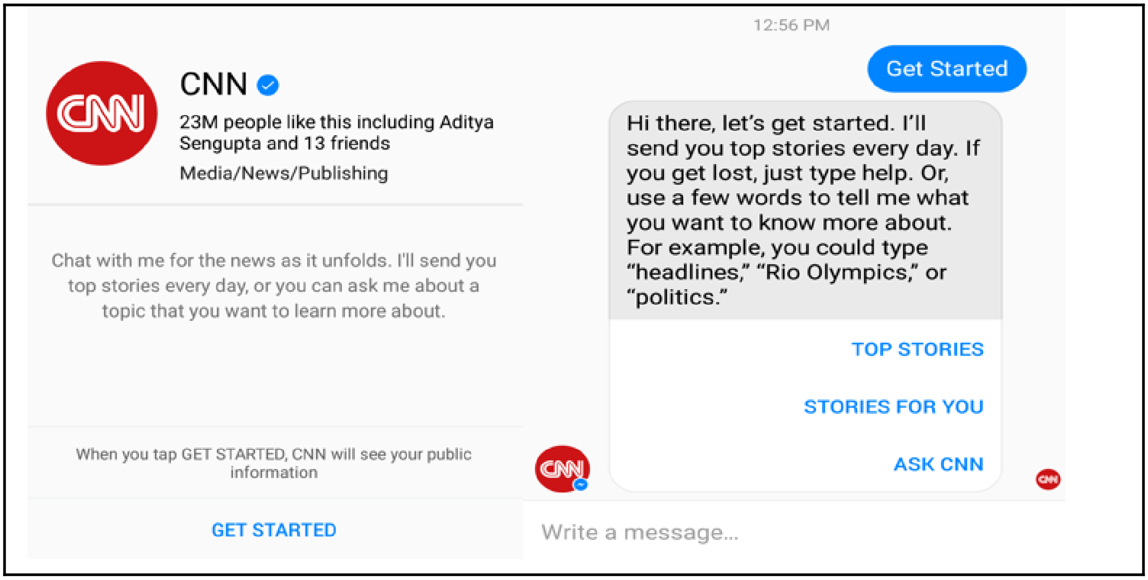
You may say, “Yes, we understand the use case, but what exactly is a chatbot?”

A chatbot is a service powered by rules and AI that you, as a customer, interact with via a chat (text) interface. Bots carry out semi-intelligent or mundane tasks and run them as software applications. Chatbots can provide you with multiple services and can run on social platforms such as **Facebook**, **Telegram**, **Slack**, and many more. Chatbots are still in active research and are an emerging computer science field.

How do bots work?

Based on what we have discussed so far, you might be thinking, “How these bots function? How do they understand human words or sentiments? How do they understand the context?” So here's the answer. There are typically two types of chatbots:

- **Bots that work on a rule engine**: This type of bot understands certain words or commands (so to speak) and has very limited behavior. It is pretty straightforward: if *x* is the input, then *y* should be the output. They're very useful in cases where there are fixed sets of questions or when questions act as queries. For example, the CNN chatbot helps you get the top stories for that moment, and furthermore, you have the luxury of asking the bot about the top stories on certain topics, such as **politics** or **business**. (Great! Then why should I even go to the CNN website?) Look at some of the screenshots that I took from my Facebook Messenger app regarding my interaction with the CNN chatbot. The first screen asks you to click on **GET STARTED**, and when you do this, the bot takes you to the next screen where it gives you an option to look at the top stories:



CNN

23M people like this including Aditya Sengupta and 13 friends

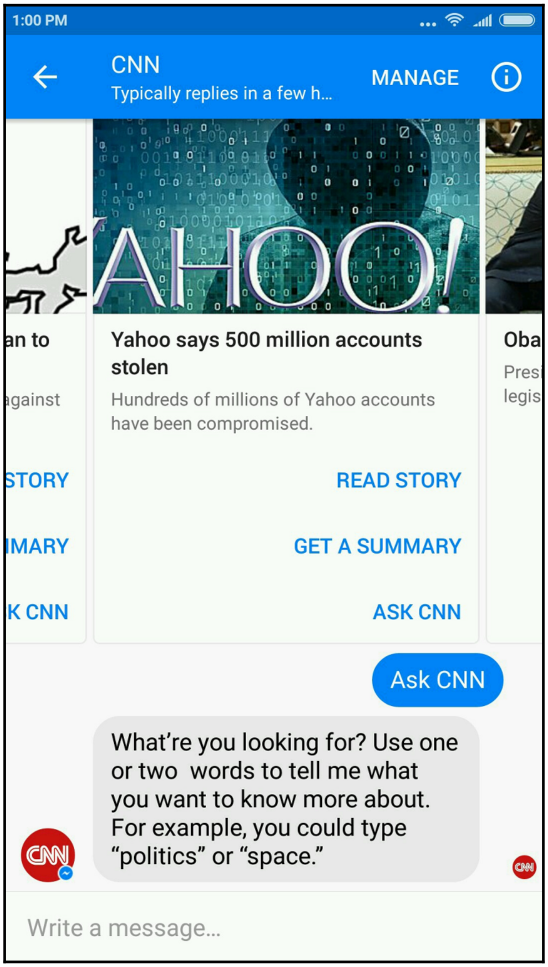
Media/News/Publishing

Chat with me for the news as it unfolds. I'll send you top stories every day, or you can ask me about a topic that you want to learn more about.

When you tap GET STARTED, CNN will see your public information

Hi there, let’s get started. I’ll send you top stories every day. If you get lost, just type help. Or, use a few words to tell me what you want to know more about. For example, you could type “headlines,” “Rio Olympics,” or “politics.”

When you click on **TOP STORIES**, it shows you the **Yahoo!** story and asks you whether you're interested in certain topics, politics for example:



Yahoo says 500 million accounts stolen

Hundreds of millions of Yahoo accounts have been compromised.

Ask CNN

What’re you looking for? Use one or two words to tell me what you want to know more about. For example, you could type “politics” or “space.”

- **A smart bot that works on machine learning**: Smart bots use AI and sentiment analysis to understand the context of a conversation and respond to language semantics. They are hence applicable to sophisticated use cases, such as purchasing products or answering customer support queries. What's more, these bots can learn from past interactions. Amazing, isn't it?

Sentiment analysis is also referred to as opinion mining and aims at identifying and extracting subjective information from the available text and determining the emotion of the writer, taking care of the contextual properties of the text.

Why bots now?

You may ask, “The world has been talking about machine learning for a while now and the chat feature has been around a long time now, so why are bots becoming so relevant now?” That's because of the following reasons:

- **Usage patterns**: Companies have figured out that users tend to spend more time on chats than on social media platforms or websites. Hence, businesses can engage with users in better ways via a chat platform.
- **Cost-effective**: No need for humans–sounds like no cost at all! Businesses are taking advantage of bots to automate processes, such as customer service, without human resource investments.
- **Scale**: It's easy to reach out to millions of users through Facebook or Telegram, which act as distribution channels for bots. This way, businesses can target as many potential customers as possible without looking at the human costs involved.
- **Efficient technology**: Growth in AI or **Natural Language Processing** (**NLP**) has made it even easier to plug algorithms into these bots. Algorithms can, or will, mature over time, and they will serve customers even better.

OK, great! Now that we understand bots and their utility much better, let's get our hands dirty and develop our own bot.

Building a moody Telegram bot

Before we start developing a bot, we should be clear about our objectives: what is my bot going to do? We take a simple example of creating a bot that would respond with an emoji based on the user's mood. It's a moody bot for the simple reason that it represents the mood of the user. Sounds like an interesting use case? Let's go for it!

In this recipe, we shall use the python-telegram-bot (https://github.com/python-telegram-bot/) library to develop a Telegram bot. So, let's get started by first installing the python-telegram-bot module with our favorite utility, namely python pip:

```
(bots)chetans-MacBookPro:ch09 Chetan$ pip install python-telegram-bot --
upgrade

Collecting python-telegram-bot
  Downloading python_telegram_bot-5.1.0-py2.py3-none-any.whl (134kB)
    100% |█████████████████████████████████████████████████████| 135kB
681kB/s
Collecting certifi (from python-telegram-bot)
  Downloading certifi-2016.8.31-py2.py3-none-any.whl (379kB)
    100% |█████████████████████████████████████████████████████| 380kB
612kB/s
Collecting future>=0.15.2 (from python-telegram-bot)
  Downloading future-0.15.2.tar.gz (1.6MB)
    100% |█████████████████████████████████████████████████████| 1.6MB
251kB/s
Collecting urllib3>=1.10 (from python-telegram-bot)
  Downloading urllib3-1.17-py2.py3-none-any.whl (101kB)
    100% |█████████████████████████████████████████████████████| 102kB
1.2MB/s
Building wheels for collected packages: future
  Running setup.py bdist_wheel for future
  Stored in directory:
/Users/chetan/Library/Caches/pip/wheels/11/c5/d2/ad287de27d0f0d646f119dcffb
921f4e63df128f28ab0a1bda
Successfully built future
Installing collected packages: certifi, future, urllib3, python-telegram-
bot
Successfully installed certifi-2016.8.31 future-0.15.2 python-telegram-
bot-5.1.0 urllib3-1.17
```

We also install the emoji (https://github.com/carpedm20/emoji) library to work with emoji icons so that we can return appropriate expressions to the user based on their mood:

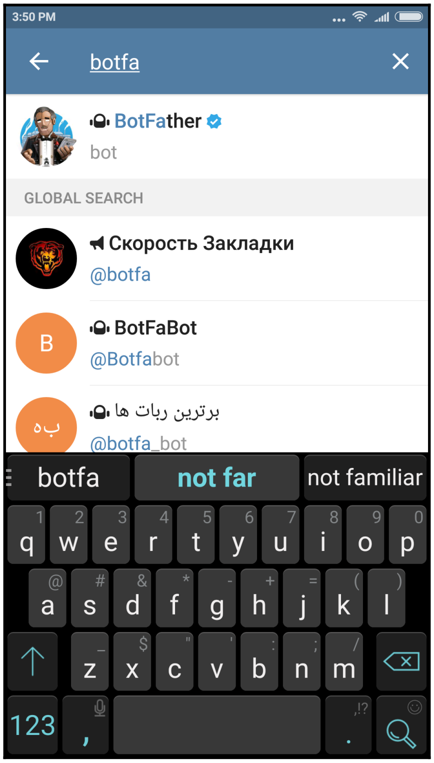
```
(bots)chetans-MacBookPro:ch09 Chetan$ pip install emoji --upgrade

Collecting emoji
  Downloading emoji-0.3.9.tar.gz
Building wheels for collected packages: emoji
  Running setup.py bdist_wheel for emoji
  Stored in directory:
/Users/chetan/Library/Caches/pip/wheels/94/fc/67/441fb0ca2ed262d6db44d9ac2d
fc953e421f57730004dff44d
Successfully built emoji
```

```
Installing collected packages: emoji
Successfully installed emoji-0.3.9
```

Have you installed the modules already? Cool! Let's move ahead.

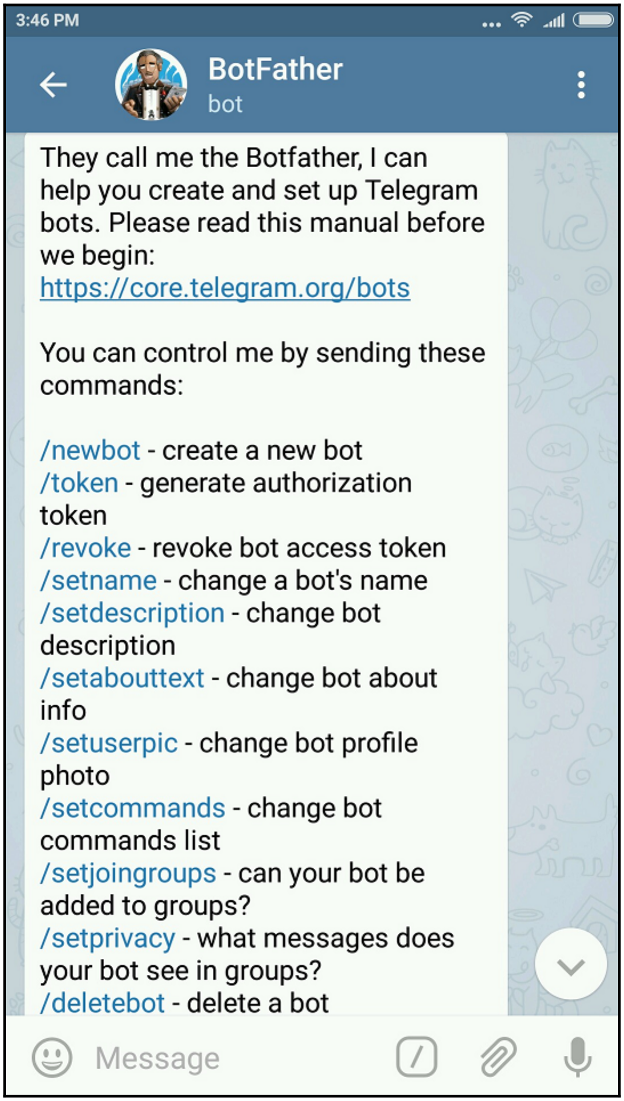
How to do it...

1. To develop a bot of your own, first download the Telegram app on your mobile. Sign up for an account and verify your number. Assuming you have done this, congrats! You are one step closer to creating a Telegram bot.
2. Now, the next thing you need to do to register your bot is to contact another bot called **BotFather**. On your Telegram app, search for **BotFather** and click on it to start a conversation with it (or him?). This is how it will look:



botfa

3. Once you start the conversation with **BotFather**, follow the steps and use commands such as `/newbot` or `/enable` to configure your bot. Follow the steps carefully and you'll create a new bot. The following screenshot will guide you through the process of creating a new bot:

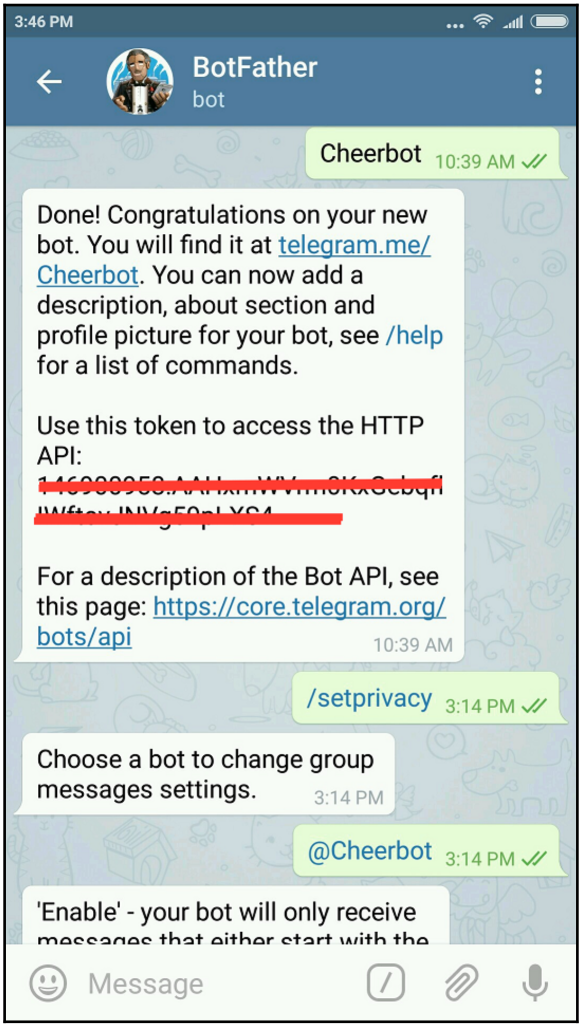


They call me the Botfather, I can help you create and set up Telegram bots. Please read this manual before we begin:
https://core.telegram.org/bots

You can control me by sending these commands:

/newbot - create a new bot
/token - generate authorization token
/revoke - revoke bot access token
/setname - change a bot's name
/setdescription - change bot description
/setabouttext - change bot about info
/setuserpic - change bot profile photo
/setcommands - change bot commands list
/setjoingroups - can your bot be added to groups?
/setprivacy - what messages does your bot see in groups?
/deletebot - delete a bot

4. When you create a new bot, you'll get a token specific to your bot. Keep this handy and safe with you; do not share it with anybody. The following screenshot shows how **BotFather** works and how the token looks:



Cheerbot 10:39 AM

Done! Congratulations on your new bot. You will find it at telegram.me/Cheerbot. You can now add a description, about section and profile picture for your bot, see /help for a list of commands.

Use this token to access the HTTP API:
[redacted]

For a description of the Bot API, see this page: https://core.telegram.org/bots/api 10:39 AM

/setprivacy 3:14 PM

Choose a bot to change group messages settings. 3:14 PM

@Cheerbot 3:14 PM

'Enable' - your bot will only receive
messages that either start with the

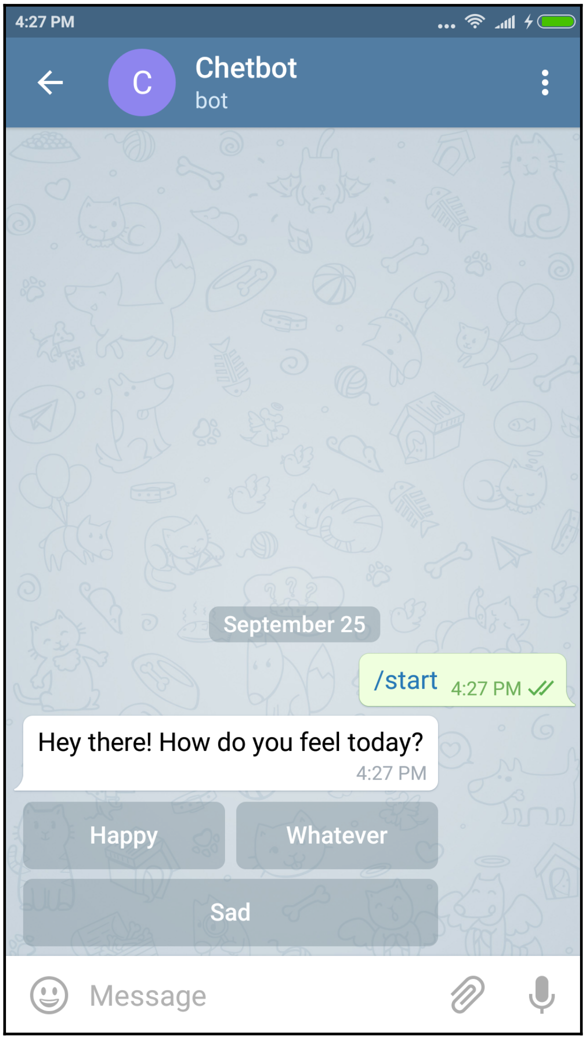
5. Nice! So you have created your own bot. But the bot is not functional and doesn't do anything spectacular yet. Let's make it do some cool stuff like we planned at the beginning of the recipe. Create a file called `bot.py` and copy the following code to it. Also, make sure you change the **token** to your bot's token ID:

```
import logging
from telegram import InlineKeyboardButton,
  InlineKeyboardMarkup
from telegram.ext import Updater,
  CommandHandler, CallbackQueryHandler
import emoji

logging.basicConfig(format='%(asctime)s
  - %(name)s - %(levelname)s - %(message)s',
level=logging.INFO)


def start(bot, update):
    keyboard = [
        [InlineKeyboardButton("Happy", callback_data='1'),
        InlineKeyboardButton("Whatever", callback_data='2')],
        [InlineKeyboardButton("Sad", callback_data='3')]]

    reply_markup = InlineKeyboardMarkup(keyboard)

    update.message.reply_text('Hey there!
      How do you feel today?', reply_markup=reply_markup)


def button(bot, update):
    query = update.callback_query
if query.data == "1":
        em = emoji.emojize(':smile:', use_aliases=True)
        bot.editMessageText(text="Oh wow! %s " % em,
chat_id=query.message.chat_id,
message_id=query.message.message_id)

if query.data == "2":
        em = emoji.emojize(':expressionless:', use_aliases=True)
        bot.editMessageText(text="Does it matter? %s " % em,
chat_id=query.message.chat_id,
message_id=query.message.message_id)


if query.data == "3":
    em = emoji.emojize(':disappointed:', use_aliases=True)
    bot.editMessageText(text="Oh man! %s " % em,
chat_id=query.message.chat_id,
message_id=query.message.message_id)

def help(bot, update):
    update.message.reply_text("Use /start to test this bot.")


def error(bot, update, error):
    logging.warning('Update "%s" caused error "%s"' % (update,
    error))


# Create the Updater and pass it your bot's token.
updater = Updater('Token')

updater.dispatcher.add_handler(
  CommandHandler('start', start))
updater.dispatcher.add_handler(
  CallbackQueryHandler(button))
updater.dispatcher.add_handler(
  CommandHandler('help', help))
updater.dispatcher.add_error_handler(error)

# Start the Bot
updater.start_polling()

# Run the bot until the user presses Ctrl-C
  or the process receives SIGINT,
# SIGTERM or SIGABRT
updater.idle()
```

6. OK, cool! We have now added the required functionality to our bot and we expect it to run well. But how do we test it? First, run the Python file with the following command:

```
python bot.py
```

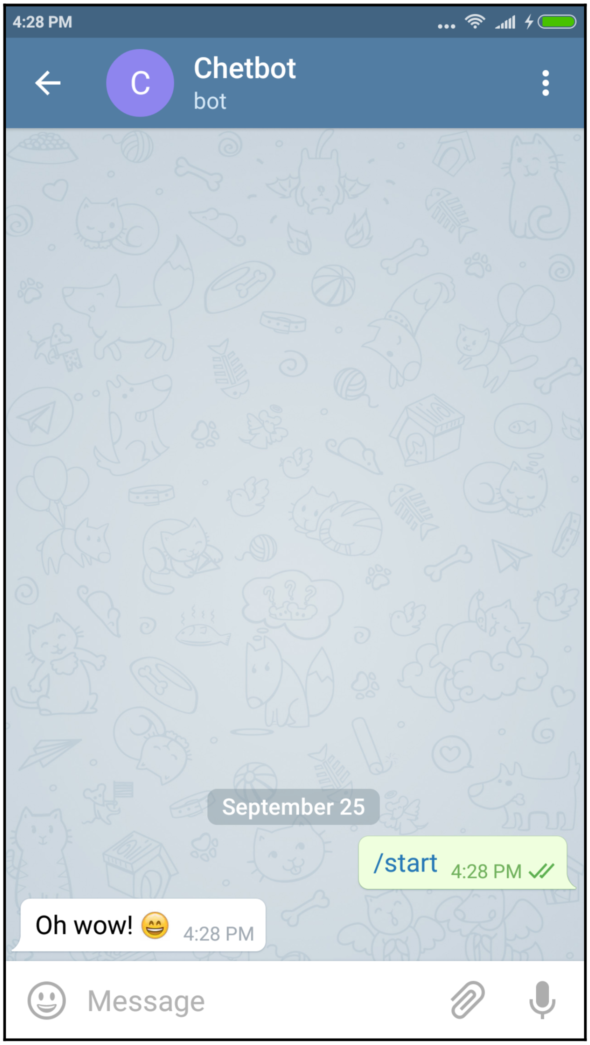
7. We then search for our bot and start a conversation with it. In my case, the bot is called **Chetbot**, and I started a conversation with it using the standard `/start` command:



/start 4:27 PM

Hey there! How do you feel today? 4:27 PM

8. In the preceding screenshot, when I started the conversation with my bot, it asked me about my mood for the day and gave me three options. The three options were **Happy**, **Whatever**, and **Sad**.
9. That's neat! But what happens when I click on one of these options? Voila! It returns me my mood for the day with an emoticon. Awesome!

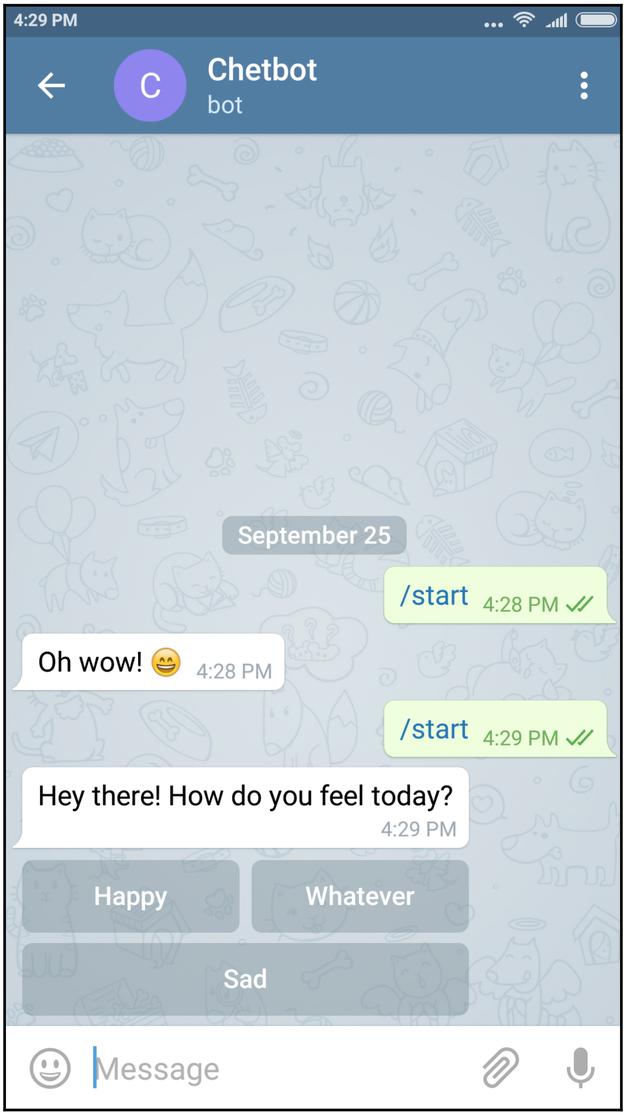


September 25

/start 4:28 PM

Oh wow! 😄 4:28 PM

10. Observe that if I have to start the conversation again, I would need to reenter the `/start` command to converse with the bot. In the following screenshot, the bot recognizes the start command and asks me my mood again. Neat, huh?



4:29 PM

Chetbot
bot

September 25

/start 4:28 PM

Oh wow! 😄 4:28 PM

/start 4:29 PM

Hey there! How do you feel today? 4:29 PM

How it works…

The python-telegram-bot module works on standard event-driven philosophy. A bot can be considered as a single-threaded event loop, which keeps polling for events. An event loop is also registered with command handlers, also referred to as dispatchers. As soon as an event gets triggered, a callback handles the event and returns the desired response to the user.

In the preceding code snippet, we registered two command handlers: start() and help(). The start() method gets called when a user starts the conversation with the bot (the /start command) or asks for help (the /help command).

We have also added a callback query handler with button() as the callback method; this gets invoked when the user responds to the bot's options.

So, initially, the bot is running, waiting for an input. When a user says /start, the request is dispatched to the start() method, which in turn asks the user **Hey there! How do you feel today?** and presents an inline keyboard with three options: **Happy**, **Whatever**, or **Sad**.

When the user selects either of the options, an event gets generated, which is handled by the callback method button(). The callback has preloaded data that acts according to the option that is chosen. Based on the choice made by the user, the bot sends back the right emotion to the user. Emoticons are retuned to the user with the help of the emoji library that has all the expressions implemented.

There's more…

Cool! Have you created your own bot already? Can you think of other simple examples where Telegram bots will be useful? There are many Python modules using which you can develop bots with Telegram, such as telepot (https://github.com/nickoala/telepot) or twx.botapi (https://github.com/datamachine/twx.botapi); both are good. You can use either one of them to get your bot up and running. Why not try them and see what they have to offer?

Different types of bots

Taking confidence from having built a bot ourselves, let's take a step forward and look at how bots can be classified.

The bot we developed in the last recipe can be tagged unintelligent. By unintelligent, I mean it questioned the user, and based on the option, responded with an emoji. But when the user says `/start` again, the bot asked the same question. Not helpful, is it?

How about a scenario where the bot would remember your previous choice and try to motivate you with some nice articles or places you can go to within the city? Just to change your mood? How about actually increasing the happiness quotient?

To put the preceding discussion in perspective, bots can be classified into three different categories based on the implementation:

- **Stateless bots**: These can also be referred to as don't-remember-anything bots. They don't persist information; that is, for them, every interaction is a new session, and they treat every question in isolation. For example, a news bot can keep returning updates on the latest stories or return the top stories in the *politics* category all the time; however, if it doesn't remember the state of the conversation, it will be deemed stateless and will not be considered useful. Most bots built today come under this category, because of which the value offered by them is very limited.
- **Stateful bots**: We discussed the news bot in the preceding point. What if the news bot remembers the news category of the user's interest and accordingly recommends more stories from the past, which the user may find interesting to read? Yeah, now we're talking business. This way, we keep the user engaged with the bot for a longer period of time.

  Such bots keep track of the user's identity and also persist information of the current and previous sessions. For example, these bots may store the news categories searched today and from the past, and can then recommend users news feeds that match the searched categories.

  Such bots are useful, but they are not smart; they don't understand context and language semantics.

- **Smart bots**: Smart bots have many batteries plugged in. They use machine learning, understand language semantics, and can build predictive algorithms based on the data they have.

  Let's take the famous example of diaper and beer. It is said that if you analyze the pattern of purchase, there is a high correlation between the purchase of beer and that of the diaper, which means a person who buys a diaper more or less definitely buys beer. Smart bots can persist data and come up with such patterns that will result in meaningful insights into conversations. Let's take another example of language semantics. Think about the phrase "*filthy awesome*"; now filthy, means dirty and awesome is a very positive word. Smart bots will understand these phrases and can comprehend the users' context much better.

Based on the preceding categorization, it's up to us to decide what kind of bot we need to develop for a particular use case. Smart bots are often needed in cases where interaction is far more humane and involved, as in the case of customer support, but imagine the productivity gains a business can harness by using smart bots.

A smart bot with artificial intelligence

With the knowledge of different types of bots in the previous section, let's try to write a bot that uses artificial intelligence and sentiment analysis in Python. But before that, let's understand both these fields in brief.

Artificial intelligence (**AI**) is an area of computer science that emphasizes on the creation of machines that can react like humans. Essentially, artificial intelligence relates to machines that perceive its context and take an action relevant to the content in order to maximize the chances of success. For instance, a machine can take decisions based on certain rules and a certain context to maximize the results of the decision.

Sentiment analysis, on the other hand, is about identifying and categorizing a piece of text to determine whether the opinion or attitude of the person involved is positive, neutral, or negative to a product or event. It refers to the use of natural language processing algorithms to perform text analysis and extract subjective information, or the sentiment, of the content.

I think, by now, you must have already started thinking of how AI and sentiment analysis can be used in our bots for various needs. In this recipe, let's build a smart bot with these technologies.

Smart bots can be built on multiple technologies, such as predictive intelligence, AI, NLP, and more; however, it's completely up to you to decide which technology you need to use to meet your objectives. Also, bots don't need to be on the Web or an app; they can be simple CLI-based bots. A web UI, CLI, or a mobile app can be used as a distributor for a bot, but it is not a necessity to build a bot.

Getting ready

To include AI in our bot, we will use a well-known Python module called `aiml`. **AIML** stands for **Artificial Intelligence Markup Language**, but it's essentially an XML file. AIML is a form of XML that defines the rules for matching patterns and determining responses. So, let's get started by installing the `aiml` module:

```
chetans-MacBookPro:ch09 Chetan$ source bots/bin/activate
(bots)chetans-MacBookPro:ch09 Chetan$
(bots)chetans-MacBookPro:ch09 Chetan$ pip install aiml

Collecting aiml
Installing collected packages: aiml
Successfully installed aiml-0.8.6
```

How to do it…

1. As step 1, we start by creating the AIML file. Go to your favorite editor and create an AIML file, just like a normal XML file, with the following content:

```
<aiml version="1.0.1" encoding="UTF-8">
<!-chat.aiml à

  <category>
    <pattern>HELLO</pattern>
    <template>
        Hi, hello!
    </template>
  </category>

  <category>
    <pattern>WHO ARE *</pattern>
    <template>
      <random>
        <li>I'm a bot!</li>
```

```
            <li>Bad guy!</li>
            <li>My name is superman!</li>
          </random>
        </template>
      </category>

      <category>
        <pattern>AWESOME *</pattern>
        <template>
            You're nice too! J
        </template>
      </category>

    </aiml>
```

2. Next, we create a startup XML file that will load the AIML file; this will also load the artificial intelligence we added to the preceding AIML file. Let's call this file `init.xml`:

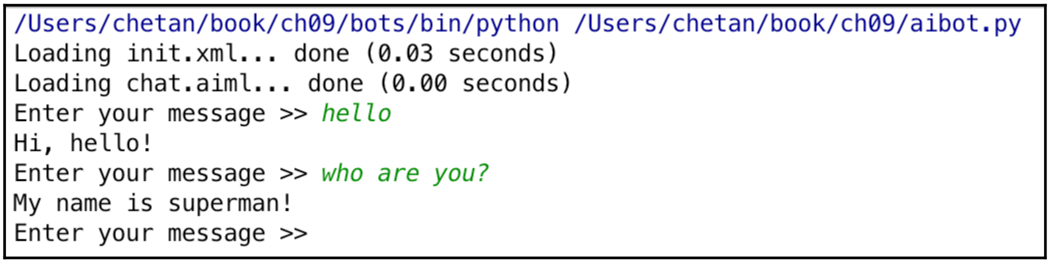
```
    <aiml version="1.0.1" encoding="UTF-8">
        <!-- init.xml -->

        <!-- Category is an atomic AIML unit -->
        <category>

            <!-- Pattern to match in user input -->
            <!-- If user enters "LOAD AIML B" -->
            <pattern>LOAD AIML B</pattern>

            <!-- Template is the response to the pattern -->
            <!-- This learn an aiml file -->
            <template>
                <learn>chat.aiml</learn>
                <!-- You can add more aiml files here -->
                <!--<learn>more_aiml.aiml</learn>-->
            </template>

        </category>

    </aiml>
```

3. Now let's develop the Python code to run our chatbot. The following code does exactly what we need. We call this file `aibot.py`:

```
    import aiml

    # Create the kernel and learn AIML files
```

```
kernel = aiml.Kernel()
kernel.learn("init.xml")
kernel.respond("load aiml b")

# Press CTRL-C to break this loop
while True:
    print kernel.respond(raw_input("Enter your message >>"))
```

4. If we run this bot with the `python aibot.py` command, it presents an input screen, waiting for a user's input. Look at the following screenshot to see how it works:



```
/Users/chetan/book/ch09/bots/bin/python /Users/chetan/book/ch09/aibot.py
Loading init.xml... done (0.03 seconds)
Loading chat.aiml... done (0.00 seconds)
Enter your message >> hello
Hi, hello!
Enter your message >> who are you?
My name is superman!
Enter your message >>
```

How it works…

The preceding Python code mimics a typical bot built on AI. When we run the Python code, `amil.Kernel()` will load the AI kernel.

Once the kernel is loaded, `kernel.learn()` will call the start up `xml` file. The AIML rule engine is loaded when the `load aiml b` command is sent to the kernel.

Once the engine is loaded into the kernel, we are free to chat with the bot.

In the preceding screenshot, when we say **hello**, the bot understands it (from the `chat.aiml` file) and responds to it with **Hi, hello!**, which is configured in `chat.aiml`.

In the second case, the AI bot matches the pattern `WHO ARE *` when **who are you?** is asked by the user; the pattern is again defined in `chat.aiml`.

If you observe, the `WHO ARE *` pattern is configured for multiple responses in the `chat.aiml` file, so the bot, at runtime, chooses a random response and returns **My name is superman!**.

Automating business processes with bots

So far in this chapter, you have learned what bots are, how they are built, and a few simple use cases where bots can be used. Let's see how we can solve Jay's problem with the knowledge we have developed so far and maybe learn more about building bots.

Jay is a marketing manager at a famous book publishing company, *MyBooks*. His task is to come up with book promotion e-mails. He feels that the promotional e-mails he sends are too generic and are not targeted to the readers effectively. For instance, an e-mail on the Python learning path may not encourage a Java developer to spend money. He thinks he can do a much better job if he understands the interests of the audience and makes his interaction more relevant; the reader would be much more inclined to buy the book this way. He also feels that a lot of readers (potential buyers) are on Facebook, but they are not currently being reached out to by the publishing house. Can we help Jay here?

Getting ready

Yes, let's help Jay by developing a fantastic bot for him. If you look into Jay's problem, he needs to understand the audience (in this case, the readers who would be interested in buying a book) and suggest them books based on their interest. So, our bot should be smart enough to get the relevant information from the readers.

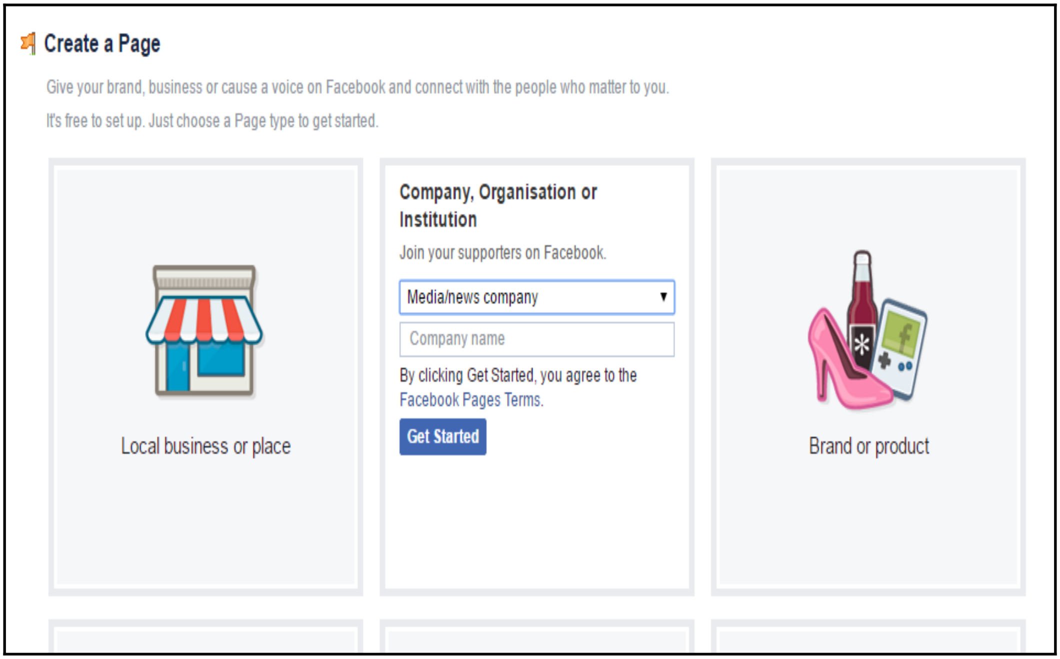
Also, since the readers are already on Facebook, we can create a MyBooks Facebook page and build a Facebook Messenger bot so that the readers can be contacted. Let's see how to do this.

Before we get into building the bot, let's install a few Python modules that will be needed for this exercise. We install the `flask` and `requests` module using the Python `pip`:

```
(bots)chetans-MacBookPro:ch09 Chetan$ pip install flask

Collecting flask
  Using cached Flask-0.11.1-py2.py3-none-any.whl
Collecting click>=2.0 (from flask)
Collecting itsdangerous>=0.21 (from flask)
Collecting Werkzeug>=0.7 (from flask)
  Downloading Werkzeug-0.11.11-py2.py3-none-any.whl (306kB)
    100% |█████████████████████████████████████████████████████| 307kB
1.4MB/s
Collecting Jinja2>=2.4 (from flask)
  Using cached Jinja2-2.8-py2.py3-none-any.whl
Collecting MarkupSafe (from Jinja2>=2.4->flask)
Installing collected packages: click, itsdangerous, Werkzeug, MarkupSafe,
Jinja2, flask
Successfully installed Jinja2-2.8 MarkupSafe-0.23 Werkzeug-0.11.11
click-6.6 flask-0.11.1 itsdangerous-0.24

(bots)chetans-MacBookPro:ch09 Chetan$ pip install requests

Collecting requests
  Using cached requests-2.11.1-py2.py3-none-any.whl
Installing collected packages: requests
Successfully installed requests-2.11.1
```

How to do it…

1. To develop a Facebook Messenger bot, first create a Facebook account (who doesn't have a Facebook account?). Log in to your account and go to `https://www.facebook.com/pages/create/` to create a new page.
2. In our case, since we're building a page for MyBook's company, we can call our page **MyBooks** and choose an appropriate organization type, which is **Media/news company**. This is how the page will look:



Create a Page

Give your brand, business or cause a voice on Facebook and connect with the people who matter to you.

It's free to set up. Just choose a Page type to get started.

Company, Organisation or Institution

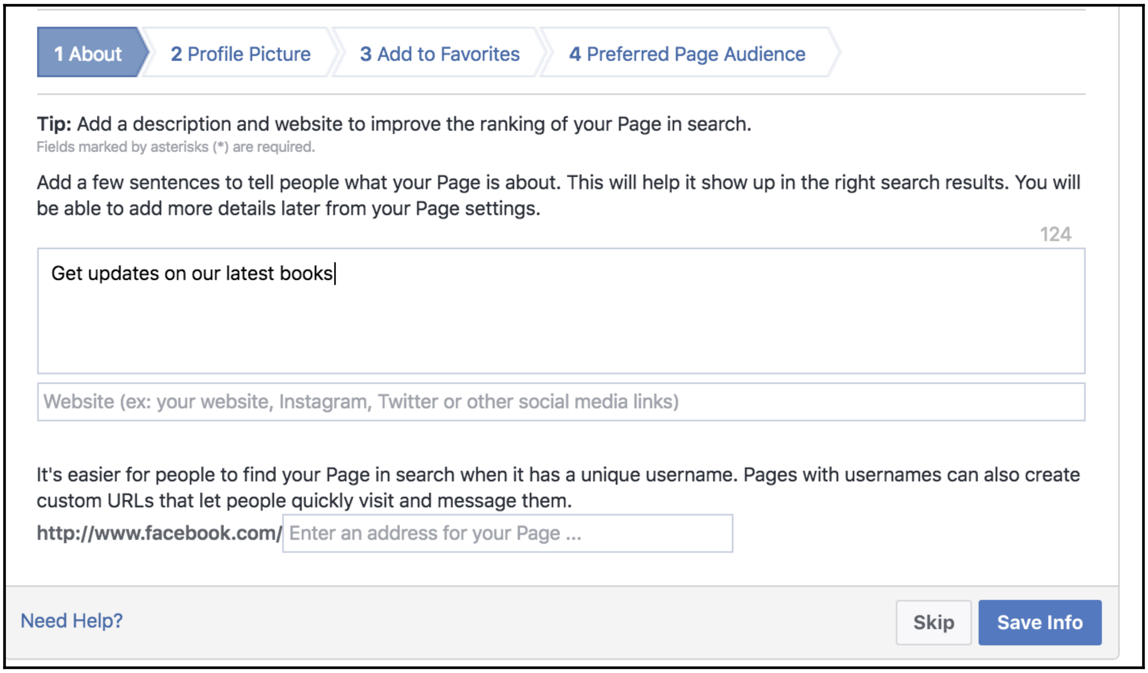
Join your supporters on Facebook.

Media/news company

Company name

By clicking Get Started, you agree to the Facebook Pages Terms.

3. The second step in creating the Facebook page is to fill out the other details requested by Facebook, as shown in the following screenshot. We have given a nice description to our page: `Get updates on our latest books`:



Tip: Add a description and website to improve the ranking of your Page in search.

Fields marked by asterisks (*) are required.

Add a few sentences to tell people what your Page is about. This will help it show up in the right search results. You will be able to add more details later from your Page settings.

124

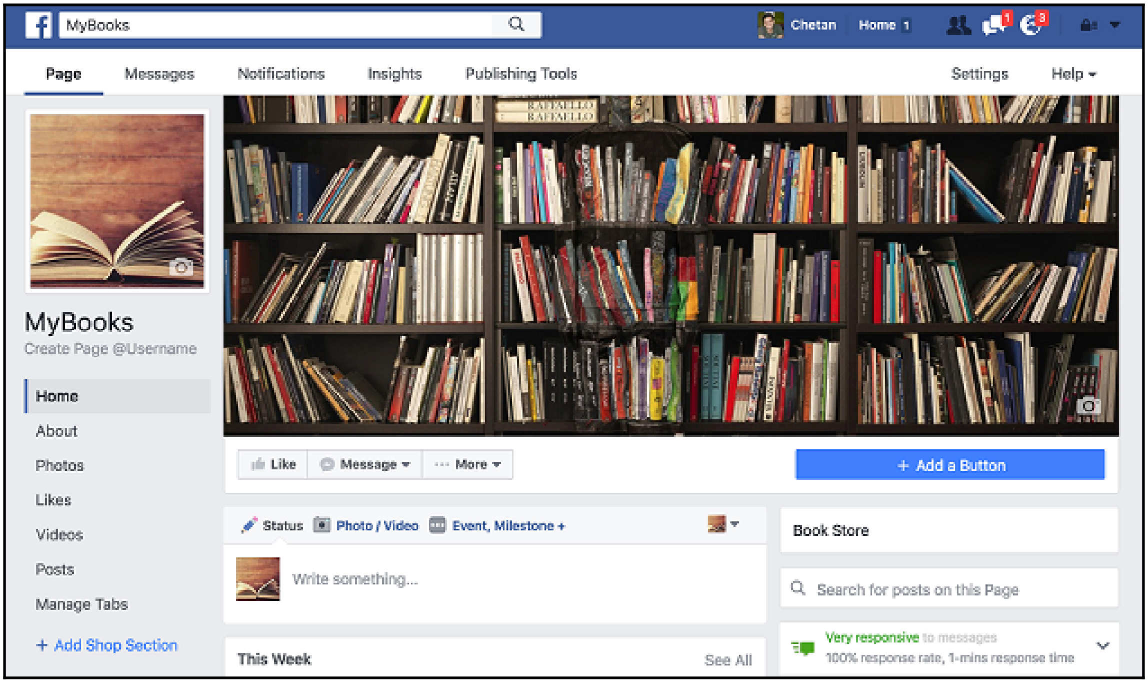
Get updates on our latest books

Website (ex: your website, Instagram, Twitter or other social media links)

It's easier for people to find your Page in search when it has a unique username. Pages with usernames can also create custom URLs that let people quickly visit and message them.

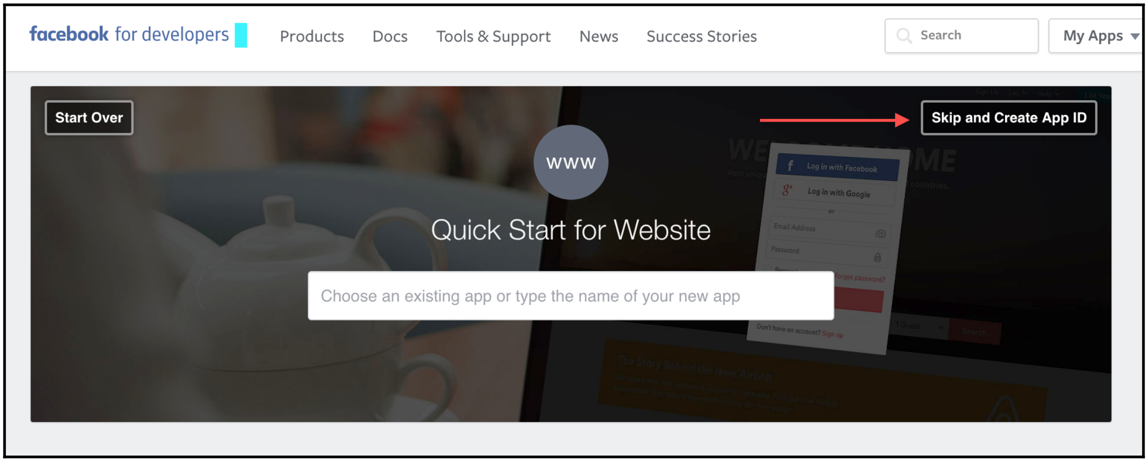
http://www.facebook.com/ Enter an address for your Page ...

4. We have filled in all the details for Jay, and the MyBooks Facebook page is ready and looks smashing:

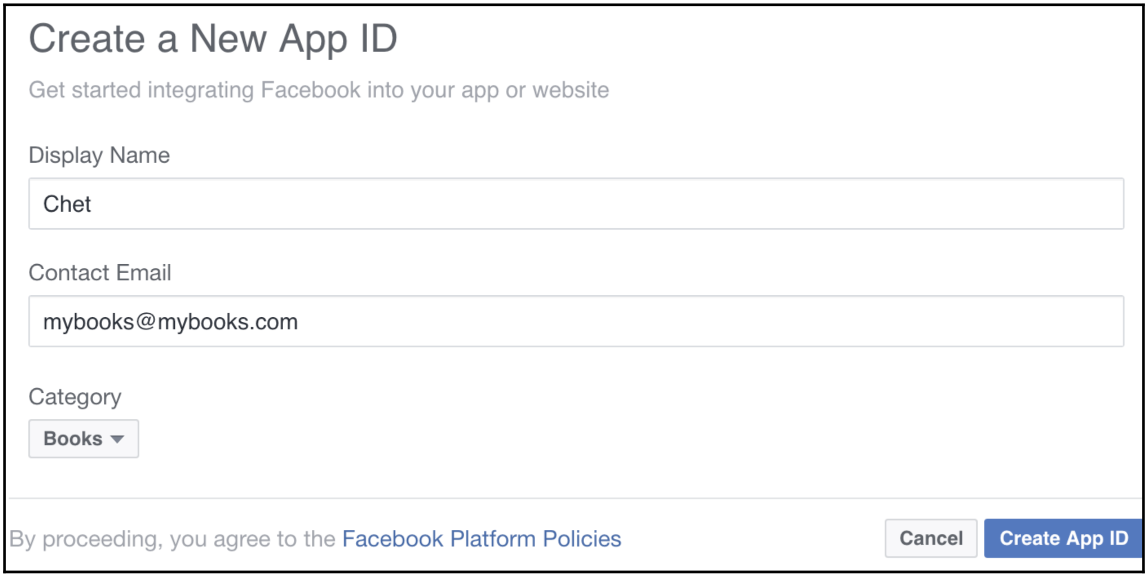


Now, that's a good start. Readers will start following this page, but we really need to add the ability for our readers to converse via a Facebook page; we do this with a Facebook Messenger bot. So let's go ahead and work on this aspect of our solution.

5. To create a Facebook Messenger bot, we need a Facebook app. We will create an app by navigating to `https://developers.facebook.com/quickstarts/?platform=web` and clicking on **Skip and Create App ID**, as shown in the following screenshot:



6. We now fill out the required details and push the **Create App ID** button to create the app. The following screenshot shows the details we added to create an app:



Create a New App ID

Get started integrating Facebook into your app or website

Display Name

Chet

Contact Email

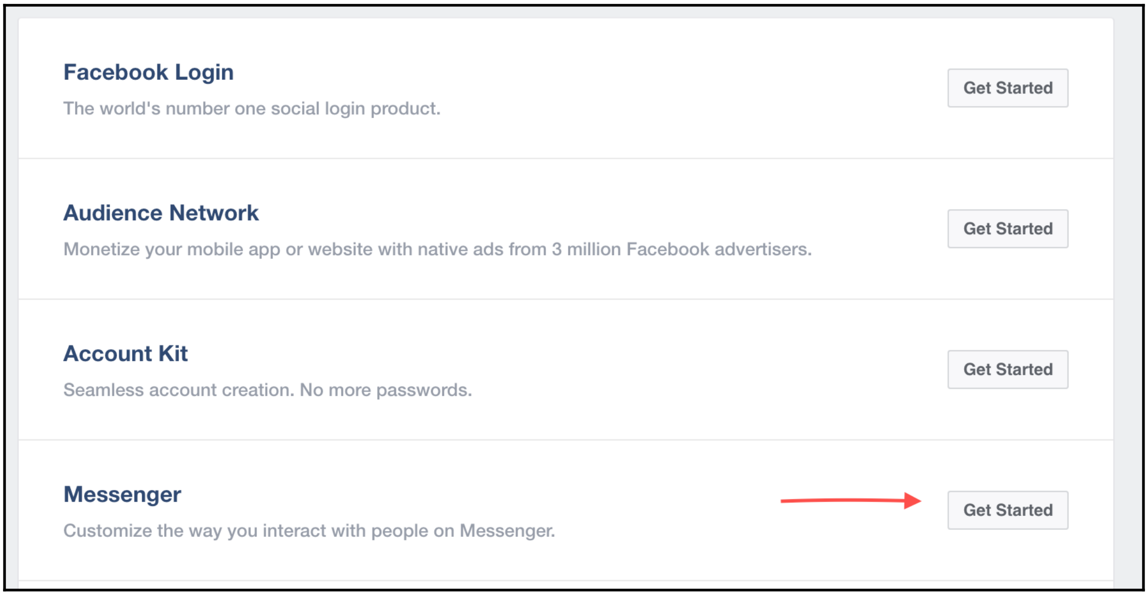
mybooks@mybooks.com

Category

Books ▾

By proceeding, you agree to the Facebook Platform Policies

7. As soon as we fill out the details and click on **Create App ID**, a new app gets created for us. This Facebook app is for our bot. We see the app ID on the top-right section of our page, but to get associated with the bot, we scroll down and click on **Get Started** in the **Messenger** section:



Facebook Login

The world's number one social login product.

Get Started

Audience Network

Monetize your mobile app or website with native ads from 3 million Facebook advertisers.

Get Started

Account Kit

Seamless account creation. No more passwords.

Get Started

Messenger

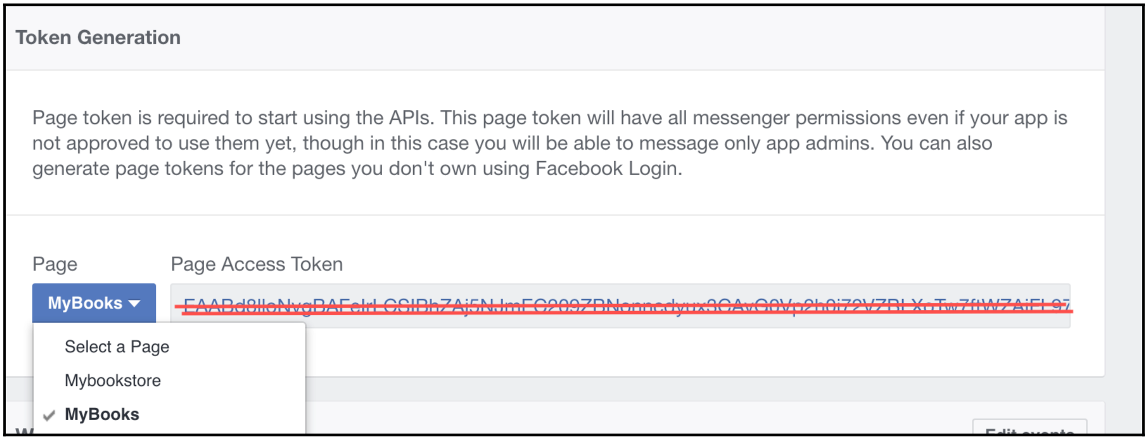
Customize the way you interact with people on Messenger.

Get Started

8. To let the bot access Messenger, we will generate **Page Access Token**, like in the following screenshot.

Keep this token safe with you and don't share it with anyone.

9. This token is used to respond to the readers who initiate a conversation with the bot from the **MyBooks** Facebook page:


Token Generation

Page token is required to start using the APIs. This page token will have all messenger permissions even if your app is not approved to use them yet, though in this case you will be able to message only app admins. You can also generate page tokens for the pages you don't own using Facebook Login.

Page: MyBooks

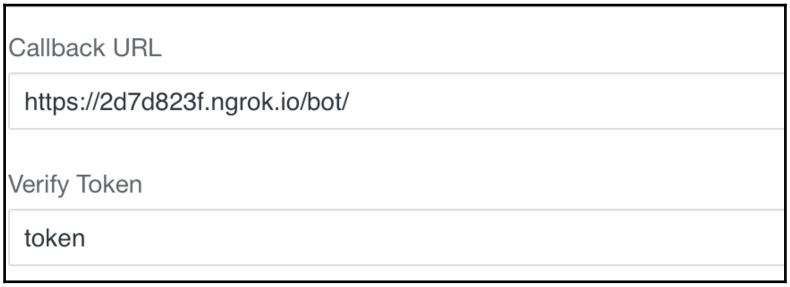
Page Access Token: [redacted]

10. OK, there's one last thing pending. We also need to receive messages from the readers; only then can we respond to them. For this, we go to the **Webhooks** section and add a few settings:
    - **Callback URL**: This is a link to our server where we receive messages from our readers via the Facebook page
    - **Verification Token**: Any set of characters can be used here, say `token`
    - **Subscription Fields**: We choose **messages** as subscription fields for our bot (this can be changed later)

As you see, we need to have a callback URL. This will be used by Facebook to verify whether our callback URL is set up fine. For this, we create a Flask server and configure the route to be used for **Callback URL**. The following code creates a route called `/bot` that is used as **Callback URL** for verification:

```
from flask import Flask
from flask import request
import sys, requests, json, os

app = Flask(__name__)

@app.route("/bot/", methods=['GET', 'POST'])
def hello():
if request.method == 'GET':
return request.args.get('hub.challenge')
```

If we run the server on port 5000 and also use `ngrok` to run on the same port, we get a callback URL that can be placed in the **Webhook** settings. This is how callback URL looks:



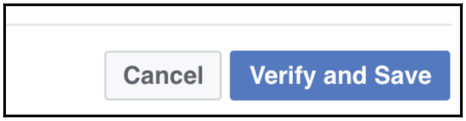
Callback URL

https://2d7d823f.ngrok.io/bot/

Verify Token

token

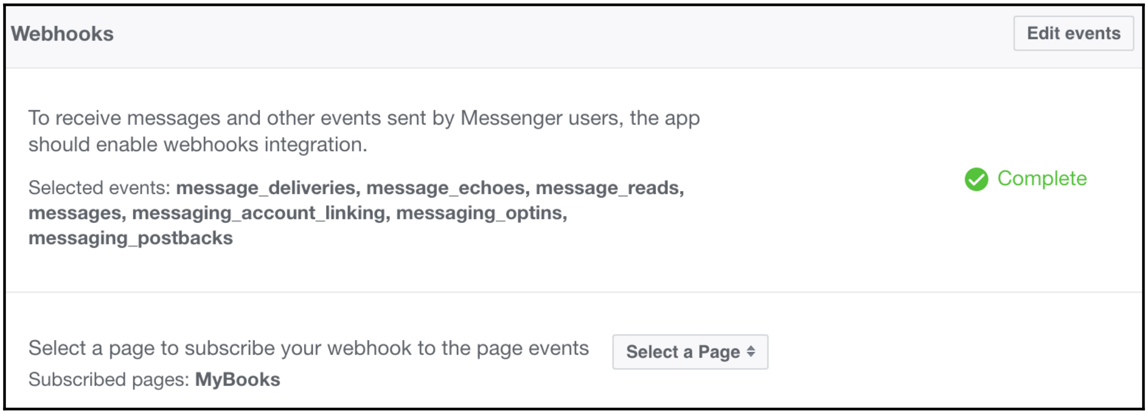
The settings can be verified and saved by clicking on the button, as shown in the following screenshot:



When we verify and save the settings, a `GET` request is sent to our Flask server with the `hub.challenge` code. We return this code to Facebook from the `flask` route and verify the **Webhook** setting:

```
/Users/chetan/book/ch09/bots/bin/python
/Users/chetan/book/ch09/bookbot.py
 * Running on http://127.0.0.1:5000/
  (Press CTRL+C to quit)

127.0.0.1 - - [01/Oct/2016 10:17:43] "GET
/bot/?hub.mode=subscribe&hub
.challenge=1742124657&hub.verify_token=
token HTTP/1.1" 200 -
```

For the bot to work fine we also need to make sure that the Facebook page allows certain events like reading or echoing messages. We enable these settings in the **Webhooks** section:



Webhooks

To receive messages and other events sent by Messenger users, the app should enable webhooks integration.

Selected events: **message_deliveries, message_echoes, message_reads, messages, messaging_account_linking, messaging_optins, messaging_postbacks**

✓ Complete

Select a page to subscribe your webhook to the page events

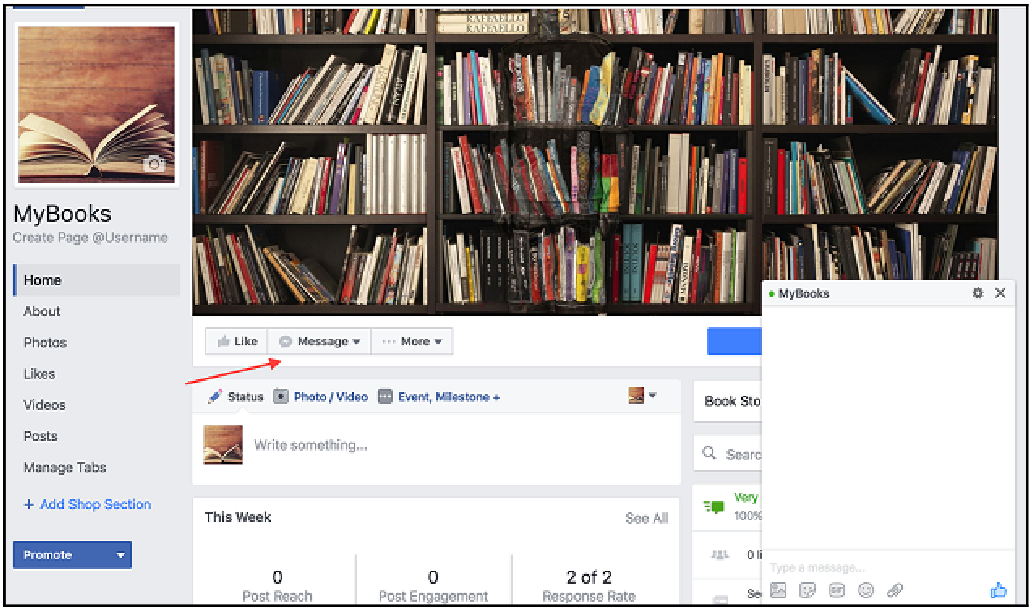
Subscribed pages: **MyBooks**

11. Cool! So now we're ready with a **Webhook** to receive messages from readers and also have an access token to respond to users. If you realize, **Webhook** is going to be our bot server! Let's just go ahead and make our bot do smarter things. The following code will make our bot do all the great things that Jay needs:

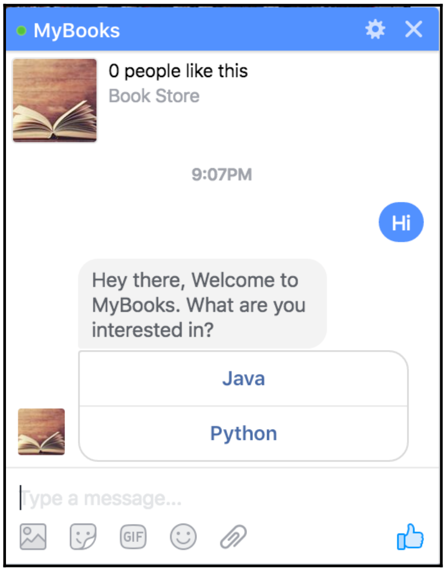
```
from flask import Flask
from flask import request
import requests, json


app = Flask(__name__)

def send_weburl(payload, recipient_id):
    headers = {
       "Content-Type": "application/json"
    }
    token = {
       "access_token":
       "TOKEN"
     }

if payload == 'Python':
  data = json.dumps({
    "recipient": {
      "id": recipient_id
    },
    "message":{
      "attachment":{
        "type":"template",
        "payload":{
          "template_type":"generic",
          "elements":[
          {
            "title":"Learn Python Design Patterns: Chetan
            Giridhar",
            "item_url":"https://www.amazon.com/Learning-Python-
            Design-Patterns-Second/dp/178588803X",
            "image_url":"https://images-na.ssl-images-
            amazon.com/images/I/51bNOsKpItL._SX404_BO1,
            204,203,200_.jpg",
            "subtitle":"Python Book for software architects and
            developers",
            "buttons":[
            {
              "type":"web_url",
              "url":"https://www.amazon.com/Learning-Python-
              Design-Patterns-Second/dp/178588803X",
              "title":"Buy",
              "webview_height_ratio":"full"
            }
            ]
          }
          ]
      }
    }
    }
  })

if payload == 'Java':
  data = json.dumps({
    "recipient": {
      "id": recipient_id
    },
    "message":{
      "attachment":{
      "type":"template",
      "payload":{
        "template_type":"generic",
        "elements":[
        {
          "title":"RESTful Java Patterns and Best
          Practices: Bhakti Mehta",
          "item_url":"https://www.amazon.com/RESTful-Java-
          Patterns-Best-Practices/dp/1783287969",
          "image_url":"https://images-na.ssl-images-
          amazon.com/images/I/51YnSP6uqeL._SX403_BO1,
          204,203,200_.jpg",
          "subtitle":"Python Book for software architects and
          developers",
          "buttons":[
            {
              "type":"web_url",
              "url":"https://www.amazon.com/RESTful-Java-
              Patterns-Best-Practices/dp/1783287969",
              "title":"Buy",
              "webview_height_ratio":"full"
            }
          ]
        }
        ]
      }
      }
    }
  })

r = requests.post("https://graph.facebook.com/v2.6/me/messages",
params=token, headers=headers, data=data)

def send_postback(recipient_id):
  headers = {
    "Content-Type": "application/json"
  }
  token = {
    "access_token":
      "TOKEN"
  }

data = json.dumps({
  "recipient": {
    "id": recipient_id
  },
  "message": {
    "attachment": {
      "type": "template",
      "payload": {
        "template_type": "button",
        "text": "Hey there, Welcome to MyBooks.
        What are you interested in?",
        "buttons": [
        {
          "type":"postback",
          "title":"Java",
          "payload":"Java"
        },
        {
          "type":"postback",
          "title":"Python",
          "payload":"Python"
```

```
                }
                ]
              }
            }
          }
        })

        r = requests.post("https://graph.facebook.com/v2.6/me/messages",
        params=token, headers=headers, data=data)


        @app.route("/bot/", methods=['GET', 'POST'])
        def hello():
          print request.data
        if request.method == 'GET':
          return request.args.get('hub.challenge')

        data = request.get_json()
        if data["object"] == "page":
          for entry in data["entry"]:
            for messaging_event in entry["messaging"]:
              if messaging_event.get("postback"):
                sender_id = messaging_event["sender"]["id"]
                payload = messaging_event["postback"]["payload"]
                send_weburl(payload, sender_id)

              if messaging_event.get("message"):  # readers send us a
              message
                sender_id = messaging_event["sender"]["id"]
                send_postback(sender_id)
                return "ok", 200

               if __name__ == "__main__":
                 app.run()
```

12. We run the preceding Flask server to activate our bot. Now, let's see how the bot works by navigating to the Facebook page. On the Facebook page, if we click on **Message**, we can start chatting with the bot on the **MyBooks** page:



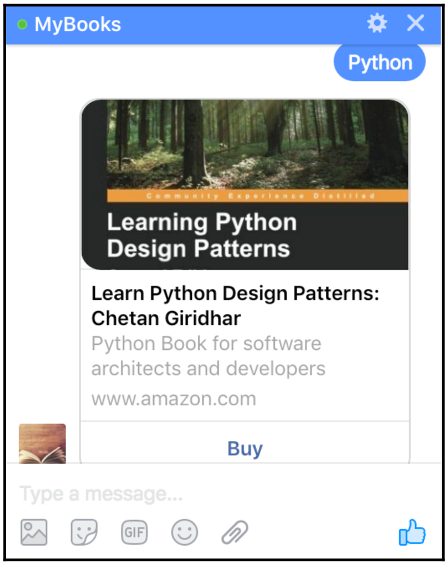
13. Let's start the conversation with the bot using a simple `Hi` message. The bot responds to us with a question about whether we'd like information on Python or Java books. Nice!



Hi

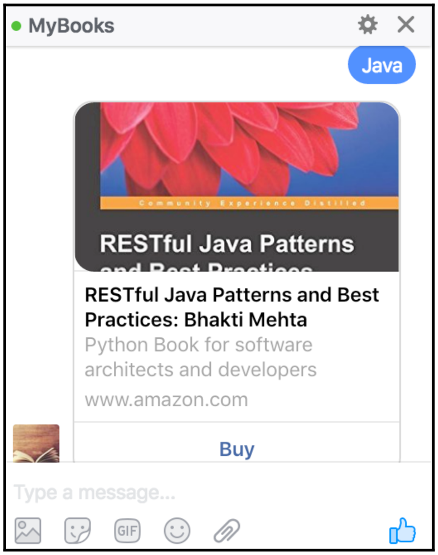
Hey there, Welcome to MyBooks. What are you interested in?

14. Now, if we click on **Python**, the bot recommends an architectural book written in Python and encourages the readers to buy it. This also happens when the reader clicks on **Java**. See the following screenshot:



Learn Python Design Patterns: Chetan Giridhar
Python Book for software architects and developers
www.amazon.com

The following screenshot demonstrates a Java example where the *RESTful Java Patterns and Best Practices* book is recommended when the user selects **Java**:



RESTful Java Patterns and Best Practices: Bhakti Mehta
Python Book for software architects and developers
www.amazon.com

15. Cool, right? This is what Jay needed. So, when the readers arrive at the **MyBooks** page, they get to talk to the bot and the bot recommends them a book based on their interest. Since the suggestion made by the bot is much more relevant to the reader, compared to a generic promotional e-mail, the chances of the reader buying a book are higher. Awesome!

How it works…

We first created a Facebook page for Jay's publishing house: MyBooks. We then associated a Facebook Messenger bot with this page and got **Access Token** to send the message back to the readers who chat with the bot. We also set up **Webhooks** so that our bot would receive messages from the readers and use **Access Token** to post messages back to them. Here, **Webhook** is the brains behind the bot.

When the reader reaches the **MyBooks** page, they click on **Messenger** to start a conversation with the bot. When he or she says `Hi`, HTTP's `POST` request is sent to the **Webhook** `https://2d7d823f.ngrok.io/bot/` with the message.

The bot reads the message from the reader and sends a *generic template* message to the reader with `postback` options. The bot sends this message using Facebook's Graph APIs.

Facebook has template messages for sending `postback` messages, buttons, images, URLs, and audio/video media files.

When the reader chooses **Python**, the bot receives this message and, based on the payload, returns the image of the book along with the URL so users can buy it. Users can then click on **Buy** to go to the book's URL and buy the book from there, exactly what Jay hoped for!

There's more…

In this chapter, we build bots based on CLI, Web UI, and mobile apps. The bots can reside on other chat systems, such as Slack, which has a nice set of APIs. You may want to try writing one. If you do write one, send me the pointers; I would love to try them out.

You can reach out to me on Twitter or send me a direct message and I will get back to you.

9
Imagining Images

We work with images almost on a daily basis. Uploading images to your Facebook profile page, or manipulating images while developing your mobile or web applications; there are abundant use cases. With so many advancements in the field of computer vision, imaging has become a critical field. Working with images is trivial with Python.

In this chapter, we will cover the following recipes:

- Converting images
- Resizing, cropping, and generating thumbnails
- Copy-pasting and watermarking images
- Image differences and comparison
- Face detection
- Imaging as a business process

Introduction

Images in the electronic world are a series of bits comprised of 0s and 1s. They are an electronic snapshot of a scene or a document. Even paintings or photographs can be digitized to form images. Let's go deeper into images and understand how they're structured.

Every image is sampled and is represented by a grid of dots called **pixels**. These pixels represent the smallest controllable elements of a picture shown on the screen. The greater the number of pixels available in an image, the more accurate is the representation of the image on the device screen.

The intensity of each pixel is variable. In the digital world, the color of an image is represented by three or four intensities of different colors: **red, green, and blue** (**RGB**) or **cyan, magenta, yellow, and black** (**CMYK**), respectively. Computer programs often represent colors in RGBA format, where A stands for alpha (or the transparency of a color). Each pixel is represented in the binary representation in the RGBA format and is stored by the computer as a sequence. The computer then reads this sequence for display and, in some cases, converts it to its analog version for printing. Let's look at specific image attributes in detail.

Image attributes

Let's have a look at some of the image attributes:

- **Image size**: As you learned earlier, computer images are stored as a series of 0s and 1s and are measured in pixels (rectangular dots). The file size of an image is calculated based on the number of pixels it contains and the amount of color information that is stored. The file size is, in fact, the space taken up by the image on the computer hard drive.
- **Bit depth**: It is the number of bits used to indicate the color of a single pixel. This concept can be defined as the bits per pixel, which denotes the number of bits used to describe a pixel. The greater the bit depth of an image, greater the number of colors it can store. A 1-bit image can store only two (2^1) colors–0 and 1–and is hence black and white in color. When compared with it, an 8-bit image can store 256 (2^8) colors.
- **Image resolution**: Resolution refers to the number of pixels in an image. Resolution is sometimes identified by the width and height of the image. For example, an image of 1,920 by 1,024 pixel resolution contains 1,966,080 pixels or is a 1.9-megapixel image.
- **Image quality**: It can be changed based on the information an image stores. Not all images need to store all the pixels to represent an image. For instance, a continuous block of blue sea in a picture need not have all the pixels, and the image can be compressed to reduce the disk space of the image without compromising on the image quality. This reduction in disk space is termed as compression. A higher compression means noticeable loss of detail. A typical compression type used in today's world is the JPG compression, which reduces the size and also sacrifices the image's quality.

- **Image formats**: Images are stored in a computer with different extensions. Formats such as BMP or TIF are not compressed at all; hence, they occupy more disk space. Files such as JPG can be compressed and you can also choose the compression level. Hence, TIF images are lossless and JPG compression is termed as lossy compression. It is interesting to note that lossy compressions make use of humans' inability to differentiate minor differences in shades. Multiple lossy conversions will result in image degradation, whereas multiple lossless conversions will preserve the image quality. But generally, it's a trade-off between image degradation and size when it comes to compression.

OK, that's a good start. In this chapter, we will go through some more concepts of images and look at the various operations that you can perform on images with Python. We'll take a look at multiple modules that will help us manipulate images the way we need them. At the end of the chapter, we will also look at a typical business process that can be automated with the knowledge we built with the recipes covered in the chapter.

During the course of this chapter, we will use the following Python modules:

- `Pillow` (`https://pypi.python.org/pypi/Pillow`)
- `scipy` (`https://www.scipy.org/`)
- `opencv` (`https://pypi.python.org/pypi/opencv-python`)

Converting images

Let's begin our journey of images with simple examples. But before going on, let's build our virtual environment.

1. We will build the virtual environment using the `virtualenv` command and also activate it:

```
chetans-MacBookPro:~ Chetan$ cd book/ch10/
chetans-MacBookPro:ch10 Chetan$ virtualenv ch10
New python executable in ch10/bin/python2.7
Also creating executable in ch10/bin/python
Installing setuptools, pip, wheel...done.
chetans-MacBookPro:ch10 Chetan$ source ch10/bin/activate
(ch10)chetans-MacBookPro:ch10 Chetan$
```

2. Nice! So, we now have an independent environment to work on our recipes for this chapter. For the first set of examples, we will use Python's Pillow module. Let's install this module first before going to the recipe. We will use our favorite `python-pip` to install the Pillow module:

```
        (ch10)chetans-MacBookPro:ch10 Chetan$ pip install pillow
        You are using pip version 7.1.0, however version 8.1.2 is
        available.
        You should consider upgrading via the 'pip install --upgrade
        pip' command.
        Collecting pillow
          Downloading Pillow-3.4.2.tar.gz (10.8MB)
          100% |████████████████████████████████|
10.8MB 39kB/s
        Building wheels for collected packages: pillow
          Running setup.py bdist_wheel for pillow
        Installing collected packages: pillow
        Successfully installed pillow-3.4.2
```

So, we now have our environment, and the `Pillow` module is also installed. We're now ready to work with the recipe.

How to do it…

In this section, we will deal with converting images to a different format.

1. First, let's just download an image that can be used as a sample image to perform all our operations. I love sunsets and that's the reason I have used a sunset image for most of the recipes in this chapter. This is how it looks. I store it on my laptop and name it `beach_sunset.png`:

2. Now, let's go ahead and write the Python code to convert this image to the JPEG format. The following code does exactly what we need. We store the Python code in a file named `convert.py`:

```
from PIL import Image
img = Image.open('beach_sunset.png')
img.save('beach-sunset-conv.jpg','jpeg')
```

   When you run this program with Python's `convert.py` command it will take the original PNG image, convert it into JPG format, and store it as `beach-susnset-conv.jpg`.

3. Neat! Now, let's perform one more operation on this image and convert it to grayscale (black and white format). It's common for people to convert images to black and white format to give them a nostalgic look; this can be easily achieved with the following set of command lines:

   ```
   from PIL import Image
   img = Image.open('beach_sunset.png').convert('L')
   img.show()
   img.save('beach-sunset-gray.png','png')
   ```

   Now, when you run this program, you will see another image generated on your disk, with the name `beach-sunset-gray.png`, which looks as follows:

4. Cool, let's go further and perform a few more operations such as rotating and flipping the image. These actions are often used in fun websites where you can simply play around with your images. The following code will help you rotate the images by 180 degrees:

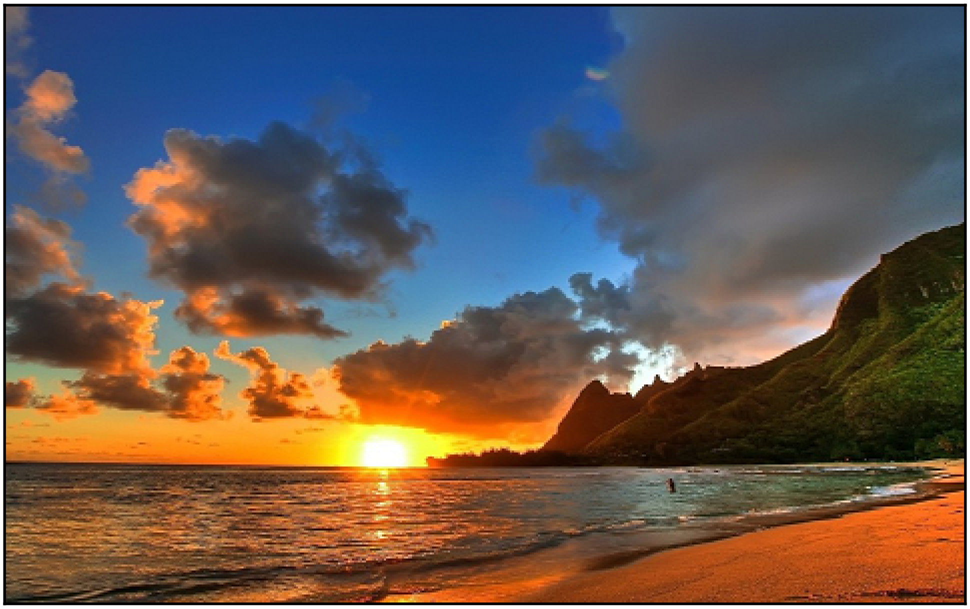
   ```
   from PIL import Image
   img = Image.open('sunset.jpg')
   img.rotate(180).save('sunset180deg.jpg')
   ```

   If you run this code with our base image, you will see an image rotated by 180 degrees, that is, the image is seen upside down, as shown here:

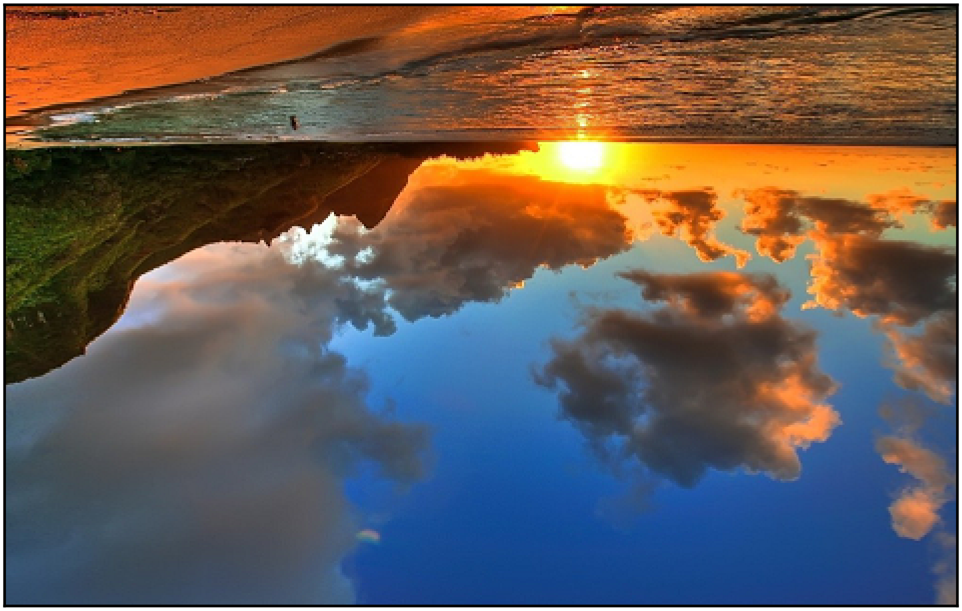
5. While rotating images is nice, it'd be awesome fun if we could flip the images. PIL doesn't disappoint here and provides you with the options of flipping the images horizontally and vertically. The following code will help us perform the flipping operations:

   ```
   from PIL import Image
   img = Image.open('sunset.jpg')
   img.transpose(Image.FLIP_LEFT_RIGHT).save(
                 'sunset_horizontal_flip.png')
   img.transpose(Image.FLIP_TOP_BOTTOM).save(
                    'sunset_vertical_flip.png')
   ```

   Now, if you run this piece of code, it will generate two images. The following image is the same as the original image, but is flipped horizontally (as if there is a mirror kept on the right side of the image). Note how the mountain has moved to the right of the image:



The following screenshot is a mirror image of the original image, which is flipped vertically. Notice, the mountain is still on the left-hand side of the image but it's upside down. Similar is the fate of the setting sun. Is it looking like sunrise?



How it works…

In this section, we dealt with two different image formats: PNG and JPEG. **Portable Network Graphics** (**PNG**) files are non-lossy files, compressing photographic images without degrading quality. It is an excellent file format for Internet graphics; it can be used with multiple backgrounds, and it supports transparency. For the image used in first code example, `beach_sunset.png`, the file size is 550KB.

Joint Photographic Experts Group (**JPEG**) uses lossy compression techniques to compress images. JPGs compress images by reducing the sections of images to pixels or tiles. JPG images can be compressed at a ratio of N:1, depending on the settings. Since images get compressed easily and can reduce the Internet bandwidth for accessing images on the Internet, JPG has become a standard for images on the Internet. For the converted image, I see the file size is 450KB–almost 20% smaller size than the PNG files.

Now, let's understand the Python code. We import the `Image` class from the `PIL` module. The `Image` class is responsible for opening, loading, and converting images, among other operations such as saving images on the disk. In our example, we open the PNG image with the `open()` method and save the image in the JPEG format with the `save()` method.

In the second example, we convert the image to the black and white format. Just as we have the RGB and CMYK formats, we also have the *L* format, which denotes black and white. While converting the image to the *L* format, it uses the ITU-R luma format, where *L=R*299/1000 + G*587/1000 + B*114/1000.*

In terms of Python code, again we use the `Image` class to `open()` the file and use the `convert()` method with its argument as *L* to convert the image to black and white. Finally, we save the file on the disk with `save()` method. Here we maintain the file format to PNG.

In the third example, we use the same `Image` class and `open()` the image to get the `img` object. This object is then used to call the `rotate()` method with the angle of rotation being the argument. In our example, we rotated the image by 180 degrees and finally called `save()` to save the rotated image on the disk with the name, `sunset180deg.jpg`.

In the last example, we make use of the `transpose()` method of the `PIL` module and flip the images both ways, that is, left-right and top-bottom, using the attributes `Image.FLIP_LEFT_RIGHT` and `Image.FLIP_TOP_BOTTOM`, and then save the flipped images with the names `sunset_horizontal_flip.png` and `sunset_vertical_flip.png`, respectively.

There's more…

The Pillow module has many more methods that help us perform more complex operations on our images, such as resizing, pasting, cropping, and what not. We will look at them in the next recipe of this chapter.

Resizing, cropping, and generating thumbnails

Operations such as resizing images and cropping them to get a selected part of the image are very common, but these operations may become tedious when tried programmatically. See how we can achieve these tasks.

Getting ready

In this recipe, we will use the `Pillow` library to resize and crop images. As we already have the Pillow module installed, we don't have to worry about any installations. Let's jump into doing stuff.

How to do it…

1. First, let's look at how to resize an image to the given dimensions. Create a Python file, `resize.py`, and paste the following code snippet:

```
from PIL import Image
img = Image.open('sunset.jpg')
resized = img.resize((256,256))
resized.save('sunset-resize.jpg', 'jpeg')
```

2. Also, download an image from the Internet and name it sunset.jpg. My image looks like this:

3. Now, run the Python code with the `python resize.py` command, and look at your disk for the image, `sunset-resize.jpg`. You'll see that the image gets resized and looks similar to the following screenshot:

The image also has the dimensions of 256 pixels by 256 pixels as expected:

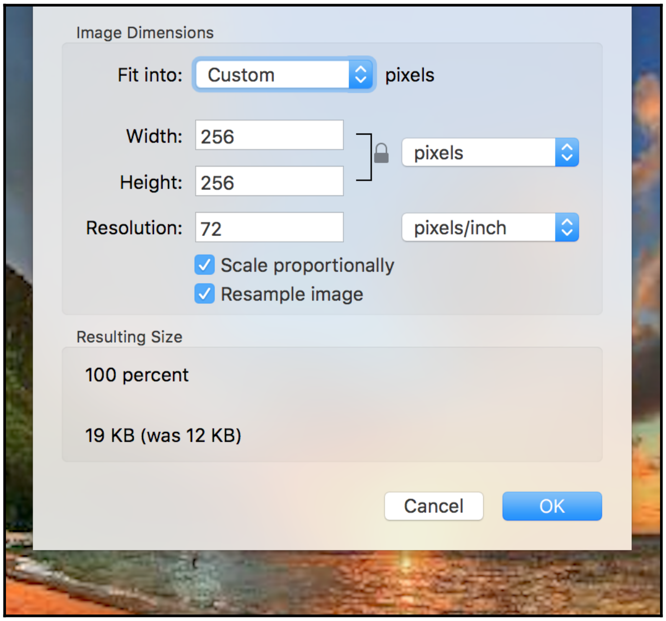


Image Dimensions

Fit into: Custom pixels

Width: 256

Height: 256

Units: pixels

Resolution: 72 pixels/inch

☑ Scale proportionally

☑ Resample image

Resulting Size

100 percent

19 KB (was 12 KB)

4. Another operation often needed in programming is to generate thumbnails for images. Thumbnails are used as the preview of an original image and are typically used in movie review websites or book publishing websites. Let's see if we can easily generate thumbnails with the Pillow module. Create a Python file and add this piece of code:

```
import os, sys
from PIL import Image

size = 128, 128
infile = "sunset.jpg"
outfile = os.path.splitext(infile)[0] + ".thumbnail.jpg"
if infile != outfile:
    try:
        im = Image.open(infile)
        im.thumbnail(size, Image.BICUBIC)
        im.save(outfile, "JPEG")
    except IOError:
        print "cannot create thumbnail for '%s'" % infile
```

Now, if you run this piece of code, you will get an image, `sunset.thumbnail.jpg`, which is a thumbnail of the original image and will look as shown in the screenshot below. If you look at the size of the image, it will not be 128 x 128 (for me, it's 128 x 80 pixels). We will cover the reasons for this in a bit.

Nice! So, we have the thumbnail generated for the image and it can be used on a website as a profile thumbnail or a preview image:

5. Another operation that we will cover in this recipe is the cropping of images. The following code does exactly what we need:

```
from PIL import Image
img = Image.open('sunset.jpg')
cropImg = img.crop((965, 700, 1265, 960))
cropImg.save('sunset-crop.jpg')
```

If you run the preceding Python snippet, you will see an image generated on your disk, `sunset-crop.jpg`, which has a cropped image of the sun from the original sunset image. This is how it looks:

It was nice to see how we could perform multiple operations on the images with Pillow so easily and so intuitively. But how do these operations work; what are the methods used? Let's look at them.

How it works…

In this recipe, we again used Pillow's `Image` class to resize and crop images and generate thumbnails from our original image.

In the first code snippet, we opened the sunset.jpg image with the `open()` method. We then used the `resize()` method with a tuple argument listing the width and height of the resized image. We then used the `save()` method with the filename, `sunset-esize.jpg` and the JPEG file format to save the file on the disk.

In the second snippet, we opened the image with the `open()` method and got an image object. We then used the `thumbnail()` method of the `Image` class on the image object for generating the thumbnail. The `thumbnail()` method takes the size of the image (we used 128 x 128) and uses the BICUBIC image filtering mechanism. Finally, we saved the image with the `save()` method with the target filename set to `sunset.thumbnail.jpg`. We looked at the size of the thumbnail and figured that it's not exactly 128 x 128; in fact, it's 128 x 80. This is because PIL keeps the width of the image to 128 pixels and then recalculates the height to maintain the aspect ratio of the image.

And in the third example, we cropped the image with the `crop()` method of the `Image` class from the Pillow module. The `crop()` method takes all the four coordinates from where the image needs to be carved out from the original image. In our example, we have given the coordinates `left = 965`, `top = 700`, `right = 1265`, `bottom = 960` to crop the original image and the result obtained is the image of the sun as we saw in the examples.

There's more…

In the thumbnail generation example, I briefly mentioned filters that are applied to images for better clarity. I'm not covering these in detail in this chapter, but if you're interested, you can look at them in detail at `http://pillow.readthedocs.io/en/3.0.x/releasenotes/2.7.0.html#default-filter-for-thumbnails`.

Copy-pasting and watermarking images

In this recipe, we will cover one operation that is highly used by designers and marketers, that is, watermarking images. We will also see an interesting use of copy-pasting images over one another. Let's go ahead and look at them.

Getting ready

In this recipe, we will continue to use Pillow to copy-paste images, but we will use another Python module, `wand`, for watermarking. So, as is the normal practice, let's install the `wand` module first before we start writing any code. We install wand with our favorite tool, Python's `pip`:

```
        (ch10)chetans-MacBookPro:ch10 Chetan$ pip install wand
        You are using pip version 7.1.0, however version 8.1.2 is
        available.
        You should consider upgrading via the 'pip install --upgrade
        pip' command.
        Collecting wand
          Downloading Wand-0.4.3.tar.gz (65kB)
            100% |████████████████████████████████|
65kB 101kB/s
        Building wheels for collected packages: wand
          Running setup.py bdist_wheel for wand
          Stored in directory:
             /Users/chetan/Library/Caches/pip/wheels/77/
             c2/a3/6cfc4bb3e21c3103df1ce72d7d301b1965657ee6f81cd3738c
        Successfully built wand
        Installing collected packages: wand
        Successfully installed wand-0.4.3
```

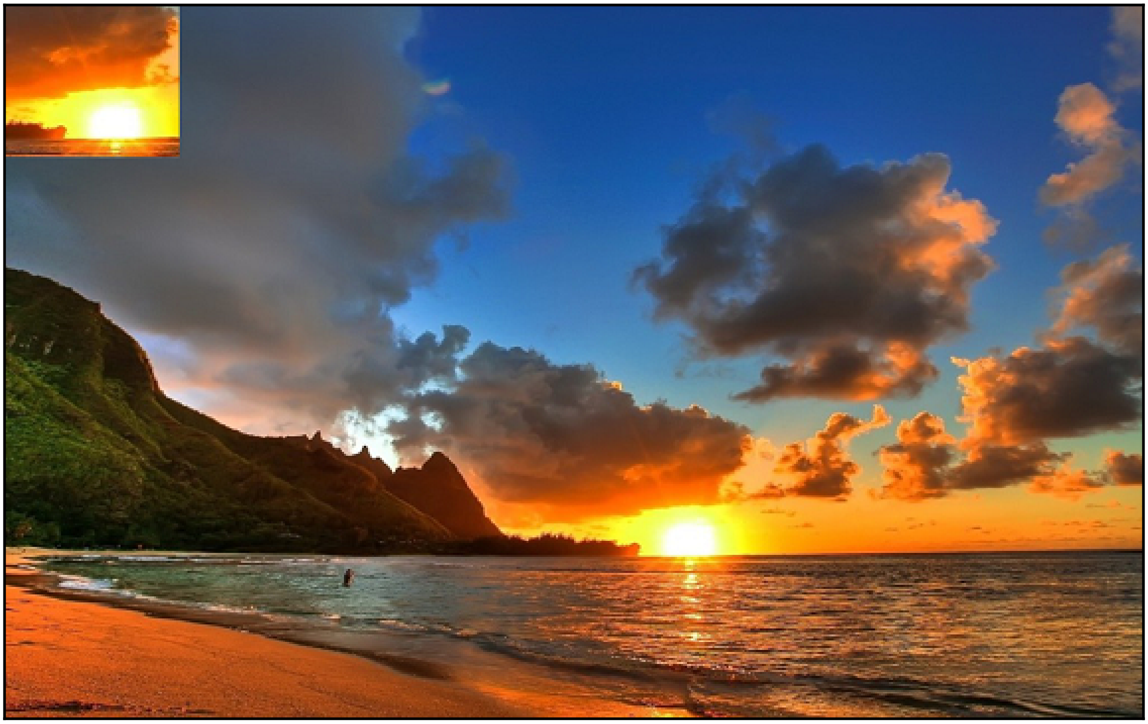
Installed the module already? OK then, let's dive in.

How to do it…

1. First, let's look at how to perform the copy-paste operation with Pillow. Remember, from the previous sections we have two images: the original image, `sunset.jpg`, and the image of the sun that was cropped from the original image, `sunset-crop.jpg`. We will use these images in the following Python code:

```
from PIL import Image
img = Image.open('sunset-crop.jpg')
pasteImg = Image.open('sunset.jpg')
pasteImg.paste(img, (0,0))
pasteImg.save('pasted.jpg')
```

2. Let's store the code in a file by the name, `copy_paste.py`, and run the code with the Python command, `copy_paste.py`. Once we run the code, we will see a new file being generated, called `pasted.jpg`, which looks like the following screenshot:



What we have managed to do is copy the cropped image, paste it on the original image, and save the pasted image as `pasted.jpg`. Cool, isn't it?

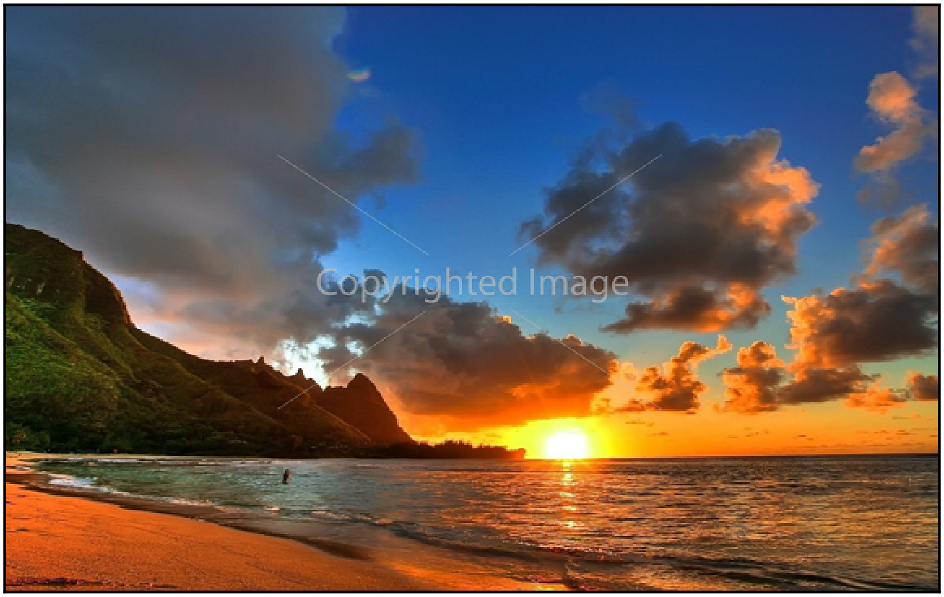
Now, let's look at an interesting example that has commercial use. In this example, we will add a watermark to an existing image and store it under a different name. But before we get into the Python code, let's look at how the watermark image looks:

Copyrighted Image

The following Python code helps us in adding the preceding watermark to our original `sunset.jpg` image file:

```
from wand.image import Image

with Image(filename='sunset.jpg') as background:
with Image(filename='watermark.jpg') as watermark:
  background.watermark(image=watermark, transparency=0.25,
  left=560, top=300)
background.save(filename='result.jpg')
```

3. Run this code and you will see a `result.jpg` file being generated in your project. It will look similar to the following screenshot. Look at how the image is watermarked with the **Copyrighted Image** text on top:



How it works…

For the first code snippet, we used the `PIL` module and the Image class to `open()` the cropped image and original image and get the file handles of both the files, namely: `img` and `pasteImg`.

As the name suggests, we opened the cropped image `img` and pasted it on `pasteImg`, using the file handles.

For pasting the image, we used Pillow's `paste()` module and passed the `img` file handle to it as the source image. We also passed the coordinates where the cropped image is to be pasted on the original image. Since we have chosen the coordinates to be (0, 0), the cropped image is pasted on the upper-left corner of the original image. Finally, we saved this image as `pasted.jpg`.

In the second example, we opened the original image, `sunset.jpg`, and the watermark image, `watermark.jpg`, and created the file handles, `background` and `watermark`, respectively. We then used the `wand` module's `watermark()` method to add the watermark to the original image.

The `watermark()` method works on the background image object (in this case, background, our original image object). It uses image as the `keyword` argument, which indicates the object of the watermark image. You can also set the transparency of the watermark image, where `0` indicates that the watermark is completely visible while `1` indicates that it is invisible. Another useful thing you can achieve with the `watermark()` method is that you can choose the location of the watermark on the original image. In this example, we have chosen it to be at the coordinates `560` from the left and `300` from the top.

Cool; that's it in this recipe. Let's go ahead and see what we have in store in the remaining recipes of this chapter.

Image differences and comparison

You must have definitely used a text-based search or even implemented one. But do you know, you can now even perform an image-based search? Of course, Google does that quite nicely. How do you think it does that? If you have to implement one yourself, you better know how to compare two images. Based on your use case, you may also want to get a `diff` or the difference between two images. In this recipe, we will cover two use cases:

- How to get the difference between two images and store the difference as an image
- How to objectively compare two images with scientific methods

Getting ready

In this recipe, we will continue using Pillow to compare images. Along with our `Image` class, we will also use the `ImageChops` class to get the difference between two images. We will use the `scipy` module to compare the images at the pixel level.

1. We already have the Pillow module installed, so let's go ahead and install the `scipy` module using Python `pip`. On Mac OS X machine, you'll need to have a compiler to install the `scipy` module. We will install the GCC compiler on Mac with Mac's `brew` command:

   ```
   (ch10)chetans-MacBookPro:ch10 Chetan$ brew install gcc
   Warning: Building gcc from source:
   The bottle needs the Xcode CLT to be installed.
   ==> Using the sandbox
   ==> Downloading https://ftpmirror.gnu.org/gcc/gcc-6.2.0/gcc-
   6.2.0.tar.bz2
   ```

```
Already downloaded: /Users/chetan/Library/Caches/Homebrew/gcc-
6.2.0.tar.bz2
==> Downloading
https://raw.githubusercontent.com/Homebrew/formula-
patches/e9e0ee09389a54cc4c8fe1c24ebca3cd765ed0ba/gcc/6.1.0-
jit.patch
Already downloaded: /Users/chetan/Library/Caches/Homebrew/gcc--
patch-     863957f90a934ee8f89707980473769cff47
ca0663c3906992da6afb242fb220.patch
==> Patching
==> Applying 6.1.0-jit.patch
patching file gcc/jit/Make-lang.in
==> ../configure --build=x86_64-apple-darwin15.5.0 --
    prefix=/usr/local/Cellar/gcc/6.2.0 --
    libdir=/usr/local/Cellar/gcc/6.2.0/lib/gcc/6 --enable-
    languages=c,c++,objc,obj-c++,fortran
    ==> make bootstrap


==> make install
==> Caveats
GCC has been built with multilib support. Notably,
    OpenMP may not
work:
    https://gcc.gnu.org/bugzilla/show_bug.cgi?id=60670
If you need OpenMP support you may want to
brew reinstall gcc --without-multilib
==> Summary
  /usr/local/Cellar/gcc/6.2.0: 1,436 files, 282.6M, built in 70
  minutes 47 seconds
  (ch10)chetans-MacBookPro:ch10 Chetan$
```

2. Now that we have GCC installed, let's install `scipy` with `python-pip`. This is how the installation logs look on my system:

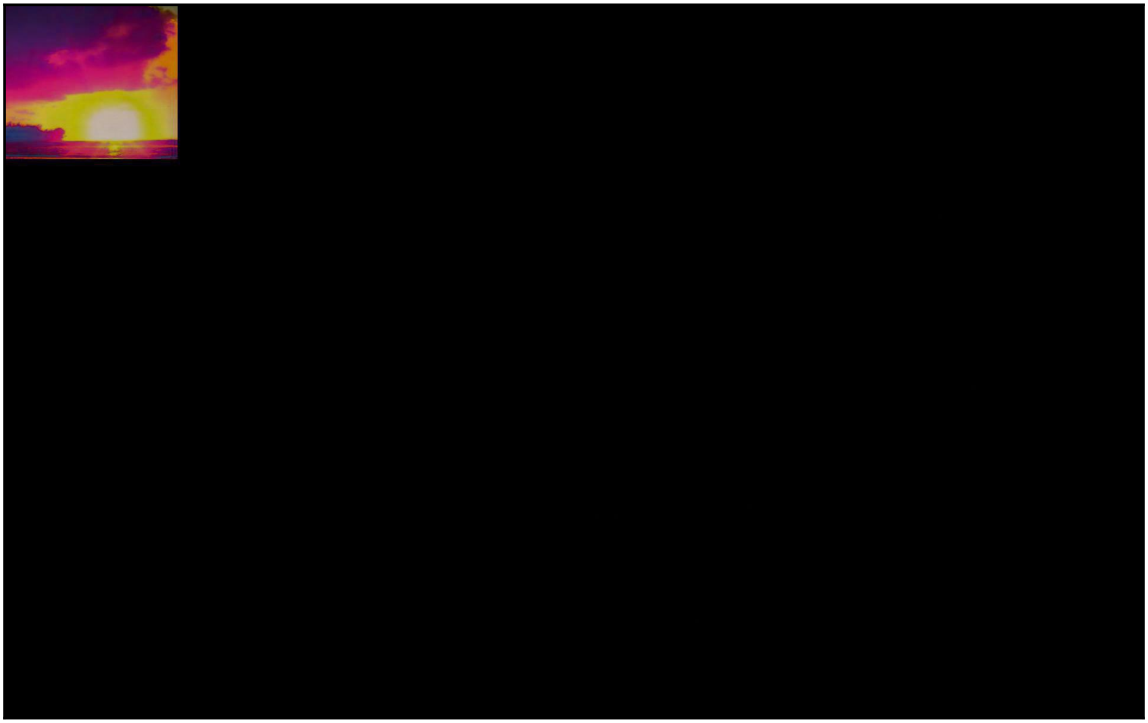
```
(ch10)chetans-MacBookPro:ch10 Chetan$ pip install scipy
You are using pip version 7.1.0, however version 8.1.2 is
available.
You should consider upgrading via the 'pip install --upgrade
pip'
command.
Collecting scipy
  Using cached scipy-0.18.1.tar.gz
Building wheels for collected packages: scipy
  Running setup.py bdist_wheel for scipy


  Stored in directory:
     /Users/chetan/Library/Caches/pip/wheels/33/
     c4/f5/e00fe242696eba9e5f63cd0f30eaf5780b8c98067eb164707c
Successfully built scipy
Installing collected packages: scipy
Successfully installed scipy-0.18.1
```

How to do it…

1. Now that the modules are already installed, let's start utilizing them for our needs. First, let's look at getting the difference between two images and storing the difference as an image itself. The following code does this operation:

```
from PIL import Image, ImageChops

def differnce_images(path_one, path_two, diff_save_location):
    image_one = Image.open(path_one)
    image_two = Image.open(path_two)

    diff = ImageChops.difference(image_one, image_two)

    if diff.getbbox() is None:
        print "No difference in the images"
        return
    else:
        print diff.getbbox()
        diff.save(diff_save_location)

differnce_images('sunset.jpg','pasted.jpg',
                 'diff.jpg')
```

In the preceding code example, we calculated the difference between the original image, `sunset.jpg`, and the copy-pasted image, `pasted.jpg` (if you remember the previous recipe, `pasted.jpg` is obtained after pasting the cropped sun image on the original sunset image). This is how the difference image looks:



Observe how the difference is only the cropped image of the sun since the base original image remains the same. Cool! What does the black region indicate? We will talk about it in the *How it works...* section.

2. Now, let's move ahead and look at calculating the difference between images in an objective manner. For this, we will use the `scipy` module. The following code example will help us with what we need:

```
from scipy.misc import imread
from scipy.linalg import norm

def compare_images(img1, img2):
    diff = img1 - img2
    z_norm = norm(diff.ravel(), 0)
    return z_norm

img1 = imread("sunset.jpg").astype(float)
img2 = imread("pasted.jpg").astype(float)
z_norm = compare_images(img1, img2)
print "Pixel Difference:", z_norm
```

If we run the preceding Python code, we will get the difference in pixels in both these images. The output of our example is as follows:

```
Pixel Difference: 246660.0
```

How it works…

In the first code snippet of this section, we calculated the difference between the two images using the `ImageChops` class of the Pillow library. As usual, we opened both the images with the `open()` method and got the image objects `image_one` and `image_two`, respectively.

We then used the `difference()` method of the `ImageChops` class and passed the image objects as arguments to this method. The `difference()` method returns the `diff` object, which essentially is the object representing the difference between the two images.

Finally, we saved the difference as an image on the disk with the name, `diff.jpg`. We also used the `getbbox()` method on the `diff` object, which calculates the bounding box of the nonzero regions in the image. Here nonzero regions indicate the pixels where the difference between `sunset.jpg` and `pasted.jpg` is 0.

Now, if you look at `diff.jpg`, it contains a huge black region. These are the pixels where the difference is 0, hence the color black. For same images, the `getbbox()` method returns `None`.

In the second example, we compared the two images based on zero norm, which indicates the number of pixels not equal to zero, or in other words, indicates how many pixels differ between the two images. For comparing images, we first read both the images using the `imread()` method of the `scipy` module. Both the image objects are `img1` and `img2`.

We then calculated the difference between the two images with `diff = img1 - img2`. This difference returned is of the `ndarray` type of `scipy`. When we pass this difference to the `norm()` method, it returns the number of pixels that are different between images.

There's more…

There are multiple ways of comparing images, which we haven't covered in this chapter. I suggest you do a deeper reading into this if you're really interested. But for all practical purposes, I think this chapter should suffice.

Face detection

We covered a lot of operations on images in the preceding sections. In this recipe, let's delve deeper and cover an advanced operation such as face detection in images.

Getting ready

In this recipe, we will use Python's `opencv` module, so let's start by installing the required module.

1. For using `opencv` Python bindings, we have to first install `opencv` on our computer. On my Mac OS X machine, I use the `brew` utility to install `opencv` this way:

   ```
   (ch10)chetans-MacBookPro:ch10 Chetan$ brew install
   homebrew/science/opencv
   ==> Tapping homebrew/science
   Cloning into '/usr/local/Homebrew/Library/Taps/homebrew/homebrew-
   science'...
   ...
   ...
   ==> Summary
     /usr/local/Cellar/opencv/2.4.13.1: 277 files, 35.5M
   ```

2. Just installing `opencv` on your computer doesn't help. You also need to point the `cv2.so` (`.so` stands for shared object or library) to glue it with a virtual environment using the following commands:

   ```
   cd ch10/lib/python2.7/site-packages/
   ln -s /usr/local/Cellar/opencv/2.4.13.1/lib/python2.7/site-
   packages/cv.py
   ln -s /usr/local/Cellar/opencv/2.4.13.1/lib/python2.7/site-
   packages/cv2.so
   ```

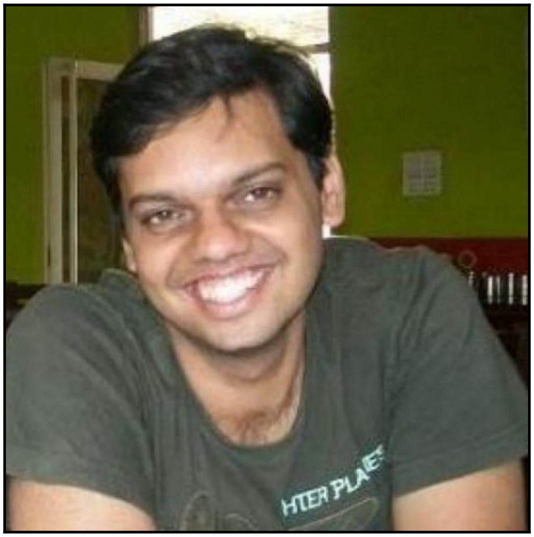
Cool! So, we now have `opencv` installed, which is required for our examples in this recipe.

How to do it…

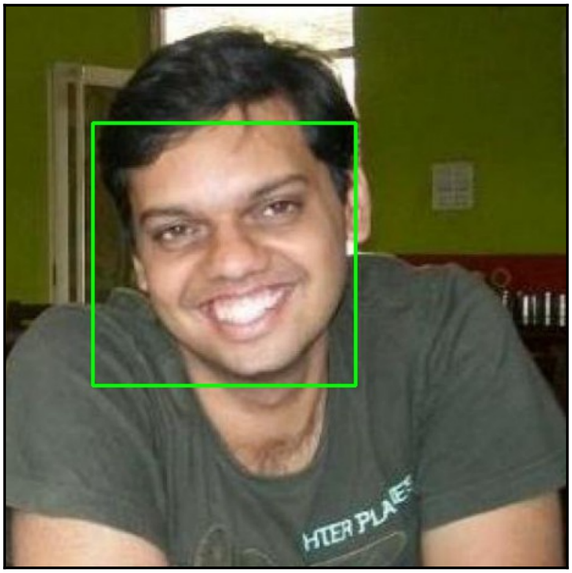
1. Go to your favorite editor, create a Python file, and name it face_detection.py. Now, copy the following code into the Python file:

```
import cv2

face_cascade = cv2.CascadeClassifier('haarcascade.xml')
original_image_path = 'Chetan.jpeg'

image = cv2.imread(original_image_path)

faces = face_cascade.detectMultiScale(
    image, scaleFactor=1.1, minNeighbors=3,
    minSize=(30, 30), flags=cv2.cv.CV_HAAR_SCALE_IMAGE)

for (x, y, w, h) in faces:
    cv2.rectangle(image, (x, y), (x + w, y + h), (0, 255, 0), 2)

cv2.imwrite('chetan_face.jpg', image)
```

2. Now, create an XML file by the name haarcascade.xml and copy the contents from the code repository of this book. In my example, I am using one of the pictures of myself, Chetan.jpeg, but you can use any of your pictures for this example. Here's how Chetan.jpeg looks:



3. Now, let's run the Python code and see if our code is able to recognize my face from the image. We run the code with the command `python face_detection.py` and it generates an image, `Chetan_face.jpg`, which looks like the following. Indeed, it did detect my face.



How it works…

In our recipe, we used the `opencv` module to first create a cascade classifier object with the `haarcascade.xml` file. We called this object `face_cascade`.

Object detection using Haar's feature-based cascade classifiers is an effective object detection method proposed by Paul Viola and Michael Jones in 2001. It is a machine learning-based approach, where a cascade function is trained from a lot of positive and negative images. It is then used to detect objects in other images.

Haar features are fed into a standard XML file, which is what we used in our code example. You can actually train your classifier to detect the objects you wish. For instance, eye detection uses another classifier.

Next, we read the original base image, `Chetan.jpeg`, using the `opencv` module's `imread()` method and defined the smallest window for detection.

In fact, Haar cascade classifier works on a sliding window approach and hence needs the smallest window for detection. The classifier also needs to have `minNeighbors` configured.

Settings such as these are configured in the `detectMultiScale()` method of the cascade object. We have set `minSize=(30,30)` and `minNeighbors=3`.

Finally, we stored the detected image on the disk and the original image has a green rectangle as an indication of face detection on the image.

There's more…

We looked at a very simple example of face detection with `opencv` and learned a bit about classifiers. `opencv` has more things that you may want to learn about.

Here's a link to a resource that you will find interesting to read:
`http://docs.opencv.org/trunk/index.html`.

Imaging as a business process

Peter is an IT manager at MBI Inc, a big corporation. His company has been in existence for long enough for a majority of the contractual finance documents, standard operating procedures, and supply chain documents to be paper-based. He is tasked with this humongous responsibility of making his company go paperless.

This means that he is responsible for eliminating the hassle and cost of managing paper archives. With the imaging knowledge we have gathered so far (and we will learn more) in this chapter, let's see if we can help Peter.

If you carefully analyze, Peter needs to achieve two important tasks:

- Scan the papers and store them in an electronic format as images
- Generate text files from these documents so that they can be easily indexed

Getting ready

For this exercise, let's start by installing the required modules. We will need the following modules:

- `scikit-image` (`http://scikit-image.org/`)
- `pyimagesearch` (`http://www.pyimagesearch.com/`)
- `tessaract` and `pytesseract` (`https://pypi.python.org/pypi/pytesseract/`)

Let's start installing the modules:

1. Let's first start with `scikit-image`:

```
        (ch10)chetans-MacBookPro:ch10 Chetan$ pip install scikit-image
        You are using pip version 7.1.0, however version 8.1.2 is
        available.
        You should consider upgrading via the 'pip install --upgrade
        pip' command.
        Collecting scikit-image
          Downloading scikit-image-0.12.3.tar.gz (20.7MB)
            100% |█████████████████████████████████████████████████████|
20.7MB 19kB/s
        Requirement already satisfied (use --upgrade to upgrade):
        six>=1.7.3 in ./lib/python2.7/site-packages (from scikit-image)
        Collecting networkx>=1.8 (from scikit-image)
          Downloading networkx-1.11-py2.py3-none-any.whl (1.3MB)
            100% |█████████████████████████████████████████████████████|
1.3MB 325kB/s
        Requirement already satisfied (use --upgrade to upgrade):
        pillow>=2.1.0 in ./lib/python2.7/site-packages (from scikit-
        image)
        Collecting dask[array]>=0.5.0 (from scikit-image)
          Downloading dask-0.11.1-py2.py3-none-any.whl (375kB)
            100% |█████████████████████████████████████████████████████|
376kB 946kB/s
        Collecting decorator>=3.4.0 (from networkx>=1.8->scikit-image)
        Using cached decorator-4.0.10-py2.py3-none-any.whl
        Collecting toolz>=0.7.2 (from dask[array]>=0.5.0->scikit-image)
        Using cached toolz-0.8.0.tar.gz
        Requirement already satisfied (use --upgrade to upgrade):
        numpy in
        ./lib/python2.7/site-packages (from dask[array]>=0.5.0->scikit-
        image)
        Building wheels for collected packages: scikit-image, toolz
          Running setup.py bdist_wheel for scikit-image
          Stored in directory:
```

```
/Users/chetan/Library/Caches/pip/wheels/d5/e8/77/925fe026d562a74a0bccf1c7dd
47d00f5f6ab2d395f247e674
          Running setup.py bdist_wheel for toolz
          Stored in directory:
/Users/chetan/Library/Caches/pip/wheels/b0/84/bf/7089262387e8ea60bdefb1fdb8
4d2ee99427f6d09c9c7ba37d
        Successfully built scikit-image toolz
        Installing collected packages: decorator, networkx, toolz,
        dask,
        scikit-image
        Successfully installed dask-0.11.1 decorator-4.0.10 networkx-
        1.11
        scikit-image-0.12.3 toolz-0.8.0
```

2. Next, let's install `pyimagesearch`. This is a nice set of libraries developed by Adrian Rosebrock. He has his work open-sourced at `https://github.com/jrosebr1`. We, in fact, leverage the scanner example of `pyimagesearch` in this code recipe.
3. Lastly, let's install `tesseract` and `pytesseract`. We need to install `tesseract`, an **Optical Character Reader** (**OCR**) module, and `pytesseract`, a Python module to work with the OCR module:

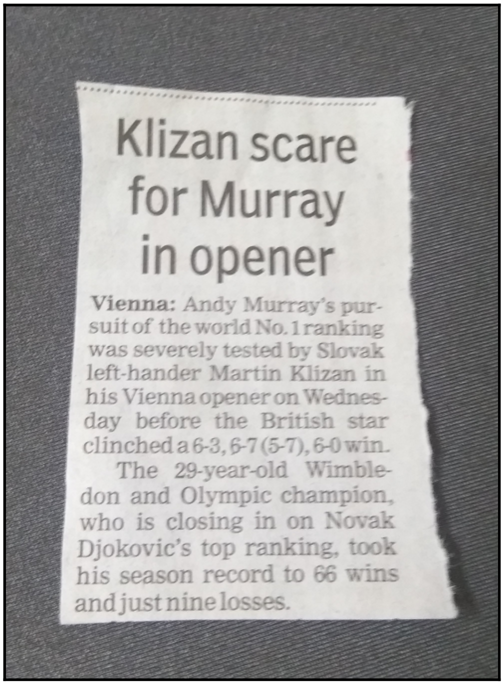
   ```
   (ch10)chetans-MacBookPro:ch10 Chetan$ brew install tesseract
   ==> Auto-updated Homebrew!
   Updated 4 taps (homebrew/core, homebrew/dupes, homebrew/python,
   homebrew/science).
   ..
   ..
   ..
   ==> Installing dependencies for tesseract: leptonica
   ==> Installing tesseract dependency: leptonica
   ==> Downloading https://homebrew.bintray.com/bottles/leptonica-
   1.73.el_capitan.bottle.tar.gz
     ########################################################
     ################    # 100.0%
   ==> Pouring leptonica-1.73.el_capitan.bottle.tar.gz
    /usr/local/Cellar/leptonica/1.73: 50 files, 5.4M
   ==> Installing tesseract
   ==> Downloading https://homebrew.bintray.com/bottles/tesseract-
   3.04.01_2.el_capitan.bottle.tar.gz
     #########################################################
     ###############    # 100.0%
   ==> Pouring tesseract-3.04.01_2.el_capitan.bottle.tar.gz
     /usr/local/Cellar/tesseract/3.04.01_2: 76 files, 39M
   (ch10)chetans-MacBookPro:ch10 Chetan$ pip install pytesseract
   Collecting pytesseract
     Downloading pytesseract-0.1.6.tar.gz (149kB)
   ```

```
                100% |████████████████████████████████| 
   151kB 201kB/s
           Building wheels for collected packages: pytesseract
             Running setup.py bdist_wheel for pytesseract
             Stored in directory:
             /Users/chetan/Library/Caches/pip/wheels/f2/27/64/
             a8fa99a36b38980aaf8d1d2c87f5dd6b5a0a274b8706e3df36
           Successfully built pytesseract
           Installing collected packages: pytesseract
           Successfully installed pytesseract-0.1.6
```

OK, cool! Now, let's look at the code in the *How to do it…* section.

How to do it…

1. Go to your favorite editor, create a Python file, and name it `scanner.py`. For Peter, it would be all about his financial documents, which are in the image format, but for the sake of this example, it'll be an image that I have handy with me. Here's how my image looks. It's the picture of a newspaper article on Andy Murray and I'm trying to digitalize it:



Klizan scare
for Murray
in opener

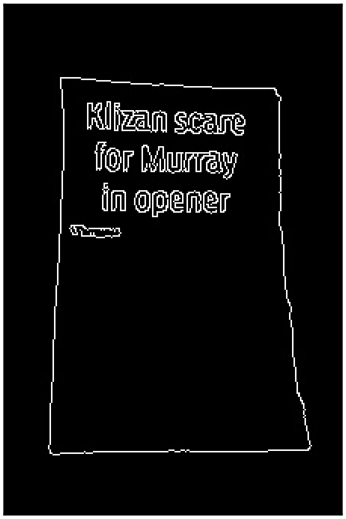
Vienna: Andy Murray's pursuit of the world No.1 ranking was severely tested by Slovak left-hander Martin Klizan in his Vienna opener on Wednesday before the British star clinched a 6-3, 6-7 (5-7), 6-0 win.

The 29-year-old Wimbledon and Olympic champion, who is closing in on Novak Djokovic's top ranking, took his season record to 66 wins and just nine losses.

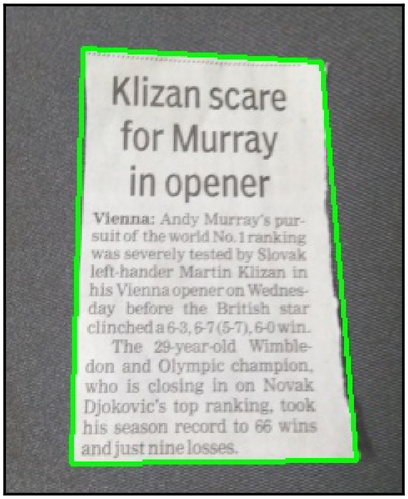
2. Now, copy the following code in `scanner.py` and run the code with the command, `python scanner.py`:

```
from pyimagesearch.transform import four_point_transform
from pyimagesearch import imutils
from skimage.filters import threshold_adaptive
import cv2


image = cv2.imread("images/murray.jpg")
ratio = image.shape[0] / 500.0
orig = image.copy()
image = imutils.resize(image, height = 500)
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
gray = cv2.GaussianBlur(gray, (5, 5), 0)
edged = cv2.Canny(gray, 75, 200)
cv2.imwrite('scan_edge.jpg', edged)


(cnts, _) = cv2.findContours(edged.copy(), cv2.RETR_LIST,
cv2.CHAIN_APPROX_SIMPLE)
cnts = sorted(cnts, key = cv2.contourArea, reverse = True)[:5]
for c in cnts:
   peri = cv2.arcLength(c, True)
   approx = cv2.approxPolyDP(c, 0.02 * peri, True)
   if len(approx) == 4:
      screenCnt = approx
      break
cv2.drawContours(image,
        [screenCnt], -1, (0, 255, 0), 2)
cv2.imwrite('scan_contours.jpg', image)


warped = four_point_transform(orig, screenCnt.reshape(4, 2) *
ratio)
warped = cv2.cvtColor(warped, cv2.COLOR_BGR2GRAY)
warped = threshold_adaptive(warped, 251, offset = 10)
warped = warped.astype("uint8") * 255
cv2.imwrite('scanned.jpg', warped)

print "Printing the contents of the image:"
from PIL import Image
img = Image.open("scanned.jpg")
import pytesseract
print(pytesseract.image_to_string(img))
```

Once you run the Python code, you will see three images being created on your hard disk. The first one is the edge-detected image, which, in my case, looks as follows. It's saved as `scan_edge.jpg`:



3. Next, we get another image, which detects the whole area of the image where text is found. This is called the contours image and is generated as `scan_contours.jpg`. See how it highlights the portion of the image where text is available:

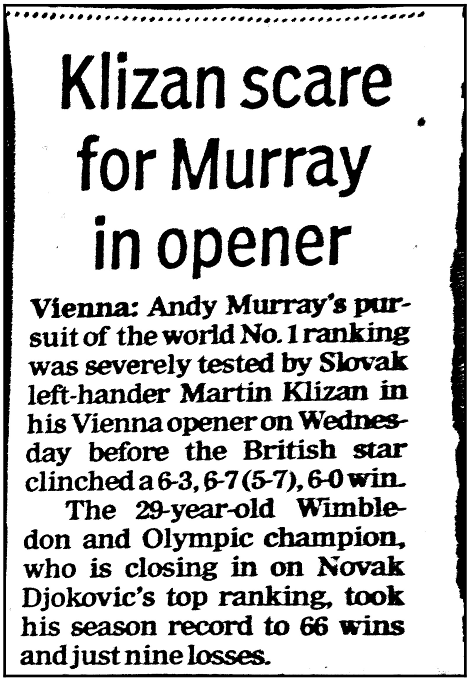


Klizan scare
for Murray
in opener

Vienna: Andy Murray's pursuit of the world No. 1 ranking was severely tested by Slovak left-hander Martin Klizan in his Vienna opener on Wednesday before the British star clinched a 6-3, 6-7 (5-7), 6-0 win.

The 29-year-old Wimbledon and Olympic champion, who is closing in on Novak Djokovic's top ranking, took his season record to 66 wins and just nine losses.

4. Finally, we get the scanned copy of our news article on Andy Murray, and it is saved as `scanned.jpg`. Look how well the news article got scanned in the following screenshot:



Klizan scare for Murray in opener

Vienna: Andy Murray's pursuit of the world No. 1 ranking was severely tested by Slovak left-hander Martin Klizan in his Vienna opener on Wednesday before the British star clinched a 6-3, 6-7 (5-7), 6-0 win.

The 29-year-old Wimbledon and Olympic champion, who is closing in on Novak Djokovic's top ranking, took his season record to 66 wins and just nine losses.

Cool, this is the exact thing we wanted to achieve when we first started, isn't it? Give this to Peter and he will be very happy. He might be thinking of some costly consulting companies or scanners to do this job, while we could do it quickly and that too for free!

5. As if this was not enough, we have done more for Peter here. If you run the program, you will also get a text output, which gives out the text of the complete article. Using this text, Peter can choose to classify the documents and index them accordingly:

```
Printing the contents of the image:
Vienna: Andy Murray's pm"
suitof the world No. Iranking
was severely tested by Slovak
lefi-hander Martin Klizan in
his Vienna openeron Wednwo
day before the British star
```

```
clinched a 6-3, 64 (57), 6-0 win.
The 29-year-old Wimble-
don and Olympic champion,
who is closing in on Novak
, Djokovic's top ranking, took
his season record to 66 wins
and just ninelosses. ﬂ
```

Awesome! Let's look at the internals of our program in the “How it works” section.

How it works…

In our recipe, we started by taking a picture of our newspaper article. We named this as `murray.jpg`. I used a simple camera phone to take this image. We then went ahead and read the image using the `opencv` module's `imread()` method.

We also computed the ratio of the original height to the new height, cloned the original image, and resized it. We used the `copy()` method to clone our image, and we used the `resize()` method to resize it to a new height.

We then converted the image to the grayscale format with the `cvtColor()` method and then applied the Gaussian filter to blur the image.

We subjected the blurred image to detect the edges of the text using the `Canny()` method and finally stored the edge-detected image as `scan_edge.jpg`.

Next, we found the contours of the image using the `findContours()` method and stored the outline of the image as `scan_contours.jpg`.

We then ran a couple of transforms on the image. The four-point transform helped us get a top-down view of the original image. For this, we used the `four_point_transform()` method.

We also converted the image to grayscale and then thresholded it to give a black and white paper style feel. The `cvtColor()` method converts the image to grayscale and the `threshold_adaptive()` method applies the appropriate thresholds. And we're done; the image is now ready and is already scanned and saved as `scanned.jpg`.

But, as we saw in the previous section, we also printed the text of the newspaper column. This we could achieve by first reading the scanned image with Pillow's `Image` class and with the `pytessaract` module by using the `image_to_string()` method on the image object.

Cool, so we have automated the business process of converting paper-based documents into the electronic format and added a facility to index the files so that they can be easily fed into the company's ERP processes. Peter is damn happy with you! Congrats!

There's more…

We looked at OCR to extract text data from scanned images, but there are many more things that can be achieved, such as intelligent character recognition (extracts hand-written text) and barcode recognition (recognition of many types of barcodes), among others. Also, in this chapter we haven't dealt much with image filtering. If you're really interested, you can do lot of reading on these topics, which is beyond the scope of this chapter.

10

Data Analysis and Visualizations

You have so much of data and its just lying around? Ever wondered how you could easily analyze data and generate insights? Curious about the data analysis process? Well, you are at the right place!

In this chapter, we will cover the following recipes:

- Reading, selecting, and interpreting data with visualizations
- Generating insights with data filtering and aggregation
- Automating social media analysis for businesses

Introduction

> *In God we trust. All others must bring data*
>
> *-W. Edwards Demming, Statistician*

Today, businesses heavily rely on data to get insights into what customers need, what channels they will use to buy, and so on. This way, businesses can take informed decisions about launching a new product or coming up with new offers. But how do businesses achieve this? What does decision making actually involve?

Data-based decision making refers to the process of data inspection, scrubbing or cleaning, data transformation, and generating models on top of data for the purpose of generating insights, discovering useful information, and drawing conclusions. For instance, an e-commerce company would use this process to analyze consumer buying patterns and suggest appropriate time slots for coming up with promotional offers for a select group of products. In fact, businesses analyze static or real-time data for multiple purposes, such as generating trends, building forecast models, or simply to extract structured information from raw data. Data analysis has multiple facets and approaches and can be briefly categorized under business intelligence, predictive analytics, and text mining.

Business intelligence (**BI**) is capable of handling large amounts of structured and, sometimes, unstructured data to allow for the easy interpretation of these large volumes of data. Identifying new opportunities based on insights into data can provide businesses with a competitive advantage and stability.

Predictive analytics encompasses the application of various statistical models, such as machine learning, to analyze historical data and current trends, in order to come up with predictions for future or unknown events. It involves generating models and capturing relationships among data features for risk assessment and decision making.

Text analytics is the process of deriving quality information from structured or unstructured text data. Text analytics involves linguistic, statistical, and contextual techniques to extract and classify information for businesses.

However, data-based decisions are not easy and cannot be taken in a jiffy. Data-based decision making is a step-by-step process involving multiple operations. Let's understand the complete process in detail in the next section.

Steps to data-based decision making

At a high level, the process can be categorized into the following phases. Of course, you can customize the process to suit your objectives:

- **Define hypothesis and data requirements**: Before you start the process, you should be clear with your business goals–they should be **Specific, Measurable, Acceptable, Relevant, and Timely** (**SMART**). You don't want to start collecting data without being clear about the problem you're solving. As far as possible, come up with clear problem statements, such as “What has been the trend for mobile sales in the consumer space for the last three quarters?” Or something futuristic such as “Will I be able to sell electronic goods this winter with a profit margin of 150%?” Can you come up with a statement like this for your company?

- **Data source**: You should also be clear about the source of your data. Are you relying on your own company database to perform the data analysis? Are you also relying on any third-party market research or trends to base your analysis on? If you are using third-party data, how do you plan to extract the data from the source (possibly through an API) and put it in your data store?
- **Data collection**: Now that you're clear about what you want to generate insights into, the next step is to collect data in the required format. For instance, if you want data on the trend of mobile sales, you should collect data for the factors that influence mobile sales, such as new product introductions (product), offers (price), payment options, and date/time of purchase, among other relevant factors. Also, you should have an agreeable or a standard way of storing data; for instance, I may store the mobile sales per unit in USD and not in EUR, or I may store the sales in days and not in hours. Identifying a representative sample is really useful in such cases. Representative samples accurately reflect the entire population and definitely help in analysis.
- **Data transformation**: Now you know where to collect the data from and in what format, it's time to decide where you want to load the data. It could be a plain old CSV or an SQL database; you need to know beforehand so that you can organize the data in the best way and get ready for analysis. This step can be referred to as transformation, as it involves extracting data from the source data system to a destination data system. In large scales, data is stored in a data warehouse system.
- **Data cleansing**: Once the data is processed and organized, it's time to look at the data sanity. Transformed data may be incompatible, contain duplicates, or may at least contain measurement, sampling, and data entry errors. Data cleansing involves the removal of inaccurate data, adding default values for missing data, removing outliers, and resolving any other data inconsistency issues. You really have to be careful while dumping outliers; you should decide on the ways you want to remove them–is it a simple deletion of records or imputing them with the mean/mode of the other observations? You're the best decision maker in this case.
- **Data analysis**: Once we have the data cleansed and ready for use, it's time for deeper analysis. You can analyze the data for business intelligence, or generate predictive models, using statistical techniques such as Logistic Regression. You could also perform text analysis on top of it to generate insights and arrive at decisions.

- **Data visualization**: Once the analysis is done, it can be reported in many formats so that the analysis can be effectively communicated to the audience. Data visualization uses information display, such as tables and charts, to help communicate key messages contained in the data. Visualizations also help users to interpret their assumptions and generate meaningful information from the analysis.
- **Data interpretation and feedback**: This phase helps you answer three main questions. Does the analysis answer the questions you began with? Does it help you validate your assumptions, that is, accept or reject your hypothesis? Do you need more data to improve your models or conclusions? It's not complete until your conclusions don't flow back into the system. The feedback loop makes sure that the predictive models are enriched and trained well for future use.

OK, that's a good start! I think you must have got a fair idea of the complete process: data collection to generating insights. You will realize that a few of these steps, such as defining objectives, data collection, and transforming data, are custom to the market context and the problem being solved.

In this chapter, we will focus on a few generic aspects, such as collecting real-time data, reading data, performing data analysis, and data visualization. We will take a look at the popular Python modules that will help us read the data efficiently and analyze the data to generate insights. You will also learn about the Python modules that help interpret data and generate visualizations (charts).

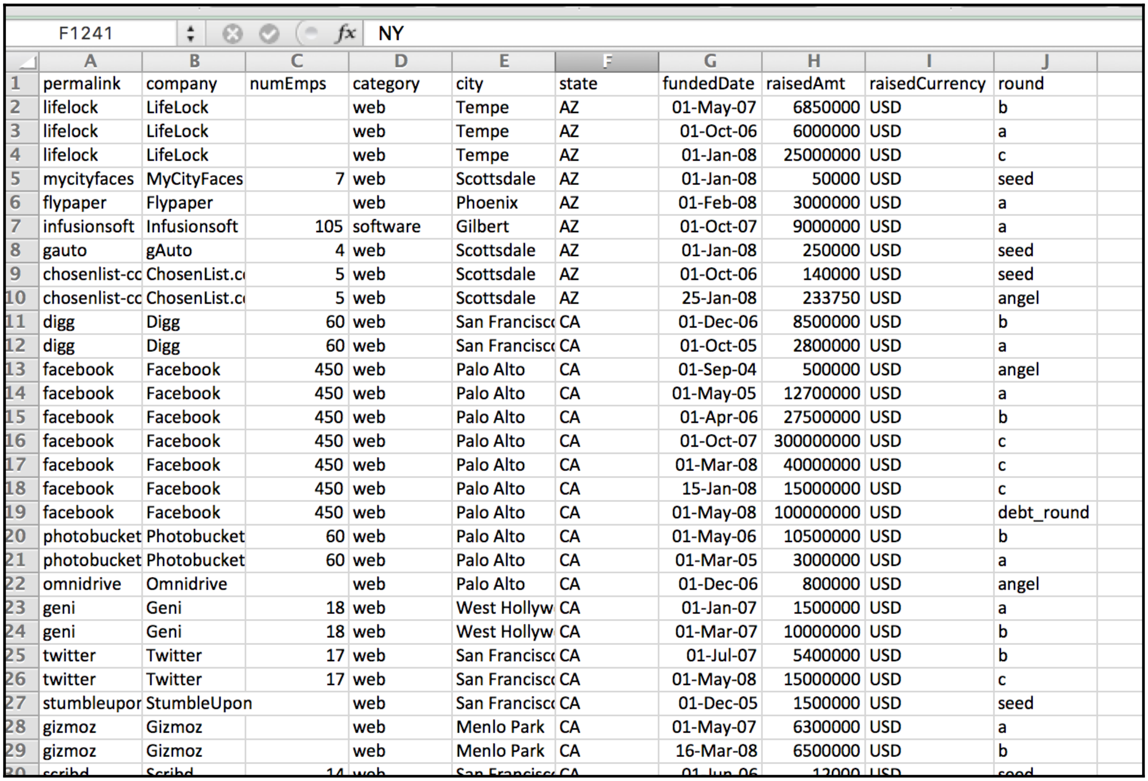
At the end of this chapter, we will also look at a typical business process that can be automated with the knowledge we built with the recipes covered in the chapter. This chapter will help you start your journey as a data scientist, but doesn't cover extensive topics, such as statistical techniques or predictive modeling.

During the course of this chapter, we will use the following Python modules:

- `pandas` (`http://pandas.pydata.org/pandas-docs/version/0.15.2/tutorials.html`)
- `numpy` (`http://www.numpy.org/`)
- `matplotlib` (`http://matplotlib.org/`)
- `seaborn` (`https://pypi.python.org/pypi/seaborn/`)

Reading, selecting, and interpreting data with visualizations

In this recipe, we will have the help of a known dataset. We will use TechCrunch's Continental USA CSV file that contains the listing of 1,460 company funding rounds. This is how it looks. It contains data points, such as the company name, number of employees, funding date, amounts raised, and type of funding (series A or angel funding):



F1241 fx NY

	A	B	C	D	E	F	G	H	I	J
1	permalink	company	numEmps	category	city	state	fundedDate	raisedAmt	raisedCurrency	round
2	lifelock	LifeLock		web	Tempe	AZ	01-May-07	6850000	USD	b
3	lifelock	LifeLock		web	Tempe	AZ	01-Oct-06	6000000	USD	a
4	lifelock	LifeLock		web	Tempe	AZ	01-Jan-08	25000000	USD	c
5	mycityfaces	MyCityFaces	7	web	Scottsdale	AZ	01-Jan-08	50000	USD	seed
6	flypaper	Flypaper		web	Phoenix	AZ	01-Feb-08	3000000	USD	a
7	infusionsoft	Infusionsoft	105	software	Gilbert	AZ	01-Oct-07	9000000	USD	a
8	gauto	gAuto	4	web	Scottsdale	AZ	01-Jan-08	250000	USD	seed
9	chosenlist-cc	ChosenList.c	5	web	Scottsdale	AZ	01-Oct-06	140000	USD	seed
10	chosenlist-cc	ChosenList.c	5	web	Scottsdale	AZ	25-Jan-08	233750	USD	angel
11	digg	Digg	60	web	San Francisc	CA	01-Dec-06	8500000	USD	b
12	digg	Digg	60	web	San Francisc	CA	01-Oct-05	2800000	USD	a
13	facebook	Facebook	450	web	Palo Alto	CA	01-Sep-04	500000	USD	angel
14	facebook	Facebook	450	web	Palo Alto	CA	01-May-05	12700000	USD	a
15	facebook	Facebook	450	web	Palo Alto	CA	01-Apr-06	27500000	USD	b
16	facebook	Facebook	450	web	Palo Alto	CA	01-Oct-07	300000000	USD	c
17	facebook	Facebook	450	web	Palo Alto	CA	01-Mar-08	40000000	USD	c
18	facebook	Facebook	450	web	Palo Alto	CA	15-Jan-08	15000000	USD	c
19	facebook	Facebook	450	web	Palo Alto	CA	01-May-08	100000000	USD	debt_round
20	photobucket	Photobucket	60	web	Palo Alto	CA	01-May-06	10500000	USD	b
21	photobucket	Photobucket	60	web	Palo Alto	CA	01-Mar-05	3000000	USD	a
22	omnidrive	Omnidrive		web	Palo Alto	CA	01-Dec-06	800000	USD	angel
23	geni	Geni	18	web	West Hollyw	CA	01-Jan-07	1500000	USD	a
24	geni	Geni	18	web	West Hollyw	CA	01-Mar-07	10000000	USD	b
25	twitter	Twitter	17	web	San Francisc	CA	01-Jul-07	5400000	USD	b
26	twitter	Twitter	17	web	San Francisc	CA	01-May-08	15000000	USD	c
27	stumbleupor	StumbleUpon		web	San Francisc	CA	01-Dec-05	1500000	USD	seed
28	gizmoz	Gizmoz		web	Menlo Park	CA	01-May-07	6300000	USD	a
29	gizmoz	Gizmoz		web	Menlo Park	CA	16-Mar-08	6500000	USD	b
30	scribd	Scribd	14	web	San Francisc	CA	01-Jun-06	12000	USD	seed

1. Now, let's install the modules that we will use to read and select the data from this CSV file. Before doing that, we will set up a virtual environment and activate it:

   ```
   chetans-MacBookPro:ch11 Chetan$ virtualenv analyze
   New python executable in analyze/bin/python2.7
   Also creating executable in analyze/bin/python
   Installing setuptools, pip, wheel...done.
   chetans-MacBookPro:ch11 Chetan$ source analyze/bin/activate
   (analyze)chetans-MacBookPro:ch11 Chetan$
   ```

2. OK, cool! Now, let's install `pandas`. We will use `pandas` to read our CSV file and select the data to perform analysis. We install `pandas` with our favorite utility, `python-pip`. The following are the installation logs for `pandas` on my Mac OSX:

   ```
   (analyze)chetans-MacBookPro:ch11 Chetan$ pip install pandas
   Collecting pandas
   Collecting pytz>=2011k (from pandas)
     Using cached pytz-2016.7-py2.py3-none-any.whl
   Collecting python-dateutil (from pandas)
     Using cached python_dateutil-2.6.0-py2.py3-none-any.whl
   Collecting numpy>=1.7.0 (from pandas)
   Collecting six>=1.5 (from python-dateutil->pandas)
     Using cached six-1.10.0-py2.py3-none-any.whl
   Installing collected packages: pytz, six,
   python-dateutil, numpy, pandas
   Successfully installed numpy-1.11.2 pandas-0.19.1
   python-dateutil-2.6.0 pytz-2016.7 six-1.10.0
   ```

Installing the `pandas` module also installs the `numpy` module for me. In fact, I had installed these modules on my machine earlier as well; hence, a lot of these modules get picked up from cache. On your machine, the installation logs may differ.

3. Next, let's install `matplotlib` and `seaborn`; libraries that will be used by us for visualizations. The following are the installation logs on my machine, first for `matplotlib`:

   ```
   (analyze)chetans-MacBookPro:ch11 Chetan$ pip install matplotlib
   Collecting matplotlib
   Requirement already satisfied (use --upgrade to upgrade):
     numpy>=1.6 in ./analyze/lib/python2.7/site-packages
     (from matplotlib)
   Requirement already satisfied (use --upgrade to upgrade):
     pytz in ./analyze/lib/python2.7/site-packages
     (from matplotlib)
   ```

```
Requirement already satisfied (use --upgrade to upgrade):
  python-dateutil in ./analyze/lib/python2.7/site-packages
  (from matplotlib)
Collecting cycler (from matplotlib)
  Using cached cycler-0.10.0-py2.py3-none-any.whl
Collecting pyparsing!=2.0.0,!=2.0.4,!=2.1.2,>=1.5.6
(from matplotlib)
  Using cached pyparsing-2.1.10-py2.py3-none-any.whl
Requirement already satisfied (use --upgrade to upgrade):
six>=1.5 in ./analyze/lib/python2.7/site-packages
(from python-dateutil->matplotlib)
Installing collected packages: cycler, pyparsing,
matplotlib
Successfully installed cycler-0.10.0
matplotlib-1.5.3 pyparsing-2.1.10
```

As you can see these modules are installed on my machine, so the installation logs may differ when you install these modules for the first time on your machine. Here are the logs for seaborn:

```
(analyze)chetans-MacBookPro:ch11 Chetan$ pip install seaborn
Collecting seaborn
Collecting scipy (from seaborn)
Requirement already satisfied (use --upgrade to upgrade):
numpy>=1.7.1 in ./analyze/lib/python2.7/site-packages
(from scipy->seaborn)
Installing collected packages: scipy, seaborn
Successfully installed scipy-0.18.1 seaborn-0.7.1
```

How to do it…

1. First, let's just download the CSV file from `https://support.spatialkey.com/spatialkey-sample-csv-data/`. The direct download link for the TechCrunch file is `http://samplecsvs.s3.amazonaws.com/TechCrunchcontinentalUSA.csv`. You can download this file with the `wget` command as follows:

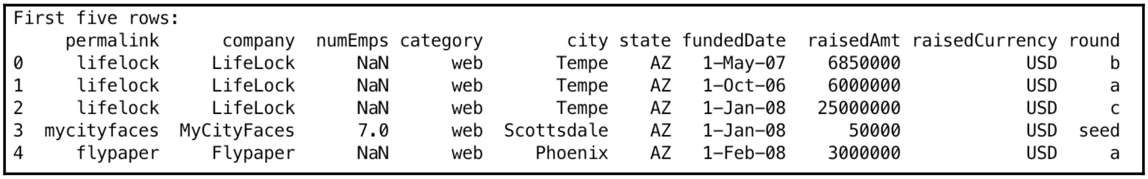
```
(analyze)chetans-MacBookPro:ch11 Chetan$
  wget http://samplecsvs.s3.amazonaws.com/
  TechCrunchcontinentalUSA.csv
--2016-11-20 16:01:57--
  http://samplecsvs.s3.amazonaws.com/
  TechCrunchcontinentalUSA.csv
Resolving samplecsvs.s3.amazonaws.com... 54.231.97.224
Connecting to samplecsvs.s3.amazonaws.com
```

```
  |54.231.97.224|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 93536 (91K) [application/x-csv]
Saving to: 'TechCrunchcontinentalUSA.csv'
TechCrunchcontinentalUSA.csv             100%
[=======================================================
 =======================================>]
                             91.34K  20.3KB/s   in 4.5s
2016-11-20 16:02:03 (20.3 KB/s) -
'TechCrunchcontinentalUSA.csv' saved [93536/93536]
```

2. Now, let's go ahead and write our first piece of Python code to read the CSV file. We read the CSV file and print the first five rows:

```
import pandas as pd
pd.set_option('display.line_width', 5000)
pd.set_option('display.max_columns', 60)
df = pd.read_csv('TechCrunchcontinentalUSA.csv')
print "First five rows:\n", df[:5]
```

In the preceding code example, we read the first five records of the CSV file:

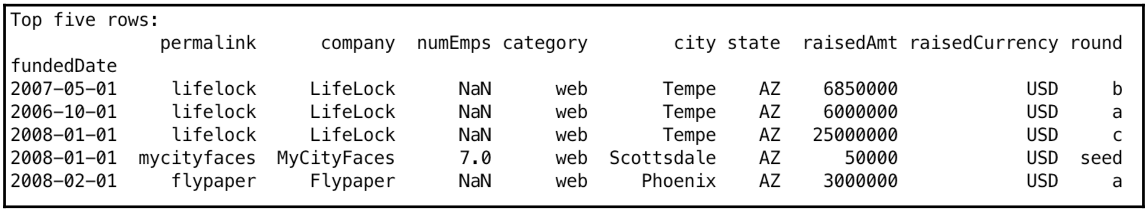


```
First five rows:
     permalink      company  numEmps category        city state fundedDate  raisedAmt raisedCurrency round
0     lifelock     LifeLock      NaN      web       Tempe    AZ   1-May-07    6850000            USD     b
1     lifelock     LifeLock      NaN      web       Tempe    AZ   1-Oct-06    6000000            USD     a
2     lifelock     LifeLock      NaN      web       Tempe    AZ   1-Jan-08   25000000            USD     c
3  mycityfaces  MyCityFaces      7.0      web  Scottsdale    AZ   1-Jan-08      50000            USD  seed
4     flypaper     Flypaper      NaN      web     Phoenix    AZ   1-Feb-08    3000000            USD     a
```

3. The `pandas` module reads the file's contents and converts them to a data frame of rows and columns. Now, if you look at the output of the preceding code, you will notice that an index column gets added to the file contents. With `pandas`, it's easy to parse the date, tell if the dates in our CSV file have the date first or month first (UK or US format), and make the date column as the index column:

```
import pandas as pd

pd.set_option('display.line_width', 5000)
pd.set_option('display.max_columns', 60)

df = pd.read_csv('TechCrunchcontinentalUSA.csv',
          index_col='fundedDate', \
          parse_dates=['fundedDate'], dayfirst=True,)
print "Top five rows:\n", df[:5]
```

If you run the preceding code snippet, you should be able to see the index column, **fundedDate**, as shown in the following screenshot:

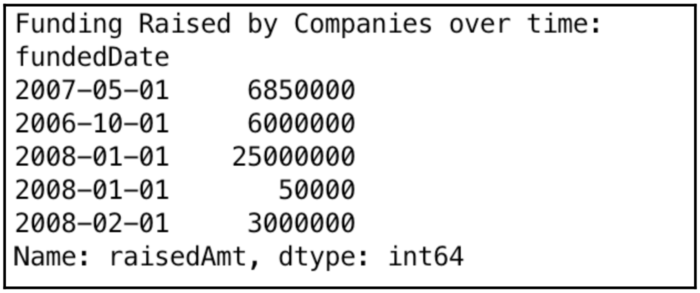


```
Top five rows:
              permalink      company  numEmps category        city state  raisedAmt raisedCurrency round
fundedDate
2007-05-01     lifelock     LifeLock      NaN      web       Tempe    AZ    6850000            USD     b
2006-10-01     lifelock     LifeLock      NaN      web       Tempe    AZ    6000000            USD     a
2008-01-01     lifelock     LifeLock      NaN      web       Tempe    AZ   25000000            USD     c
2008-01-01  mycityfaces  MyCityFaces      7.0      web  Scottsdale    AZ      50000            USD  seed
2008-02-01     flypaper     Flypaper      NaN      web     Phoenix    AZ    3000000            USD     a
```

4. Neat! Now, we're able to read the data, but how about selecting some data so that we can perform some analysis on top of it. Let's select the column that depicts the amount of funding raised by the companies (the **raisedAmt** column):

```
import pandas as pd

pd.set_option('display.line_width', 5000)
pd.set_option('display.max_columns', 60)
df = pd.read_csv('TechCrunchcontinentalUSA.csv',
          index_col='fundedDate', \
          parse_dates=['fundedDate'], dayfirst=True,)

raised = df['raisedAmt'][:5]
print "Funding Raised by Companies over time:\n", raised
```

Note that in the following screenshot, we have printed the top five records of the companies that have raised funding:

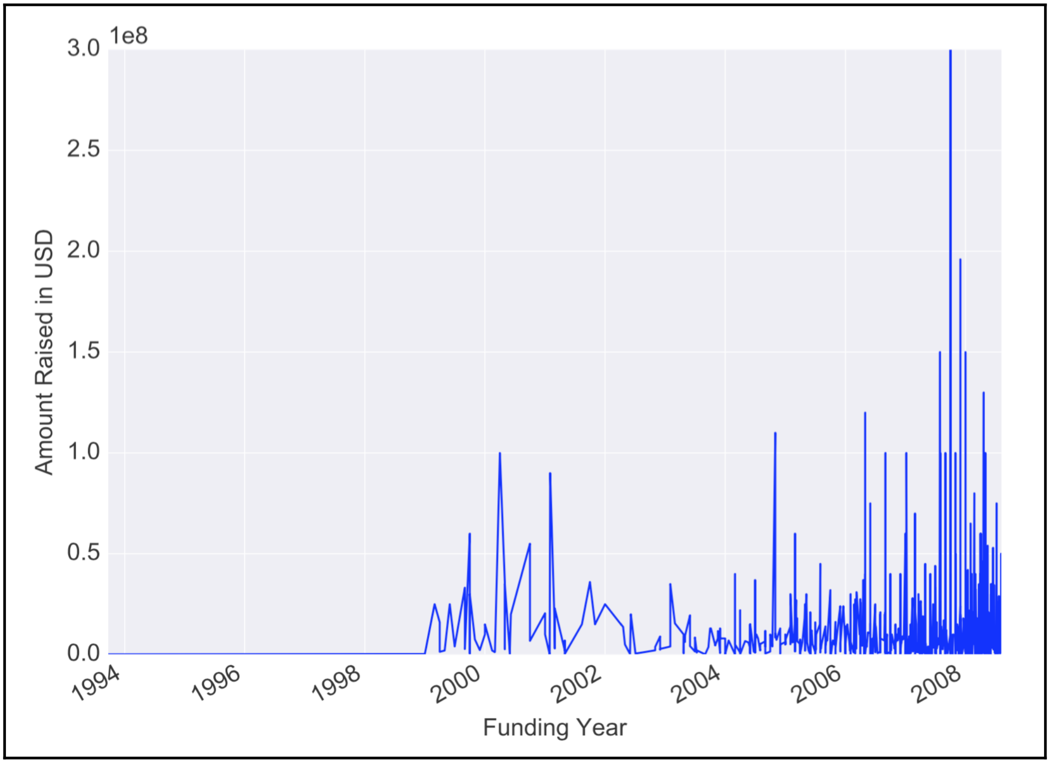


```
Funding Raised by Companies over time:
fundedDate
2007-05-01     6850000
2006-10-01     6000000
2008-01-01    25000000
2008-01-01       50000
2008-02-01     3000000
Name: raisedAmt, dtype: int64
```

5. OK, cool! So we can select the column of our choice and get the data we wanted to analyze. Let's see if we can generate some nice visualizations for it. The following recipe generates a line chart for the funding rounds reported for all the years (*x* axis), based on the amount raised (*y* axis):

```
import pandas as pd
from matplotlib import pyplot as plt
import seaborn as sns

plt.style.use('default')
pd.set_option('display.line_width', 5000)
pd.set_option('display.max_columns', 60)

df = pd.read_csv('TechCrunchcontinentalUSA.csv')
print "First five rows:\n", df[:5]

df = pd.read_csv('TechCrunchcontinentalUSA.csv',
          index_col='fundedDate', \
          parse_dates=['fundedDate'], dayfirst=True,)
print "Top five rows:\n", df[:5]
raised = df['raisedAmt'][:5]
print "Funding Raised by Companies over time:\n", raised

sns.set_style("darkgrid")
sns_plot = df['raisedAmt'].plot()
plt.ylabel("Amount Raised in USD");
plt.xlabel("Funding Year")
plt.savefig('amountRaisedOverTime.pdf')
```

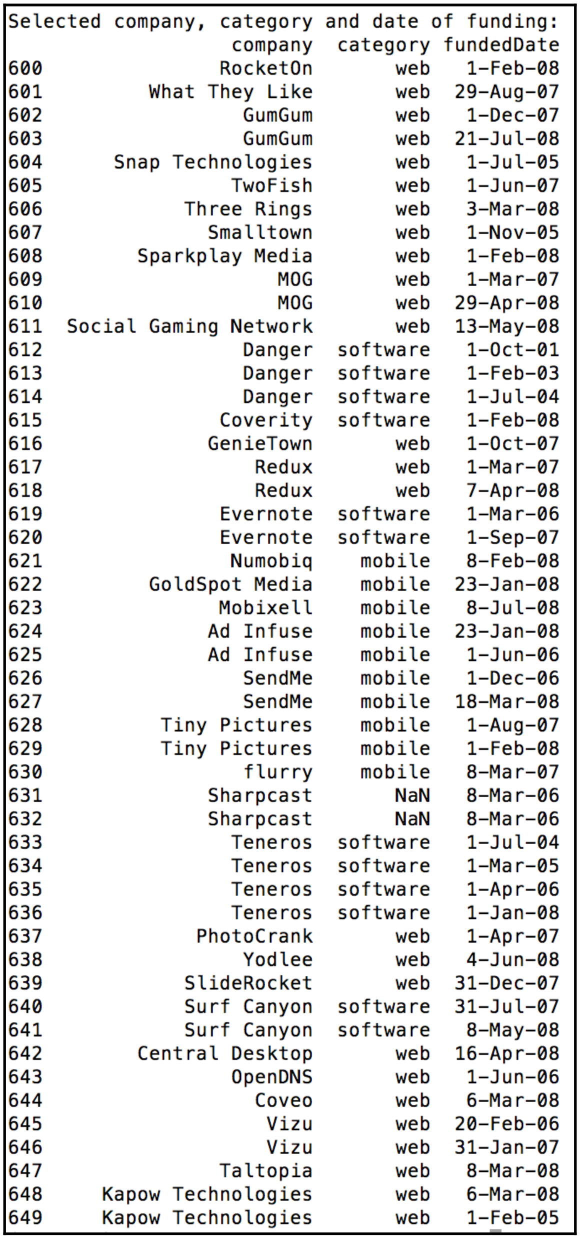
In the following screenshot, see how the rate of funding (or the rate of reporting) increased, and with that, the amounts raised also saw a steady increase!



6. Fantastic! I know you have already started to like what we're doing here. Let's move forward and see whether we can select multiple columns from the CSV file. In the following example, we get the data for 50 rows, with the column names being **company**, **category**, and **fundedDate**:

```
import pandas as pd
from matplotlib import pyplot as plt
plt.style.use('default')
pd.set_option('display.line_width', 5000)
pd.set_option('display.max_columns', 60)

fundings = pd.read_csv('TechcrunchcontinentalUSA.csv')
print "Type of funding:\n", fundings[:5]['round']

# Selecting multiple columns
print "Selected company, category and date of
       funding:\n",\
fundings[['company', 'category',
       'fundedDate']][600:650]
```

The output of the preceding code snippet is as follows:



```
Selected company, category and date of funding:
                   company  category fundedDate
600               RocketOn       web   1-Feb-08
601         What They Like       web  29-Aug-07
602                 GumGum       web   1-Dec-07
603                 GumGum       web  21-Jul-08
604      Snap Technologies       web   1-Jul-05
605                TwoFish       web   1-Jun-07
606            Three Rings       web   3-Mar-08
607              Smalltown       web   1-Nov-05
608        Sparkplay Media       web   1-Feb-08
609                    MOG       web   1-Mar-07
610                    MOG       web  29-Apr-08
611  Social Gaming Network       web  13-May-08
612                 Danger  software   1-Oct-01
613                 Danger  software   1-Feb-03
614                 Danger  software   1-Jul-04
615               Coverity  software   1-Feb-08
616              GenieTown       web   1-Oct-07
617                  Redux       web   1-Mar-07
618                  Redux       web   7-Apr-08
619               Evernote  software   1-Mar-06
620               Evernote  software   1-Sep-07
621                Numobiq    mobile   8-Feb-08
622         GoldSpot Media    mobile  23-Jan-08
623               Mobixell    mobile   8-Jul-08
624              Ad Infuse    mobile  23-Jan-08
625              Ad Infuse    mobile   1-Jun-06
626                 SendMe    mobile   1-Dec-06
627                 SendMe    mobile  18-Mar-08
628          Tiny Pictures    mobile   1-Aug-07
629          Tiny Pictures    mobile   1-Feb-08
630                 flurry    mobile   8-Mar-07
631              Sharpcast       NaN   8-Mar-06
632              Sharpcast       NaN   8-Mar-06
633                Teneros  software   1-Jul-04
634                Teneros  software   1-Mar-05
635                Teneros  software   1-Apr-06
636                Teneros  software   1-Jan-08
637             PhotoCrank       web   1-Apr-07
638                 Yodlee       web   4-Jun-08
639            SlideRocket       web  31-Dec-07
640            Surf Canyon  software  31-Jul-07
641            Surf Canyon  software   8-May-08
642        Central Desktop       web  16-Apr-08
643                OpenDNS       web   1-Jun-06
644                  Coveo       web   6-Mar-08
645                   Vizu       web  20-Feb-06
646                   Vizu       web  31-Jan-07
647               Taltopia       web   8-Mar-08
648     Kapow Technologies       web   6-Mar-08
649     Kapow Technologies       web   1-Feb-05
```

7. OK, great! Now let's select one of these columns and perform some analysis on top of it. In the following code example, we select the **category** column that gives us the categories of all the reported funding rounds. We then process the selected column to get the most common category of the company that got funded:

```
import pandas as pd
from matplotlib import pyplot as plt
plt.style.use('default')

pd.set_option('display.line_width', 5000)
pd.set_option('display.max_columns', 60)

fundings = pd.read_csv('TechcrunchcontinentalUSA.csv')
print "Type of funding:\n", fundings[:5]['round']

# Selecting multiple columns
print "Selected company, category and date of funding:\n",\
    fundings[['company', 'category', 'fundedDate']][600:650]

# Most common category of company that got funded
counts = fundings['category'].value_counts()
print "Count of common categories of company
       that raised funds:\n", \
       counts
```

The output of the preceding code snippet is:

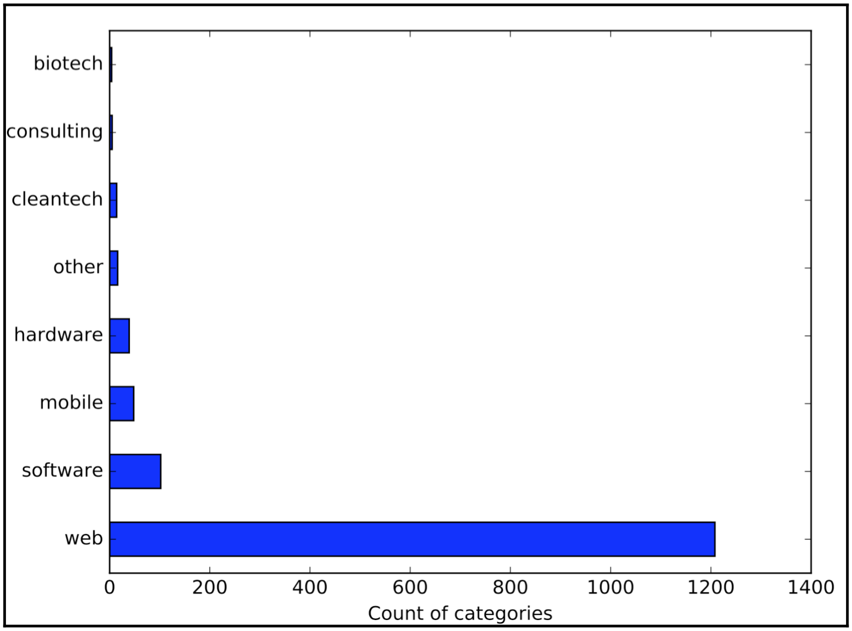
```
Count of common categories of company that raised funds:
web           1208
software       102
mobile          48
hardware        39
other           16
cleantech       14
consulting       5
biotech          4
Name: category, dtype: int64
```

8. Data and numbers give a lot of information, but the impact can actually only be seen through visualizations. Let's see whether we can plot the above data as a horizontal bar chart. The following recipe does the job for us:

```
import pandas as pd
from matplotlib import pyplot as plt
plt.style.use('default')
pd.set_option('display.line_width', 5000)
pd.set_option('display.max_columns', 60)
```

```
fundings = pd.read_csv('TechcrunchcontinentalUSA.csv')
print "Type of funding:\n", fundings[:5]['round']

# Selecting multiple columns
print "Selected company, category and date of funding:\n",\
    fundings[['company', 'category', 'fundedDate']][600:650]

# Most common category of company that got funded
counts = fundings['category'].value_counts()
print "Count of common categoris of company
       that raised funds:\n", \
       counts
counts.plot(kind='barh')
plt.xlabel("Count of categories")

plt.savefig('categoriesFunded.pdf')
```

On the *y* axis, we have the category of the company that got funded, and the *x* axis is the total count of companies in a given category. Also, we save the plotted chart in a PDF file named `categoriesFunded.pdf`:



Whoa! So many web companies got funded? Awesome! I too should start a web company–it increases the chances of getting funded.

How it works…

In this section we dealt with the two major aspects of data analysis. First, we covered how to read the dataset from a CSV file and select the appropriate data (rows or columns) from our dataset.

In the first code snippet, we used the help of the `pandas` module to read the CSV file. In `pandas`, we have the `read_csv(csv_file)` method that takes the CSV file path as an argument. The `pandas` module reads the file and stores the file contents as data frames. DataFrame is a two-dimensional, labeled-in-memory data structure with columns of potentially different types. It mimics the structure of a spreadsheet or an SQL table or a dictionary of series objects. It has a nice set of methods and attributes to select, index, and filter data. In our first recipe, we read the CSV file and generated a DataFrame object `df`. Using `df` we selected the first five rows of our CSV file with `df[:5]`. See how easy it becomes to select the rows of a CSV file with `pandas`.

We can do a few more things with the `read_csv()` method. By default, `pandas` adds another index column to our dataset, but we can specify which column from the CSV file should be used for indexing our data. We achieved this by passing the `index_col` parameter to the `read_csv()` method. We also converted the string dates present in the **fundedDate** column of the CSV file to a datetime format with the `parse_dates` parameter and said that the date is in a format where the day is the first part of the date with the `dayfirst` parameter.

After getting the DataFrame and using **fundedDate** as index, we used `df['raisedAmt'][:5]` to select the **raisedAmt** column and print the first five rows. We then used the `seaborn` library to set the style of our plot with `sns.set_style("darkgrid")` and generated the bar chart with the `plot()` method. The `seaborn` library is used for generating nice visualizations and is implemented on `matplotlib`.

Using the `matplotlib` library, we created an object, `plt`, which was then used to label our chart with the `ylabel()` and `xlabel()` methods. The `plt` object was also used to finally store the resulting chart in the PDF format with the `savefig()` method.

In the second example, we selected multiple columns with `fundings[['company', 'category' and 'fundedDate']]`. We selected three columns from a CSV file in one line of code. We then plotted a horizontal bar chart with the `plot()` method and specified the type of chart with `kind=barh`. Finally, we made use of the `matplotlib` library to label the *x* axis with the `xlabel()` method and saved the chart with the `savefig()` method. As you can see, we didn't have to use the `seaborn` library to plot the chart; we could simply do it with `matplotlib`.

Generating insights using data filtering and aggregation

Reading CSV files and selecting multiple columns is easy with `pandas`. In this section, we will take a look at how to slice and dice data, essentially filtering data with `pandas`.

Getting ready

In this section, we will use the same set of libraries (the following ones) that we used in the previous recipe:

- `pandas` for filtering and data analysis
- `matplotlib` and `seaborn` for plotting charts and saving the data in a PDF file

How to do it…

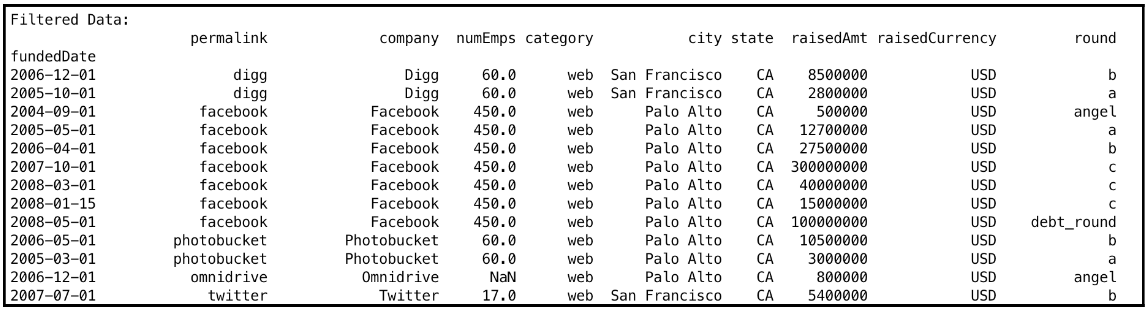
1. Let's start by importing the required libraries and reading the CSV file with the `read_csv()` method. The following code carries out the operations:

```
import pandas as pd

from matplotlib import pyplot as plt
import seaborn as sns
plt.style.use('default')

pd.set_option('display.line_width', 5000)
pd.set_option('display.max_columns', 60)

fundings = pd.read_csv(
           'TechcrunchcontinentalUSA.csv',
           index_col='fundedDate', \
           parse_dates=['fundedDate'], dayfirst=True)
```

2. Now, let's filter on the data frame and use multiple columns to filter data. Say we filter funding records on the funding category, state, and the selected city in the state. This can be achieved with the following piece of code:

```
 import pandas as pd
 from matplotlib import pyplot as plt
 import seaborn as sns

 plt.style.use('default')

 pd.set_option('display.line_width', 5000)
 pd.set_option('display.max_columns', 60)

funding = pd.read_csv(
           'TechcrunchcontinentalUSA.csv',
           index_col='fundedDate', \
           parse_dates=['fundedDate'], dayfirst=True)

#Web fundings in CA
web_funding = funding['category'] == 'web'
in_CA = funding['state'] == 'CA'
in_city = funding['city'].isin(['Palo Alto',
                  'San Francisco', 'San Mateo',
                  'Los Angeles', 'Redwood City'])
```

The preceding piece of code returns all the funding records in the State of California (CA) for the cities Palo Alto, San Francisco, San Mateo, Los Angeles, and Redwood City for the web companies. The following is a partial screenshot of the output:

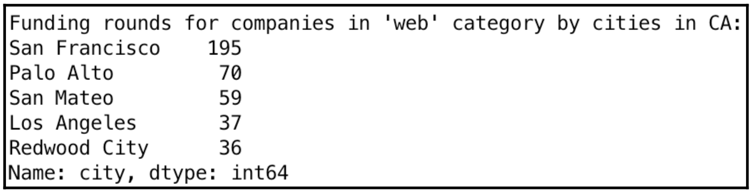


```
Filtered Data:
                permalink          company  numEmps category          city state  raisedAmt raisedCurrency        round
fundedDate
2006-12-01           digg             Digg     60.0      web  San Francisco    CA    8500000            USD            b
2005-10-01           digg             Digg     60.0      web  San Francisco    CA    2800000            USD            a
2004-09-01       facebook         Facebook    450.0      web      Palo Alto    CA     500000            USD        angel
2005-05-01       facebook         Facebook    450.0      web      Palo Alto    CA   12700000            USD            a
2006-04-01       facebook         Facebook    450.0      web      Palo Alto    CA   27500000            USD            b
2007-10-01       facebook         Facebook    450.0      web      Palo Alto    CA  300000000            USD            c
2008-03-01       facebook         Facebook    450.0      web      Palo Alto    CA   40000000            USD            c
2008-01-15       facebook         Facebook    450.0      web      Palo Alto    CA   15000000            USD            c
2008-05-01       facebook         Facebook    450.0      web      Palo Alto    CA  100000000            USD   debt_round
2006-05-01    photobucket      Photobucket     60.0      web      Palo Alto    CA   10500000            USD            b
2005-03-01    photobucket      Photobucket     60.0      web      Palo Alto    CA    3000000            USD            a
2006-12-01      omnidrive        Omnidrive      NaN      web      Palo Alto    CA     800000            USD        angel
2007-07-01        twitter          Twitter     17.0      web  San Francisco    CA    5400000            USD            b
```

3. OK cool, now let's see if we can get the count of funding for the companies in the web category by city names. The following code will get us the details we need:

```
web_funding = funding[web_funding & in_CA & in_city]
web_counts = web_funding['city'].value_counts()
print "Funding rounds for companies in 'web'
           category by cities in CA:\n", web_counts
```

The output of the preceding piece of code is the number of funding rounds received by the companies in the **web** category in the selected cities:



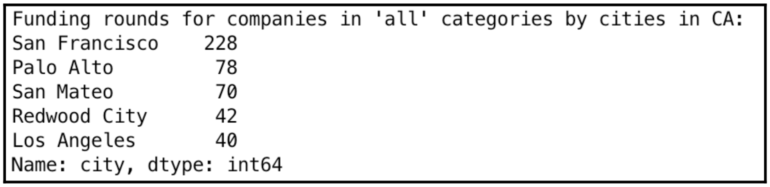
```
Funding rounds for companies in 'web' category by cities in CA:
San Francisco    195
Palo Alto         70
San Mateo         59
Los Angeles       37
Redwood City      36
Name: city, dtype: int64
```

Wow! The preceding analysis was quite useful; you got to know that the web companies in San Francisco city have received the funding 195 times (from the data we have). Looks like if you are a web company in San Francisco, all your funding worries are over. Well, that sounds logical and simple.

4. But wait, isn't this information incomplete? How about gathering data for companies in all categories including **web** and then representing the data for the companies in the **web** category as a percentage of all the categories? This way, we will know whether you should have your company in 'San Francisco' or in any other city. OK then, let's get the count of funding rounds for companies in all categories (including **web**), for all the selected cities in CA. The following code will get us the information we need:

```
total_funding = funding[in_CA & in_city]
total_counts = total_funding['city'].value_counts()
print "Funding rounds for companies in 'all'
       categories by cities in CA:\n",total_counts
```

Here is the output of the preceding code snippet:



```
Funding rounds for companies in 'all' categories by cities in CA:
San Francisco    228
Palo Alto         78
San Mateo         70
Redwood City      42
Los Angeles       40
Name: city, dtype: int64
```

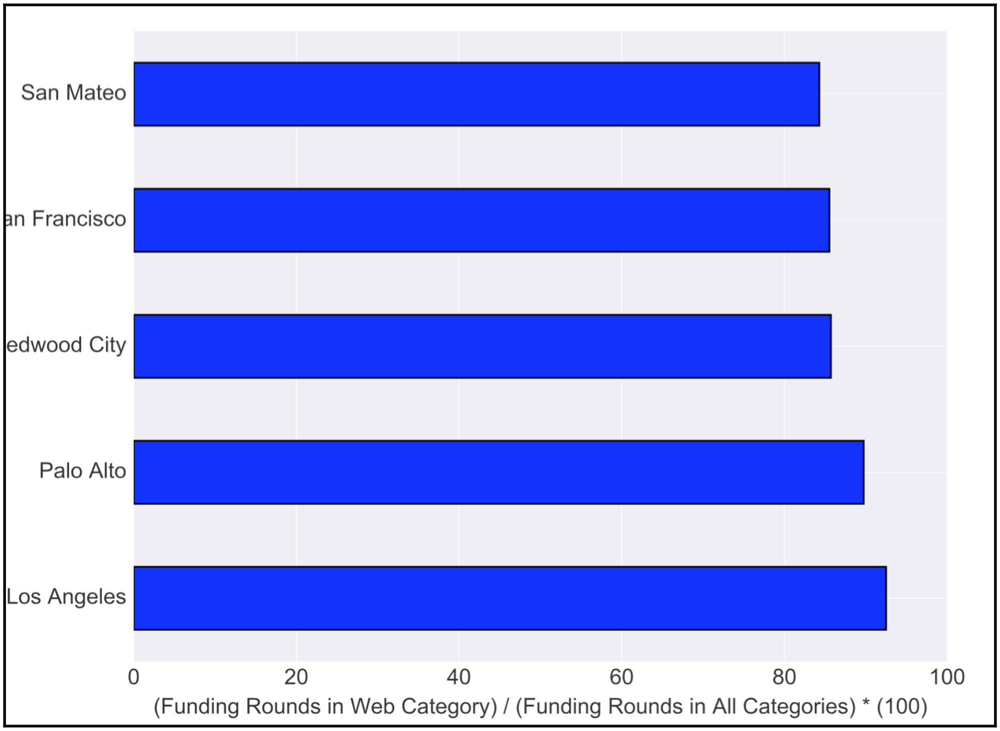
5. Nice! Now, let's get the data for the companies in the **web** category as a percentage of companies in all the categories for the selected cities of CA. We can get this data by simply dividing the data for the **web** categories by all the categories and then multiplying by 100 to represent it as a percentage. The following code snippet will help us in this case:

```
sns.set_style("darkgrid")
sns_plot = (web_counts*100/total_counts.astype(
                     float)).plot(kind='barh')
```

6. Now, let's plot this data as a horizontal bar chart with the following code:

```
plt.xlabel("(Funding Rounds in Web Category) / (
     Funding Rounds in All Categories) * (100)")
plt.savefig('webFundedByCity.pdf')
```

The following screenshot helps us compare the funding rounds for web companies with respect to the funding rounds for companies in all other categories and for the cities in California:



After the analysis what did you figure? Do you still want to set up your company in San Francisco? If you are a web company in Los Angeles, even though the funding rounds are limited, there is a higher chance (0.925) of your company getting funded than when you're in San Francisco (0.855), at least from the data points we have.

Now, let's take a look at another example. Say we want to analyze our data to see what months have historically received more funding than others. Additionally, can we also relate this with the rounds of funding [such as series A or angel funding]? Interesting thought! But how do we get there? The `pandas` module has support for grouping the data and data aggregation, which will come to our rescue for this analysis. Let's solve this problem step by step:

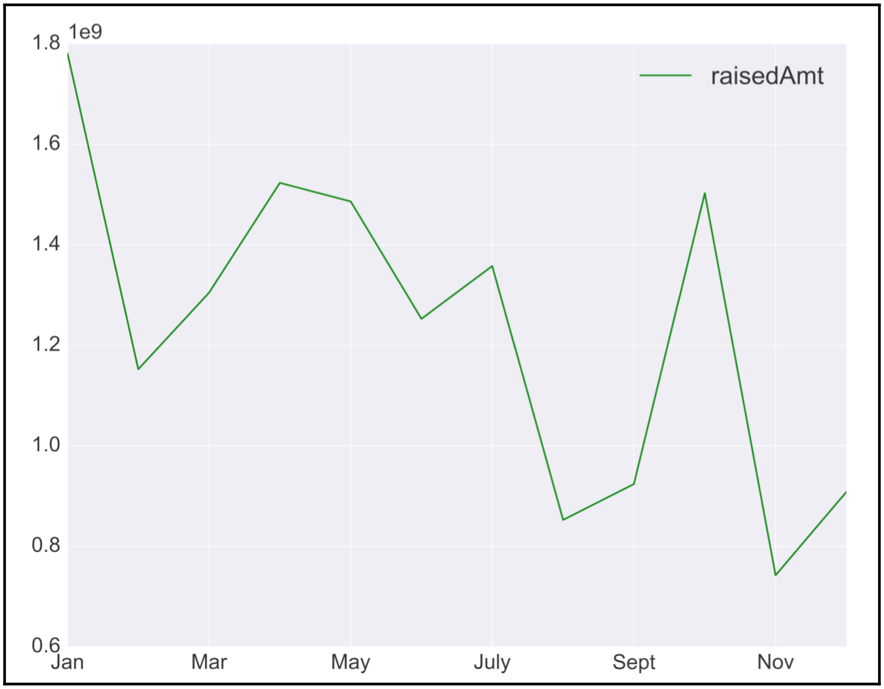
1. First, let's read the CSV file and select two columns: **raisedAmt** and **rounds**. Let's also add another column, `month`, as the index column to this data frame. The following code gets the data frame ready for further analysis:

```
import pandas as pd
from matplotlib import pyplot as plt
import seaborn as sns

plt.style.use('default')

pd.set_option('display.line_width', 5000)
pd.set_option('display.max_columns', 60)

df = pd.read_csv('TechCrunchcontinentalUSA.csv',
             index_col='fundedDate', \
             parse_dates=['fundedDate'], dayfirst=True,)

funds = df[['raisedAmt', 'round']]
funds['month'] = funds.index.month
print "Funding Rounds with Month Index:\n", funds
```

2. Now we need to get the data for the funds raised based on the month. The following code does exactly what we need:

```
funding_by_month = funds.groupby('month').aggregate('sum')
funding_by_month.index = ['Jan', 'Feb', 'Mar',
                         'Apr', 'May', 'June', 'July', \
                         'Aug', 'Sept', 'Oct', 'Nov', 'Dec']
print "Funding Rounds Grouped By Month:\n", funding_by_month
```

3. Now, if we plot the data for this analysis, we will see how the funding fluctuates on a monthly basis for all the years of data we have:

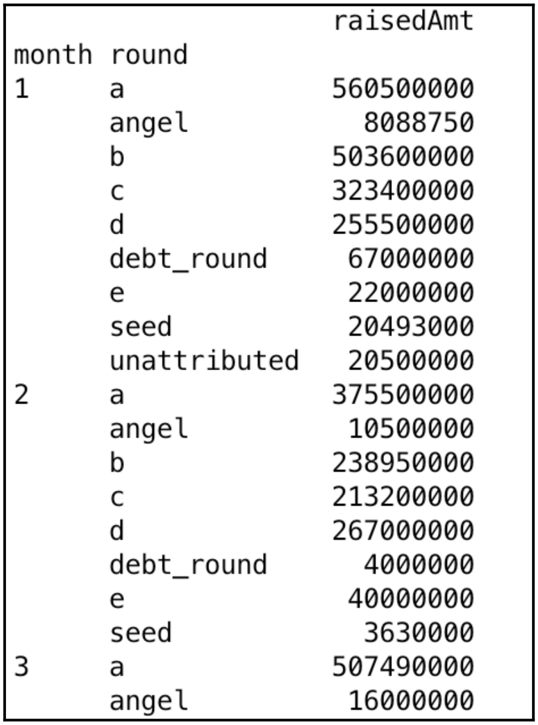


Cool, looks like it's better to ask for funding in January. Maybe, the investors are in a good mood after the Christmas and New Year vacation; what do you think?

4. Now, if you want to analyze and build a correlation between the month of year, amount raised, and the round of funding, we can get the data with the following piece of code:

```
funds['month'] = funds.index.month
funding_by_stage = funds.groupby(['month',
                    'round']).aggregate('sum')
print funding_by_stage
```

The output of the preceding code snippet is a data frame arranged as shown in the following screenshot. The data is grouped by **month** and **round** columns of funding, and **raisedAmt** is aggregated accordingly:



```
                    raisedAmt
month round
1     a             560500000
      angel           8088750
      b             503600000
      c             323400000
      d             255500000
      debt_round     67000000
      e              22000000
      seed           20493000
      unattributed   20500000
2     a             375500000
      angel          10500000
      b             238950000
      c             213200000
      d             267000000
      debt_round      4000000
      e              40000000
      seed            3630000
3     a             507490000
      angel          16000000
```

How it works…

For the first problem, we load the data from the CSV file as a data frame with the `read_csv()` method. Then, we filter the data based on multiple factors, where the state is `CA` and the company category is `web` and the cities are `Palo Alto`, `San Francisco`, `San Mateo`, `Los Angeles`, and `Redwood City`. Filtering is a column-based operation and is pretty straight forward; it gets you the relevant data frame after applying the criteria.

Then, we calculated the count of funding rounds, grouped by cities for the **web** companies with the `value_counts()` method. We did the same exercise for the funding rounds for companies in all the categories, including **web**.

Finally, we simply divided the data and got the data for the **web** companies as a percentage of the data for all the categories. The `pandas` module handled this operation for us seamlessly. It used both the data points for the same city for analysis without us even worrying about it.

Finally, we plotted the horizontal bar chart with the `plot()` method, depicted the percentages for each city individually, and got the insight we were looking for.

In the second example, we first got the data frame by selecting multiple columns: **raisedAmt** and **round**. We also added a new column to the **month** DataFrame and treated it as an index column.

Then, we grouped the data based on **month** with the help of the **groupby()** method. We then summed up the funding amounts to get the amount of funds raised based on **month**. To get the total funds, we used the **aggregation()** method and added the data points to get the required information.

Also, to build the correlation between the funds raised with respect to **month** and **round**, we grouped the data frame by **month** and **round** and again applied the aggregation on **raisedAmt**.

There's more…

In the preceding two recipes, we learned about data analysis and visualization, and we extensively used the `pandas` Python module in our recipes. The `pandas` module is a very comprehensive library and has capabilities such as working with time series, advanced indexing techniques, merging and concatenating objects, and working with different datasets (JSON and Excel), amongst others. I highly encourage you to go through the `pandas` API Reference to learn more about this awesome library.

In the next recipe, let's see whether we can apply the knowledge we have gained so far in this chapter by helping Judy automate her task.

Automating social media analysis for businesses

Judy is a columnist at a leading magazine in London. As a writer, she is always interested in recent topics. She collects data and analyzes it to come up with insights that would be interesting to her readers.

Currently, Judy is interested in the battle between Apple's iPhone and Samsung's Note and wants to publish an article in her magazine. She plans to collect the data by talking to people on the streets and reading blog posts, but she knows she will get an awful lot of information from social media. She is aware of the fact that people take to Twitter these days to express their pleasure or disappointment about using a product and also refer products to their friends on social media. However, she is worried about the fact that she has to go through this huge volume of social media data for her article.

You are a data scientist and Judy's colleague. Can you help Judy with her needs? It is an opportunity for you to show off your data skills!

Let's begin by analyzing Judy's problems. To begin with, Judy needs to collect data from an ever-growing social media platforms such as Twitter. Secondly, she needs to analyze this data to generate interesting insights. So, we should be able to build a system that caters to both her problems. Also, you may want to build a system that only solves her current needs, but she should be able to use it for any projects in the future.

Getting ready

Let's install all the modules that we'll need to work on this problem. We already have `pandas`, `matplotlib`, and `seaborn` installed. For this problem, we will also install `tweepy`, a module to work with Twitter data. Let's install tweepy with our very own `python-pip`:

```
  (analyze)chetans-MacBookPro:ch11 Chetan$ pip install tweepy
  You are using pip version 7.1.0, however version 9.0.1 is available.
  You should consider upgrading via the 'pip install --upgrade pip'
command.
  Collecting tweepy
   Downloading tweepy-3.5.0-py2.py3-none-any.whl
  Collecting requests>=2.4.3 (from tweepy)
   Downloading requests-2.12.1-py2.py3-none-any.whl (574kB)
    100% |████████████████████████████████| 577kB
161kB/s
  Requirement already satisfied (use --upgrade to upgrade): six>=1.7.3 in
./analyze/lib/python2.7/site-packages (from tweepy)
  Collecting requests-oauthlib>=0.4.1 (from tweepy)
   Downloading requests_oauthlib-0.7.0-py2.py3-none-any.whl
  Collecting oauthlib>=0.6.2 (from requests-oauthlib>=0.4.1->tweepy)
   Downloading oauthlib-2.0.0.tar.gz (122kB)
    100% |████████████████████████████████| 122kB
345kB/s
  Building wheels for collected packages: oauthlib
   Running setup.py bdist_wheel for oauthlib
   Stored in directory:
```

```
/Users/chetan/Library/Caches/pip/wheels/e4/e1/92/68af4b20ac26182fbd623647af
92118fc4cdbdb2c613030a67
    Successfully built oauthlib
    Installing collected packages: requests, oauthlib, requests-oauthlib,
tweepy
    Successfully installed oauthlib-2.0.0 requests-2.12.1 requests-
oauthlib-0.7.0 tweepy-3.5.0
```

How to do it…

OK nice, we're now armed with all the modules. So, let's get started by collecting data from Twitter. Twitter has this amazing set, Streaming APIs, which helps developers collect tweets in real time. We will use this library for our data collection needs too.

1. The following code uses the Twitter Streaming APIs to collect data and store it in a text file with each tweet stored in the JSON format. We look for tweets that have two keywords, `iPhone 7` and `Note 5`. For this chapter, I ran the code for around 5 minutes, but for Judy we may have to run it for hours, or even days, to collect the maximum data to generate accurate insights:

```
from tweepy import Stream
from tweepy import OAuthHandler
from tweepy.streaming import StreamListener
import json
#consumer key, consumer secret, access token, access secret.
ckey="<>"
csecret="<>"
atoken="<>"
asecret="<>"
tweets_data_path = 'twitter_data.txt'
f = open(tweets_data_path, "w")
class listener(StreamListener):
    def on_data(self, data):
        print data
        f.write(data)
        #all_data = json.loads(data)
        #tweet = all_data["text"]
        #lang = all_data["lang"]
        #username = all_data["user"]["screen_name"]
        #print "username:%s, tweet:%s,
             language:%s" %(username, tweet, lang)
        return True
    def on_error(self, status):
        print "Error:", status
auth = OAuthHandler(ckey, csecret)
```

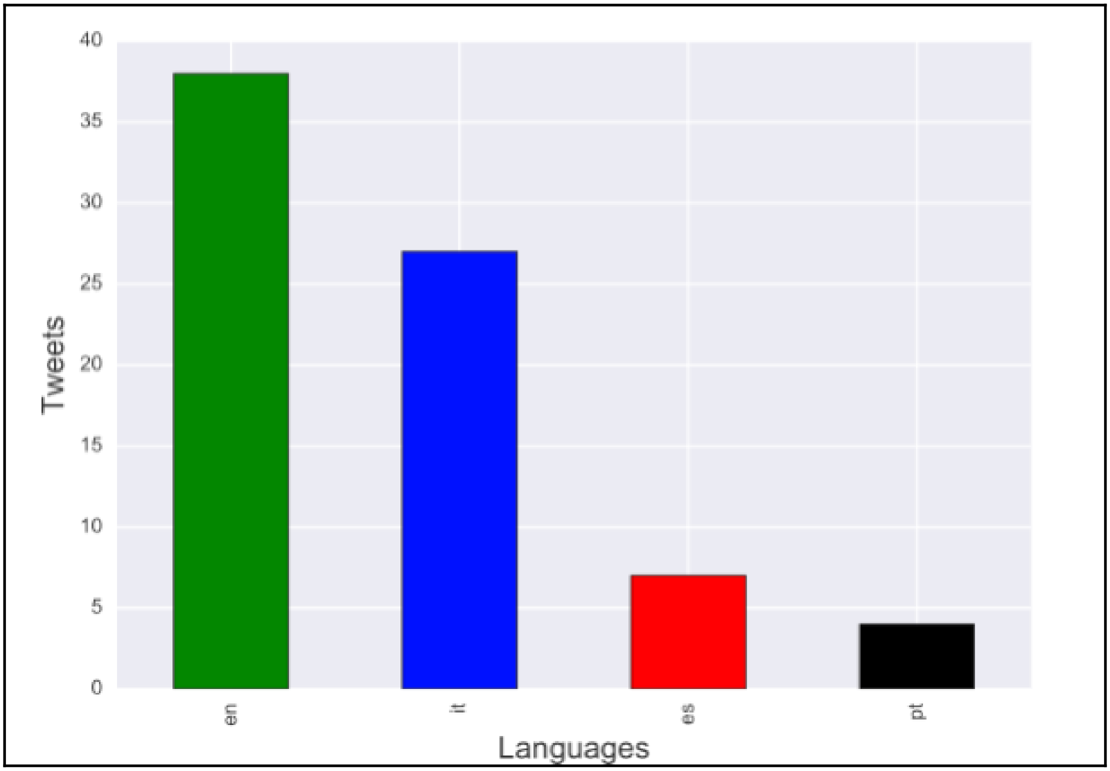
```
auth.set_access_token(atoken, asecret)
twitterStream = Stream(auth, listener())
twitterStream.filter(track=["iPhone 7","Note 5"])
f.close()
```

Okay, now that we have the data from Twitter flowing in, let's write a code to analyze this data and see whether we can find any interesting stuff that can be shared with Judy for her article.

2. Apple iPhone and Samsung Note are such popular products on a global level that people talk about these products from all around the world. It'd be really interesting to find the different languages used by consumers to talk about these products on Twitter. The following code does exactly what we wanted to do with Twitter data. It goes through the stored tweets, figures out the languages of all the tweets, and groups them to plot the top four languages:

```
import json
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
tweets = []
fh = open("twitter_data.txt", "r")
for data in fh:
    try:
        tweets.append(json.loads(data))
    except:
        continue
tweet_df = pd.DataFrame()
tweet_df['lang'] = map(lambda x: x['lang'], tweets)
tweets_by_lang = tweet_df['lang'].value_counts()
fig, axis = plt.subplots()
sns.set_style("darkgrid")
axis.set_xlabel('Languages', fontsize=15)
axis.set_ylabel('Tweets' , fontsize=15)
clrs = ['green', 'blue', 'red', 'black']
sns_plot = tweets_by_lang[:4].plot(ax=axis, kind='bar', color=clrs)
plt.savefig('language.pdf')
```

If we run the preceding code snippet, it plots the bar chart for the top languages people have used to tweet about iPhone 7 and Note 5.



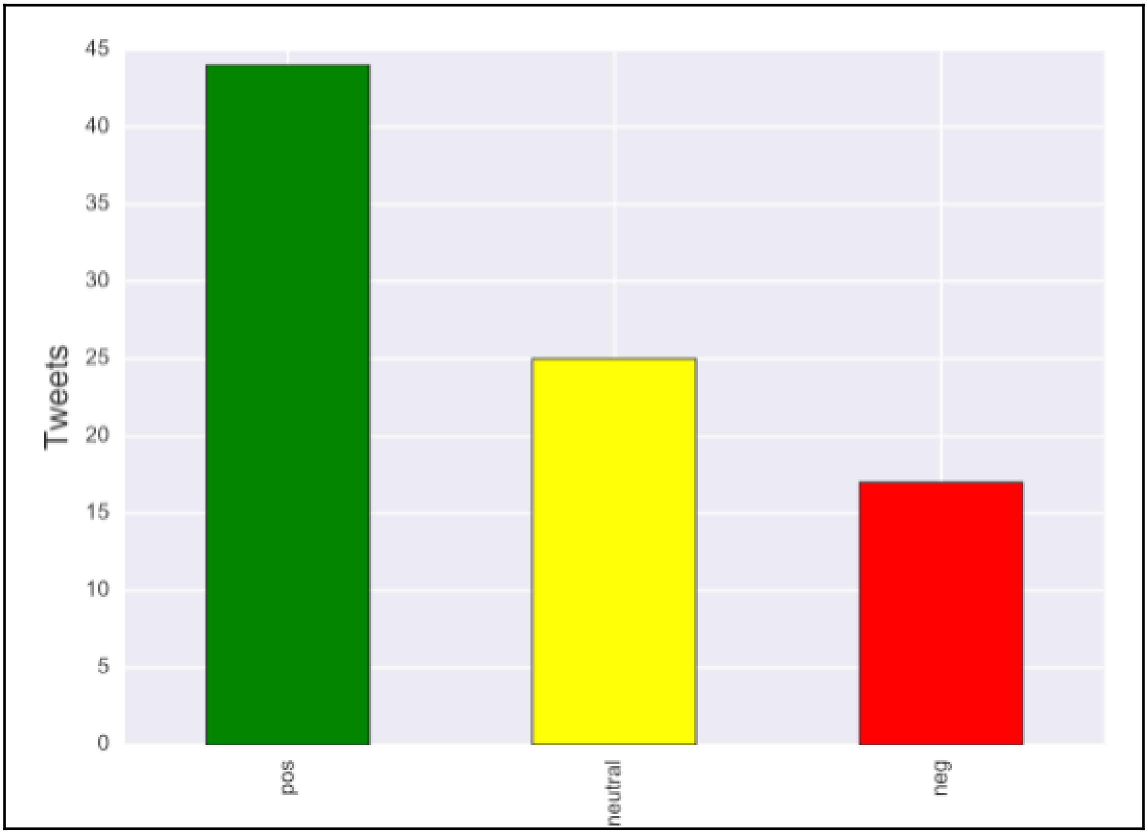
Awesome! I think Judy will love this analysis. Even though the top language is English (**en**), as expected, it is quite interesting to see that the other three languages are Italian (**it**), Spanish (**es**), and Portuguese (**pt**). This will be a cool feed for her article.

The results that you will get from this exercise will depend on when you run the data collection program. For instance, if you run it for a short duration between 2 and 8 a.m. GMT time, you will see more tweets in Chinese or Japanese as it is daytime in these countries. By the way, wouldn't it be interesting to analyze what are the best times of the day when people tweet? You may find some correlation.

3. Let's go further and do some more cool stuff with this data. How about performing text-based analysis on the tweets to get consumer sentiments about these products? By sentiment, I mean, is a tweet cursing these products, appreciating a product's feature, or just a passing comment? But wait; is it possible to get this kind of data? Yes, absolutely. The following code uses Python's NTLK-based APIs to perform sentiment analysis on tweets (text) to determine its polarity: positive, negative or neutral sentiment. It then groups this data to represent it with a bar chart and saves it in a PDF file:

```
        import json
        import pandas as pd
        import matplotlib.pyplot as plt
        import seaborn as sns
        import requests

        tweets = []
        fh = open("twitter_data.txt", "r")
        for data in fh:
            try:
                tweets.append(json.loads(data))
            except:
                continue

        probablities = pd.DataFrame()
        prob = []
        for tweet in tweets:
            text = tweet['text']
            r =
requests.post(url="http://text-processing.com/api/sentiment/",
                              data={"text":text},)
            print r.text
            if r.status_code == 200:
                ans = json.loads(r.text)
                prob.append(ans["label"])
        probablities['data'] = map(lambda x: x, prob)
        p_df = probablities['data'].value_counts()

        fig, axis = plt.subplots()
        sns.set_style("darkgrid")
        axis.set_xlabel('Sentiments', fontsize=15)
        axis.set_ylabel('Tweets' , fontsize=15)
        clrs = ['green', 'yellow', 'red']
        sns_plot = p_df.plot(ax=axis, kind='bar', color=clrs)
        plt.savefig('sentiments.pdf')
```

If you run the preceding piece of code, you will get a bar chart with the sentiment data for all the stored tweets. Just by looking at the graph, you will know that the consumers have generally spoken positively about both products:



There are a few complaints about them, hence the sentiments with negative polarity, but then there are neutral comments that could just be product referrals or comments on the products. Neat!

While this is nice, don't you think Judy might be interested to know how many of these negative tweets are about the iPhone or Note? Well, I will leave that to you; I'm sure you will figure that out for Judy.

How it works…

For the first problem, we first collected the data required for analysis. Twitter's Streaming APIs set helps us collect this information in real time. To use Streaming APIs, you need to register for a developer app with Twitter and collect your consumer key, consumer secret, auth token, and auth secret. It is a straightforward process and can be easily looked up on `https://dev.twitter.com`. So, in the data collection example (the first example in this recipe), we instantiated the `OAuthHandler` class to get Twitter's authorization object, `auth`, and then used it to set the authorization token and secret with the `set_access_token()` method. The `Stream` class of Twitter's Streaming APIs is bound to the `listener` class and returns the `twitterStream` object. The `listener` class inherits the `StreamListener` class, which will monitor incoming tweets and take actions on the arriving tweets in the `on_data()` method. The filtering of tweets is done with the `twitterStream.filter()` method.

Now, we know that the incoming tweets are available in the `on_data()` method; we hooked on to it to store the tweets in the `twitter_data.txt` file. For this, we opened the file in the write (`w`) mode and used the `write()` method to write the tweet to the file in the JSON format. With this, we finished the first recipe and collected the data required by Judy. Now, it's time to perform the analysis.

For the first insight on getting the languages, we started by opening the `twitter_data.txt` file in the read(`r`) mode. We read through all the tweets (the JSON format) and appended them to the `tweets` array. Using `pandas`, we created an empty DataFrame object, `tweet_df`, with `pd.DataFrame()`. With Python's `map()` method, we operated on the `tweets` array and added a new column, `lang`, to our empty DataFrame. The `value_counts()` method was then used to get the count of all the languages of the tweets under analysis and was stored in the variable, `tweets_by_lang`.

The other part of the code is as usual, where we created a `plt` object from matplotlib and used the `plot()` method to generate a bar chart using the `seaborn` library. We set the axis labels with the `set_xlabel()` and `set_ylabel()` methods and used the colors `green`, `blue`, `red`, and `black` to manifest the different languages. Finally, we saved the plot in a PDF file with the `savefig()` method.

For the second insight involving sentiment analysis, we started by reading through all the tweets from `twitter_data.txt` and storing them to the `tweets` array. We then created an empty data frame, probabilities, processed all the tweets for sentiment analysis, and stored the analysis in the `prob` array. We then added a column, `text`, to our empty data frame using the `map()` method on the `prob` array.

Our data frame, `probabilities['text']`, now contains sentiments for all the tweets we analyzed. Following the regular set of operations, we got the set of values for `positive`, `negative`, and `neutral` sentiments for the analyzed tweets and plotted them as a bar chart.

If you look at all the examples in this recipe, we have divided the task of data collection and analysis as separate programs. If our visualizations are massive, we can even separate them out. This makes sure that Judy can use the data collection Python program to gather information on another set of keywords for her articles in the future.

She can also run the analysis on the data sets she has by making small changes to the analysis and visualization parameters from our program. So, for all her articles in the future, we have managed to automate the data collection, analysis, and visualization process for her.

I'm already seeing a smile on Judy's face. I hope you enjoyed this chapter. I'm confident that the knowledge you gained in this chapter will get you started into the world of data and visualizations.

There's more…

We have just scratched the surface of text-based sentiment analysis in the preceding recipe. Sentiment analysis involves text classifications, tokenization, semantic reasoning, and much more interesting stuff. I highly encourage you to go through a book on NLTK to learn more about working with text in Python at `http://www.nltk.org/book/`.

11
Time in the Zone

Time calculations are fun, interesting, and tedious at the same time. Fun, when you first learn to read time, interesting, when you learn about daylight saving and tedious, when a customer complains about not able to schedule meetings across time zones through your web application.

In this chapter, we will cover the following recipes:

- Working with time, date, and calendar
- Comparing and combining dates, and date arithmetic
- Formatting and parsing dates
- Dealing with time zone calculations
- Automating invoicing based on time zone

Introduction

> *"If you love life, don't waste time, for time is what life is made up of."*
>
> *–Bruce Lee.*

Time is the measure of all things. We have so many things to do in life, and yet the irony is we have so much less time on our hands. These days, we plan for time intuitively: what time should I travel at to avoid traffic, what's my deadline for this task among many other things, and so on. Businesses plan their activities for the complete year even before the calendar begins.

Time calculations are almost everywhere. Want to schedule a meeting with your colleague in Australia? Get the time zone right, work on a good time for you and your colleague, and then schedule it. Want to write code to perform a task for your customer when the time is right? Manage time objects in the database and keep track of all the tasks for your users. Even in the Hollywood movie, *National Treasure*, Nicholas Cage had to depend on time zone calculations to get to the next clue that took him closer to the treasure.

Essentially, you can't run away from time calculations wherever you are and whatever you do. In this chapter, we will work with `date` and `time` objects in Python. We will also learn how to perform arithmetic operations on dates and work with time zone calculations. We will also learn how to automate a business process based on users' time zone.

During the course of this chapter, we will majorly use built-in Python modules. The following built-in modules will be used in this chapter:

- `datetime` (`https://docs.python.org/2/library/datetime.html`)
- `calendar` (`https://docs.python.org/2/library/calendar.html`)

We will also use this external module to work with time zones:

- `pytz` (`http://pytz.sourceforge.net/`)

The `pytz` library brings the Olson timezone database into Python. It allows accurate and cross-platform timezone calculations in Python 2. It also helps in performing calculations with respect to daylight savings.

Before we get into the recipes, let's check if we have the relevant modules in our Python installation and also install the ones we need in this chapter. We start by creating a virtual environment for this chapter and activating it:

```
chetans-MacBookPro:ch12 Chetan$ virtualenv date
New python executable in date/bin/python2.7
Also creating executable in date/bin/python
Installing setuptools, pip, wheel...done.
chetans-MacBookPro:ch12 Chetan$ source date/bin/activate
(date)chetans-MacBookPro:ch12 Chetan$
```

Now, let's install the `pytz` module in our virtual environment using Python `pip`. Once we install the module, we will move ahead to the first recipe and start working with `time` and `date` objects:

```
(date)chetans-MacBookPro:ch12 Chetan$ pip install pytz
Collecting pytz
  Using cached pytz-2016.7-py2.py3-none-any.whl
```

```
Installing collected packages: pytz
Successfully installed pytz-2016.7
```

Working with time, date, and calendar

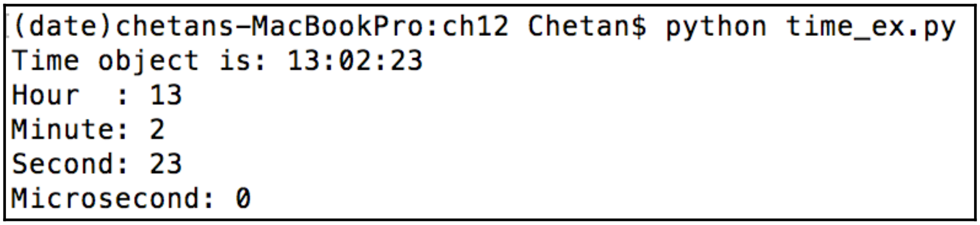
In this recipe, we will use built in Python modules and we don't need to install anything explicitly. So let's get started.

How to do it…

1. Go to your favorite editor, create a file named `time_ex.py`, and write the following code in the Python file:

```
import datetime
time_obj = datetime.time(13, 2, 23)
print "Time object is:", time_obj
print 'Hour  :', time_obj.hour
print 'Minute:', time_obj.minute
print 'Second:', time_obj.second
print 'Microsecond:', time_obj.microsecond
```

If we run the preceding Python code, we will see the following output. Observe how we created a given time object using Python and retrieved the *hour, minute, second,* and *microsecond* details for the given time object:

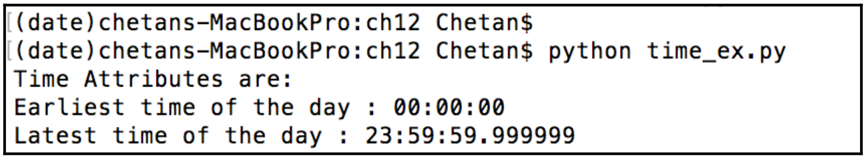


```
(date)chetans-MacBookPro:ch12 Chetan$ python time_ex.py
Time object is: 13:02:23
Hour  : 13
Minute: 2
Second: 23
Microsecond: 0
```

2. Python's `Time` class has some more attributes that can be effectively used for time calculations. For instance, in the following code snippet, I can get the valid time range for a given day:

```
import datetime
print "Time Attributes are:"
print "Earliest time of the day :", datetime.time.min
print "Latest time of the day :", datetime.time.max
```

The output of the preceding code snippet can be seen in the following screenshot. Observe how we get the first and the last available time for the day:

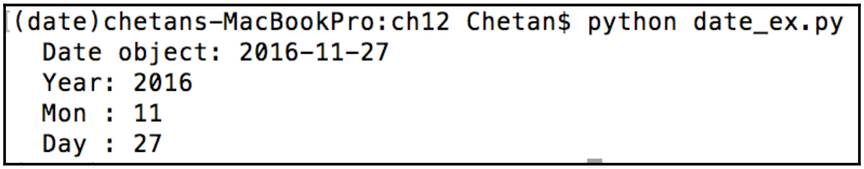


```
(date)chetans-MacBookPro:ch12 Chetan$
(date)chetans-MacBookPro:ch12 Chetan$ python time_ex.py
Time Attributes are:
Earliest time of the day : 00:00:00
Latest time of the day : 23:59:59.999999
```

3. Neat! Now, let's look at the `date` object. The following Python code gets today's date for you. We also retrieve the `year`, `month`, and `day` attributes with the following code:

```
import datetime
today = datetime.date.today()
print 'Date object:', today
print 'Year:', today.year
print 'Mon :', today.month
print 'Day :', today.day
```

The output of the preceding code snippet is shown in the following screenshot:

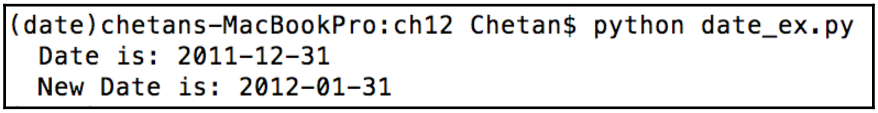


```
(date)chetans-MacBookPro:ch12 Chetan$ python date_ex.py
  Date object: 2016-11-27
  Year: 2016
  Mon : 11
  Day : 27
```

4. Ok, cool! We can also create a new `date` object or modify an existing one with Python's `date()` and `replace()` methods. The following code demonstrates how to use these methods:

```
import datetime
date_1 = datetime.date(2011, 12, 31)
print '  Date is:', date_1
date_2 = date_1.replace(year=2012, month=1)
print '  New Date is:', date_2
```

The output of the preceding snippet is as follows:

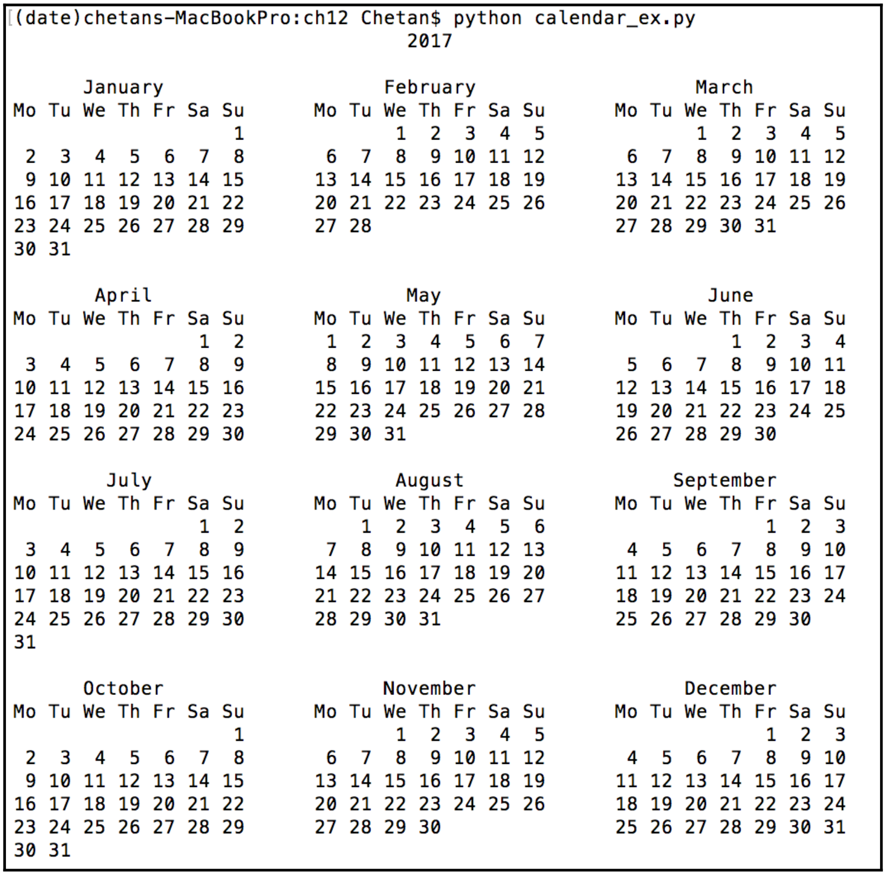


```
(date)chetans-MacBookPro:ch12 Chetan$ python date_ex.py
  Date is: 2011-12-31
  New Date is: 2012-01-31
```

5. Fantastic! Let's move ahead and see if we can work with months or even years. With Python, it's very easy to work with the whole calendar. The following code prints out the calendar for the complete year on the console. In this case, it returns the calendar for the year 2017. Let me check my birthday… oh it's on Tuesday this year and I have to go to the office:

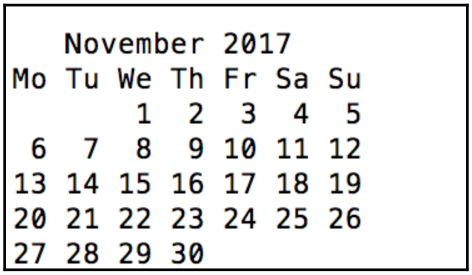
```
import calendar
from calendar import TextCalendar
cal = TextCalendar()
cal.pryear(2017)
cal.prmonth(2017, 11)
```

The output of the preceding code snippet is shown in the following screenshot. The first screenshot returns the complete calendar for the year 2017:



```
[(date)chetans-MacBookPro:ch12 Chetan$ python calendar_ex.py
                                  2017

      January                   February                   March
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
                   1             1  2  3  4  5             1  2  3  4  5
 2  3  4  5  6  7  8       6  7  8  9 10 11 12       6  7  8  9 10 11 12
 9 10 11 12 13 14 15      13 14 15 16 17 18 19      13 14 15 16 17 18 19
16 17 18 19 20 21 22      20 21 22 23 24 25 26      20 21 22 23 24 25 26
23 24 25 26 27 28 29      27 28                     27 28 29 30 31
30 31

       April                      May                     June
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
                1  2       1  2  3  4  5  6  7                1  2  3  4
 3  4  5  6  7  8  9       8  9 10 11 12 13 14       5  6  7  8  9 10 11
10 11 12 13 14 15 16      15 16 17 18 19 20 21      12 13 14 15 16 17 18
17 18 19 20 21 22 23      22 23 24 25 26 27 28      19 20 21 22 23 24 25
24 25 26 27 28 29 30      29 30 31                  26 27 28 29 30

        July                     August                September
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
                1  2          1  2  3  4  5  6                   1  2  3
 3  4  5  6  7  8  9       7  8  9 10 11 12 13       4  5  6  7  8  9 10
10 11 12 13 14 15 16      14 15 16 17 18 19 20      11 12 13 14 15 16 17
17 18 19 20 21 22 23      21 22 23 24 25 26 27      18 19 20 21 22 23 24
24 25 26 27 28 29 30      28 29 30 31               25 26 27 28 29 30
31

      October                   November                  December
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
                   1             1  2  3  4  5                   1  2  3
 2  3  4  5  6  7  8       6  7  8  9 10 11 12       4  5  6  7  8  9 10
 9 10 11 12 13 14 15      13 14 15 16 17 18 19      11 12 13 14 15 16 17
16 17 18 19 20 21 22      20 21 22 23 24 25 26      18 19 20 21 22 23 24
23 24 25 26 27 28 29      27 28 29 30               25 26 27 28 29 30 31
30 31
```

The following screenshot only returns the calendar for the eleventh month of 2017, that is, November 2017:



```
   November 2017
Mo Tu We Th Fr Sa Su
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30
```

How it works…

In this recipe, we started working with `time` objects. We created a `time` object with the `datetime.time()` method that takes hour, minute, and second as input parameters. We also read the time object with `hour`, `minute`, and `second` attributes, and got the earliest and latest times of the day with `datetime.time.min` and `datetime.time.max` attributes.

Next, we moved onto working with dates using the `datetime.date()` method. We got today's date with the `datetime.date()` method and printed the year, month, and day for today with the `today.year`, `today.month`, and `today.day` attributes.

We also created a new date with the `datetime.date()` method by passing the `year`, `month`, and `day` as parameters. Once the `date` object was available, we used it to get the new date with the `replace()` method. We passed `year` and `month` as parameters to the `replace()` method to create the new `date` object.

We also worked with the `calendar` object in this section. We used the `calendar` module available in Python installation for this purpose. First, we instantiated the `TextCalendar` class to create a text calendar object. Later, we used this object to print the calendar for the year 2017 on the console with `pryear()` method.

We could also show the calendar for just the month of November for the year, 2017, using the `prmonth()` method. Nice!

Comparing and combining the date and time objects, and date arithmetic

Creating date objects and using them is great, but the tasks we need to perform for business use cases are often related to comparing or calculating differences in the date and time objects. In this recipe, we will learn how to perform these operations in Python. It is, however, important to note a major change that we will see in this recipe. In the last recipe, we worked with the time and date objects independently. But Python's datetime module provides a great benefit, in the sense that we get to work on objects that include both date and time attributes. You will see this difference in the How to do it section.

Getting ready

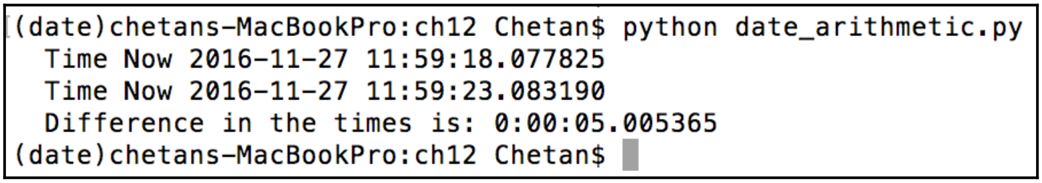
In this recipe, we will work with the datetime Python module that we used for the last couple of examples. We don't have any new installations to be done for this recipe.

How to do it…

1. Let's start by getting the difference between two datetime objects. The following code does this operation and calculates the difference between two datetime objects. Though this operation only tells you the difference in seconds, you can also use these to get the difference in months or years. In the following screenshot, notice how datetime.now() returns a string with both today's date and the current time. It is important to understand that we are working on an object that comprises the date and time attributes. If you think about it, in the actual world also, when we have to calculate the time difference between two events, working on the date and time objects together will be the most useful to us. Even if we independently work on the date object or the time object, we will end up performing the same calculation that we will perform with the datetime object, so imagine the benefits we get with this approach:

```
from datetime import datetime
import time
now_1 = datetime.now()
print "  Time Now", now_1
time.sleep(5)
now_2 = datetime.now()
print "  Time Now", now_2
print "  Difference in the times is:", (now_2 - now_1).seconds
```

The output of the preceding code snippet is shown in the following screenshot. See how we get the difference between the datetime objects in seconds:

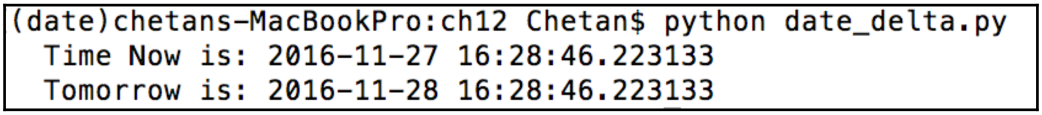


```
(date)chetans-MacBookPro:ch12 Chetan$ python date_arithmetic.py
  Time Now 2016-11-27 11:59:18.077825
  Time Now 2016-11-27 11:59:23.083190
  Difference in the times is: 0:00:05.005365
(date)chetans-MacBookPro:ch12 Chetan$
```

2. This is fine, but you may ask what happens if the difference in the `datetime` objects is negative. In this example, if we calculate `now_1 - now_2`, we get a high number and not the actual difference. For this, we have a nice trick to get the difference between two `datetime` objects. We can use `(now_1 - now_2).total_seconds()` to get the negative value, that is, `-5` seconds.
3. OK, now let's move forward to perform more calculations on the `datetime` objects. For instance, how about getting the time from the past or into the future? The following code helps us perform these operations; see how we got the date and time at this moment and also could return the date and time for the next day, that is, tomorrow:

```
from datetime import datetime, timedelta
now = datetime.now()
print "  Time Now is:", now
one_day_later = now + timedelta(days=1)
print "  Tomorrow is:", one_day_later
```

The output of the preceding code is as follows:

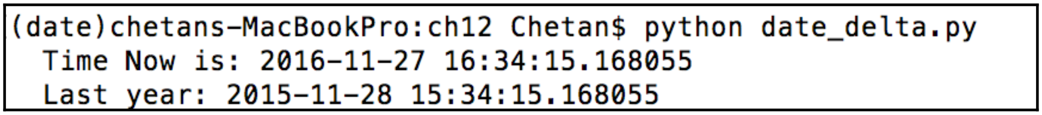


```
(date)chetans-MacBookPro:ch12 Chetan$ python date_delta.py
  Time Now is: 2016-11-27 16:28:46.223133
  Tomorrow is: 2016-11-28 16:28:46.223133
```

4. If we want to get the time in the past, we could do it in the same way as shown in the following code snippet:

```
from datetime import datetime, timedelta
now = datetime.now()
print "  Time Now is:", now
days_in_past = now - timedelta(days=365, hours=1)
print "  Last year:", days_in_past
```

The output of the preceding code is shown in the following screenshot. Note that we asked for a past date, which is 365 days back. However, it shows November 28, 2015. Why? Shouldn't it show the same day? Oh, of course, 2016 was a leap year!

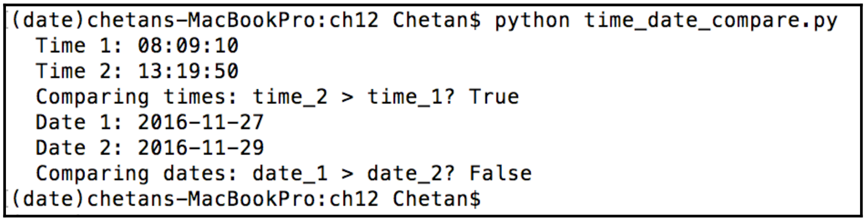


```
(date)chetans-MacBookPro:ch12 Chetan$ python date_delta.py
  Time Now is: 2016-11-27 16:34:15.168055
  Last year: 2015-11-28 15:34:15.168055
```

5. OK, now we're comfortable working with getting the difference or adding time to `date` and `time` objects. But we also often need to compare between times, right? Let's learn this with the help of a code snippet. The following Python code compares `time` and `date` objects separately:

```
import datetime

time_1 = datetime.time(8, 9, 10)
print "  Time 1:", time_1
time_2 = datetime.time(13, 19, 50)
print "  Time 2:", time_2
print "  Comparing times: time_2 > time_1?", time_2 > time_1

date_1 = datetime.date.today()
print "  Date 1:", date_1
date_2 = date_1 + datetime.timedelta(days=2)
print "  Date 2:", date_2
print "  Comparing dates: date_1 > date_2?", date_1 > date_2
```

The output of the preceding piece of code can be seen in the following screenshot:

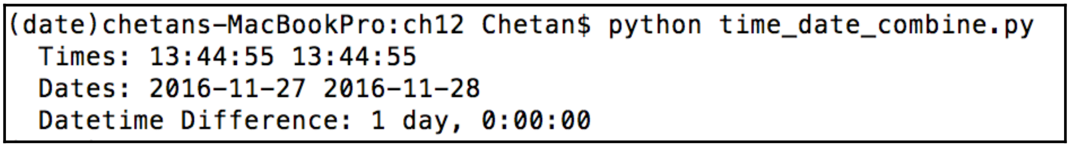


```
(date)chetans-MacBookPro:ch12 Chetan$ python time_date_compare.py
  Time 1: 08:09:10
  Time 2: 13:19:50
  Comparing times: time_2 > time_1? True
  Date 1: 2016-11-27
  Date 2: 2016-11-29
  Comparing dates: date_1 > date_2? False
(date)chetans-MacBookPro:ch12 Chetan$
```

6. As we saw in the previous recipe, you also feel the need to combine your `time` objects with the `date` objects. For instance, you have developed your program for a use case where you want to compare `time` objects and take some action. But you may end up doing the date comparison also since the `time` objects you are comparing fall on different dates. Since you're already aware how easy it is to work with `datetime` objects, you may want to combine your `time` and `date` objects into a single `datetime` object and easily work on them. We can easily achieve this in Python; the following code demonstrates combining independent times and dates to `datetime` objects:

```
import datetime

time_1 = datetime.time(13, 44, 55)
time_2 = datetime.time(13, 44, 55)
print "  Times:", time_1, time_2

date_1 = datetime.date.today()
date_2 = date_1 + datetime.timedelta(days=1)
print "  Dates:", date_1, date_2

datetime_1 = datetime.datetime.combine(date_1, time_1)
datetime_2 = datetime.datetime.combine(date_2, time_2)
print "  Datetime Difference:", datetime_2 - datetime_1
```

The output of the preceding piece of code is:



```
(date)chetans-MacBookPro:ch12 Chetan$ python time_date_combine.py
  Times: 13:44:55 13:44:55
  Dates: 2016-11-27 2016-11-28
  Datetime Difference: 1 day, 0:00:00
```

How it works…

In the last section, we worked on `time`, `date`, and `calendar` objects independently. In this recipe, we started working on the complete `datetime` object.

In the first code example of this recipe, we calculated the difference between the `datetime` objects. We could easily do that with the same subtraction (-) operator we're used to. That means the `__sub__()` method has been overridden for the `datetime` class.

Then, in the second and third code snippets, we used the `timedelta()` method to get to the future `datetime` objects or move in the past. The `timedelta()` method supports conveniently named attributes such as `days` or `hours` to shift the current `datetime` objects to the past or future. We get to the past with `- timedelta()` and move ahead with `+ timedelta()` operations.

Next, we understood how to compare the `datetime` objects. This was again simply done like any other Python object. In Python, we check if an integer is less than or greater than another integer with the < and > operators, respectively. Same is the case with the `datetime` objects. We simply use these operators to compare even the `datetime` objects.

Finally, we looked at the use case where we needed to work on `date` and `time` objects to get the difference or compare them. For this, we wrote a Python code to combine the `date` and `time` objects and used the `datetime.combine()` method. This made sure that the comparison or the difference operation can be easily done on the `datetime` objects instead of doing them individually on `date` or `time` objects and then merging the results.

Formatting and parsing dates

In all the recipes so far, we performed multiple operations on the `date` or `time` objects. But the objects themselves are represented in certain formats. For instance, by default, the `date()` object is represented in a YYYY-MM-DD format and the `time()` object is represented in a HH:MM:SS format. While these representations are good, we can't always use these formats for representing data to the users on a website or while scheduling meetings from a web page.

In this section, we quickly look at the different formats in which the `date` and `time` objects can be manifested to the users.

Getting ready

For this recipe, we end up using the same `datetime` module that gets packaged with the default Python installation.

How to do it…

1. Let's get started with something we know. The following Python code will print the date and time in ISO format. This format is the most used format around the world and is universally acceptable:

   ```
   import datetime
   today = datetime.datetime.today()
   print "  ISO datetime: ", today
   ```

2. However, as you would have already imagined, this format is not quite readable. For instance, it reads the month in digits (11 for November) and returns the time even till microseconds (which I don't think is very useful). How about formats where we solve these issues and make the date more readable? Yes, we can easily do that with the following code snippet. In this code, with the help of certain format specifiers such as `'%b'`, we manage to make the month readable:

   ```
   import datetime
   today = datetime.datetime.today()
   print "  ISO datetime: ", today
   format = "%a %b %d %H:%M:%S %Y"
   string_format = today.strftime(format)
   print "  Datetime in String format:", string_format
   ```

3. You must have seen some web applications using Unix timestamp or epochs to store time. Even though this is a nice way to store objects, you still need to represent the actual time or date to the user in the format she understands.

4. Unix time, also known as POSIX time or epoch time, is a system for describing times defined as seconds that have elapsed since Thursday, January 01, 1970, 00:00:00 UTC. Unix timestamps are useful as they represent time independent of time zones. For example, a Unix time can represent 1:00 pm in London and 8:00 am in New York.

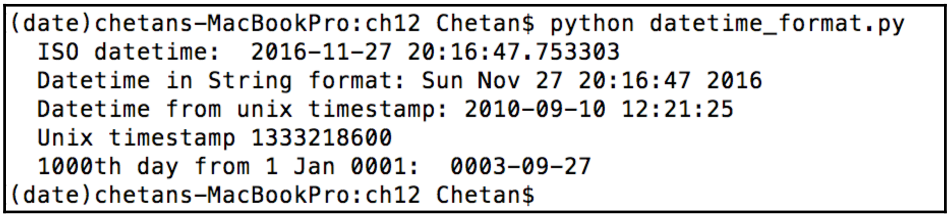
The following code snippets show how to convert timestamps to `datetime` objects and vice versa:

```
import datetime
import time

time_1 = time.time()
print "  Datetime from unix timestamp:",
datetime.datetime.fromtimestamp(1284101485)

date_1 = datetime.datetime(2012,4,1,0,0)
print "  Unix timestamp", date_1.strftime('%s')
```

5. Another fun representation of the `datetime` objects can be to show the date from the *n*th day from when the world began. For instance, can you print the date for the 1000th day after Jan 1, 0001? It is the date corresponding to the propletic Gregorian calendar where 01/01/01 has an ordinal 1:

```
import datetime
date_1 = datetime.date.fromordinal(1000)
print "  1000th day from 1 Jan 0001: ", date_1
```

If you run the preceding Python code snippet, you will be able to see the desired objects as in the following screenshot:



```
(date)chetans-MacBookPro:ch12 Chetan$ python datetime_format.py
  ISO datetime:  2016-11-27 20:16:47.753303
  Datetime in String format: Sun Nov 27 20:16:47 2016
  Datetime from unix timestamp: 2010-09-10 12:21:25
  Unix timestamp 1333218600
  1000th day from 1 Jan 0001:  0003-09-27
(date)chetans-MacBookPro:ch12 Chetan$
```

How it works…

In this recipe, we looked at the various ways of representing `datetime` objects. In the first example, we printed the date and time in ISO format. This is the most used format and you can read more about the ISO format at `https://en.wikipedia.org/wiki/ISO_8601`. As you can see, we need not use a new method for this representation; we simply used `datetime.today()` to get the date in ISO format.

In the second example, we looked at defining our own format for representing the date in the string format. We took the help of format specifiers, such as `%a`, `%b`, and `%d` to work with date and `%H`, `%M`, and `%S` to work with time. We specified the format in the `format` variable and used it to pass it to `strftime()` method that formatted the ISO `datetime` object to our custom String format.

The next two examples helped us convert a Unix timestamp or an epoch to a `datetime` object and vice versa. For the first use case, we use the `datetime.fromtimestamp(<unixtimestamp>)` method to convert a Unix time stamp to a `datetime` object, and in the successive example, we converted a `datetime` object to a Unix timestamp with the `strftime()` method. The Unix time (1284101485) used in this example is the number of seconds that have elapsed since January 01, 1970.

In the last and interesting example, we get the date and time in a Gregorian calendar ordinal format with `fromordinal()` method. You will not use this method, in all likelihood, but I have included it in this chapter as an interesting date format for you to know.

Dealing with time zone calculations

One of the trickiest calculations that you will have to perform on `date` or `time` objects is the one that involves time zones. Your colleague works in San Francisco and you are in Sydney, how do you plan to do a conference call? When you set up a meeting, you should be aware of your colleague's time zone, else you may set up a meeting for 8 pm Sydney time while for your colleague in San Francisco, it is already past midnight. Time zone calculations are often tedious and need to be handled cautiously while developing business applications. Let's see how Python can help us in this regard.

Getting ready

For this recipe, we will use the `pytz` module that we installed at the beginning of this chapter. In fact, the Python standard library doesn't have a time zone library, but we can completely rely on the modules contributed by the Python community for our needs.

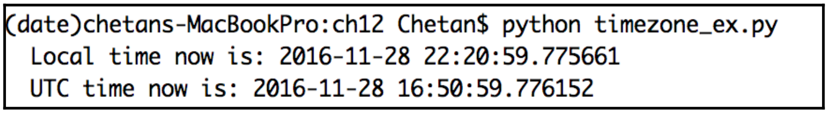
How to do it…

You need to perform the following steps:

1. Let's get started with a trivial operation of getting the local time in UTC. **UTC** stands for **Universal Time Converter**, a worldwide standard for regulating clocks and time measurements. UTC is also popularly known as Greenwich Mean Time (GMT).

   ```
   from datetime import datetime, timedelta
   now = datetime.now()
   print "  Local time now is:", now
   utcnow = datetime.utcnow()
   print "  UTC time now is:", utcnow
   ```

   The output of the preceding code snippet is shown in the following screenshot. Look at how my local time is +5:30 hours ahead of UTC:

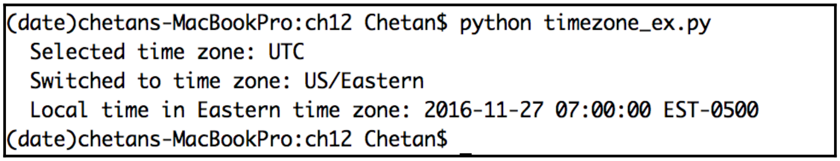
   

   ```
   (date)chetans-MacBookPro:ch12 Chetan$ python timezone_ex.py
     Local time now is: 2016-11-28 22:20:59.775661
     UTC time now is: 2016-11-28 16:50:59.776152
   ```

2. OK, this is nice. So, you can convert your local time to UTC, but this is not always enough. Your customers (for whom you develop the application) can be from anywhere in the world. Their accounts also need to be managed with respect to their time zones and local times. Let's see how we can figure out the local time for a user in a given time zone:

   ```
   from pytz import timezone
   import pytz
   utc = pytz.utc
   print "  Selected time zone:", utc
   eastern = timezone('US/Eastern')
   print "  Switched to time zone:", eastern
   loc_dt = datetime(2016, 11, 27, 12, 0, 0, tzinfo=pytz.utc)
   est = loc_dt.astimezone(eastern)
   fmt = '%Y-%m-%d %H:%M:%S %Z%z'
   print "  Local time in Eastern time zone:", est.strftime(fmt)
   ```

The output of the preceding piece of code is shown in the following screenshot. Observe how we converted a local UTC time to **Eastern Standard Time** (**EST**) by getting the eastern time zone. In fact, UTC works out to be the best way to convert time across time zones:

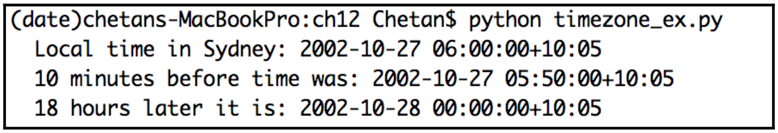


```
(date)chetans-MacBookPro:ch12 Chetan$ python timezone_ex.py
  Selected time zone: UTC
  Switched to time zone: US/Eastern
  Local time in Eastern time zone: 2016-11-27 07:00:00 EST-0500
(date)chetans-MacBookPro:ch12 Chetan$ _
```

3. Performing arithmetic calculations on `datetime` objects with time zone information is also trivial in Python. Look at the following code and see how we perform arithmetic operations on `date` objects:

```
from datetime import datetime, timedelta
au_tz = timezone('Australia/Sydney')
local = datetime(2002, 10, 27, 6, 0, 0, tzinfo=au_tz)
print "  Local time in Sydney:", local
past = local - timedelta(minutes=10)
print "  10 minutes before time was:", past
future = local + timedelta(hours=18)
print "  18 hours later it is:", future
```

Now, if we run this piece of code on our Python interpreter, we get the following output:

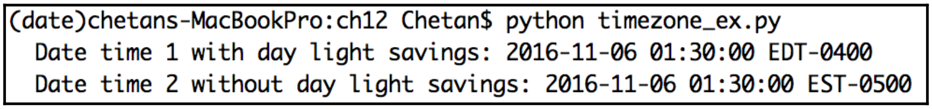


```
(date)chetans-MacBookPro:ch12 Chetan$ python timezone_ex.py
  Local time in Sydney: 2002-10-27 06:00:00+10:05
  10 minutes before time was: 2002-10-27 05:50:00+10:05
  18 hours later it is: 2002-10-28 00:00:00+10:05
```

4. We can't finish any topic on time zones without really talking about this, can we? Yes, how do we deal with daylight savings in time zone calculations? Thanks to Benjamin Franklin for the gift he gave to the world on daylight savings. Let's understand this with the help of a code example:

```
eastern = timezone('US/Eastern')
dt = datetime(2016, 11, 06, 1, 30, 0)
dt1 = eastern.localize(dt, is_dst=True)
print "  Date time 1 with day light savings:", dt1.strftime(fmt)
dt2 = eastern.localize(dt, is_dst=False)
print "  Date time 2 without day light savings:", dt2.strftime(fmt)
```

If you run the code snippet, you'd see two `datetime` objects represented in the String format. The first one takes care of the daylight savings and the second one disregards it. November 6, 2016 is when the daylight savings ended this year in the eastern time zone and the clock moved backwards:

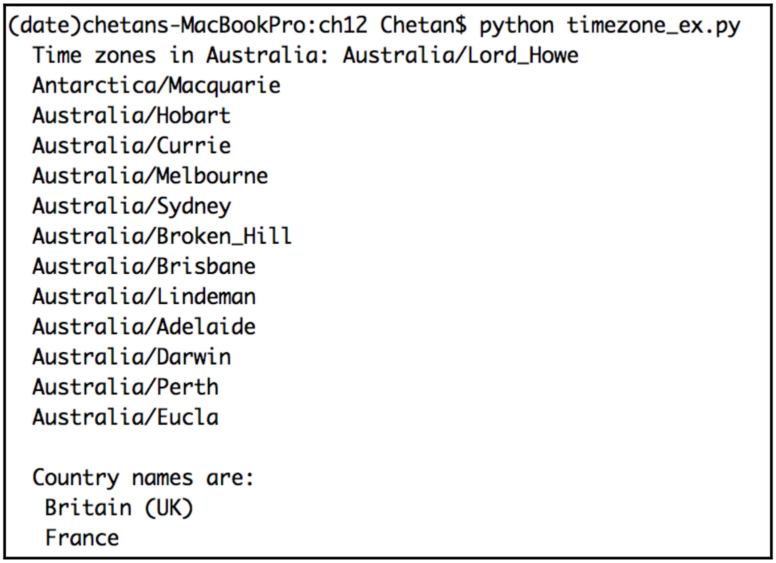


```
(date)chetans-MacBookPro:ch12 Chetan$ python timezone_ex.py
  Date time 1 with day light savings: 2016-11-06 01:30:00 EDT-0400
  Date time 2 without day light savings: 2016-11-06 01:30:00 EST-0500
```

5. Lastly, there are a few helper methods that are available in the `pytz` module that are often useful, for instance, getting the time zones for a given country based on the ISO country code, or simply getting the country name from the ISO country code. Let's look at the following examples:

```
  tz_au = '\n  '.join(pytz.country_timezones['au'])
  print "  Time zones in Australia:", tz_au
  country_gb, country_fr = pytz.country_names['gb'],
                           pytz.country_names['fr']
print "\n  Country names are:\n", "  ",
country_gb, "\n  ", "  ", country_gb, "\n  ", country_fr
```

The output of the preceding code snippet can be viewed in the following screenshot:



```
(date)chetans-MacBookPro:ch12 Chetan$ python timezone_ex.py
  Time zones in Australia: Australia/Lord_Howe
  Antarctica/Macquarie
  Australia/Hobart
  Australia/Currie
  Australia/Melbourne
  Australia/Sydney
  Australia/Broken_Hill
  Australia/Brisbane
  Australia/Lindeman
  Australia/Adelaide
  Australia/Darwin
  Australia/Perth
  Australia/Eucla

  Country names are:
   Britain (UK)
   France
```

How it works…

In this recipe, we looked at the various ways of working with time zones, which are integral to date-time calculations. In the first code example of this recipe, we calculate the current local time with `datetime.now()` and then got the same local time in UTC with `datetime.utcnow()`. The `utcnow()` method becomes very handy when we have to store date/time objects in the database for further processing, such as scheduling events.

Next, we looked at how to switch to a different time zone and retrieve the local time in that time zone. The `pytz` class has a simple attribute, `utc`, to set the time zone to UTC; we used it to set our current time zone to UTC. Later we used the `timezone()` method of the `pytz` module to switch to the eastern time zone with `timezone('US/Eastern')`.

In all the recipes before this one, we created a `datetime` object with the `datetime()` method; in this recipe also, we used the `datetime` method but with the `tzinfo` parameter in this way: `datetime(YYYY, MM, DD, HH, MM, SS, tzinfo=<timezone>)`. The `tzinfo` parameter makes sure to add the time zone information to the `datetime` object, which is important while performing calculations across time zones.

The `datetime` class has another convenient method that will represent the `datetime` object to a time zone of our choice: the `astimezone()` method. Using this method, we converted the UTC `datetime` object to eastern time with this code, `loc_dt.astimezone(eastern)`.

Finally, we created a custom string format to represent the eastern time with the `strftime(format)` method.

We can also add or remove time/days during the time zone calculations like we did with `datetime` objects. In the third code sample of this recipe, we switched to the Australia/Sydney time zone and created a `datetime` object for this time zone; this operation returned us the local time in Sydney. With the help of the `timedelta()` method, we then removed ten minutes from the local time with `local - timedelta(mins=10)` and also added 18 hours to the time with `local + timedelta(hours=18)`. This way, we could access the time from the past or in the future. Think of it as time travel.

In the fourth code snippet, we understood how to work with daylight savings. To understand this, we created a `datetime` object without any time zone information and assigned it to the `dt` variable. We also created a time zone object for eastern time with the code, `eastern = timezone('US/Eastern')`. We then used the `localize()` method on the time zone object to convert the `dt` object to eastern time. Here is where we add another parameter, `is_dst`, to the `localize(is_dst=<True/False>)` method to return the local time in eastern time zone, with or without considering daylight savings.

In 2016, November 6 was the day when the clock moved backwards at 2 am. So, in our example, when we queried for 1:30 am eastern time with `is_dst=True`, it returned time in **Eastern Daylight Time** (**EDT**), which is four hours behind **Coordinated Universal Time** (UTC-0400 hours). When we queried for the same time with `is_dst=False`, it returns the time in EST, which is UTC-0500 hours.

In the last example of this recipe, we looked at a few useful helper methods provided by the `pytz` module. For instance, `pytz.country_timezones['au']` returned all the time zones available in Au (Australia) and `pytz.country_names['gb']` returned the name of the country, that is, Britain (UK), based on the ISO country code `gb`. You will realize the utility of these libraries when you actually solve some of the time zone problems.

Automating invoicing based on user time zone

Jacob is a Finance Manager at Anzee Corporation in North America and is responsible for customer invoicing. Anzee Corporation provides a **Software as a Service** (**SaaS**) platform to its customers and charges customers based on the platform usage. Anzee's customers have raised complaints regarding incorrect monthly invoices. In their words, *"Invoices for the previous month are available on the 1st day of the next month, which is fine, but a part of our usage is not accounted for. This messes up our accounting."*

Currently, Jacob generates invoices in a manual way by getting the data for customer's payments and platform usage from the platform's database records. With an increasing number of customers each month, Jacob realizes that the manual process is going to be laborious and time consuming. He also wants someone to look at the issue the customers are complaining about. Can we help Jacob?

Getting ready

In this recipe, we'll use all the built-in Python modules that we've used in the previous recipes as well as install the `fpdf` module for generating the PDF invoice to solve Jacob's need for an automated way to prepare invoices for his customers. We install the module using Python `pip`:

```
(date)chetans-MacBookPro:ch12 Chetan$ pip install fpdf
You are using pip version 7.1.0, however version 9.0.1 is
available.
You should consider upgrading via the
```

```
    'pip install --upgrade pip' command.
Collecting fpdf
Downloading fpdf-1.7.2.tar.gz
Building wheels for collected packages: fpdf
Running setup.py bdist_wheel for fpdf
Stored in directory: /Users/chetan/Library/Caches/pip/wheels/
c9/22/63/16731bdbcccd4a91f5f9e9bea98b1e51855a678f2c6510ae76
Successfully built fpdf
Installing collected packages: fpdf
Successfully installed fpdf-1.7.2
```

How to do it…

1. Let's get started by looking at the database records. Anzee Corporation uses MongoDB to store the records of customer payments and charges for the month. For this example, lets assume, the payment records of Anzee Corporation are stored in JSON format.
2. We use a `users` document that contains the list of all the users with fields such as `id`, `name`, `city`, and `timezone`, like any other user table would. It also maintains the records of all the payments done by the users in `payments`, which contains the `id` of the user who paid for the platform services, the amount paid, and the date on which the amount is paid.
3. The timestamp for all the payments done is in the UTC format. Like payments, it also maintains the `usage` records that again contain the user ID of the person who used the platform, the amount she was charged for the usage, and the time at which she was charged:

```
users = [{"id":12, "name":"John", "city":"New York",
          "timezone":"US/Eastern"},
         {"id":13, "name":"Johny", "city":"Indiana",
          "timezone":"US/Central"}]

#All time stamps are in UTC
payments = [{"id":12, "amount":12.00,
             "created_at":"2016-11-29T11:46:07.141Z"},
            {"id":13, "amount":22.00,
             "created_at":"2016-11-30T23:46:07.141Z"},
            {"id":12, "amount":5.00,
             "created_at":"2016-12-01T01:00:00.141Z"}]

usage = [{"id":12, "charge":5.00,
          "created_at":"2016-11-29T11:46:07.141Z"}]
```

4. OK cool, now that we have all the data, let's work on writing the code to generate our invoice. We start by writing methods to get the payments and usages for the users for a given month. The following code snippet does this task for us:

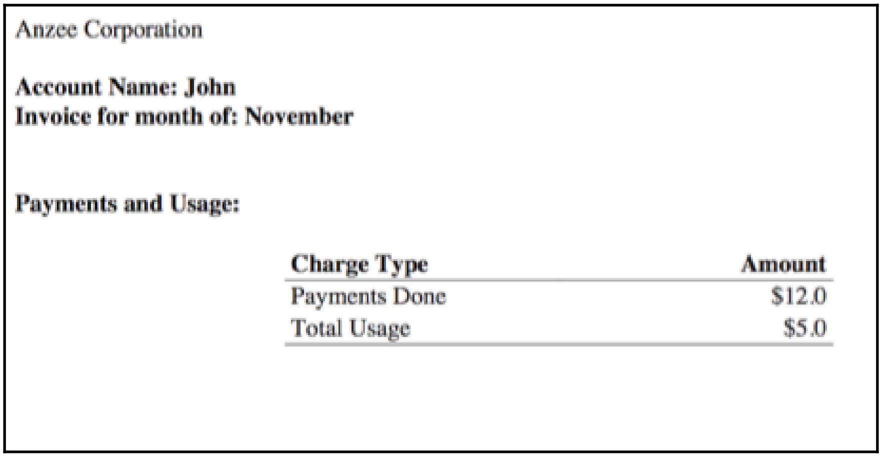
```
user_ids = []
user_names = []
for usr in users:
user_ids.append(usr["id"])
user_names.append(usr["name"])

def get_payments(user_id, month):
    tz = [ x for x in users if x["id"] == user_id]
    tot_payment = 0.00
    for p in payments:
        dt = datetime.strptime(p["created_at"],
                               '%Y-%m-%dT%H:%M:%S.%fZ')
        if p["id"] == user_id and dt.month == month:
           tot_payment += p["amount"]
    return tot_payment

def get_usage(user_id, month):
tz = [ x for x in users if x["id"] == user_id]
    tot_usage = 0.00
    for u in usage:
      dt = datetime.strptime(u["created_at"],
                             '%Y-%m-%dT%H:%M:%S.%fZ')
      if u["id"] == user_id and dt.month == month:
         tot_usage += u["charge"]
    return tot_usage
```

5. Next, let's write the code to generate a PDF invoice in an automated manner, as Jacob wanted for his platform. We use the `fpdf` module for this purpose. The following code generates the invoice:

```
def get_invoice(user_name, user_id, month):
    html = """
<p>Anzee Corporation</p><br>
<b><p>Account Name: """ + user_name + """</p>
<p>Invoice for month of: """ +
   str(calendar.month_name[month]) + """</p></b>
<br><br>
<p><b>Payments and Usage:</b></p>
<table align="center" width="50%">
  <thead>
    <tr>
      <th align="left" width="50%">Charge Type</th>
```

```
                <th align="right" width="50%">Amount</th>
              </tr>
            </thead>
            <tbody>
              <tr>
                <td>Payments Done</td>
                <td align="right">$""" + str(get_payments
                                 (user_id, month)) + """</td>
              </tr>
              <tr>
                <td>Total Usage</td>
                <td align="right">$""" + str(get_usage
                                 (user_id, month)) + """</td>
              </tr>
            </tbody>
          </table>
          <br><br>
            """
            return html

          class MyFPDF(FPDF, HTMLMixin):
          pass

          html = get_invoice("John", 12, 11)
          pdf=MyFPDF()
          pdf.add_page()
          pdf.write_html(html)
          pdf.output('invoice.pdf','F')
```

6. If we run the preceding code snippet in entirety, we get the generated invoice, which looks like the following screenshot:



Anzee Corporation

Account Name: John
Invoice for month of: November

Payments and Usage:

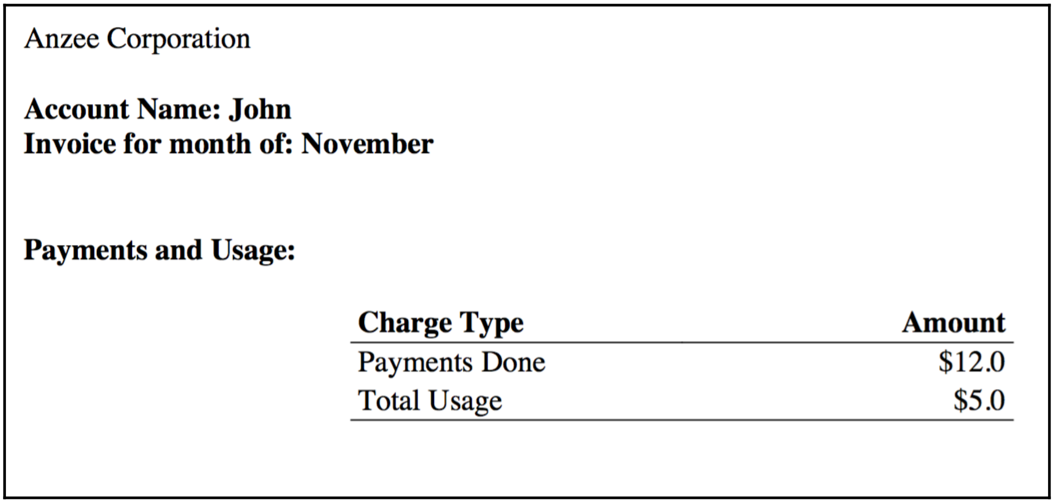
Charge Type	**Amount**
Payments Done	$12.0
Total Usage	$5.0

7. OK, this is cool! We could generate the invoice like Jacob expected. We are now able to save a lot of time for him by automating the complete process. But then he also wanted us to look at the customer complaints regarding the invoice not containing accurate information. Let's see what could have happened.
8. Now, Anzee Corporation stores all the transactions and usage timestamps in UTC so that it becomes very easy for them to retrieve the time and show it to the user based on the user's time zone. So when we look up all records to get the transactions in that month for the invoice, we're looking at the UTC timestamps and not the time in the user's time zones.
9. For instance, if you look deeper into the JSON data, there is a mention of another payment made by `John`, user ID `12`, which has a `created_at` timestamp as `2016-12-01T01:00:00.141Z`. This time may not fall under the month of November from a UTC perspective, but the user who made the payment belongs to the US/Eastern time zone. So, 1 am on December 1, 2016 in UTC is actually 8 pm on November 30 in the Eastern time zone. Obviously, the user doesn't find his payment featuring in the invoice.
10. The following code snippet solves the problem by generating invoices based on the user time zones:

```
from datetime import datetime
import pytz
from pytz import timezone
from fpdf import FPDF, HTMLMixin
import calendar

users = [{"id":12, "name":"John",
          "city":"New York", "timezone":"US/Eastern"},
         {"id":13, "name":"Johny",
          "city":"Indiana", "timezone":"US/Central"}]

#All time stamps are in UTC
payments = [{"id":12, "amount":12.00,
             "created_at":"2016-11-29T11:46:07.141Z"},
            {"id":13, "amount":22.00,
             "created_at":"2016-11-30T23:46:07.141Z"},
            {"id":12, "amount":5.00,
             "created_at":"2016-12-01T01:00:00.141Z"}]
usage = [{"id":12, "charge":5.00,
          "created_at":"2016-11-29T11:46:07.141Z"}]

user_ids = []
user_names = []
for usr in users:
    user_ids.append(usr["id"])
```

```
user_names.append(usr["name"])

def get_payments(user_id, month):
    tz = [ x for x in users if x["id"] == user_id]
    tot_payment = 0.00
    for p in payments:
        dt = datetime.strptime(p["created_at"],
                               '%Y-%m-%dT%H:%M:%S.%fZ')
        dt = dt.replace(tzinfo=pytz.UTC)
        dt = dt.astimezone(timezone(tz[0]["timezone"]))
        if p["id"] == user_id and dt.month == month:
           tot_payment += p["amount"]
    return tot_payment
def get_usage(user_id, month):
    tz = [ x for x in users if x["id"] == user_id]
    tot_usage = 0.00
    for u in usage:
        dt = datetime.strptime(u["created_at"],
                               '%Y-%m-%dT%H:%M:%S.%fZ')
        dt = dt.replace(tzinfo=pytz.UTC)
        dt = dt.astimezone(timezone(tz[0]["timezone"]))
        if u["id"] == user_id and dt.month == month:
           tot_usage += u["charge"]
return tot_usage

def get_invoice(user_name, user_id, month):
  html = """
<p>Anzee Corporation</p><br>
<b><p>Account Name: """ + user_name + """</p>
<p>Invoice for month of: """ +
   str(calendar.month_name[month]) + """</p></b>
<br><br>
<p><b>Payments and Usage:</b></p>
<table align="center" width="50%">
  <thead>
    <tr>
      <th align="left" width="50%">Charge Type</th>
      <th align="right" width="50%">Amount</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Payments Done</td>
      <td align="right">$""" + str(get_payments(
                     user_id, month)) + """</td>
    </tr>
    <tr>
      <td>Total Usage</td>
      <td align="right">$""" + str(get_usage(
                  user_id, month)) + """</td>
    </tr>
  </tbody>
</table>
<br><br>
  """
  return html


class MyFPDF(FPDF, HTMLMixin):
      pass

html = get_invoice("John", 12, 11)
pdf=MyFPDF()
pdf.add_page()
pdf.write_html(html)
pdf.output('invoice.pdf','F')
```

The output for the preceding code snippet is shown in the following screenshot. See how the payments column now reflects the correct data and includes the $5 payment done at 8 pm on November 30, 2016, taking the total to *$12 + $5 = $17*:



Anzee Corporation

Account Name: John
Invoice for month of: November

Payments and Usage:

Charge Type	**Amount**
Payments Done	$12.0
Total Usage	$5.0

How it works…

We first looked at automating the invoice generation for Jacob. We parsed the JSON data for all the users and calculated the payments and usage for all the users for the month of November.

We developed `get_payments(user_id, month)` and `get_usage(user_id, month)` to go through the `payments` and `usage` records and picked the records for the month of November. We did this by working on the `created_at` JSON strings and converting them to date/time objects, with `dt = datetime.strptime(u["created_at"], '%Y-%m-%dT%H:%M:%S.%fZ')`.

But as we understood in the previous section, just converting the string to a timestamp didn't help, as we didn't consider the time with respect to the user's time zone. For this, we used the date/time object, `dt`, to convert it to the UTC time zone with `dt.replace(tzinfo=pytz.UTC)`, and then converted `dt` to reflect the time in the user's time zone with the `dt.astimezone(timezone(<>))` method. This way, we could get the payment time in the user's time zone and the invoice data reflected correct figures for the month of November.

Next, we created HTML content for the invoice by adding the appropriate username, time of invoice, and stored this in the `html` variable. Later, we created a `MyFPDF` class that inherited `FPDF` and `HTMLMixin` classes from the `fpdf` module. The `MyFPDP` class was then used to create a `pdf` object, which represented an empty PDF file object. We added a page to the `pdf` object with the `add_page()` method and updated it with HTML content (our invoice content) with the `write_html(html)` method. Eventually, we dumped the `pdf` object with all the data on the disk with the `output(<filename>)` method.

There's more…

There are many other interesting use cases with time and time zone operations in Python. It can get tricky if not used well, as we saw in our last examples. As a general guideline, I recommend you to:

- Always use `datetime` objects that are time zone aware. You will never go wrong with this approach. It will always serve you as a reminder.
- Return the `datetime` in an ISO format that also returns you the time zone information for the given object.

Hope you liked this chapter and enjoyed the examples! Stay tuned.

Index