Module 5: Web Scrapping

What is Web Scraping?

- Web Scripting is an automatic method to obtain large amounts of data from websites.
- Most of this data is unstructured data in an HTML format which is then converted into structured data in a spreadsheet or a database so that it can be used in various applications.

Crawler and Scrapper

- Web scraping requires two parts namely the **crawler** and the **scraper**.
- The **crawler** is an artificial intelligence algorithm that browses the web to search the data required by following the links across the internet.
- The **scraper,** on the other hand, is a specific tool created to extract the data from the website. The design of the scraper can vary greatly according to the complexity and scope of the project so that it can quickly and accurately extract the data.

Uses of Web Scraping

- Price Monitoring
- Market Research
- News Monitoring
- Sentiment Analysis
- Email Marketing

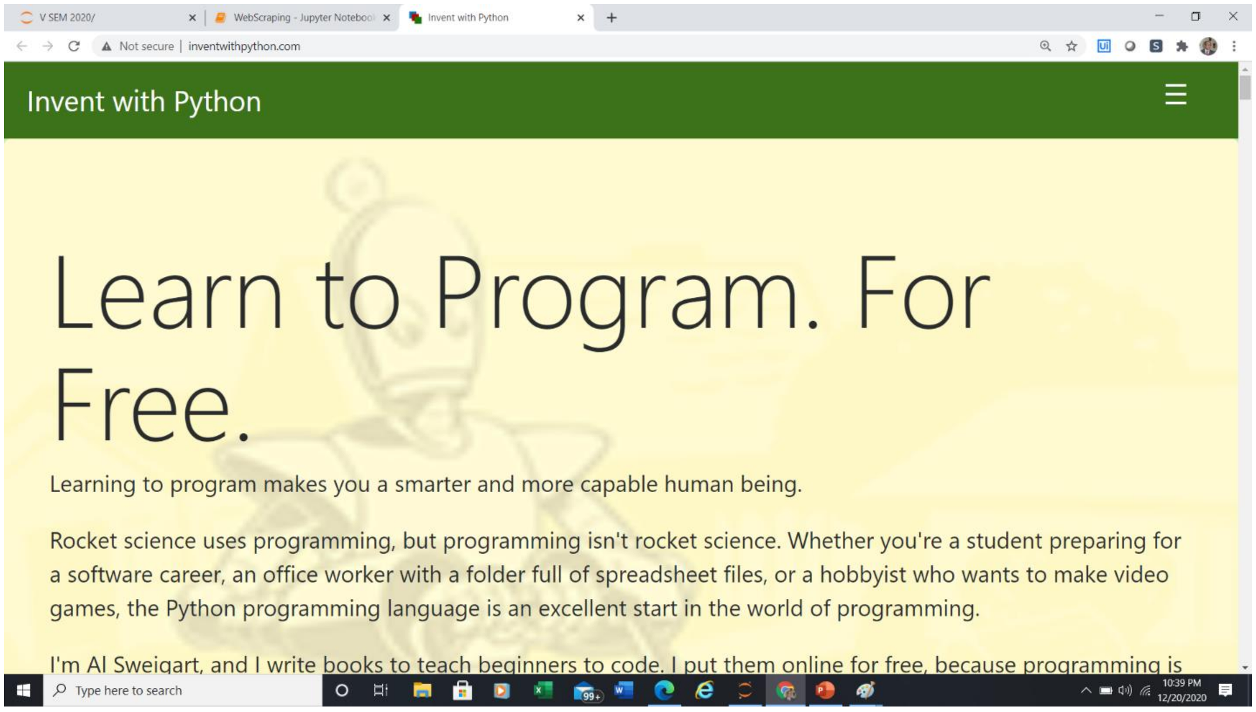
WEB SCRAPING

Web scraping is the term for using a program to download and process content from the Web. For example, Google runs many web scraping programs to index web pages for its search engine. In this chapter, you will learn about several modules that make it easy to scrape web pages in Python.

- `webbrowser`. Comes with Python and opens a browser to a specific page.
- **Requests**. Downloads files and web pages from the Internet.
- **Beautiful Soup**. Parses HTML, the format that web pages are written in.
- **Selenium**. Launches and controls a web browser. Selenium is able to fill in forms and simulate mouse clicks in this browser.

Project: mapIt.py with the webbrowser Module

```
import webbrowser
webbrowser.open('http://inventwithpython.com/')
```



Invent with Python

Learn to Program. For Free.

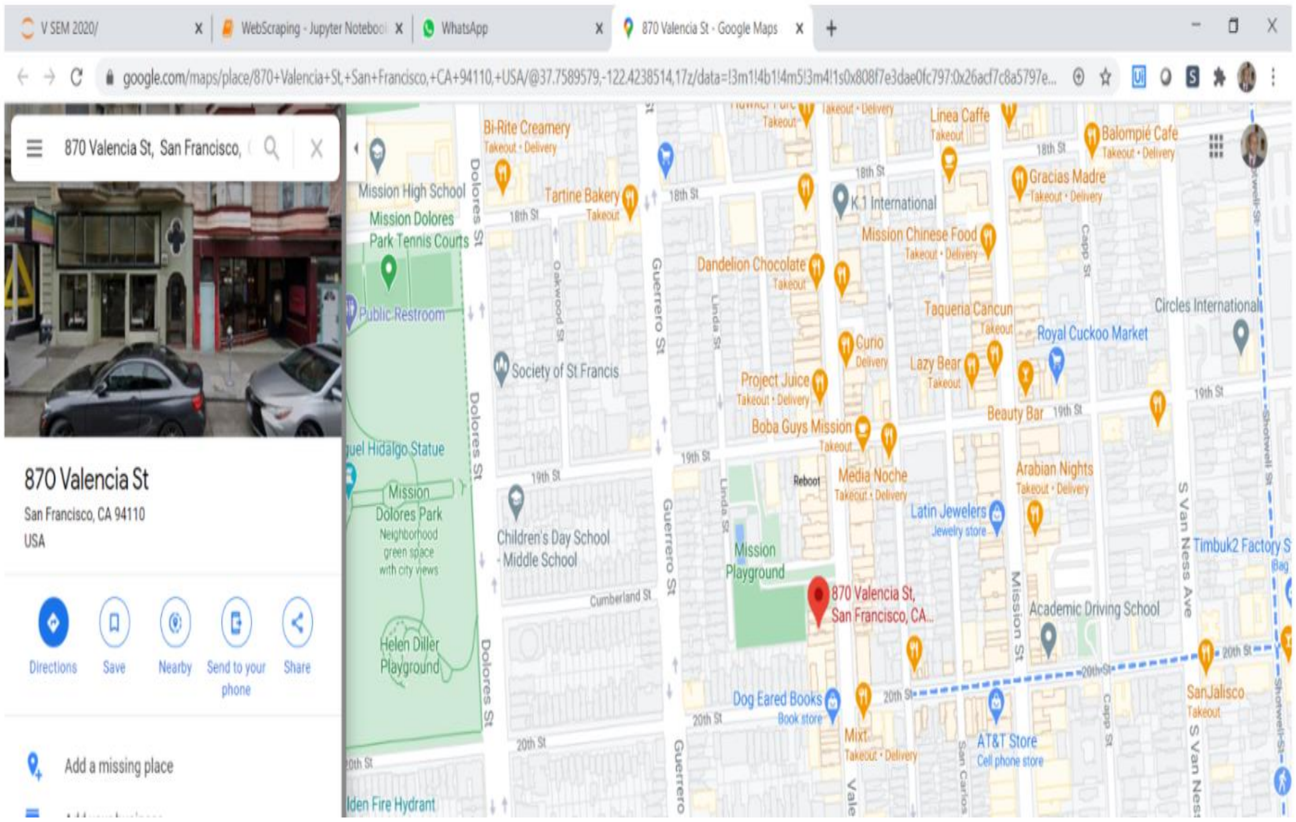
Learning to program makes you a smarter and more capable human being.

Rocket science uses programming, but programming isn't rocket science. Whether you're a student preparing for a software career, an office worker with a folder full of spreadsheet files, or a hobbyist who wants to make video games, the Python programming language is an excellent start in the world of programming.

I'm Al Sweigart, and I write books to teach beginners to code. I put them online for free, because programming is

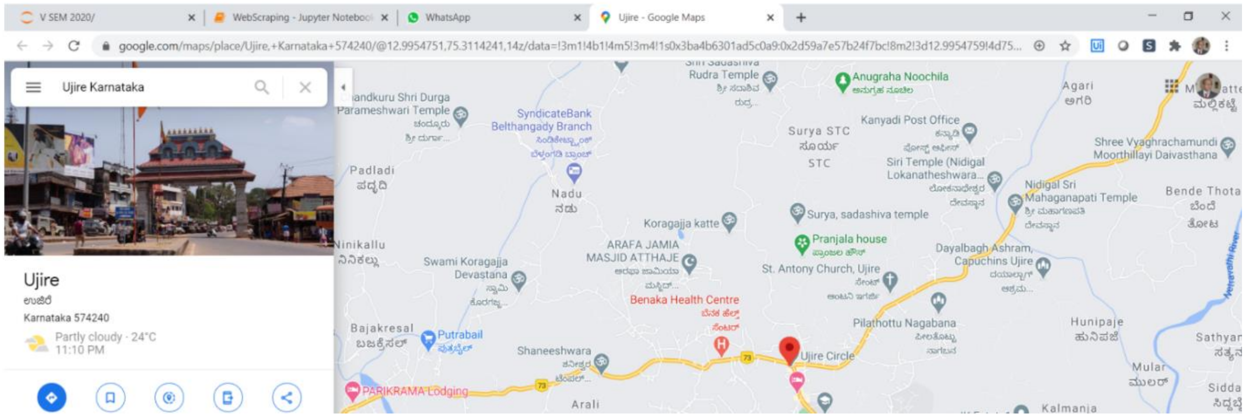
```python
import webbrowser
addressline = ['870', 'Valencia', 'St, ', 'San', 'Francisco, ', 'CA', '94110']
address =  ' '.join(addressline[:])
webbrowser.open('https://www.google.com/maps/place/' + address)
```

```
True
```



V SEM 2020/
WebScraping - Jupyter Noteboo
WhatsApp
870 Valencia St - Google Maps
google.com/maps/place/870+Valencia+St,+San+Francisco,+CA+94110,+USA/@37.7589579,-122.4238514,17z/data=!3m1!4b1!4m5!3m4!1s0x808f7e3dae0fc797:0x26acf7c8a5797e...
870 Valencia St, San Francisco,
870 Valencia St
San Francisco, CA 94110
USA
Directions
Save
Nearby
Send to your phone
Share
Add a missing place
870 Valencia St, San Francisco, CA...
Mission High School
Mission Dolores Park
Mission Playground
Valencia St
Guerrero St
Dolores St
Mission St
18th St
19th St
20th St

```python
import webbrowser
addressline = ['Ujire', 'Karnataka']
address =  ' '.join(addressline[:])
webbrowser.open('https://www.google.com/maps/place/' + address)
```



Ujire - Google Maps
google.com/maps/place/Ujire,+Karnataka+574240/@12.9954751,75.3114241,14z/data=!3m1!4b1!4m5!3m4!1s0x3ba4b6301ad5c0a9:0x2d59a7e57b24f7bc!8m2!3d12.9954759!4d75...
Ujire Karnataka
Ujire
Karnataka 574240
Partly cloudy · 24°C
11:10 PM

Ideas for Similar Programs

As long as you have a URL, the `webbrowser` module lets users cut out the step of opening the browser and directing themselves to a website. Other programs could use this functionality to do the following:

- Open all links on a page in separate browser tabs.
- Open the browser to the URL for your local weather.
- Open several social network sites that you regularly check.

Downloading Files from the Web with the requests Module

The `requests` module lets you easily download files from the Web without having to worry about complicated issues such as network errors, connection problems, and data compression. The `requests` module doesn't come with Python, so you'll have to install it first. From the command line, run **`pip install requests`**. (Appendix A has additional details on how to install third-party modules.)

The `requests` module was written because Python's `urllib2` module is too complicated to use. In fact, take a permanent marker and black out this entire paragraph. Forget I ever mentioned `urllib2`. If you need to download things from the Web, just use the `requests` module.

Next, do a simple test to make sure the `requests` module installed itself correctly. Enter the following into the interactive shell:

```
>>> import requests
```

If no error messages show up, then the `requests` module has been successfully installed.

DOWNLOADING A WEB PAGE WITH THE REQUESTS.GET() FUNCTION

The `requests.get()` function takes a string of a URL to download. By calling `type()` on `requests.get()`'s return value, you can see that it returns a `Response` object, which contains the response that the web server gave for your request. I'll explain the `Response` object in more detail later, but for now, enter the following into the interactive shell while your computer is connected to the Internet:

```
   >>> import requests
   >>> res = requests.get('https://automatetheboringstuff.com/files/rj.txt')
   >>> type(res)
   <class 'requests.models.Response'>
❶ >>> res.status_code == requests.codes.ok
   True
   >>> len(res.text)
   178981
   >>> print(res.text[:250])
   The Project Gutenberg EBook of Romeo and Juliet, by William Shakespeare

   This eBook is for the use of anyone anywhere at no cost and with
   almost no restrictions whatsoever. You may copy it, give it away or
   re-use it under the terms of the Proje
```

The URL goes to a text web page for the entire play of *Romeo and Juliet*. You can tell that the request for this web page succeeded by checking the `status_code` attribute of the `Response` object. If it is equal to the value of `requests.codes.ok`, then everything went fine ❶. (Incidentally, the status code for "OK" in the HTTP protocol is 200. You may already be familiar with the 404 status code for "Not Found.")

If the request succeeded, the downloaded web page is stored as a string in the `Response` object's `text` variable. This variable holds a large string of the entire play; the call to `len(res.text)` shows you that it is more than

178,000 characters long. Finally, calling `print(res.text[:250])` displays only the first 250 characters.

CHECKING FOR ERRORS

As you've seen, the `Response` object has a `status_code` attribute that can be checked against `requests.codes.ok` to see whether the download succeeded. A simpler way to check for success is to call the `raise_for_status()` method on the `Response` object. This will raise an exception if there was an error downloading the file and will do nothing if the download succeeded. Enter the following into the interactive shell:

```
>>> res =
requests.get('http://inventwithpython.com/page_that_does_not_exist')
>>> res.raise_for_status()
Traceback (most recent call last):
  File "<pyshell#138>", line 1, in <module>
    res.raise_for_status()
  File "C:\Python34\lib\site-packages\requests\models.py", line 773, in
raise_for_status
    raise HTTPError(http_error_msg, response=self)
requests.exceptions.HTTPError: 404 Client Error: Not Found
```

The `raise_for_status()` method is a good way to ensure that a program halts if a bad download occurs. This is a good thing: You want your program to stop as soon as some unexpected error happens. If a failed download *isn't* a deal breaker for your program, you can wrap the `raise_for_status()` line with `try` and `except` statements to handle this error case without crashing.

```
import requests
res = requests.get('http://inventwithpython.com/page_that_does_not_exist')
try:
    res.raise_for_status()
except Exception as exc:
    print('There was a problem: %s' % (exc))
```

This `raise_for_status()` method call causes the program to output the following:

```
There was a problem: 404 Client Error: Not Found
```

Always call `raise_for_status()` after calling `requests.get()`. You want to be sure that the download has actually worked before your program continues.

Saving Downloaded Files to the Hard Drive

From here, you can save the web page to a file on your hard drive with the standard `open()` function and `write()` method. There are some slight differences, though. First, you must open the file in *write binary* mode by passing the string `'wb'` as the second argument to `open()`. Even if the page is in plaintext (such as the *Romeo and Juliet* text you downloaded earlier), you need to write binary data instead of text data in order to maintain the *Unicode encoding* of the text.

To write the web page to a file, you can use a `for` loop with the `Response` object's `iter_content()` method.

```
>>> import requests
>>> res = requests.get('https://automatetheboringstuff.com/files/rj.txt')
>>> res.raise_for_status()
>>> playFile = open('RomeoAndJuliet.txt', 'wb')
>>> for chunk in res.iter_content(100000):
        playFile.write(chunk)

100000
78981
>>> playFile.close()
```

The `iter_content()` method returns "chunks" of the content on each iteration through the loop. Each chunk is of the *bytes* data type, and you get to specify how many bytes each chunk will contain. One hundred thousand bytes is generally a good size, so pass `100000` as the argument to `iter_content()`.

The file *RomeoAndJuliet.txt* will now exist in the current working directory. Note that while the filename on the website was *rj.txt*, the file on your hard drive has a different filename. The `requests` module simply handles downloading the contents of web pages. Once the page is downloaded, it is simply data in your program. Even if you were to lose your Internet connection after downloading the web page, all the page data would still be on your computer.

The `write()` method returns the number of bytes written to the file. In the previous example, there were 100,000 bytes in the first chunk, and the remaining part of the file needed only 78,981 bytes.

To review, here's the complete process for downloading and saving a file:

1. Call `requests.get()` to download the file.
2. Call `open()` with `'wb'` to create a new file in write binary mode.
3. Loop over the `Response` object's `iter_content()` method.
4. Call `write()` on each iteration to write the content to the file.
5. Call `close()` to close the file.

That's all there is to the `requests` module! The `for` loop and `iter_content()` stuff may seem complicated compared to the `open()`/`write()`/`close()` workflow you've been using to write text files, but it's to ensure that the `requests` module doesn't eat up too much memory even if you download massive files. You can learn about the `requests` module's other features from *http://requests.readthedocs.org/*.

HTML

Before you pick apart web pages, you'll learn some HTML basics. You'll also see how to access your web browser's powerful developer tools, which will make scraping information from the Web much easier.

Resources for Learning HTML

Hypertext Markup Language (HTML) is the format that web pages are written in. This chapter assumes you have some basic experience with HTML, but if you need a beginner tutorial, I suggest one of the following sites:

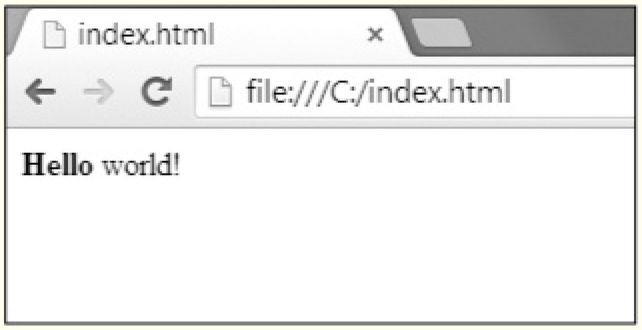
- *http://htmldog.com/guides/html/beginner/*
- *http://www.codecademy.com/tracks/web/*
- *https://developer.mozilla.org/en-US/learn/html/*

A Quick Refresher

In case it's been a while since you've looked at any HTML, here's a quick overview of the basics. An HTML file is a plaintext file with the *.html* file extension. The text in these files is surrounded by *tags*, which are words enclosed in angle brackets. The tags tell the browser how to format the web page. A starting tag and closing tag can enclose some text to form an *element*. The *text* (or *inner HTML*) is the content between the starting and closing tags. For example, the following HTML will display *Hello world!* in the browser, with *Hello* in bold:

```
<strong>Hello</strong> world!
```

This HTML will look like Figure 11-1 in a browser.



Hello world!

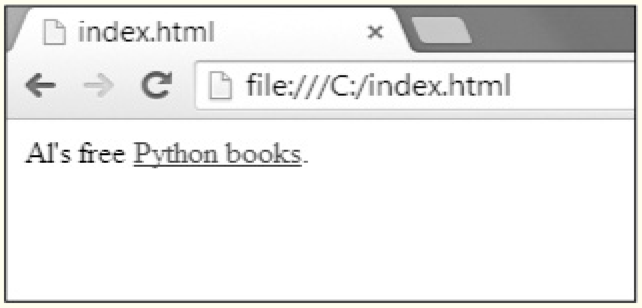
Figure 11-1. *Hello world!* rendered in the browser

The opening `<strong>` tag says that the enclosed text will appear in bold. The closing `</strong>` tags tells the browser where the end of the bold text is.

There are many different tags in HTML. Some of these tags have extra properties in the form of *attributes* within the angle brackets. For example, the `<a>` tag encloses text that should be a link. The URL that the text links to is determined by the `href` attribute. Here's an example:

```
Al's free <a href="http://inventwithpython.com">Python books</a>.
```

This HTML will look like Figure 11-2 in a browser.



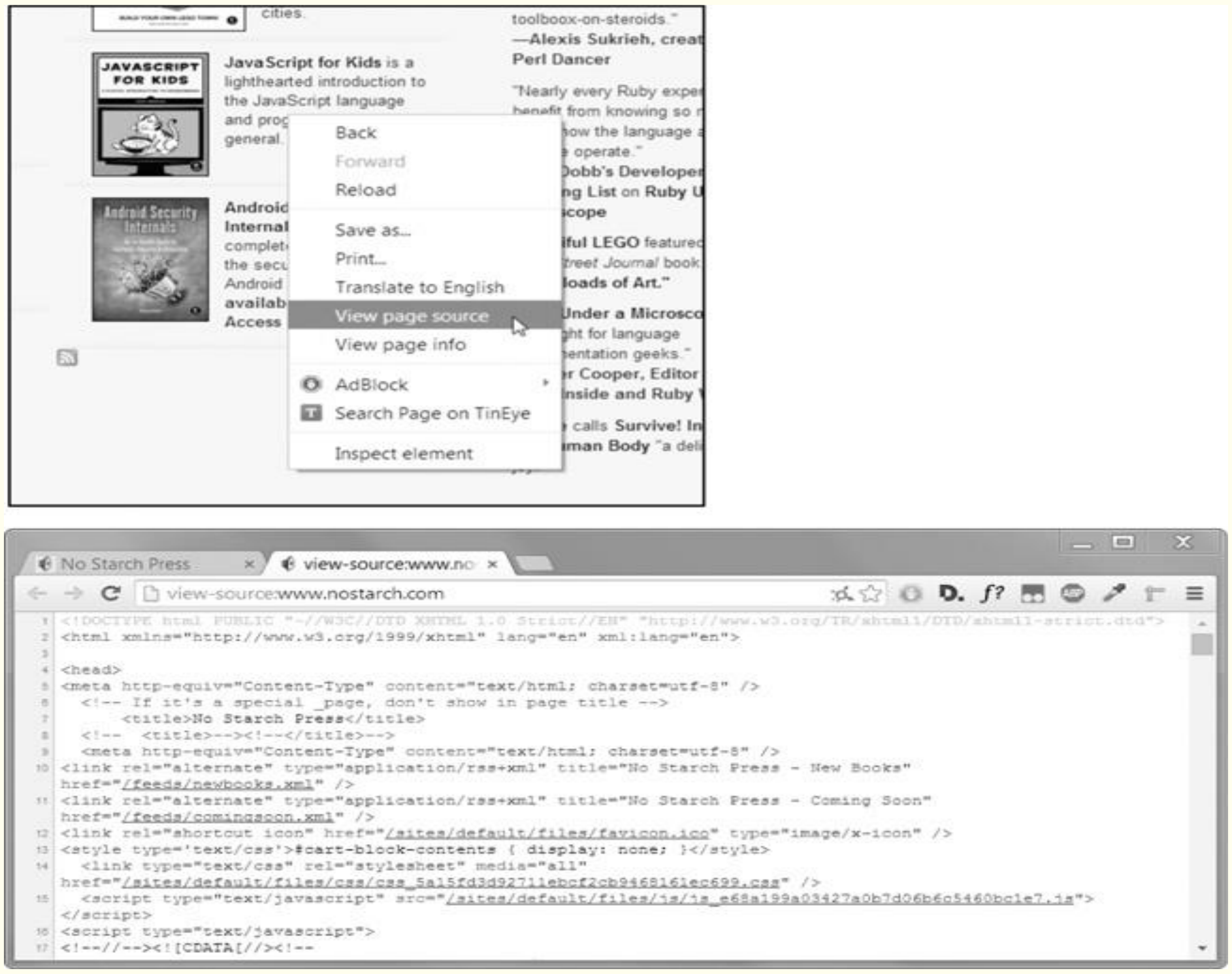
Al's free Python books.

Figure 11-2. The link rendered in the browser

Some elements have an `id` attribute that is used to uniquely identify the element in the page. You will often instruct your programs to seek out an element by its `id` attribute, so figuring out an element's `id` attribute using the browser's developer tools is a common task in writing web scraping programs.

VIEWING THE SOURCE HTML OF A WEB PAGE

You'll need to look at the HTML source of the web pages that your programs will work with. To do this, right-click (or CTRL-click on OS X) any web page in your web browser, and select **View Source** or **View page source** to see the HTML text of the page (see Figure 11-3). This is the text your browser actually receives. The browser knows how to display, or *render*, the web page from this HTML.



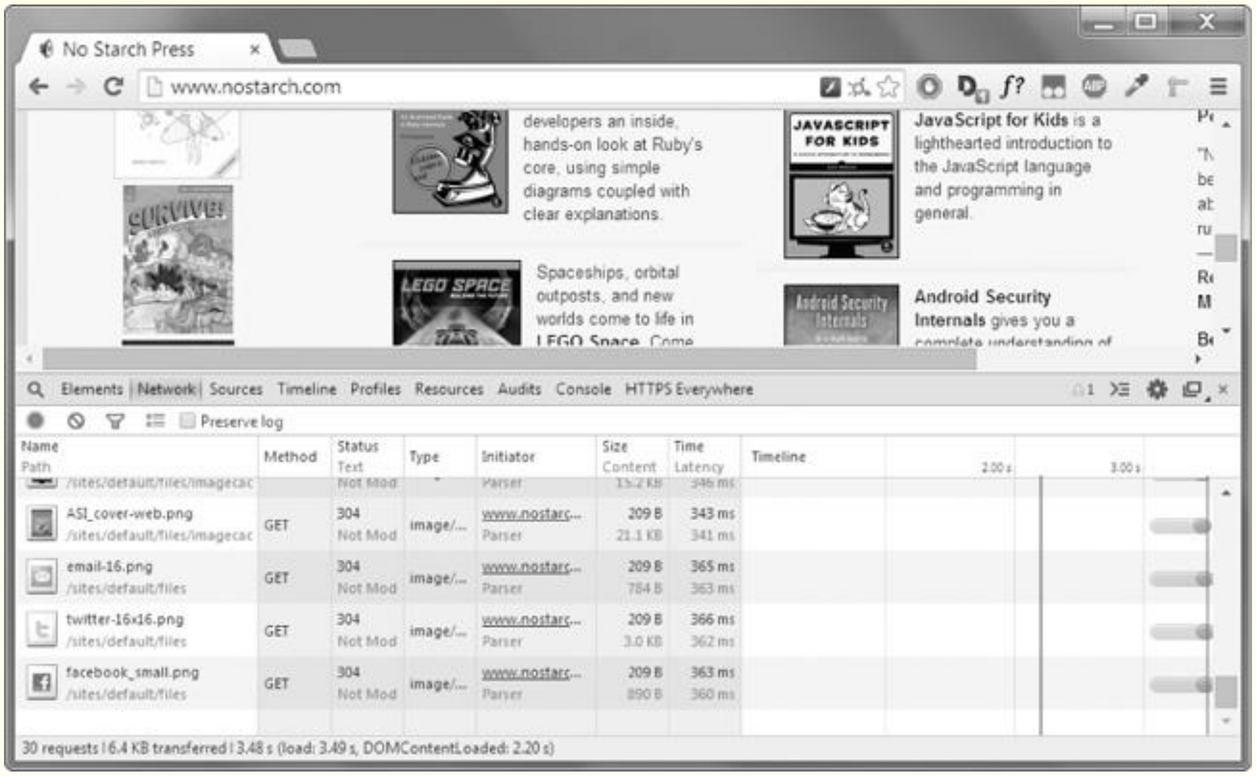
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">

<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <!-- If it's a special _page, don't show in page title -->
      <title>No Starch Press</title>
  <!--  <title>--><!--</title>-->
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<link rel="alternate" type="application/rss+xml" title="No Starch Press - New Books"
href="/feeds/newbooks.xml" />
<link rel="alternate" type="application/rss+xml" title="No Starch Press - Coming Soon"
href="/feeds/comingsoon.xml" />
<link rel="shortcut icon" href="/sites/default/files/favicon.ico" type="image/x-icon" />
<style type='text/css'>#cart-block-contents { display: none; }</style>
  <link type="text/css" rel="stylesheet" media="all"
href="/sites/default/files/css/css_5a15fd3d92711ebcf2cb946816lec699.css" />
  <script type="text/javascript" src="/sites/default/files/js/js_e68a199a03427a0b7d06b6c5460bc1e7.js">
</script>
<script type="text/javascript">
<!--//--><![CDATA[//><!--
```

Figure 11-3. Viewing the source of a web page

I highly recommend viewing the source HTML of some of your favorite sites. It's fine if you don't fully understand what you are seeing when you look at the source. You won't need HTML mastery to write simple web scraping programs—after all, you won't be writing your own websites. You just need enough knowledge to pick out data from an existing site.

OPENING YOUR BROWSER'S DEVELOPER TOOLS

In addition to viewing a web page's source, you can look through a page's HTML using your browser's developer tools. In Chrome and Internet Explorer for Windows, the developer tools are already installed, and you can press F12 to make them appear (see Figure 11-4). Pressing F12 again will make the developer tools disappear. In Chrome, you can also bring up the developer tools by selecting View ▸ Developer ▸ Developer Tools. In OS X, pressing ⌘-OPTION-I will open Chrome's Developer Tools.



Name / Path	Method	Status / Text	Type	Initiator	Size / Content	Time / Latency
ASI_cover-web.png /sites/default/files/imagecac	GET	304 Not Mod	image/...	www.nostarc... Parser	209 B 21.1 KB	343 ms 341 ms
email-16.png /sites/default/files	GET	304 Not Mod	image/...	www.nostarc... Parser	209 B 784 B	365 ms 363 ms
twitter-16x16.png /sites/default/files	GET	304 Not Mod	image/...	www.nostarc... Parser	209 B 3.0 KB	366 ms 362 ms
facebook_small.png /sites/default/files	GET	304 Not Mod	image/...	www.nostarc... Parser	209 B 890 B	363 ms 360 ms

30 requests | 6.4 KB transferred | 3.48 s (load: 3.49 s, DOMContentLoaded: 2.20 s)

Figure 11-4. The Developer Tools window in the Chrome browser

In Firefox, you can bring up the Web Developer Tools Inspector by pressing CTRL-SHIFT-C on Windows and Linux or by pressing ⌘-

OPTION-C on OS X. The layout is almost identical to Chrome's developer tools.

In Safari, open the Preferences window, and on the Advanced pane check the **Show Develop menu in the menu bar** option. After it has been enabled, you can bring up the developer tools by pressing ⌘-OPTION-I.

After enabling or installing the developer tools in your browser, you can right-click any part of the web page and select **Inspect Element** from the context menu to bring up the HTML responsible for that part of the page. This will be helpful when you begin to parse HTML for your web scraping programs.

Don't Use Regular Expressions to Parse HTML

Locating a specific piece of HTML in a string seems like a perfect case for regular expressions. However, I advise you against it. There are many different ways that HTML can be formatted and still be considered valid HTML, but trying to capture all these possible variations in a regular expression can be tedious and error prone. A module developed specifically for parsing HTML, such as Beautiful Soup, will be less likely to result in bugs.

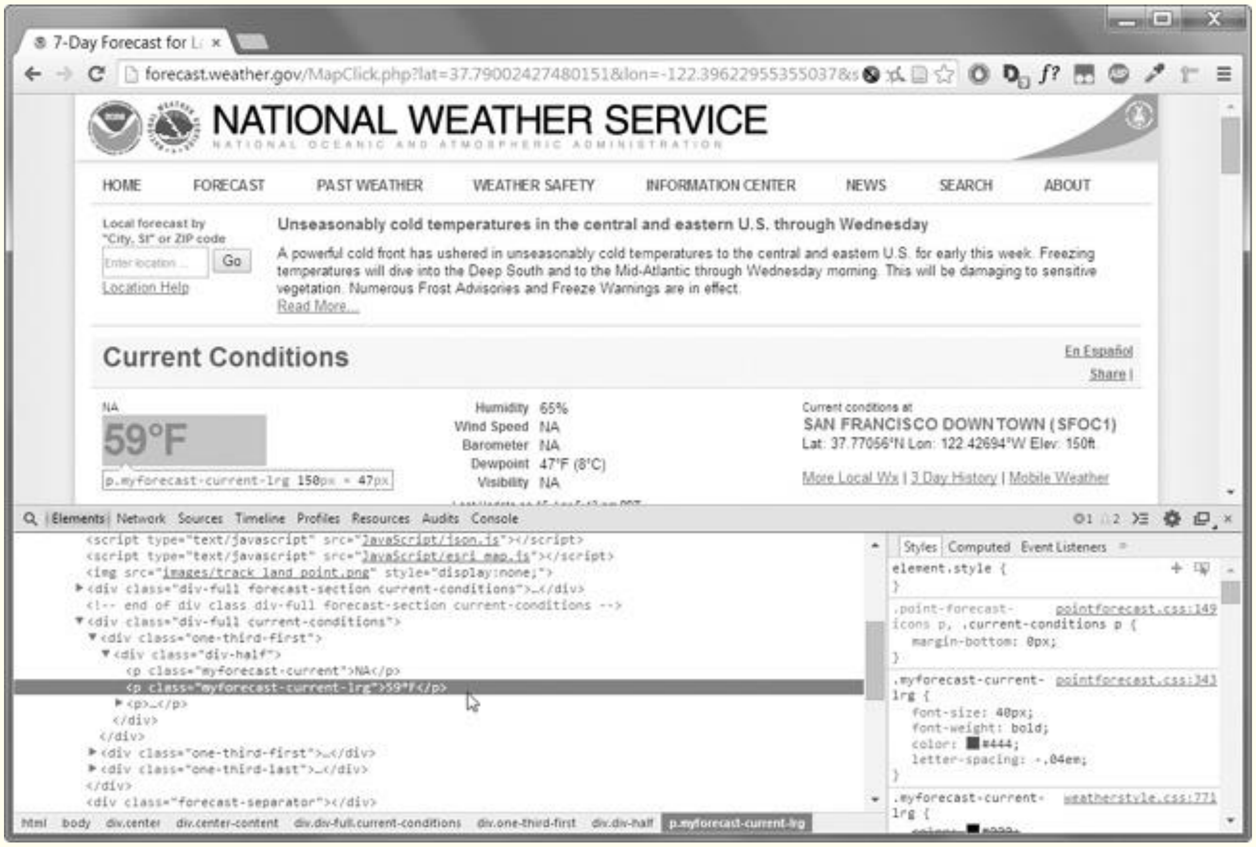
You can find an extended argument for why you shouldn't to parse HTML with regular expressions at *http://stackoverflow.com/a/1732454/1893164/*.

USING THE DEVELOPER TOOLS TO FIND HTML ELEMENTS

Once your program has downloaded a web page using the `requests` module, you will have the page's HTML content as a single string value. Now you need to figure out which part of the HTML corresponds to the information on the web page you're interested in.

This is where the browser's developer tools can help. Say you want to write a program to pull weather forecast data from *http://weather.gov/*. Before writing any code, do a little research. If you visit the site and search for the 94105 ZIP code, the site will take you to a page showing the forecast for that area.

What if you're interested in scraping the temperature information for that ZIP code? Right-click where it is on the page (or CONTROL-click on OS X) and select **Inspect Element** from the context menu that appears. This will bring up the Developer Tools window, which shows you the HTML that produces this particular part of the web page. Figure 11-5 shows the developer tools open to the HTML of the temperature.



Unseasonably cold temperatures in the central and eastern U.S. through Wednesday

A powerful cold front has ushered in unseasonably cold temperatures to the central and eastern U.S. for early this week. Freezing temperatures will dive into the Deep South and to the Mid-Atlantic through Wednesday morning. This will be damaging to sensitive vegetation. Numerous Frost Advisories and Freeze Warnings are in effect.
Read More...

Current Conditions

NA
59°F
p.myforecast-current-lrg 150px × 47px

Humidity	65%
Wind Speed	NA
Barometer	NA
Dewpoint	47°F (8°C)
Visibility	NA

Current conditions at
SAN FRANCISCO DOWNTOWN (SFOC1)
Lat: 37.77056°N Lon: 122.42694°W Elev: 150ft

More Local Wx | 3 Day History | Mobile Weather

```
<script type="text/javascript" src="JavaScript/json.js"></script>
<script type="text/javascript" src="JavaScript/esri_map.js"></script>
<img src="images/track_land_point.png" style="display:none;">
▶<div class="div-full forecast-section current-conditions">…</div>
<!-- end of div class div-full forecast-section current-conditions -->
▼<div class="div-full current-conditions">
  ▼<div class="one-third-first">
    ▼<div class="div-half">
       <p class="myforecast-current">NA</p>
       <p class="myforecast-current-lrg">59°F</p>
     ▶<p>…</p>
     </div>
   </div>
  ▶<div class="one-third-first">…</div>
  ▶<div class="one-third-last">…</div>
</div>
<div class="forecast-separator"></div>
```

```
element.style {
}
.point-forecast-        pointforecast.css:149
icons p, .current-conditions p {
  margin-bottom: 0px;
}
.myforecast-current-    pointforecast.css:343
lrg {
  font-size: 40px;
  font-weight: bold;
  color: ■#444;
  letter-spacing: -.04em;
}
.myforecast-current-    weatherstyle.css:771
lrg {
```

Figure 11-5. Inspecting the element that holds the temperature text with the developer tools

From the developer tools, you can see that the HTML responsible for the temperature part of the web page is `<p class="myforecast-current -lrg">59°F</p>`. This is exactly what you were looking for! It seems that the temperature information is contained inside a `<p>` element with the `myforecast-current-lrg` class. Now that you know what you're looking for, the `BeautifulSoup` module will help you find it in the string.

Parsing HTML with the BeautifulSoup Module

Beautiful Soup is a module for extracting information from an HTML page (and is much better for this purpose than regular expressions). The `BeautifulSoup` module's name is `bs4` (for Beautiful Soup, version 4). To install it, you will need to run `pip install beautifulsoup4` from the command line. (Check out Appendix A for instructions on installing third-party modules.) While `beautifulsoup4` is the name used for installation, to import Beautiful Soup you run `import bs4`.

For this chapter, the Beautiful Soup examples will *parse* (that is, analyze and identify the parts of) an HTML file on the hard drive. Open a new file editor window in IDLE, enter the following, and save it as *example.html*. Alternatively, download it from *http://nostarch.com/automatestuff/*.

```
<!-- This is the example.html example file. -->

<html><head><title>The Website Title</title></head>
<body>
<p>Download my <strong>Python</strong> book from <a href="http://
inventwithpython.com">my website</a>.</p>
<p class="slogan">Learn Python the easy way!</p>
<p>By <span id="author">Al Sweigart</span></p>
</body></html>
```

As you can see, even a simple HTML file involves many different tags and attributes, and matters quickly get confusing with complex websites. Thankfully, Beautiful Soup makes working with HTML much easier.

Creating a BeautifulSoup Object from HTML

The `bs4.BeautifulSoup()` function needs to be called with a string containing the HTML it will parse. The `bs4.BeautifulSoup()` function returns is a `BeautifulSoup` object. Enter the following into the interactive shell while your computer is connected to the Internet:

```
>>> import requests, bs4
>>> res = requests.get('http://nostarch.com')
>>> res.raise_for_status()
>>> noStarchSoup = bs4.BeautifulSoup(res.text)
>>> type(noStarchSoup)
<class 'bs4.BeautifulSoup'>
```

This code uses `requests.get()` to download the main page from the No Starch Press website and then passes the `text` attribute of the response to `bs4.BeautifulSoup()`. The `BeautifulSoup` object that it returns is stored in a variable named `noStarchSoup`.

You can also load an HTML file from your hard drive by passing a `File` object to `bs4.BeautifulSoup()`. Enter the following into the interactive shell (make sure the *example.html* file is in the working directory):

```
>>> exampleFile = open('example.html')
>>> exampleSoup = bs4.BeautifulSoup(exampleFile)
>>> type(exampleSoup)
<class 'bs4.BeautifulSoup'>
```

Once you have a `BeautifulSoup` object, you can use its methods to locate specific parts of an HTML document.

FINDING AN ELEMENT WITH THE SELECT() METHOD

You can retrieve a web page element from a `BeautifulSoup` object by calling the `select()`method and passing a string of a CSS *selector* for the element you are looking for. Selectors are like regular expressions: They specify a pattern to look for, in this case, in HTML pages instead of general text strings.

A full discussion of CSS selector syntax is beyond the scope of this book (there's a good selector tutorial in the resources at *http://nostarch.com/automatestuff/*), but here's a short introduction to selectors. Table 11-2 shows examples of the most common CSS selector patterns.

Table 11-2. Examples of CSS Selectors

Selector passed to the `select()` method	**Will match...**
`soup.select('div')`	All elements named `<div>`
`soup.select('#author')`	The element with an `id` attribute of `author`
`soup.select('.notice')`	All elements that use a CSS `class` attribute named `notice`
`soup.select('div span')`	All elements named `<span>` that are within an element named `<div>`

Selector passed to the `select()` method	**Will match...**
`soup.select('div > span')`	All elements named `<span>` that are *directly* within an element named `<div>`, with no other element in between
`soup.select('input[name]')`	All elements named `<input>` that have a `name` attribute with any value
`soup.select('input[type="button"]')`	All elements named `<input>` that have an attribute named `type` with value `button`

The various selector patterns can be combined to make sophisticated matches. For example, `soup.select('p #author')` will match any element that has an `id` attribute of `author`, as long as it is also inside a `<p>` element.

The `select()` method will return a list of `Tag` objects, which is how Beautiful Soup represents an HTML element. The list will contain one `Tag` object for every match in the `BeautifulSoup` object's HTML. Tag values can be passed to the `str()` function to show the HTML tags they represent. Tag values also have an `attrs` attribute that shows all the HTML attributes of the tag as a dictionary. Using the *example.html* file from earlier, enter the following into the interactive shell:

```
>>> import bs4
>>> exampleFile = open('example.html')
>>> exampleSoup = bs4.BeautifulSoup(exampleFile.read())
>>> elems = exampleSoup.select('#author')
>>> type(elems)
<class 'list'>
>>> len(elems)
1
>>> type(elems[0])
<class 'bs4.element.Tag'>
>>> elems[0].getText()
'Al Sweigart'
>>> str(elems[0])
'<span id="author">Al Sweigart</span>'
>>> elems[0].attrs
{'id': 'author'}
```

This code will pull the element with `id="author"` out of our example HTML. We use `select('#author')` to return a list of all the elements with `id="author"`. We store this list of **`Tag`** objects in the variable `elems`, and `len(elems)` tells us there is one **`Tag`** object in the list; there was one match. Calling `getText()` on the element returns the element's text, or inner HTML. The text of an element is the content between the opening and closing tags: in this case, `'Al Sweigart'`.

Passing the element to `str()` returns a string with the starting and closing tags and the element's text. Finally, `attrs` gives us a dictionary with the element's attribute, `'id'`, and the value of the `id` attribute, `'author'`.

You can also pull all the `<p>` elements from the `BeautifulSoup` object. Enter this into the interactive shell:

```
>>> pElems = exampleSoup.select('p')
>>> str(pElems[0])
'<p>Download my <strong>Python</strong> book from <a href="http://
inventwithpython.com">my website</a>.</p>'
>>> pElems[0].getText()
'Download my Python book from my website.'
>>> str(pElems[1])
'<p class="slogan">Learn Python the easy way!</p>'
>>> pElems[1].getText()
'Learn Python the easy way!'
>>> str(pElems[2])
'<p>By <span id="author">Al Sweigart</span></p>'
>>> pElems[2].getText()
'By Al Sweigart'
```

This time, `select()` gives us a list of three matches, which we store in `pElems`. Using `str()` on `pElems[0]`, `pElems[1]`, and `pElems[2]` shows you each element as a string, and using `getText()` on each element shows you its text.

Getting Data from an Element's Attributes

The `get()` method for `Tag` objects makes it simple to access attribute values from an element. The method is passed a string of an attribute name and returns that attribute's value. Using *example.html*, enter the following into the interactive shell:

```
>>> import bs4
>>> soup = bs4.BeautifulSoup(open('example.html'))
>>> spanElem = soup.select('span')[0]
>>> str(spanElem)
'<span id="author">Al Sweigart</span>'
>>> spanElem.get('id')
'author'
>>> spanElem.get('some_nonexistent_addr') == None
True
>>> spanElem.attrs
{'id': 'author'}
```

Here we use `select()` to find any `<span>` elements and then store the first matched element in `spanElem`. Passing the attribute name `'id'` to `get()` returns the attribute's value, `'author'`.

Project: "I'm Feeling Lucky" Google Search

Whenever I search a topic on Google, I don't look at just one search result at a time. By middle-clicking a search result link (or clicking while holding CTRL), I open the first several links in a bunch of new tabs to read later. I search Google often enough that this workflow—opening my browser, searching for a topic, and middle-clicking several links one by one—is tedious. It would be nice if I could simply type a search term on the command line and have my computer automatically open a browser with all the top search results in new tabs. Let's write a script to do this.

This is what your program does:

- Gets search keywords from the command line arguments.
- Retrieves the search results page.
- Opens a browser tab for each result.

This means your code will need to do the following:

- Read the command line arguments from `sys.argv`.
- Fetch the search result page with the `requests` module.
- Find the links to each search result.
- Call the `webbrowser.open()` function to open the web browser.

Open a new file editor window and save it as *lucky.py*.

Step 1: Get the Command Line Arguments and Request the Search Page

Before coding anything, you first need to know the URL of the search result page. By looking at the browser's address bar after doing a Google search, you can see that the result page has a URL like *https://www.google.com/search?q=SEARCH_TERM_HERE*. The `requests` module can download this page and then you can use

Beautiful Soup to find the search result links in the HTML. Finally, you'll use the `webbrowser` module to open those links in browser tabs.

Make your code look like the following:

```
#! python3
# lucky.py - Opens several Google search results.

import requests, sys, webbrowser, bs4

print('Googling...') # display text while downloading the Google page
res = requests.get('http://google.com/search?q=' + ' '.join(sys.argv[1:]))
res.raise_for_status()

# TODO: Retrieve top search result links.

# TODO: Open a browser tab for each result.
```

The user will specify the search terms using command line arguments when they launch the program. These arguments will be stored as strings in a list in `sys.argv`.

STEP 2: FIND ALL THE RESULTS

Now you need to use Beautiful Soup to extract the top search result links from your downloaded HTML. But how do you figure out the right selector for the job? For example, you can't just search for all `<a>` tags, because there are lots of links you don't care about in the HTML. Instead, you must inspect the search result page with the browser's developer tools to try to find a selector that will pick out only the links you want.

After doing a Google search for *Beautiful Soup*, you can open the browser's developer tools and inspect some of the link elements on the page. They look incredibly complicated, something like this: `<a href="/url?sa =t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&amp;ved=0CCgQFjAA&url=http%3A%2F%2Fwww.crummy.com%2Fsoftware%2FBeautifulSoup%2F&ei=LHBVU_XDD9KVyAShmYDwCw&usg=AFQjCNHAxwplurFOBqg5cehWQEVKi-TuLQ&amp;sig2=sdZu6WVlBlVSDrwhtworMA" onmousedown="return rwt(this,'','','','1','AFQ jCNHAxwplurFOBqg5cehWQEVKi-TuLQ','sdZu6WVlBlVSDrwhtworMA','0CCgQFjAA','','',ev ent)" data-href="http://www.crummy.com/software/BeautifulSoup/"><em>Beautiful Soup</em>: We called him Tortoise because he taught us.</a>`.

It doesn't matter that the element looks incredibly complicated. You just need to find the pattern that all the search result links have. But this `<a>` element doesn't have anything that easily distinguishes it from the nonsearch result `<a>` elements on the page.

Make your code look like the following:

```
#! python3
# lucky.py - Opens several google search results.

import requests, sys, webbrowser, bs4

--snip--

# Retrieve top search result links.
soup = bs4.BeautifulSoup(res.text)

# Open a browser tab for each result.
linkElems = soup.select('.r a')
```

If you look up a little from the `<a>` element, though, there is an element like this: `<h3 class="r">`. Looking through the rest of the HTML source, it looks like the `r` class is used only for search result links. You don't have to know what the CSS class `r` is or what it does. You're just going to use it as a marker for the `<a>` element you are looking for. You can create a `BeautifulSoup` object from the downloaded page's HTML text and then use the selector `'.r a'` to find all `<a>` elements that are within an element that has the `r` CSS class.

STEP 3: OPEN WEB BROWSERS FOR EACH RESULT

Finally, we'll tell the program to open web browser tabs for our results. Add the following to the end of your program:

```
#! python3
# lucky.py - Opens several google search results.

import requests, sys, webbrowser, bs4

--snip--

# Open a browser tab for each result.
linkElems = soup.select('.r a')
numOpen = min(5, len(linkElems))
for i in range(numOpen):
    webbrowser.open('http://google.com' + linkElems[i].get('href'))
```

By default, you open the first five search results in new tabs using the `webbrowser` module. However, the user may have searched for something that turned up fewer than five results. The `soup.select()` call returns a list of all the elements that matched your `'.r a'` selector, so the number of tabs you want to open is either `5` or the length of this list (whichever is smaller).

The built-in Python function `min()` returns the smallest of the integer or float arguments it is passed. (There is also a built-in `max()` function that returns the largest argument it is passed.) You can use `min()` to find out whether there are fewer than five links in the list and store the number of links to open in a variable named `numOpen`. Then you can run through a `for` loop by calling `range(numOpen)`.

On each iteration of the loop, you use `webbrowser.open()` to open a new tab in the web browser. Note that the `href` attribute's value in the returned `<a>` elements do not have the initial `http://google.com` part, so you have to concatenate that to the `href` attribute's string value.

Now you can instantly open the first five Google results for, say, *Python programming tutorials* by running `lucky python programming tutorials` on the command line! (See Appendix B for how to easily run programs on your operating system.)

IDEAS FOR SIMILAR PROGRAMS

The benefit of tabbed browsing is that you can easily open links in new tabs to peruse later. A program that automatically opens several links at once can be a nice shortcut to do the following:

- Open all the product pages after searching a shopping site such as Amazon
- Open all the links to reviews for a single product
- Open the result links to photos after performing a search on a photo site such as Flickr or Imgur

PROJECT: DOWNLOADING ALL XKCD COMICS

Blogs and other regularly updating websites usually have a front page with the most recent post as well as a Previous button on the page that takes you to the previous post. Then that post will also have a Previous button, and so on, creating a trail from the most recent page to the first post on the site. If you wanted a copy of the site's content to read when you're not online, you could manually navigate over every page and save each one. But this is pretty boring work, so let's write a program to do it instead.

XKCD is a popular geek webcomic with a website that fits this structure (see Figure 11-6). The front page at *http://xkcd.com/* has a Prev button that guides the user back through prior comics. Downloading each comic by hand would take forever, but you can write a script to do this in a couple of minutes.

Here's what your program does:

- Loads the XKCD home page.
- Saves the comic image on that page.
- Follows the Previous Comic link.
- Repeats until it reaches the first comic.

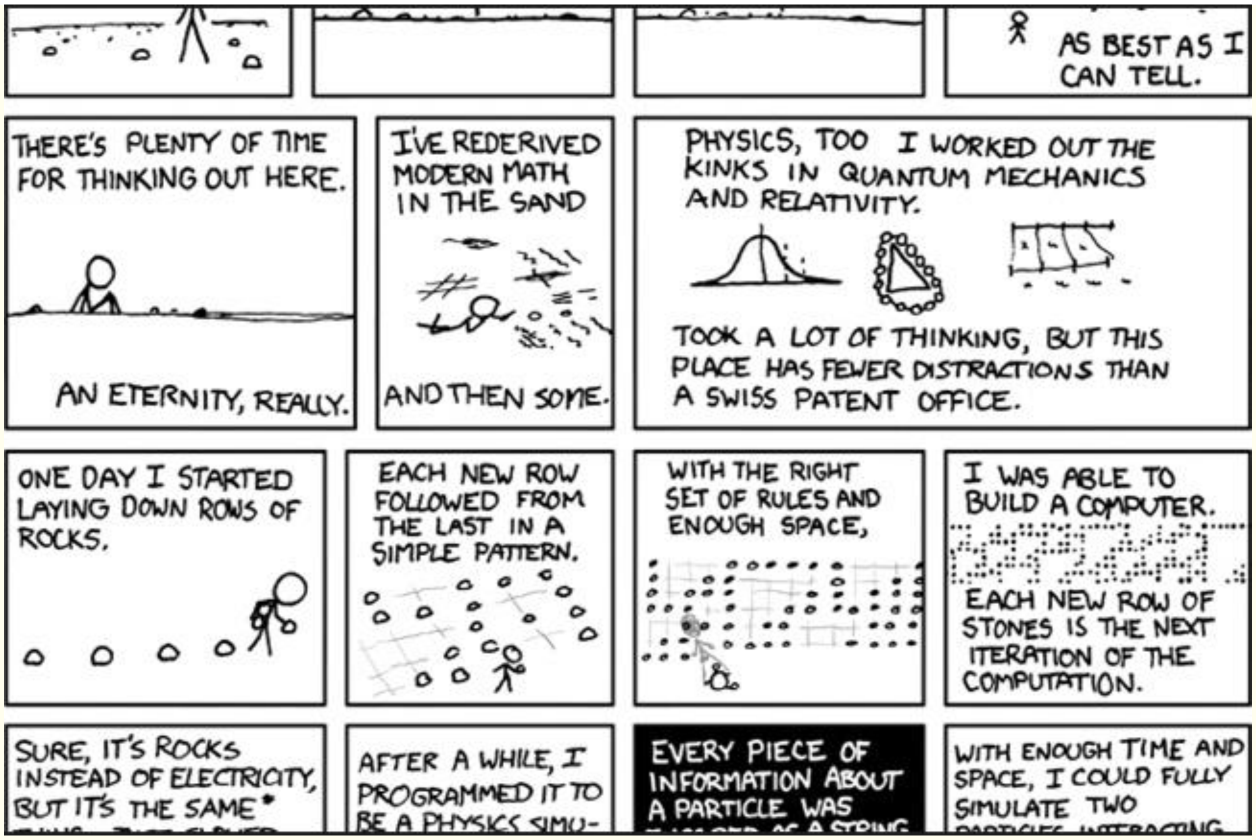

AS BEST AS I CAN TELL.
THERE'S PLENTY OF TIME FOR THINKING OUT HERE.
AN ETERNITY, REALLY.
I'VE REDERIVED MODERN MATH IN THE SAND
AND THEN SOME.
PHYSICS, TOO. I WORKED OUT THE KINKS IN QUANTUM MECHANICS AND RELATIVITY.
TOOK A LOT OF THINKING, BUT THIS PLACE HAS FEWER DISTRACTIONS THAN A SWISS PATENT OFFICE.
ONE DAY I STARTED LAYING DOWN ROWS OF ROCKS.
EACH NEW ROW FOLLOWED FROM THE LAST IN A SIMPLE PATTERN.
WITH THE RIGHT SET OF RULES AND ENOUGH SPACE,
I WAS ABLE TO BUILD A COMPUTER.
EACH NEW ROW OF STONES IS THE NEXT ITERATION OF THE COMPUTATION.
SURE, IT'S ROCKS INSTEAD OF ELECTRICITY, BUT IT'S THE SAME*
AFTER A WHILE, I PROGRAMMED IT TO BE A PHYSICS SIMU-
EVERY PIECE OF INFORMATION ABOUT A PARTICLE WAS
WITH ENOUGH TIME AND SPACE, I COULD FULLY SIMULATE TWO

Figure 11-6. XKCD, "a webcomic of romance, sarcasm, math, and language"

This means your code will need to do the following:

- Download pages with the `requests` module.
- Find the URL of the comic image for a page using Beautiful Soup.
- Download and save the comic image to the hard drive with `iter_content()`.
- Find the URL of the Previous Comic link, and repeat.

Open a new file editor window and save it as *downloadXkcd.py*.

STEP 1: DESIGN THE PROGRAM

If you open the browser's developer tools and inspect the elements on the page, you'll find the following:

- The URL of the comic's image file is given by the `href` attribute of an `<img>` element.
- The `<img>` element is inside a `<div id="comic">` element.
- The Prev button has a `rel` HTML attribute with the value `prev`.
- The first comic's Prev button links to the *http://xkcd.com/#* URL, indicating that there are no more previous pages.

Make your code look like the following:

```
#! python3
# downloadXkcd.py - Downloads every single XKCD comic.

import requests, os, bs4

url = 'http://xkcd.com'               # starting url
os.makedirs('xkcd', exist_ok=True)    # store comics in ./xkcd
while not url.endswith('#'):
    # TODO: Download the page.

    # TODO: Find the URL of the comic image.

    # TODO: Download the image.

    # TODO: Save the image to ./xkcd.

    # TODO: Get the Prev button's url.

print('Done.')
```

You'll have a `url` variable that starts with the value `'http://xkcd.com'` and repeatedly update it (in a `for` loop) with the URL of the current page's

Prev link. At every step in the loop, you'll download the comic at `url`. You'll know to end the loop when `url` ends with `'#'`.

You will download the image files to a folder in the current working directory named *xkcd*. The call `os.makedirs()` ensures that this folder exists, and the `exist_ok=True` keyword argument prevents the function from throwing an exception if this folder already exists. The rest of the code is just comments that outline the rest of your program.

STEP 2: DOWNLOAD THE WEB PAGE

Let's implement the code for downloading the page. Make your code look like the following:

```
#! python3
# downloadXkcd.py - Downloads every single XKCD comic.

import requests, os, bs4

url = 'http://xkcd.com'               # starting url
os.makedirs('xkcd', exist_ok=True)    # store comics in ./xkcd
while not url.endswith('#'):
    # Download the page.
    print('Downloading page %s...' % url)
    res = requests.get(url)
    res.raise_for_status()

    soup = bs4.BeautifulSoup(res.text)

    # TODO: Find the URL of the comic image.

    # TODO: Download the image.

    # TODO: Save the image to ./xkcd.

    # TODO: Get the Prev button's url.

print('Done.')
```

First, print `url` so that the user knows which URL the program is about to download; then use the `requests` module's `request.get()` function to download it. As always, you immediately call
the `Response` object's `raise_for_status()` method to throw an exception and end the program if something went wrong with the download. Otherwise, you create a `BeautifulSoup` object from the text of the downloaded page.

STEP 3: FIND AND DOWNLOAD THE COMIC IMAGE

Make your code look like the following:

```
#! python3
# downloadXkcd.py - Downloads every single XKCD comic.

import requests, os, bs4

--snip--

    # Find the URL of the comic image.
    comicElem = soup.select('#comic img')
    if comicElem == []:
         print('Could not find comic image.')
    else:
         try:
             comicUrl = 'http:' + comicElem[0].get('src')
             # Download the image.
             print('Downloading image %s...' % (comicUrl))
             res = requests.get(comicUrl)
             res.raise_for_status()
         except requests.exceptions.MissingSchema:
             # skip this comic
             prevLink = soup.select('a[rel="prev"]')[0]
             url = 'http://xkcd.com' + prevLink.get('href')
             continue


    # TODO: Save the image to ./xkcd.

    # TODO: Get the Prev button's url.

print('Done.')
```

From inspecting the XKCD home page with your developer tools, you know that the `<img>` element for the comic image is inside a `<div>` element with the `id` attribute set to `comic`, so the selector `'#comic img'` will get you the correct `<img>` element from the `BeautifulSoup` object.

A few XKCD pages have special content that isn't a simple image file. That's fine; you'll just skip those. If your selector doesn't find any elements, then `soup.select('#comic img')` will return a blank list. When that happens, the program can just print an error message and move on without downloading the image.

Otherwise, the selector will return a list containing one `<img>` element. You can get the `src` attribute from this `<img>` element and pass it to `requests.get()` to download the comic's image file.

STEP 4: SAVE THE IMAGE AND FIND THE PREVIOUS COMIC

Make your code look like the following:

```
#! python3
# downloadXkcd.py - Downloads every single XKCD comic.

import requests, os, bs4

--snip--

        # Save the image to ./xkcd.
        imageFile = open(os.path.join('xkcd', os.path.basename(comicUrl)),
'wb')
        for chunk in res.iter_content(100000):
            imageFile.write(chunk)
        imageFile.close()

    # Get the Prev button's url.
    prevLink = soup.select('a[rel="prev"]')[0]
    url = 'http://xkcd.com' + prevLink.get('href')

print('Done.')
```

At this point, the image file of the comic is stored in the `res` variable. You need to write this image data to a file on the hard drive.

You'll need a filename for the local image file to pass to `open()`. The `comicUrl` will have a value like `'http://imgs.xkcd.com/comics/heartbleed_explanation.png'`—which you might have noticed looks a lot like a file path. And in fact, you can call `os.path.basename()` with `comicUrl`, and it will return just the last part of the URL, `'heartbleed_explanation.png'`. You can use this as the filename when saving the image to your hard drive. You join this name with the name of your `xkcd` folder using `os.path.join()` so that your program uses backslashes (`\`) on Windows and forward slashes (`/`) on OS X and Linux. Now that you finally have the filename, you can call `open()` to open a new file in `'wb'` “write binary” mode.

Remember from earlier in this chapter that to save files you've downloaded using Requests, you need to loop over the return value of the `iter_content()` method. The code in the `for` loop writes out chunks of the image data (at most 100,000 bytes each) to the file and then you close the file. The image is now saved to your hard drive.

Afterward, the selector `'a[rel="prev"]'` identifies the `<a>` element with the `rel` attribute set to `prev`, and you can use this `<a>` element's `href` attribute to get the previous comic's URL, which gets stored in `url`. Then the `while` loop begins the entire download process again for this comic.

The output of this program will look like this:

```
Downloading page http://xkcd.com...
Downloading image http://imgs.xkcd.com/comics/phone_alarm.png...
Downloading page http://xkcd.com/1358/...
Downloading image http://imgs.xkcd.com/comics/nro.png...
Downloading page http://xkcd.com/1357/...
Downloading image http://imgs.xkcd.com/comics/free_speech.png...
Downloading page http://xkcd.com/1356/...
Downloading image http://imgs.xkcd.com/comics/orbital_mechanics.png...
Downloading page http://xkcd.com/1355/...
Downloading image http://imgs.xkcd.com/comics/airplane_message.png...
Downloading page http://xkcd.com/1354/...
Downloading image http://imgs.xkcd.com/comics/heartbleed_explanation.png...
--snip--
```

This project is a good example of a program that can automatically follow links in order to scrape large amounts of data from the Web. You can learn about Beautiful Soup's other features from its documentation at *http://www.crummy.com/software/BeautifulSoup/bs4/doc/*.

IDEAS FOR SIMILAR PROGRAMS

Downloading pages and following links are the basis of many web crawling programs. Similar programs could also do the following:

- Back up an entire site by following all of its links.
- Copy all the messages off a web forum.
- Duplicate the catalog of items for sale on an online store.

The `requests` and `BeautifulSoup` modules are great as long as you can figure out the URL you need to pass to `requests.get()`. However, sometimes this isn't so easy to find. Or perhaps the website you want your program to navigate requires you to log in first. The `selenium` module will give your programs the power to perform such sophisticated tasks.

CONTROLLING THE BROWSER WITH THE SELENIUM MODULE

The `selenium` module lets Python directly control the browser by programmatically clicking links and filling in login information, almost as though there is a human user interacting with the page. Selenium allows you to interact with web pages in a much more advanced way than Requests and Beautiful Soup; but because it launches a web browser, it is a bit slower and hard to run in the background if, say, you just need to download some files from the Web.

Appendix A has more detailed steps on installing third-party modules.

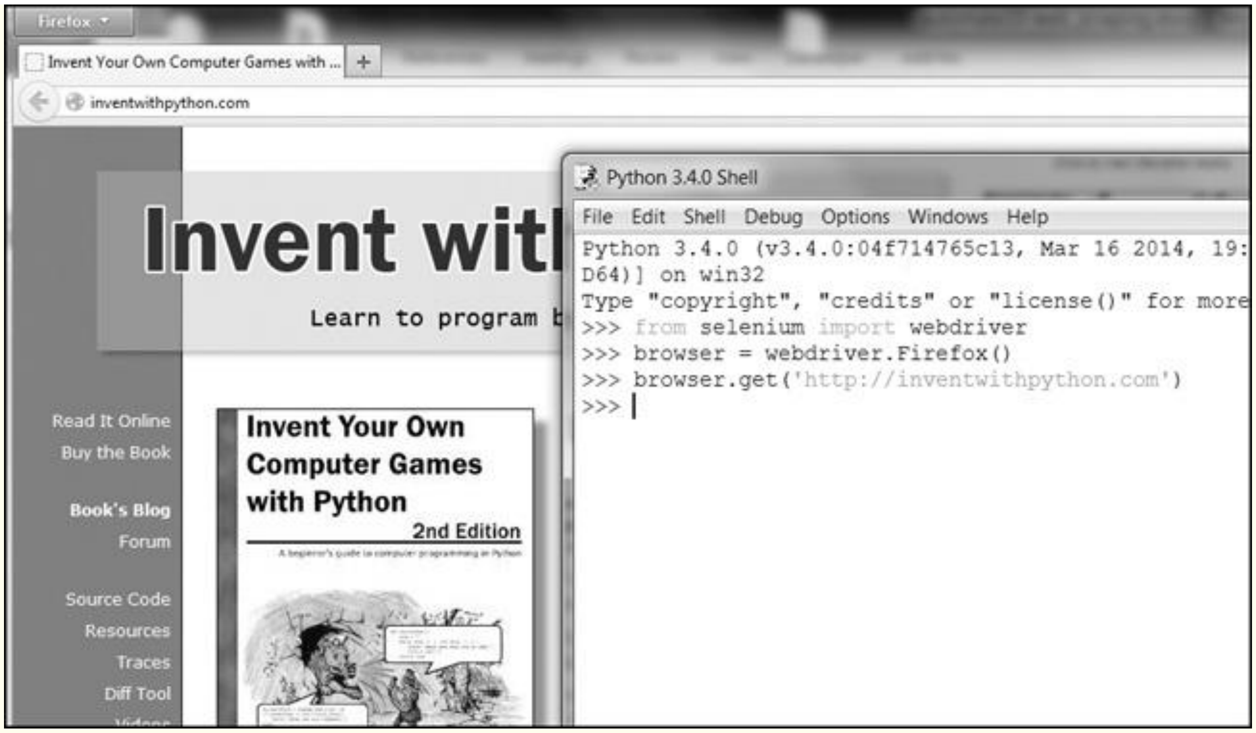
STARTING A SELENIUM-CONTROLLED BROWSER

For these examples, you'll need the Firefox web browser. This will be the browser that you control. If you don't already have Firefox, you can download it for free from *http://getfirefox.com/*.

Importing the modules for Selenium is slightly tricky. Instead of `import selenium`, you need to run `from selenium import webdriver`. (The exact reason why the `selenium` module is set up this way is beyond the scope of this book.) After that, you can launch the Firefox browser with Selenium. Enter the following into the interactive shell:

```
>>> from selenium import webdriver
>>> browser = webdriver.Firefox()
>>> type(browser)
<class 'selenium.webdriver.firefox.webdriver.WebDriver'>
>>> browser.get('http://inventwithpython.com')
```

You'll notice when `webdriver.Firefox()` is called, the Firefox web browser starts up. Calling `type()` on the value `webdriver.Firefox()` reveals it's of the `WebDriver` data type. And calling `browser.get('http://inventwithpython.com')` directs the browser to *http://inventwithpython.com/*. Your browser should look something like Figure 11-7.



```
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:
D64)] on win32
Type "copyright", "credits" or "license()" for more
>>> from selenium import webdriver
>>> browser = webdriver.Firefox()
>>> browser.get('http://inventwithpython.com')
>>> 
```

Figure 11-7. After calling `webdriver.Firefox()` and `get()` in IDLE, the Firefox browser appears.

FINDING ELEMENTS ON THE PAGE

`WebDriver` objects have quite a few methods for finding elements on a page. They are divided into the `find_element_*` and `find_elements_*` methods. The `find_element_*` methods return a single `WebElement` object, representing the first element on the page that matches your query. The `find_elements_*` methods return a list of `WebElement_*` objects for *every* matching element on the page.

Table 11-3 shows several examples of `find_element_*` and `find_elements_*` methods being called on a `WebDriver` object that's stored in the variable `browser`.

Table 11-3. Selenium's `WebDriver` Methods for Finding Elements

Method name	WebElement object/list returned

`browser.find_element_by_class_name(name)`
`browser.find_elements_by_class_name(name)`
	Elements that use the CSS class `name`

Method name	WebElement object/list returned
`browser.find_element_by_css_selector(`*`selector`*`)` `browser.find_elements_by_css_selector(`*`selector`*`)`	Elements that match the CSS *`selector`*
`browser.find_element_by_id(`*`id`*`)` `browser.find_elements_by_id(`*`id`*`)`	Elements with a matching *`id`* attribute value
`browser.find_element_by_link_text(`*`text`*`)` `browser.find_elements_by_link_text(`*`text`*`)`	`<a>` elements that completely match the *`text`* provided
`browser.find_element_by_partial_link_text(`*`text`*`)` `browser.find_elements_by_partial_link_text(`*`text`*`)`	`<a>` elements that contain the *`text`* provided
`browser.find_element_by_name(`*`name`*`)` `browser.find_elements_by_name(`*`name`*`)`	Elements with a matching *`name`* attribute value
`browser.find_element_by_tag_name(`*`name`*`)` `browser.find_elements_by_tag_name(`*`name`*`)`	Elements with a matching tag *`name`* (case insensitive; an `<a>` element is matched by `'a'` and `'A'`)

Except for the `*_by_tag_name()` methods, the arguments to all the methods are case sensitive. If no elements exist on the page that match what the method is looking for, the `selenium` module raises a `NoSuchElement` exception. If you do not want this exception to crash your program, add `try` and `except` statements to your code.

Once you have the `WebElement` object, you can find out more about it by reading the attributes or calling the methods in Table 11-4.

Table 11-4. WebElement Attributes and Methods

Attribute or method	Description
`tag_name`	The tag name, such as `'a'` for an `<a>` element
`get_attribute(`*`name`*`)`	The value for the element's `name` attribute
`text`	The text within the element, such as `'hello'` in `<span>hello</span>`
`clear()`	For text field or text area elements, clears the text typed into it
`is_displayed()`	Returns `True` if the element is visible; otherwise returns `False`
`is_enabled()`	For input elements, returns `True` if the element is enabled; otherwise returns `False`
`is_selected()`	For checkbox or radio button elements, returns `True` if the element is selected; otherwise returns `False`
`location`	A dictionary with keys `'x'` and `'y'` for the position of the element in the page

For example, open a new file editor and enter the following program:

```
from selenium import webdriver
browser = webdriver.Firefox()
browser.get('http://inventwithpython.com')
try:
    elem = browser.find_element_by_class_name('bookcover')
    print('Found <%s> element with that class name!' % (elem.tag_name))
except:
    print('Was not able to find an element with that name.')
```

Here we open Firefox and direct it to a URL. On this page, we try to find elements with the class name `'bookcover'`, and if such an element is found, we print its tag name using the `tag_name` attribute. If no such element was found, we print a different message.

This program will output the following:

```
Found <img> element with that class name!
```

We found an element with the class name `'bookcover'` and the tag name `'img'`.

Clicking the Page

`WebElement` objects returned from the `find_element_*` and `find_elements_*` methods have a `click()` method that simulates a mouse click on that element. This method can be used to follow a link, make a selection on a radio button, click a Submit button, or trigger whatever else might happen when the element is clicked by the mouse. For example, enter the following into the interactive shell:

```
>>> from selenium import webdriver
>>> browser = webdriver.Firefox()
>>> browser.get('http://inventwithpython.com')
>>> linkElem = browser.find_element_by_link_text('Read It Online')
>>> type(linkElem)
<class 'selenium.webdriver.remote.webelement.WebElement'>
>>> linkElem.click() # follows the "Read It Online" link
```

This opens Firefox to *http://inventwithpython.com/*, gets the `WebElement` object for the `<a>` element with the text *Read It Online*, and then simulates clicking that `<a>` element. It's just like if you clicked the link yourself; the browser then follows that link.

Filling Out and Submitting Forms

Sending keystrokes to text fields on a web page is a matter of finding the `<input>` or `<textarea>` element for that text field and then calling the `send_keys()` method. For example, enter the following into the interactive shell:

```
>>> from selenium import webdriver
>>> browser = webdriver.Firefox()
>>> browser.get('https://mail.yahoo.com')
>>> emailElem = browser.find_element_by_id('login-username')
>>> emailElem.send_keys('not_my_real_email')
>>> passwordElem = browser.find_element_by_id('login-passwd')
>>> passwordElem.send_keys('12345')
>>> passwordElem.submit()
```

As long as Gmail hasn't changed the `id` of the Username and Password text fields since this book was published, the previous code will fill in those text fields with the provided text. (You can always use the browser's inspector to verify the `id`.) Calling the `submit()` method on any element will have the same result as clicking the Submit button for the form that element is in. (You could have just as easily called `emailElem.submit()`, and the code would have done the same thing.)

Sending Special Keys

Selenium has a module for keyboard keys that are impossible to type into a string value, which function much like escape characters. These values are stored in attributes in the `selenium.webdriver.common.keys` module. Since that is such a long module name, it's much easier to run `from selenium.webdriver.common.keys import Keys` at the top of your program; if you do, then you can simply write `Keys` anywhere you'd normally have to write `selenium.webdriver.common.keys`. Table 11-5 lists the commonly used `Keys` variables.

Table 11-5. Commonly Used Variables in the `selenium.webdriver.common.keys` Module

Attributes	Meanings
`Keys.DOWN`, `Keys.UP`, `Keys.LEFT`, `Keys.RIGHT`	The keyboard arrow keys
`Keys.ENTER`, `Keys.RETURN`	The ENTER and RETURN keys

Attributes	Meanings
`Keys.HOME`, `Keys.END`, `Keys.PAGE_DOWN`, `Keys.PAGE_UP`	The `home`, `end`, `pagedown`, and `pageup` keys
`Keys.ESCAPE`, `Keys.BACK_SPACE`, `Keys.DELETE`	The ESC, BACKSPACE, and DELETE keys
`Keys.F1`, `Keys.F2`,..., `Keys.F12`	The F1 to F12 keys at the top of the keyboard
`Keys.TAB`	The TAB key

For example, if the cursor is not currently in a text field, pressing the HOME and END keys will scroll the browser to the top and bottom of the page, respectively. Enter the following into the interactive shell, and notice how the `send_keys()` calls scroll the page:

```
>>> from selenium import webdriver
>>> from selenium.webdriver.common.keys import Keys
>>> browser = webdriver.Firefox()
>>> browser.get('http://nostarch.com')
>>> htmlElem = browser.find_element_by_tag_name('html')
>>> htmlElem.send_keys(Keys.END)     # scrolls to bottom
>>> htmlElem.send_keys(Keys.HOME)    # scrolls to top
```

The `<html>` tag is the base tag in HTML files: The full content of the HTML file is enclosed within the `<html>` and `</html>` tags. Calling `browser.find_element_by_tag_name('html')` is a good place to send keys to the general web page. This would be useful if, for example, new content is loaded once you've scrolled to the bottom of the page.

Clicking Browser Buttons

Selenium can simulate clicks on various browser buttons as well through the following methods:

- **`browser.back()`**. Clicks the Back button.
- **`browser.forward()`**. Clicks the Forward button.
- **`browser.refresh()`**. Clicks the Refresh/Reload button.
- **`browser.quit()`**. Clicks the Close Window button.

More Information on Selenium

Selenium can do much more beyond the functions described here. It can modify your browser's cookies, take screenshots of web pages, and run custom JavaScript. To learn more about these features, you can visit the Selenium documentation at http://selenium-python.readthedocs.org/.

Summary

Most boring tasks aren't limited to the files on your computer. Being able to programmatically download web pages will extend your programs to the Internet. The `requests` module makes downloading straightforward, and with some basic knowledge of HTML concepts and selectors, you can utilize the `BeautifulSoup` module to parse the pages you download.

But to fully automate any web-based tasks, you need direct control of your web browser through the `selenium` module. The `selenium` module will allow you to log in to websites and fill out forms automatically. Since a web browser is the most common way to send and receive information over the Internet, this is a great ability to have in your programmer toolkit.

Working with Excel Spreadsheets

Excel is a popular and powerful spreadsheet application for Windows. The `openpyxl` module allows your Python programs to read and modify Excel spreadsheet files. For example, you might have the boring task of copying certain data from one spreadsheet and pasting it into another one. Or you might have to go through thousands of rows and pick out just a handful of them to make small edits based on some criteria. Or you might have to look through hundreds of spreadsheets of department budgets, searching for any that are in the red. These are exactly the sort of boring, mindless spreadsheet tasks that Python can do for you.

Although Excel is proprietary software from Microsoft, there are free alternatives that run on Windows, OS X, and Linux. Both LibreOffice Calc and OpenOffice Calc work with Excel's *.xlsx* file format for spreadsheets, which means the `openpyxl` module can work on spreadsheets from these applications as well. You can download the software
from https://www.libreoffice.org/ and http://www.openoffice.org/,

respectively. Even if you already have Excel installed on your computer, you may find these programs easier to use. The screenshots in this chapter, however, are all from Excel 2010 on Windows 7.

Excel Documents

First, let’s go over some basic definitions: An Excel spreadsheet document is called a *workbook*. A single workbook is saved in a file with the *.xlsx* extension. Each workbook can contain multiple *sheets* (also called *worksheets*). The sheet the user is currently viewing (or last viewed before closing Excel) is called the *active sheet*.

Each sheet has *columns* (addressed by letters starting at *A*) and *rows* (addressed by numbers starting at 1). A box at a particular column and row is called a *cell*. Each cell can contain a number or text value. The grid of cells with data makes up a sheet.

Installing the openpyxl Module

Python does not come with OpenPyXL, so you’ll have to install it. Follow the instructions for installing third-party modules in Appendix A; the name of the module is `openpyxl`. To test whether it is installed correctly, enter the following into the interactive shell:

```
>>> import openpyxl
```

If the module was correctly installed, this should produce no error messages. Remember to import the `openpyxl` module before running the interactive shell examples in this chapter, or you’ll get a `NameError: name 'openpyxl' is not defined` error.

This book covers version 2.3.3 of OpenPyXL, but new versions are regularly released by the OpenPyXL team. Don’t worry, though: New versions should stay backward compatible with the instructions in this book for quite some time. If you have a newer version and want to see what additional features may be available to you, you can check out the full documentation for OpenPyXL at *http://openpyxl.readthedocs.org/*.

READING EXCEL DOCUMENTS

The examples in this chapter will use a spreadsheet named *example.xlsx* stored in the root folder. You can either create the spreadsheet yourself or download it from *http://nostarch.com/automatestuff/*. Figure 12-1 shows the tabs for the three default sheets named *Sheet1*, *Sheet2*, and *Sheet3* that Excel automatically provides for new workbooks. (The number of default sheets created may vary between operating systems and spreadsheet programs.)

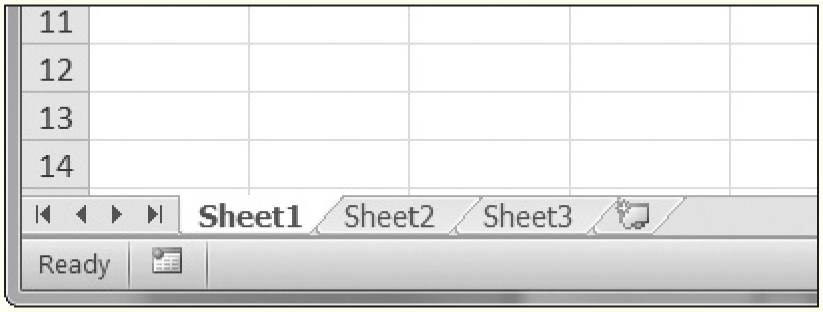


Figure 12-1. The tabs for a workbook's sheets are in the lower-left corner of Excel.

Sheet 1 in the example file should look like Table 12-1. (If you didn't download *example.xlsx* from the website, you should enter this data into the sheet yourself.)

Table 12-1. The *example.xlsx* Spreadsheet

	A	B	C
1	4/5/2015 1:34:02 PM	Apples	73
2	4/5/2015 3:41:23 AM	Cherries	85
3	4/6/2015 12:46:51 PM	Pears	14
4	4/8/2015 8:59:43 AM	Oranges	52
5	4/10/2015 2:07:00 AM	Apples	152
6	4/10/2015 6:10:37 PM	Bananas	23
7	4/10/2015 2:40:46 AM	Strawberries	98

Now that we have our example spreadsheet, let's see how we can manipulate it with the `openpyxl` module.

OPENING EXCEL DOCUMENTS WITH OPENPYXL

Once you've imported the `openpyxl` module, you'll be able to use the `openpyxl.load_workbook()` function. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> type(wb)
<class 'openpyxl.workbook.workbook.Workbook'>
```

The `openpyxl.load_workbook()` function takes in the filename and returns a value of the `workbook` data type. This `Workbook` object represents the Excel file, a bit like how a `File` object represents an opened text file.

Remember that *example.xlsx* needs to be in the current working directory in order for you to work with it. You can find out what the current working directory is by importing `os` and using `os.getcwd()`, and you can change the current working directory using `os.chdir()`.

GETTING SHEETS FROM THE WORKBOOK

You can get a list of all the sheet names in the workbook by calling the `get_sheet_names()` method. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> wb.get_sheet_names()
['Sheet1', 'Sheet2', 'Sheet3']
>>> sheet = wb.get_sheet_by_name('Sheet3')
>>> sheet
<Worksheet "Sheet3">
>>> type(sheet) <class 'openpyxl.worksheet.worksheet.Worksheet'>
>>> sheet.title
'Sheet3'
>>> anotherSheet = wb.active
>>> anotherSheet
<Worksheet "Sheet1">
```

Each sheet is represented by a `Worksheet` object, which you can obtain by passing the sheet name string to the `get_sheet_by_name()` workbook method. Finally, you can read the `active` member variable of a `Workbook` object to get the workbook's active sheet. The active sheet is the sheet that's on top when the workbook is opened in Excel. Once you have the `Worksheet` object, you can get its name from the `title` attribute.

GETTING CELLS FROM THE SHEETS

Once you have a `Worksheet` object, you can access a `Cell` object by its name. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.get_sheet_by_name('Sheet1')
>>> sheet['A1']
<Cell Sheet1.A1>
>>> sheet['A1'].value
datetime.datetime(2015, 4, 5, 13, 34, 2)
>>> c = sheet['B1']
>>> c.value
'Apples'
>>> 'Row ' + str(c.row) + ', Column ' + c.column + ' is ' + c.value
'Row 1, Column B is Apples'
>>> 'Cell ' + c.coordinate + ' is ' + c.value
'Cell B1 is Apples'
>>> sheet['C1'].value
73
```

The `Cell` object has a `value` attribute that contains, unsurprisingly, the value stored in that cell. `Cell` objects also have `row`, `column`, and `coordinate` attributes that provide location information for the cell.

Here, accessing the `value` attribute of our `Cell` object for cell B1 gives us the string `'Apples'`. The `row` attribute gives us the integer `1`, the `column` attribute gives us `'B'`, and the `coordinate` attribute gives us `'B1'`.

OpenPyXL will automatically interpret the dates in column A and return them as `datetime` values rather than strings. The `datetime` data type is explained further in Chapter 16.

Specifying a column by letter can be tricky to program, especially because after column Z, the columns start by using two letters: AA, AB, AC, and so on. As an alternative, you can also get a cell using the sheet's `cell()` method and passing integers for its `row` and `column` keyword arguments. The first row or column integer is `1`, not `0`. Continue the interactive shell example by entering the following:

```
>>> sheet.cell(row=1, column=2)
```

```
<Cell Sheet1.B1>
>>> sheet.cell(row=1, column=2).value
'Apples'
>>> for i in range(1, 8, 2):
        print(i, sheet.cell(row=i, column=2).value)

1 Apples
3 Pears
5 Apples
7 Strawberries
```

As you can see, using the sheet's `cell()` method and passing it `row=1` and `column=2` gets you a `Cell` object for cell `B1`, just like specifying `sheet['B1']` did. Then, using the `cell()` method and its keyword arguments, you can write a `for` loop to print the values of a series of cells.

Say you want to go down column B and print the value in every cell with an odd row number. By passing `2` for the `range()` function's "step" parameter, you can get cells from every second row (in this case, all the odd-numbered rows). The `for` loop's `i` variable is passed for the `row` keyword argument to the `cell()` method, while `2` is always passed for the `column` keyword argument. Note that the integer `2`, not the string `'B'`, is passed.

You can determine the size of the sheet with the `Worksheet` object's `max_row` and `max_column` member variables. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.get_sheet_by_name('Sheet1')
>>> sheet.max_row
7
>>> sheet.max_column
3
```

Note that the `max_column` method returns an integer rather than the letter that appears in Excel.

CONVERTING BETWEEN COLUMN LETTERS AND NUMBERS

To convert from letters to numbers, call the `openpyxl.cell.column_index_from_string()` function. To convert from numbers to letters, call the `openpyxl.cell.get_column_letter()` function. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> from openpyxl.cell import get_column_letter, column_index_from_string
>>> get_column_letter(1)
'A'
>>> get_column_letter(2)
'B'
>>> get_column_letter(27)
'AA'
>>> get_column_letter(900)
'AHP'
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.get_sheet_by_name('Sheet1')
>>> get_column_letter(sheet.max_column)
'C'
>>> column_index_from_string('A')
1
>>> column_index_from_string('AA')
27
```

After you import these two functions from the `openpyxl.cell` module, you can call `get_column_letter()` and pass it an integer like 27 to figure out what the letter name of the 27th column is. The function `column_index_string()` does the reverse: You pass it the letter name of a column, and it tells you what number that column is. You don't need to have a workbook loaded to use these functions. If you want, you can load a workbook, get a `Worksheet` object, and call a `Worksheet` object method like `max_column` to get an integer. Then, you can pass that integer to `get_column_letter()`.

GETTING ROWS AND COLUMNS FROM THE SHEETS

You can slice `Worksheet` objects to get all the `Cell` objects in a row, column, or rectangular area of the spreadsheet. Then you can loop over all the cells in the slice. Enter the following into the interactive shell:

```
   >>> import openpyxl
   >>> wb = openpyxl.load_workbook('example.xlsx')
   >>> sheet = wb.get_sheet_by_name('Sheet1')
   >>> tuple(sheet['A1':'C3'])
   ((<Cell Sheet1.A1>, <Cell Sheet1.B1>, <Cell Sheet1.C1>), (<Cell
Sheet1.A2>,
   <Cell Sheet1.B2>, <Cell Sheet1.C2>), (<Cell Sheet1.A3>, <Cell Sheet1.B3>,
   <Cell Sheet1.C3>))
❶ >>> for rowOfCellObjects in sheet['A1':'C3']:
❷         for cellObj in rowOfCellObjects:
              print(cellObj.coordinate, cellObj.value)
          print('--- END OF ROW ---')
   A1 2015-04-05 13:34:02
   B1 Apples
   C1 73
   --- END OF ROW ---
   A2 2015-04-05 03:41:23
   B2 Cherries
   C2 85
   --- END OF ROW ---
   A3 2015-04-06 12:46:51
   B3 Pears
   C3 14
   --- END OF ROW ---
```

Here, we specify that we want the `Cell` objects in the rectangular area from A1 to C3, and we get a `Generator` object containing the `Cell` objects in that area. To help us visualize this `Generator` object, we can use `tuple()` on it to display its `Cell` objects in a tuple.

This tuple contains three tuples: one for each row, from the top of the desired area to the bottom. Each of these three inner tuples contains the `Cell` objects in one row of our desired area, from the leftmost cell to the right. So overall, our slice of the sheet contains all the `Cell` objects in the area from A1 to C3, starting from the top-left cell and ending with the bottom-right cell.

To print the values of each cell in the area, we use two `for` loops. The outer `for` loop goes over each row in the slice ❶. Then, for each row, the nested `for` loop goes through each cell in that row ❷.

To access the values of cells in a particular row or column, you can also use a `Worksheet` object's `rows` and `columns` attribute. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.active
>>> sheet.columns[1]
(<Cell Sheet1.B1>, <Cell Sheet1.B2>, <Cell Sheet1.B3>, <Cell Sheet1.B4>,
<Cell Sheet1.B5>, <Cell Sheet1.B6>, <Cell Sheet1.B7>)
>>> for cellObj in sheet.columns[1]:
        print(cellObj.value)

Apples
Cherries
Pears
Oranges
Apples
Bananas
Strawberries
```

Using the `rows` attribute on a `Worksheet` object will give you a tuple of tuples. Each of these inner tuples represents a row, and contains the `Cell` objects in that row. The `columns` attribute also gives you a tuple of tuples, with each of the inner tuples containing the `Cell` objects in a particular column. For *example.xlsx*, since there are 7 rows and 3 columns, `rows` gives us a tuple of 7 tuples (each containing 3 `Cell` objects), and `columns` gives us a tuple of 3 tuples (each containing 7 `Cell` objects).

To access one particular tuple, you can refer to it by its index in the larger tuple. For example, to get the tuple that represents column B, you use `sheet.columns[1]`. To get the tuple containing the `Cell` objects in column A, you'd use `sheet.columns[0]`. Once you have a tuple representing one row or column, you can loop through its `Cell` objects and print their values.

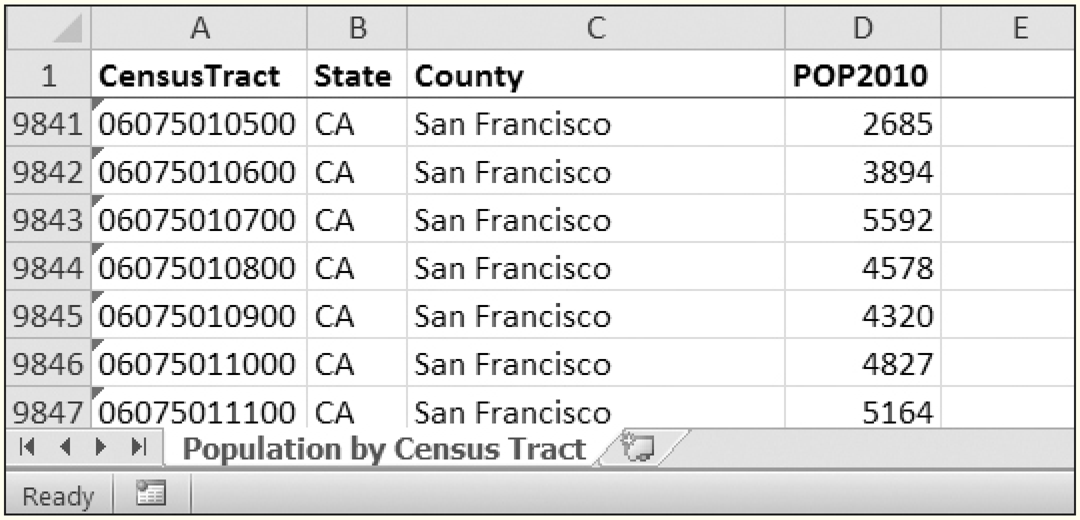
WORKBOOKS, SHEETS, CELLS

As a quick review, here's a rundown of all the functions, methods, and data types involved in reading a cell out of a spreadsheet file:

1. Import the `openpyxl` module.
2. Call the `openpyxl.load_workbook()` function.
3. Get a `Workbook` object.
4. Read the `active` member variable or call the `get_sheet_by_name()` workbook method.
5. Get a `Worksheet` object.
6. Use indexing or the `cell()` sheet method with `row` and `column` keyword arguments.
7. Get a `Cell` object.
8. Read the `Cell` object's `value` attribute.

PROJECT: READING DATA FROM A SPREADSHEET

Say you have a spreadsheet of data from the 2010 US Census and you have the boring task of going through its thousands of rows to count both the total population and the number of census tracts for each county. (A census tract is simply a geographic area defined for the purposes of the census.) Each row represents a single census tract. We'll name the spreadsheet file *censuspopdata.xlsx*, and you can download it from *http://nostarch.com/automatestuff/*. Its contents look like Figure 12-2.



	A	B	C	D	E
1	**CensusTract**	**State**	**County**	**POP2010**	
9841	06075010500	CA	San Francisco	2685	
9842	06075010600	CA	San Francisco	3894	
9843	06075010700	CA	San Francisco	5592	
9844	06075010800	CA	San Francisco	4578	
9845	06075010900	CA	San Francisco	4320	
9846	06075011000	CA	San Francisco	4827	
9847	06075011100	CA	San Francisco	5164	

Figure 12-2. The *censuspopdata.xlsx* spreadsheet

Even though Excel can calculate the sum of multiple selected cells, you'd still have to select the cells for each of the 3,000-plus counties. Even if it takes just a few seconds to calculate a county's population by hand, this would take hours to do for the whole spreadsheet.

In this project, you'll write a script that can read from the census spreadsheet file and calculate statistics for each county in a matter of seconds.

This is what your program does:

- Reads the data from the Excel spreadsheet.
- Counts the number of census tracts in each county.
- Counts the total population of each county.
- Prints the results.

This means your code will need to do the following:

- Open and read the cells of an Excel document with the `openpyxl` module.
- Calculate all the tract and population data and store it in a data structure.

- Write the data structure to a text file with the *.py* extension using the `pprint` module.

STEP 1: READ THE SPREADSHEET DATA

There is just one sheet in the *censuspopdata.xlsx* spreadsheet, named `'Population by Census Tract'`, and each row holds the data for a single census tract. The columns are the tract number (A), the state abbreviation (B), the county name (C), and the population of the tract (D).

Open a new file editor window and enter the following code. Save the file as *readCensusExcel.py*.

```
   #! python3
   # readCensusExcel.py - Tabulates population and number of census tracts
for
   # each county.

❶ import openpyxl, pprint
   print('Opening workbook...')
❷ wb = openpyxl.load_workbook('censuspopdata.xlsx')
❸ sheet = wb.get_sheet_by_name('Population by Census Tract')
   countyData = {}

   # TODO: Fill in countyData with each county's population and tracts.
   print('Reading rows...')
❹ for row in range(2, sheet.max_row + 1):
       # Each row in the spreadsheet has data for one census tract.
       state  = sheet['B' + str(row)].value
       county = sheet['C' + str(row)].value
       pop    = sheet['D' + str(row)].value

   # TODO: Open a new text file and write the contents of countyData to it.
```

This code imports the `openpyxl` module, as well as the `pprint` module that you'll use to print the final county data ❶. Then it opens the *censuspopdata.xlsx* file ❷, gets the sheet with the census data ❸, and begins iterating over its rows ❹.

Note that you've also created a variable named `countyData`, which will contain the populations and number of tracts you calculate for each county. Before you can store anything in it, though, you should determine exactly how you'll structure the data inside it.

STEP 2: POPULATE THE DATA STRUCTURE

The data structure stored in `countyData` will be a dictionary with state abbreviations as its keys. Each state abbreviation will map to another dictionary, whose keys are strings of the county names in that state. Each county name will in turn map to a dictionary with just two keys, `'tracts'` and `'pop'`. These keys map to the number of census tracts and population for the county. For example, the dictionary will look similar to this:

```
{'AK': {'Aleutians East': {'pop': 3141, 'tracts': 1},
        'Aleutians West': {'pop': 5561, 'tracts': 2},
        'Anchorage': {'pop': 291826, 'tracts': 55},
        'Bethel': {'pop': 17013, 'tracts': 3},
        'Bristol Bay': {'pop': 997, 'tracts': 1},
        --snip--
```

If the previous dictionary were stored in `countyData`, the following expressions would evaluate like this:

```
>>> countyData['AK']['Anchorage']['pop']
291826
>>> countyData['AK']['Anchorage']['tracts']
55
```

More generally, the `countyData` dictionary's keys will look like this:

```
countyData[state abbrev][county]['tracts']
countyData[state abbrev][county]['pop']
```

Now that you know how `countyData` will be structured, you can write the code that will fill it with the county data. Add the following code to the bottom of your program:

```
   #! python 3
   # readCensusExcel.py - Tabulates population and number of census tracts
for
   # each county.

   --snip--
   for row in range(2, sheet.max_row + 1):
       # Each row in the spreadsheet has data for one census tract.
       state  = sheet['B' + str(row)].value
       county = sheet['C' + str(row)].value
       pop    = sheet['D' + str(row)].value

       # Make sure the key for this state exists.
❶      countyData.setdefault(state, {})
```

```
        # Make sure the key for this county in this state exists.
❷       countyData[state].setdefault(county, {'tracts': 0, 'pop': 0})

        # Each row represents one census tract, so increment by one.
❸       countyData[state][county]['tracts'] += 1
        # Increase the county pop by the pop in this census tract.
❹       countyData[state][county]['pop'] += int(pop)

    # TODO: Open a new text file and write the contents of countyData to it.
```

The last two lines of code perform the actual calculation work, incrementing the value for `tracts` ❸ and increasing the value for `pop` ❹ for the current county on each iteration of the `for` loop.

The other code is there because you cannot add a county dictionary as the value for a state abbreviation key until the key itself exists in `countyData`. (That is, `countyData['AK']['Anchorage']['tracts'] += 1` will cause an error if the `'AK'` key doesn't exist yet.) To make sure the state abbreviation key exists in your data structure, you need to call the `setdefault()` method to set a value if one does not already exist for `state` ❶.

Just as the `countyData` dictionary needs a dictionary as the value for each state abbreviation key, each of *those* dictionaries will need its own dictionary as the value for each county key ❷. And each of *those* dictionaries in turn will need keys `'tracts'` and `'pop'` that start with the integer value `0`. (If you ever lose track of the dictionary structure, look back at the example dictionary at the start of this section.)

Since `setdefault()` will do nothing if the key already exists, you can call it on every iteration of the `for` loop without a problem.

STEP 3: WRITE THE RESULTS TO A FILE

After the `for` loop has finished, the `countyData` dictionary will contain all of the population and tract information keyed by county and state. At this point, you could program more code to write this to a text file or another Excel spreadsheet. For now, let's just use the `pprint.pformat()` function to write the `countyData` dictionary value as a massive string to a file named *census2010.py*. Add the following code to the bottom of your program (making sure to keep it unindented so that it stays outside the `for` loop):

```
#! python 3
# readCensusExcel.py - Tabulates population and number of census tracts for
# each county.get_active_sheet
--snip--

for row in range(2, sheet.max_row + 1):
--snip--

# Open a new text file and write the contents of countyData to it.
print('Writing results...')
resultFile = open('census2010.py', 'w')
resultFile.write('allData = ' + pprint.pformat(countyData))
resultFile.close()
print('Done.')
```

The `pprint.pformat()` function produces a string that itself is formatted as valid Python code. By outputting it to a text file named *census2010.py*, you've generated a Python program from your Python program! This may seem complicated, but the advantage is that you can now import *census2010.py* just like any other Python module. In the interactive shell, change the current working directory to the folder with your newly created *census2010.py* file (on my laptop, this is *C:\Python34*), and then import it:

```
>>> import os
>>> os.chdir('C:\\Python34')
>>> import census2010
>>> census2010.allData['AK']['Anchorage']
{'pop': 291826, 'tracts': 55}
>>> anchoragePop = census2010.allData['AK']['Anchorage']['pop']
>>> print('The 2010 population of Anchorage was ' + str(anchoragePop))
The 2010 population of Anchorage was 291826
```

The *readCensusExcel.py* program was throwaway code: Once you have its results saved to *census2010.py*, you won't need to run the program again. Whenever you need the county data, you can just run `import census2010`.

Calculating this data by hand would have taken hours; this program did it in a few seconds. Using OpenPyXL, you will have no trouble extracting information that is saved to an Excel spreadsheet and performing calculations on it. You can download the complete program from *http://nostarch.com/automatestuff/*.

IDEAS FOR SIMILAR PROGRAMS

Many businesses and offices use Excel to store various types of data, and it’s not uncommon for spreadsheets to become large and unwieldy. Any program that parses an Excel spreadsheet has a similar structure: It loads the spreadsheet file, preps some variables or data structures, and then loops through each of the rows in the spreadsheet. Such a program could do the following:

- Compare data across multiple rows in a spreadsheet.
- Open multiple Excel files and compare data between spreadsheets.
- Check whether a spreadsheet has blank rows or invalid data in any cells and alert the user if it does.
- Read data from a spreadsheet and use it as the input for your Python programs.

WRITING EXCEL DOCUMENTS

OpenPyXL also provides ways of writing data, meaning that your programs can create and edit spreadsheet files. With Python, it’s simple to create spreadsheets with thousands of rows of data.

CREATING AND SAVING EXCEL DOCUMENTS

Call the `openpyxl.Workbook()` function to create a new, blank `Workbook` object. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> wb.get_sheet_names()
['Sheet']
>>> sheet = wb.active
>>> sheet.title
'Sheet'
>>> sheet.title = 'Spam Bacon Eggs Sheet'
>>> wb.get_sheet_names()
['Spam Bacon Eggs Sheet']
```

The workbook will start off with a single sheet named *Sheet*. You can change the name of the sheet by storing a new string in its `title` attribute.

Any time you modify the `Workbook` object or its sheets and cells, the spreadsheet file will not be saved until you call the `save()` workbook method. Enter the following into the interactive shell (with *example.xlsx* in the current working directory):

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.active
>>> sheet.title = 'Spam Spam Spam'
>>> wb.save('example_copy.xlsx')
```

Here, we change the name of our sheet. To save our changes, we pass a filename as a string to the `save()` method. Passing a different filename than the original, such as `'example_copy.xlsx'`, saves the changes to a copy of the spreadsheet.

Whenever you edit a spreadsheet you've loaded from a file, you should always save the new, edited spreadsheet to a different filename than the original. That way, you'll still have the original spreadsheet file to work with in case a bug in your code caused the new, saved file to have incorrect or corrupt data.

CREATING AND REMOVING SHEETS

Sheets can be added to and removed from a workbook with the `create_sheet()` and `remove_sheet()` methods. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> wb.get_sheet_names()
['Sheet']
>>> wb.create_sheet()
<Worksheet "Sheet1">
>>> wb.get_sheet_names()
['Sheet', 'Sheet1']
>>> wb.create_sheet(index=0, title='First Sheet')
<Worksheet "First Sheet">
>>> wb.get_sheet_names()
['First Sheet', 'Sheet', 'Sheet1']
>>> wb.create_sheet(index=2, title='Middle Sheet')
<Worksheet "Middle Sheet">
>>> wb.get_sheet_names()
['First Sheet', 'Sheet', 'Middle Sheet', 'Sheet1']
```

The `create_sheet()` method returns a new `Worksheet` object named `Sheet`*`X`*, which by default is set to be the last sheet in the workbook. Optionally, the index and name of the new sheet can be specified with the `index` and `title` keyword arguments.

Continue the previous example by entering the following:

```
>>> wb.get_sheet_names()
['First Sheet', 'Sheet', 'Middle Sheet', 'Sheet1']
>>> wb.remove_sheet(wb.get_sheet_by_name('Middle Sheet'))
>>> wb.remove_sheet(wb.get_sheet_by_name('Sheet1'))
>>> wb.get_sheet_names()
['First Sheet', 'Sheet']
```

The `remove_sheet()` method takes a `Worksheet` object, not a string of the sheet name, as its argument. If you know only the name of a sheet you want to remove, call `get_sheet_by_name()` and pass its return value into `remove_sheet()`.

Remember to call the `save()` method to save the changes after adding sheets to or removing sheets from the workbook.

WRITING VALUES TO CELLS

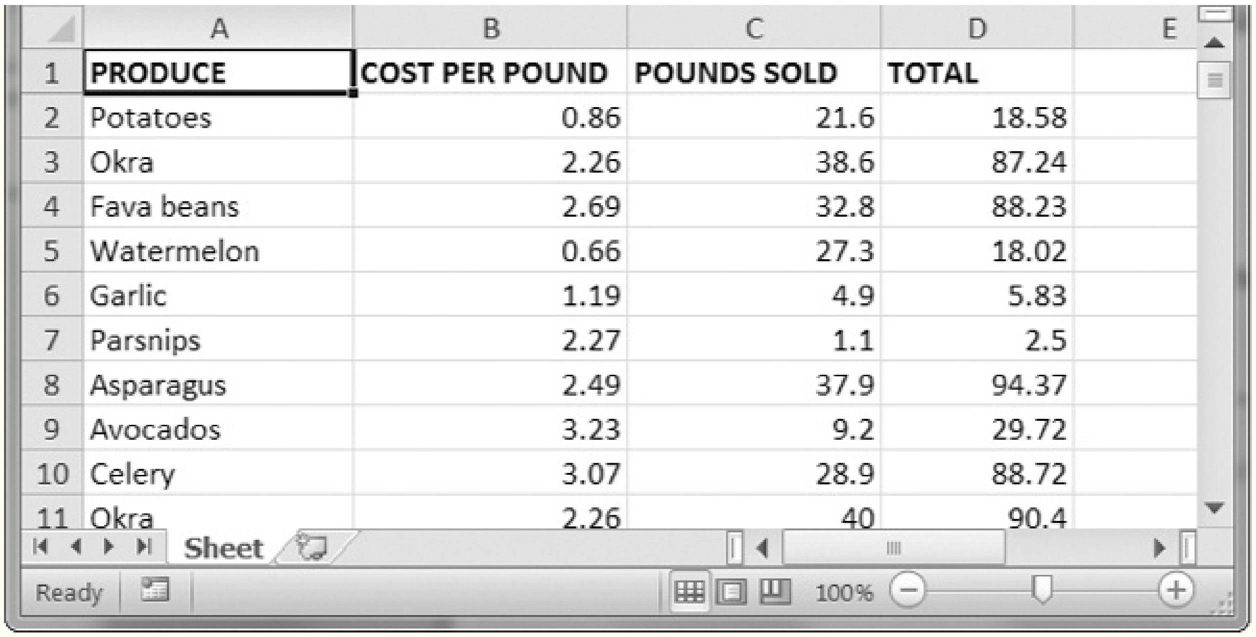
Writing values to cells is much like writing values to keys in a dictionary. Enter this into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb.get_sheet_by_name('Sheet')
>>> sheet['A1'] = 'Hello world!'
>>> sheet['A1'].value
'Hello world!'
```

If you have the cell's coordinate as a string, you can use it just like a dictionary key on the `Worksheet` object to specify which cell to write to.

PROJECT: UPDATING A SPREADSHEET

In this project, you'll write a program to update cells in a spreadsheet of produce sales. Your program will look through the spreadsheet, find specific kinds of produce, and update their prices. Download this spreadsheet from *http://nostarch.com/automatestuff/*. Figure 12-3 shows what the spreadsheet looks like.



	A	B	C	D
1	PRODUCE	COST PER POUND	POUNDS SOLD	TOTAL
2	Potatoes	0.86	21.6	18.58
3	Okra	2.26	38.6	87.24
4	Fava beans	2.69	32.8	88.23
5	Watermelon	0.66	27.3	18.02
6	Garlic	1.19	4.9	5.83
7	Parsnips	2.27	1.1	2.5
8	Asparagus	2.49	37.9	94.37
9	Avocados	3.23	9.2	29.72
10	Celery	3.07	28.9	88.72
11	Okra	2.26	40	90.4

Figure 12-3. A spreadsheet of produce sales

Each row represents an individual sale. The columns are the type of produce sold (A), the cost per pound of that produce (B), the number of pounds sold (C), and the total revenue from the sale (D). The TOTAL column is set to the Excel formula *=ROUND(B3*C3, 2)*, which multiplies the cost per pound by the number of pounds sold and rounds the result to the nearest cent. With this formula, the cells in the TOTAL column will automatically update themselves if there is a change in column B or C.

Now imagine that the prices of garlic, celery, and lemons were entered incorrectly, leaving you with the boring task of going through thousands of rows in this spreadsheet to update the cost per pound for any garlic, celery, and lemon rows. You can't do a simple find-and-replace for the price because there might be other items with the same price that you don't want to mistakenly "correct." For thousands of rows, this would take hours to do by hand. But you can write a program that can accomplish this in seconds.

Your program does the following:

- Loops over all the rows.
- If the row is for garlic, celery, or lemons, changes the price.

This means your code will need to do the following:

- Open the spreadsheet file.
- For each row, check whether the value in column A is `Celery`, `Garlic`, or `Lemon`.
- If it is, update the price in column B.
- Save the spreadsheet to a new file (so that you don't lose the old spreadsheet, just in case).

STEP 1: SET UP A DATA STRUCTURE WITH THE UPDATE INFORMATION

The prices that you need to update are as follows:

Celery	1.19
Garlic	3.07
Lemon	1.27

You could write code like this:

```
if produceName == 'Celery':
    cellObj = 1.19
if produceName == 'Garlic':
    cellObj = 3.07
if produceName == 'Lemon':
    cellObj = 1.27
```

Having the produce and updated price data hardcoded like this is a bit inelegant. If you needed to update the spreadsheet again with different prices or different produce, you would have to change a lot of the code. Every time you change code, you risk introducing bugs.

A more flexible solution is to store the corrected price information in a dictionary and write your code to use this data structure. In a new file editor window, enter the following code:

```
#! python3
# updateProduce.py - Corrects costs in produce sales spreadsheet.

import openpyxl

wb = openpyxl.load_workbook('produceSales.xlsx')
sheet = wb.get_sheet_by_name('Sheet')
```

```
# The produce types and their updated prices
PRICE_UPDATES = {'Garlic': 3.07,
                 'Celery': 1.19,
                 'Lemon': 1.27}

# TODO: Loop through the rows and update the prices.
```

Save this as *updateProduce.py*. If you need to update the spreadsheet again, you'll need to update only the `PRICE_UPDATES` dictionary, not any other code.

Step 2: Check All Rows and Update Incorrect Prices

The next part of the program will loop through all the rows in the spreadsheet. Add the following code to the bottom of *updateProduce.py*:

```
   #! python3
   # updateProduce.py - Corrects costs in produce sales spreadsheet.

   --snip--

   # Loop through the rows and update the prices.
❶ for rowNum in range(2, sheet.max_row + 1):  # skip the first row
❷     produceName = sheet.cell(row=rowNum, column=1).value
❸     if produceName in PRICE_UPDATES:
           sheet.cell(row=rowNum, column=2).value =
PRICE_UPDATES[produceName]

❹ wb.save('updatedProduceSales.xlsx')
```

We loop through the rows starting at row 2, since row 1 is just the header ❶. The cell in column 1 (that is, column A) will be stored in the variable `produceName` ❷. If `produceName` exists as a key in the `PRICE_UPDATES` dictionary ❸, then you know this is a row that must have its price corrected. The correct price will be in `PRICE_UPDATES[produceName]`.

Notice how clean using `PRICE_UPDATES` makes the code. Only one `if` statement, rather than code like `if produceName == 'Garlic':`, is necessary for every type of produce to update. And since the code uses the `PRICE_UPDATES` dictionary instead of hardcoding the produce names and updated costs into the `for` loop, you modify only the `PRICE_UPDATES` dictionary and not the code if the produce sales spreadsheet needs additional changes.

After going through the entire spreadsheet and making changes, the code saves the `Workbook` object to *updatedProduceSales.xlsx* ❹. It doesn't overwrite the old spreadsheet just in case there's a bug in your program and the updated spreadsheet is wrong. After checking that the updated spreadsheet looks right, you can delete the old spreadsheet.

You can download the complete source code for this program from *http://nostarch.com/automatestuff/*.

Ideas for Similar Programs

Since many office workers use Excel spreadsheets all the time, a program that can automatically edit and write Excel files could be really useful. Such a program could do the following:

- Read data from one spreadsheet and write it to parts of other spreadsheets.
- Read data from websites, text files, or the clipboard and write it to a spreadsheet.
- Automatically "clean up" data in spreadsheets. For example, it could use regular expressions to read multiple formats of phone numbers and edit them to a single, standard format.

Setting the Font Style of Cells

Styling certain cells, rows, or columns can help you emphasize important areas in your spreadsheet. In the produce spreadsheet, for example, your program could apply bold text to the potato, garlic, and parsnip rows. Or perhaps you want to italicize every row with a cost per pound greater than $5. Styling parts of a large spreadsheet by hand would be tedious, but your programs can do it instantly.

To customize font styles in cells, important, import the `Font()` function from the `openpyxl.styles` module.

```
from openpyxl.styles import Font
```

This allows you to type `Font()` instead of `openpyxl.styles.Font()`. (See Importing Modules to review this style of `import` statement.)

Here’s an example that creates a new workbook and sets cell A1 to have a 24-point, italicized font. Enter the following into the interactive shell:

```
   >>> import openpyxl
   >>> from openpyxl.styles import Font
   >>> wb = openpyxl.Workbook()
   >>> sheet = wb.get_sheet_by_name('Sheet')
❶ >>> italic24Font = Font(size=24, italic=True)
❷ >>> sheet['A1'].font = italic24Font
   >>> sheet['A1'] = 'Hello world!'
   >>> wb.save('styled.xlsx')
```

A cell’s style can be set by assigning the `Font` object to the `style` attribute.

In this example, `Font(size=24, italic=True)` returns a `Font` object, which is stored in `italic24Font` ❶. The keyword arguments to `Font()`, `size` and `italic`, configure the `Font` object. And when `fontObj` is assigned to the cell’s `font` attribute ❷, all that font styling information gets applied to cell A1.

Font Objects

To set font style attributes, you pass keyword arguments to `Font()`. Table 12-2 shows the possible keyword arguments for the `Font()` function.

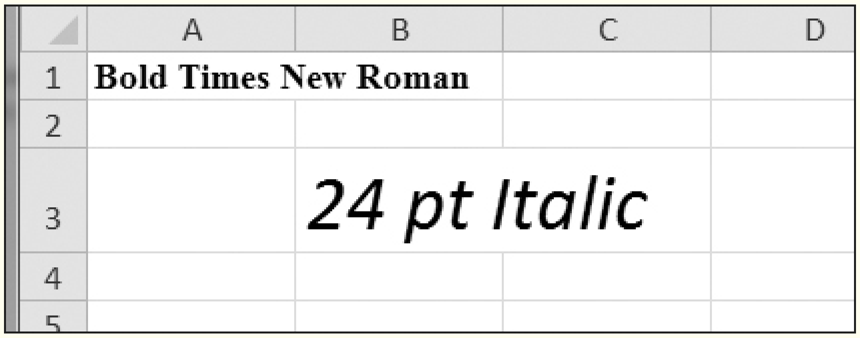
Table 12-2. Keyword Arguments for Font `style` Attributes

Keyword argument	Data type	Description
`name`	String	The font name, such as `'Calibri'` or `'Times New Roman'`
`size`	Integer	The point size
`bold`	Boolean	`True`, for bold font
`italic`	Boolean	`True`, for italic font

You can call `Font()` to create a `Font` object and store that `Font` object in a variable. You then pass that to `Style()`, store the resulting `Style` object in a variable, and assign that variable to a `Cell` object’s `style` attribute. For example, this code creates various font styles:

```
>>> import openpyxl
>>> from openpyxl.styles import Font
>>> wb = openpyxl.Workbook()
>>> sheet = wb.get_sheet_by_name('Sheet')
>>> fontObj1 = Font(name='Times New Roman', bold=True)
>>> sheet['A1'].font = fontObj1
>>> sheet['A1'] = 'Bold Times New Roman'

>>> fontObj2 = Font(size=24, italic=True)
>>> sheet['B3'].font = fontObj2
>>> sheet['B3'] = '24 pt Italic'

>>> wb.save('styles.xlsx')
```

Here, we store a `Font` object in `fontObj1` and then set the A1 `Cell` object's `font` attribute to `fontObj1`. We repeat the process with another `Font` object to set the style of a second cell. After you run this code, the styles of the A1 and B3 cells in the spreadsheet will be set to custom font styles, as shown in Figure 12-4.



	A	B	C	D
1	Bold Times New Roman			
2				
3		24 pt Italic		
4				
5				

Figure 12-4. A spreadsheet with custom font styles

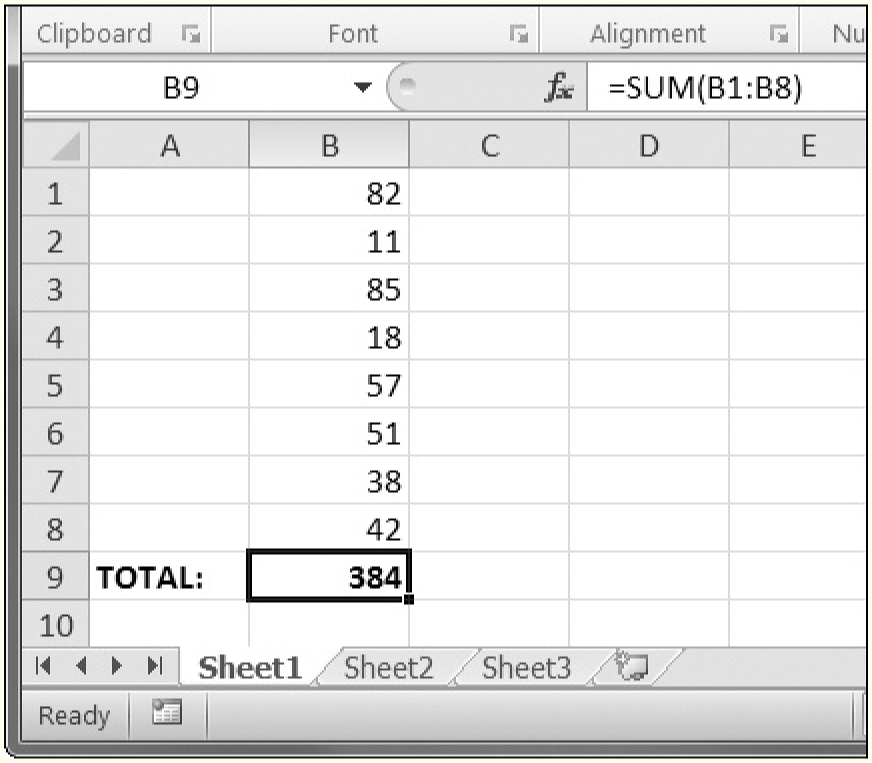
For cell A1, we set the font name to `'Times New Roman'` and set `bold` to `true`, so our text appears in bold Times New Roman. We didn't specify a size, so the `openpyxl` default, 11, is used. In cell B3, our text is italic, with a size of 24; we didn't specify a font name, so the `openpyxl` default, Calibri, is used.

FORMULAS

Formulas, which begin with an equal sign, can configure cells to contain values calculated from other cells. In this section, you'll use the `openpyxl` module to programmatically add formulas to cells, just like any normal value. For example:

```
>>> sheet['B9'] = '=SUM(B1:B8)'
```

This will store *=SUM(B1:B8)* as the value in cell B9. This sets the B9 cell to a formula that calculates the sum of values in cells B1 to B8. You can see this in action in Figure 12-5.



B9 =SUM(B1:B8)

	A	B	C	D	E
1		82			
2		11			
3		85			
4		18			
5		57			
6		51			
7		38			
8		42			
9	TOTAL:	384			
10					

Figure 12-5. Cell B9 contains the formula *=SUM(B1:B8)*, which adds the cells B1 to B8.

A formula is set just like any other text value in a cell. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb.active
>>> sheet['A1'] = 200
>>> sheet['A2'] = 300
>>> sheet['A3'] = '=SUM(A1:A2)'
>>> wb.save('writeFormula.xlsx')
```

The cells in A1 and A2 are set to 200 and 300, respectively. The value in cell A3 is set to a formula that sums the values in A1 and A2. When the spreadsheet is opened in Excel, A3 will display its value as 500.

Excel formulas offer a level of programmability for spreadsheets but can quickly become unmanageable for complicated tasks. For example, even if you’re deeply familiar with Excel formulas, it’s a headache to try to decipher what *=IFERROR(TRIM(IF(LEN(VLOOKUP(F7, Sheet2!$A$1:$B$10000, 2, FALSE))>0,SUBSTITUTE(VLOOKUP(F7, Sheet2!$A$1:$B$10000, 2, FALSE), “ ”, “”),“”)), “”)* actually does. Python code is much more readable.

ADJUSTING ROWS AND COLUMNS

In Excel, adjusting the sizes of rows and columns is as easy as clicking and dragging the edges of a row or column header. But if you need to set a row or column’s size based on its cells’ contents or if you want to set sizes in a large number of spreadsheet files, it will be much quicker to write a Python program to do it.

Rows and columns can also be hidden entirely from view. Or they can be “frozen” in place so that they are always visible on the screen and appear on every page when the spreadsheet is printed (which is handy for headers).

SETTING ROW HEIGHT AND COLUMN WIDTH

`Worksheet` objects have `row_dimensions` and `column_dimensions` attributes that control row heights and column widths. Enter this into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb.active
>>> sheet['A1'] = 'Tall row'
>>> sheet['B2'] = 'Wide column'
>>> sheet.row_dimensions[1].height = 70
>>> sheet.column_dimensions['B'].width = 20
>>> wb.save('dimensions.xlsx')
```

A sheet’s `row_dimensions` and `column_dimensions` are dictionary-like values; `row_dimensions` contains `RowDimension` objects and `column_dimensions` contains `ColumnDimension` objects. In `row_dimensions`, you can access one of the objects using the number of the row (in this case, 1 or 2). In `column_dimensions`, you can access one of the objects using the letter of the column (in this case, A or B).

The *dimensions.xlsx* spreadsheet looks like Figure 12-6.

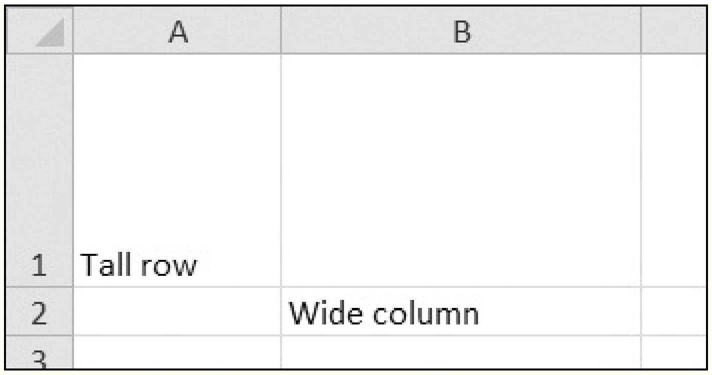


Figure 12-6. Row 1 and column B set to larger heights and widths

Once you have the `RowDimension` object, you can set its height. Once you have the `ColumnDimension` object, you can set its width. The row height can be set to an integer or float value between `0` and `409`. This value represents the height measured in *points*, where one point equals 1/72 of an inch. The default row height is 12.75. The column width can be set to an integer or float value between `0` and `255`. This value represents the number of characters at the default font size (11 point) that can be displayed in the cell. The default column width is 8.43 characters. Columns with widths of `0` or rows with heights of `0` are hidden from the user.

MERGING AND UNMERGING CELLS

A rectangular area of cells can be merged into a single cell with the `merge_cells()` sheet method. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb.active
>>> sheet.merge_cells('A1:D3')
>>> sheet['A1'] = 'Twelve cells merged together.'
>>> sheet.merge_cells('C5:D5')
>>> sheet['C5'] = 'Two merged cells.'
>>> wb.save('merged.xlsx')
```

The argument to `merge_cells()` is a single string of the top-left and bottom-right cells of the rectangular area to be merged: `'A1:D3'` merges 12 cells into a single cell. To set the value of these merged cells, simply set the value of the top-left cell of the merged group.

When you run this code, *merged.xlsx* will look like Figure 12-7.

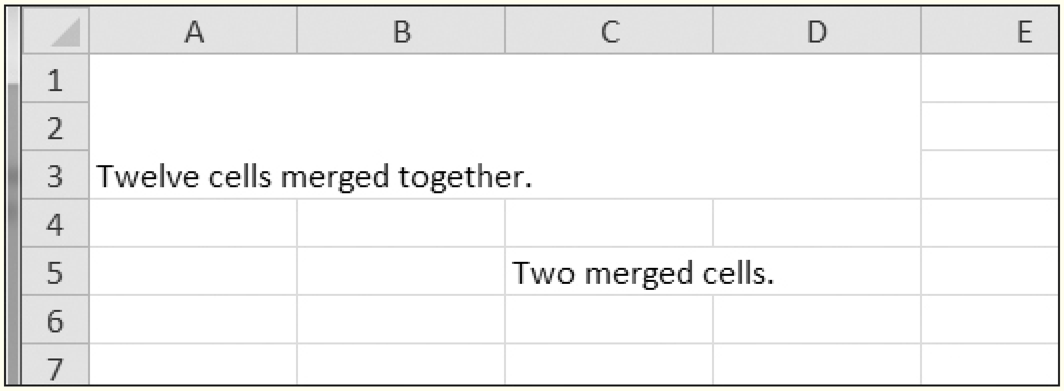


Figure 12-7. Merged cells in a spreadsheet

To unmerge cells, call the `unmerge_cells()` sheet method. Enter this into the interactive shell.

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('merged.xlsx')
>>> sheet = wb.active
>>> sheet.unmerge_cells('A1:D3')
>>> sheet.unmerge_cells('C5:D5')
>>> wb.save('merged.xlsx')
```

If you save your changes and then take a look at the spreadsheet, you'll see that the merged cells have gone back to being individual cells.

FREEZE PANES

For spreadsheets too large to be displayed all at once, it's helpful to "freeze" a few of the top rows or leftmost columns onscreen. Frozen column or row headers, for example, are always visible to the user even as they scroll through the spreadsheet. These are known as *freeze panes*. In OpenPyXL, each `Worksheet` object has a `freeze_panes` attribute that can be set to a `Cell` object or a string of a cell's coordinates. Note that all rows above and all columns to the left of this cell will be frozen, but the row and column of the cell itself will not be frozen.

To unfreeze all panes, set `freeze_panes` to `None` or `'A1'`. Table 12-3 shows which rows and columns will be frozen for some example settings of `freeze_panes`.

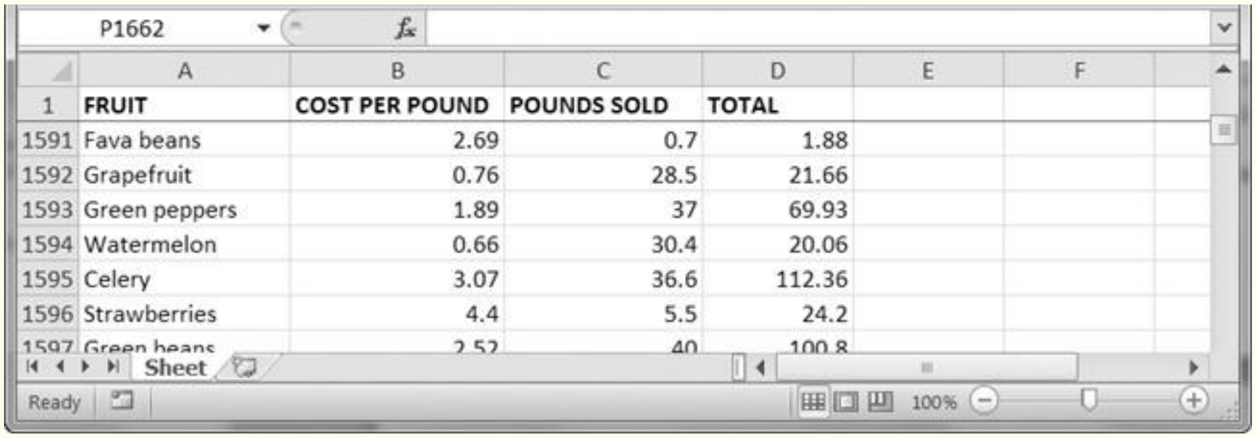
Table 12-3. Frozen Pane Examples

`freeze_panes` setting	Rows and columns frozen
`sheet.freeze_panes = 'A2'`	Row 1
`sheet.freeze_panes = 'B1'`	Column A
`sheet.freeze_panes = 'C1'`	Columns A and B
`sheet.freeze_panes = 'C2'`	Row 1 and columns A and B
`sheet.freeze_panes = 'A1'` or `sheet.freeze_panes = None`	No frozen panes

Make sure you have the produce sales spreadsheet from *http://nostarch.com/automatestuff/*. Then enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('produceSales.xlsx')
>>> sheet = wb.active
>>> sheet.freeze_panes = 'A2'
>>> wb.save('freezeExample.xlsx')
```

If you set the `freeze_panes` attribute to `'A2'`, row 1 will always be viewable, no matter where the user scrolls in the spreadsheet. You can see this in Figure 12-8.



	A	B	C	D	E	F
1	FRUIT	COST PER POUND	POUNDS SOLD	TOTAL		
1591	Fava beans	2.69	0.7	1.88		
1592	Grapefruit	0.76	28.5	21.66		
1593	Green peppers	1.89	37	69.93		
1594	Watermelon	0.66	30.4	20.06		
1595	Celery	3.07	36.6	112.36		
1596	Strawberries	4.4	5.5	24.2		
1597	Green beans	2.52	40	100.8		

Figure 12-8. With `freeze_panes` set to `'A2'`, row 1 is always visible even as the user scrolls down.

CHARTS

OpenPyXL supports creating bar, line, scatter, and pie charts using the data in a sheet's cells. To make a chart, you need to do the following:

1. Create a `Reference` object from a rectangular selection of cells.
2. Create a `Series` object by passing in the `Reference` object.
3. Create a `Chart` object.
4. Append the `Series` object to the `Chart` object.
5. Add the `Chart` object to the `Worksheet` object, optionally specifying which cell the top left corner of the chart should be positioned..

The `Reference` object requires some explaining. `Reference` objects are created by calling the `openpyxl.chart.Reference()` function and passing three arguments:

1. The `Worksheet` object containing your chart data.
2. A tuple of two integers, representing the top-left cell of the rectangular selection of cells containing your chart data: The first integer in the tuple is the row, and the second is the column. Note that `1` is the first row, not `0`.
3. A tuple of two integers, representing the bottom-right cell of the rectangular selection of cells containing your chart data: The first integer in the tuple is the row, and the second is the column.

Figure 12-9 shows some sample coordinate arguments.

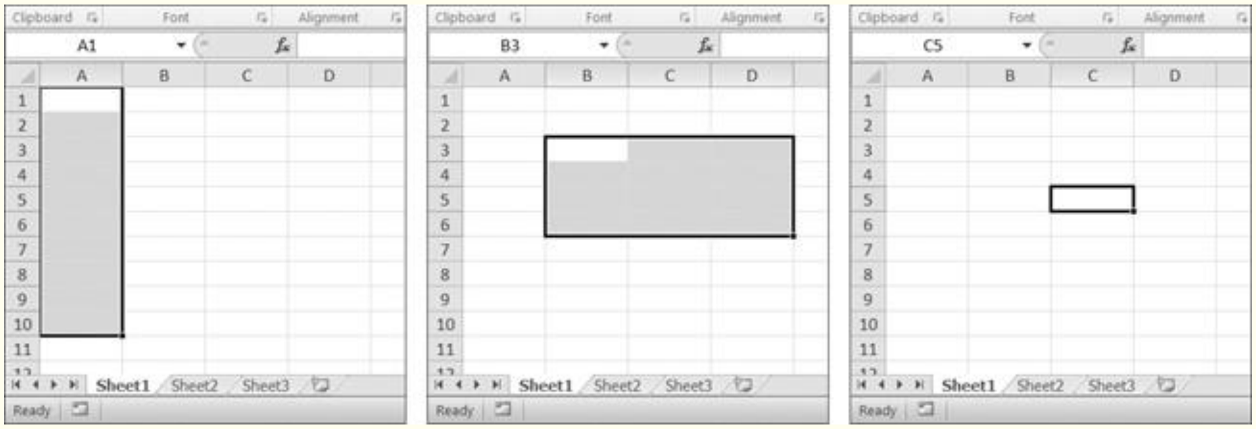
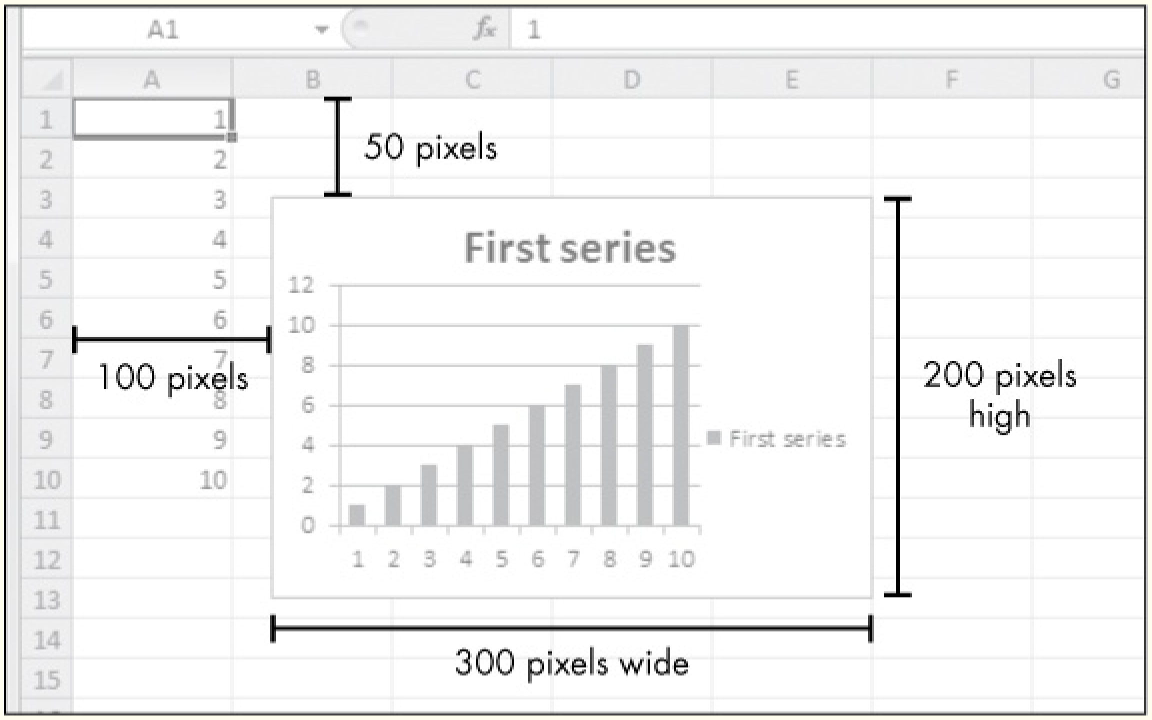


Figure 12-9. From left to right: `(1, 1), (10, 1)`; `(3, 2), (6, 4)`; `(5, 3), (5, 3)`

Enter this interactive shell example to create a bar chart and add it to the spreadsheet:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb.active
>>> for i in range(1, 11):         # create some data in column A
        sheet['A' + str(i)] = i

>>> refObj = openpyxl.chart.Reference(sheet, min_col=1, min_row=1, max_col=1,
max_row=10)

>>> seriesObj = openpyxl.chart.Series(refObj, title='First series')

>>> chartObj = openpyxl.chart.BarChart()
>>> chartObj.title = 'My Chart'
>>> chartObj.append(seriesObj)
>>> sheet.add_chart(chartObj, 'C5')
>>> wb.save('sampleChart.xlsx')
```

This produces a spreadsheet that looks like Figure 12-10.



	A
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10

Figure 12-10. A spreadsheet with a chart added

We've created a bar chart by calling `openpyxl.chart.BarChart()`. You can also create line charts, scatter charts, and pie charts by calling `openpyxl.chart.LineChart()`, `openpyxl.chart.ScatterChart()`, and `openpyxl.chart.PieChart()`.

Unfortunately, in the current version of OpenPyXL (2.3.3), the `load_workbook()` function does not load charts in Excel files. Even if the Excel file has charts, the loaded `Workbook` object will not include them. If you load a `Workbook` object and immediately save it to the same *.xlsx* filename, you will effectively remove the charts from it.

SUMMARY

Often the hard part of processing information isn’t the processing itself but simply getting the data in the right format for your program. But once you have your spreadsheet loaded into Python, you can extract and manipulate its data much faster than you could by hand.

You can also generate spreadsheets as output from your programs. So if colleagues need your text file or PDF of thousands of sales contacts transferred to a spreadsheet file, you won’t have to tediously copy and paste it all into Excel.

Equipped with the `openpyxl` module and some programming knowledge, you’ll find processing even the biggest spreadsheets a piece of cake.

WORKING WITH PDF AND WORD DOCUMENTS

PDF and Word documents are binary files, which makes them much more complex than plaintext files. In addition to text, they store lots of font, color, and layout information. If you want your programs to read or write to PDFs or Word documents, you'll need to do more than simply pass their filenames to `open()`.

Fortunately, there are Python modules that make it easy for you to interact with PDFs and Word documents. This chapter will cover two such modules: PyPDF2 and Python-Docx.

PDF DOCUMENTS

PDF stands for *Portable Document Format* and uses the *.pdf* file extension. Although PDFs support many features, this chapter will focus on the two things you'll be doing most often with them: reading text content from PDFs and crafting new PDFs from existing documents.

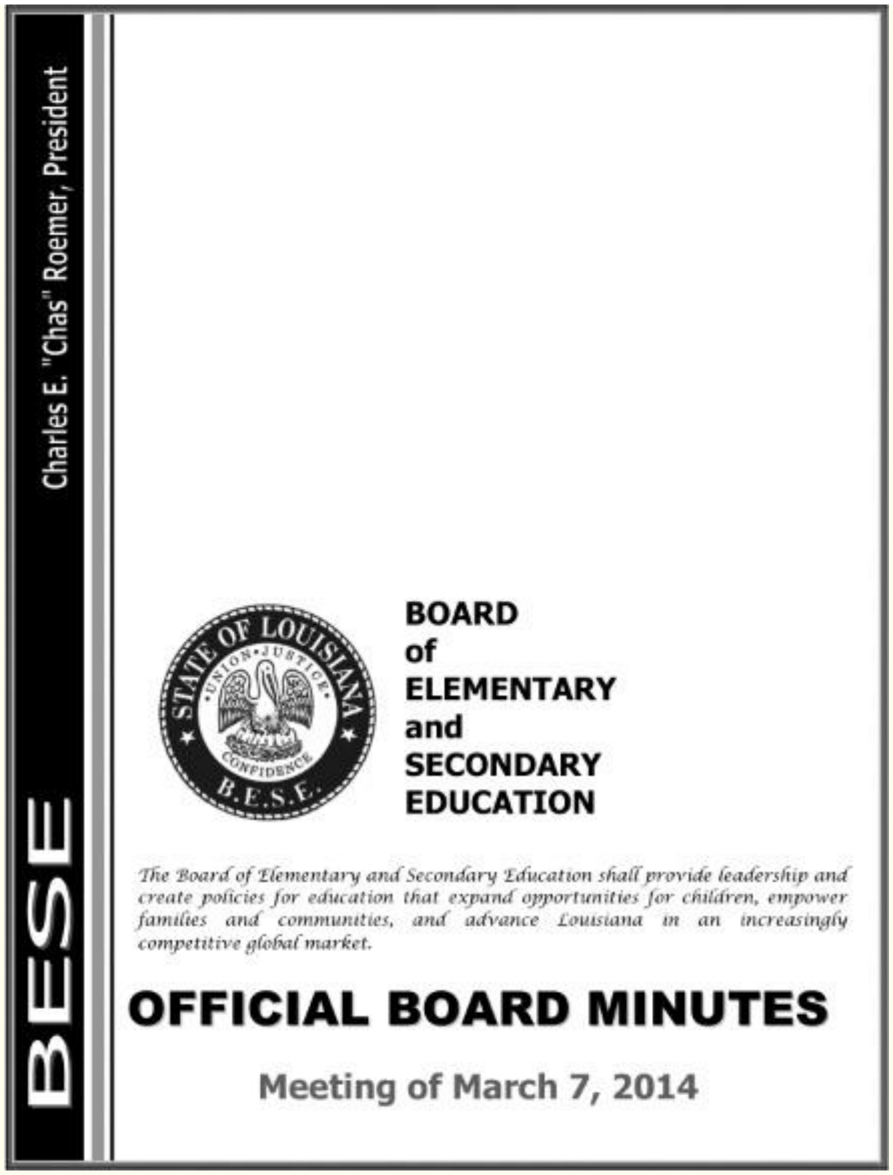
The module you'll use to work with PDFs is PyPDF2. To install it, run `pip install PyPDF2` from the command line. This module name is case sensitive, so make sure the *y* is lowercase and everything else is uppercase. (Check out Appendix A for full details about installing third-party modules.) If the module was installed correctly, running `import PyPDF2` in the interactive shell shouldn't display any errors.

The Problematic PDF Format

While PDF files are great for laying out text in a way that’s easy for people to print and read, they’re not straightforward for software to parse into plaintext. As such, PyPDF2 might make mistakes when extracting text from a PDF and may even be unable to open some PDFs at all. There isn’t much you can do about this, unfortunately. PyPDF2 may simply be unable to work with some of your particular PDF files. That said, I haven’t found any PDF files so far that can’t be opened with PyPDF2.

Extracting Text from PDFs

PyPDF2 does not have a way to extract images, charts, or other media from PDF documents, but it can extract text and return it as a Python string. To start learning how PyPDF2 works, we’ll use it on the example PDF shown in Figure 13-1.



Charles E. "Chas" Roemer, President

BESE

STATE OF LOUISIANA · UNION · JUSTICE · CONFIDENCE · B.E.S.E.

BOARD of ELEMENTARY and SECONDARY EDUCATION

The Board of Elementary and Secondary Education shall provide leadership and create policies for education that expand opportunities for children, empower families and communities, and advance Louisiana in an increasingly competitive global market.

OFFICIAL BOARD MINUTES

Meeting of March 7, 2014

Figure 13-1. The PDF page that we will be extracting text from

Download this PDF from http://nostarch.com/automatestuff/, and enter the following into the interactive shell:

```
   >>> import PyPDF2
   >>> pdfFileObj = open('meetingminutes.pdf', 'rb')
   >>> pdfReader = PyPDF2.PdfFileReader(pdfFileObj)
❶ >>> pdfReader.numPages
   19
❷ >>> pageObj = pdfReader.getPage(0)
❸ >>> pageObj.extractText()
   'OOFFFFIICCIIAALL BBOOAARRDD MMIINNUUTTEESS Meeting of March 7, 2015
   \n     The Board of Elementary and Secondary Education shall provide
leadership
```

```
and create policies for education that expand opportunities for children,
empower families and communities, and advance Louisiana in an increasingly
competitive global market. BOARD of ELEMENTARY and SECONDARY EDUCATION '
```

First, import the `PyPDF2` module. Then open *meetingminutes.pdf* in read binary mode and store it in `pdfFileObj`. To get a `PdfFileReader` object that represents this PDF, call `PyPDF2.PdfFileReader()` and pass it `pdfFileObj`. Store this `PdfFileReader` object in `pdfReader`.

The total number of pages in the document is stored in the `numPages` attribute of a `PdfFileReader` object ❶. The example PDF has 19 pages, but let's extract text from only the first page.

To extract text from a page, you need to get a `Page` object, which represents a single page of a PDF, from a `PdfFileReader` object. You can get a `Page` object by calling the `getPage()` method ❷ on a `PdfFileReader` object and passing it the page number of the page you're interested in—in our case, 0.

PyPDF2 uses a *zero-based index* for getting pages: The first page is page 0, the second is Introduction, and so on. This is always the case, even if pages are numbered differently within the document. For example, say your PDF is a three-page excerpt from a longer report, and its pages are numbered 42, 43, and 44. To get the first page of this document, you would want to call `pdfReader.getPage(0)`, not `getPage(42)` or `getPage(1)`.

Once you have your `Page` object, call its `extractText()` method to return a string of the page's text ❸. The text extraction isn't perfect: The text *Charles E. "Chas" Roemer, President* from the PDF is absent from the string returned by `extractText()`, and the spacing is sometimes off. Still, this approximation of the PDF text content may be good enough for your program.

Decrypting PDFs

Some PDF documents have an encryption feature that will keep them from being read until whoever is opening the document provides a password. Enter the following into the interactive shell with the PDF you downloaded, which has been encrypted with the password *rosebud*:

```
>>> import PyPDF2
```

```
   >>> pdfReader = PyPDF2.PdfFileReader(open('encrypted.pdf', 'rb'))
❶ >>> pdfReader.isEncrypted
   True
   >>> pdfReader.getPage(0)
❷ Traceback (most recent call last):
     File "<pyshell#173>", line 1, in <module>
       pdfReader.getPage()
     --snip--
     File "C:\Python34\lib\site-packages\PyPDF2\pdf.py", line 1173, in
getObject
       raise utils.PdfReadError("file has not been decrypted")
   PyPDF2.utils.PdfReadError: file has not been decrypted
❸ >>> pdfReader.decrypt('rosebud')
   1
   >>> pageObj = pdfReader.getPage(0)
```

All `PdfFileReader` objects have an `isEncrypted` attribute that is `True` if the PDF is encrypted and `False` if it isn't ❶. Any attempt to call a function that reads the file before it has been decrypted with the correct password will result in an error ❷.

To read an encrypted PDF, call the `decrypt()` function and pass the password as a string ❸. After you call `decrypt()` with the correct password, you'll see that calling `getPage()` no longer causes an error. If given the wrong password, the `decrypt()` function will return `0` and `getPage()` will continue to fail. Note that the `decrypt()` method decrypts only the `PdfFileReader` object, not the actual PDF file. After your program terminates, the file on your hard drive remains encrypted. Your program will have to call `decrypt()` again the next time it is run.

CREATING PDFS

PyPDF2's counterpart to `PdfFileReader` objects is `PdfFileWriter` objects, which can create new PDF files. But PyPDF2 cannot write arbitrary text to a PDF like Python can do with plaintext files. Instead, PyPDF2's PDF-writing capabilities are limited to copying pages from other PDFs, rotating pages, overlaying pages, and encrypting files.

PyPDF2 doesn't allow you to directly edit a PDF. Instead, you have to create a new PDF and then copy content over from an existing document. The examples in this section will follow this general approach:

1. Open one or more existing PDFs (the source PDFs) into `PdfFileReader` objects.

2. Create a new `PdfFileWriter` object.
3. Copy pages from the `PdfFileReader` objects into the `PdfFileWriter` object.
4. Finally, use the `PdfFileWriter` object to write the output PDF.

Creating a `PdfFileWriter` object creates only a value that represents a PDF document in Python. It doesn't create the actual PDF file. For that, you must call the PdfFileWriter's `write()` method.

The `write()` method takes a regular `File` object that has been opened in *write-binary* mode. You can get such a `File` object by calling Python's `open()` function with two arguments: the string of what you want the PDF's filename to be and `'wb'` to indicate the file should be opened in write-binary mode.

If this sounds a little confusing, don't worry—you'll see how this works in the following code examples.

Copying Pages

You can use PyPDF2 to copy pages from one PDF document to another. This allows you to combine multiple PDF files, cut unwanted pages, or reorder pages.

Download *meetingminutes.pdf* and *meetingminutes2.pdf* from http://nostarch.com/automatestuff/ and place the PDFs in the current working directory. Enter the following into the interactive shell:

```
  >>> import PyPDF2
  >>> pdf1File = open('meetingminutes.pdf', 'rb')
  >>> pdf2File = open('meetingminutes2.pdf', 'rb')
❶ >>> pdf1Reader = PyPDF2.PdfFileReader(pdf1File)
❷ >>> pdf2Reader = PyPDF2.PdfFileReader(pdf2File)
❸ >>> pdfWriter = PyPDF2.PdfFileWriter()

  >>> for pageNum in range(pdf1Reader.numPages):
❹         pageObj = pdf1Reader.getPage(pageNum)
❺         pdfWriter.addPage(pageObj)

  >>> for pageNum in range(pdf2Reader.numPages):
❻         pageObj = pdf2Reader.getPage(pageNum)
❼         pdfWriter.addPage(pageObj)

❽ >>> pdfOutputFile = open('combinedminutes.pdf', 'wb')
  >>> pdfWriter.write(pdfOutputFile)
```

```
   >>> pdfOutputFile.close()
   >>> pdf1File.close()
   >>> pdf2File.close()
```

Open both PDF files in read binary mode and store the two resulting `File` objects in `pdf1File` and `pdf2File`. Call `PyPDF2.PdfFileReader()` and pass it `pdf1File` to get a `PdfFileReader` object for *meetingminutes.pdf* ❶. Call it again and pass it `pdf2File` to get a `PdfFileReader` object for *meetingminutes2.pdf* ❷. Then create a new `PdfFileWriter` object, which represents a blank PDF document ❸.

Next, copy all the pages from the two source PDFs and add them to the `PdfFileWriter` object. Get the `Page` object by calling `getPage()` on a `PdfFileReader` object ❹. Then pass that `Page` object to your PdfFileWriter's `addPage()` method ❺. These steps are done first for `pdf1Reader` and then again for `pdf2Reader`. When you're done copying pages, write a new PDF called *combinedminutes.pdf* by passing a `File` object to the PdfFileWriter's `write()` method ❻.

NOTE

PyPDF2 cannot insert pages in the middle of a `PdfFileWriter` object; the `addPage()` method will only add pages to the end.

You have now created a new PDF file that combines the pages from *meetingminutes.pdf* and *meetingminutes2.pdf* into a single document. Remember that the `File` object passed to `PyPDF2.PdfFileReader()` needs to be opened in read-binary mode by passing `'rb'` as the second argument to `open()`. Likewise, the `File` object passed to `PyPDF2.PdfFileWriter()` needs to be opened in write-binary mode with `'wb'`.

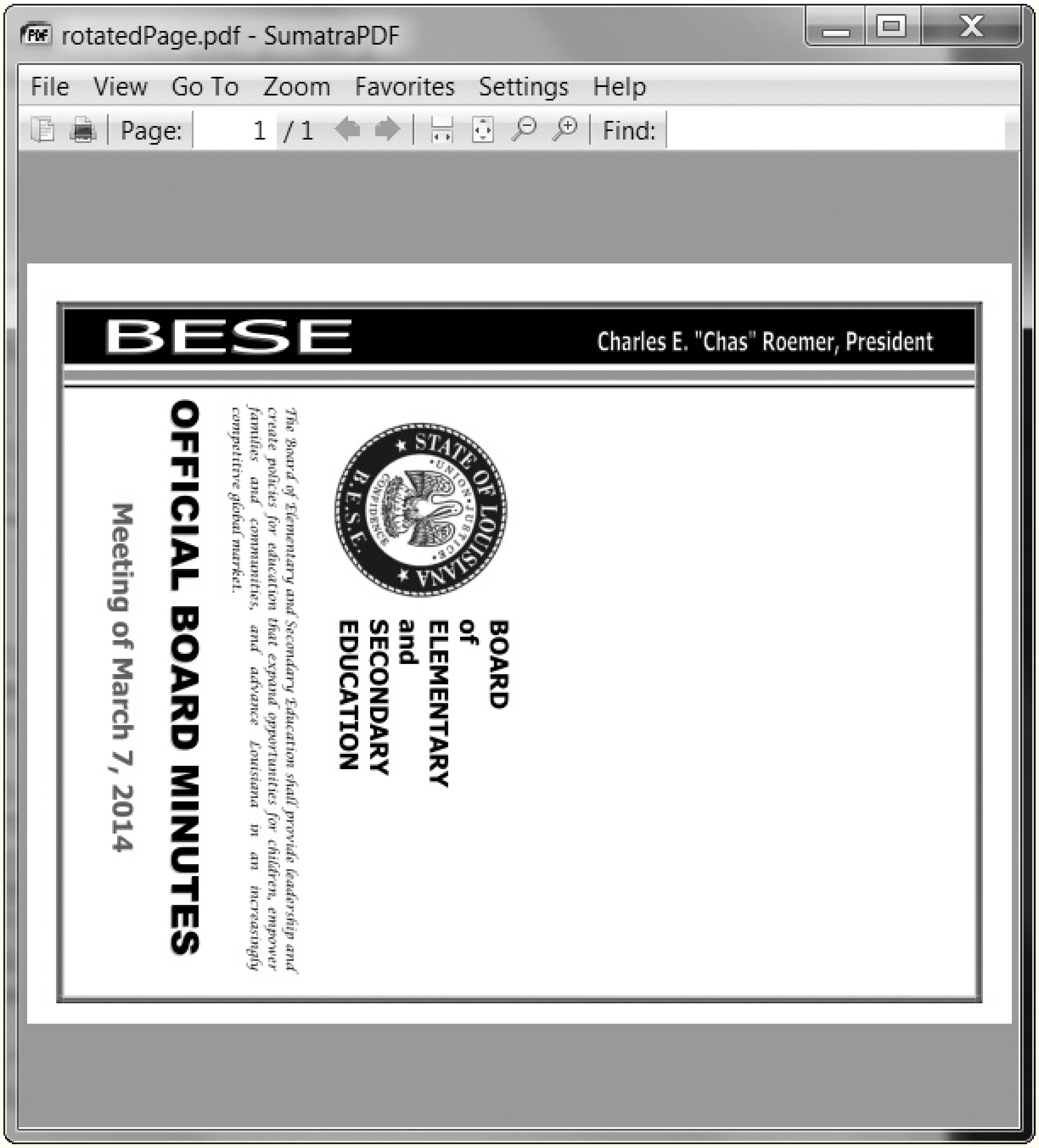
ROTATING PAGES

The pages of a PDF can also be rotated in 90-degree increments with the `rotateClockwise()` and `rotateCounterClockwise()` methods. Pass one of the integers `90`, `180`, or `270` to these methods. Enter the following into the interactive shell, with the *meetingminutes.pdf* file in the current working directory:

```
   >>> import PyPDF2
   >>> minutesFile = open('meetingminutes.pdf', 'rb')
   >>> pdfReader = PyPDF2.PdfFileReader(minutesFile)
❶ >>> page = pdfReader.getPage(0)
❷ >>> page.rotateClockwise(90)
   {'/Contents': [IndirectObject(961, 0), IndirectObject(962, 0),
   --snip--
   }
   >>> pdfWriter = PyPDF2.PdfFileWriter()
   >>> pdfWriter.addPage(page)
❸ >>> resultPdfFile = open('rotatedPage.pdf', 'wb')
   >>> pdfWriter.write(resultPdfFile)
   >>> resultPdfFile.close()
   >>> minutesFile.close()
```

Here we use `getPage(0)` to select the first page of the PDF ❶, and then we call `rotateClockwise(90)` on that page ❷. We write a new PDF with the rotated page and save it as *rotatedPage.pdf* ❸.

The resulting PDF will have one page, rotated 90 degrees clockwise, as in Figure 13-2. The return values from `rotateClockwise()` and `rotateCounterClockwise()` contain a lot of information that you can ignore.


rotatedPage.pdf - SumatraPDF
File View Go To Zoom Favorites Settings Help
Page: 1 / 1 Find:
STATE OF LOUISIANA
UNION JUSTICE CONFIDENCE
B.E.S.E.

BESE

Charles E. "Chas" Roemer, President

BOARD of ELEMENTARY and SECONDARY EDUCATION

The Board of Elementary and Secondary Education shall provide leadership and create policies for education that expand opportunities for children, empower families and communities, and advance Louisiana in an increasingly competitive global market.

OFFICIAL BOARD MINUTES

Meeting of March 7, 2014

Figure 13-2. The *rotatedPage.pdf* file with the page rotated 90 degrees clockwise

Overlaying Pages

PyPDF2 can also overlay the contents of one page over another, which is useful for adding a logo, timestamp, or watermark to a page. With

Python, it’s easy to add watermarks to multiple files and only to pages your program specifies.

Download *watermark.pdf* from http://nostarch.com/automatestuff/ and place the PDF in the current working directory along with *meetingminutes.pdf*. Then enter the following into the interactive shell:

```
  >>> import PyPDF2
  >>> minutesFile = open('meetingminutes.pdf', 'rb')
❷ >>> pdfReader = PyPDF2.PdfFileReader(minutesFile)
❷ >>> minutesFirstPage = pdfReader.getPage(0)
❸ >>> pdfWatermarkReader = PyPDF2.PdfFileReader(open('watermark.pdf', 'rb'))
❹ >>> minutesFirstPage.mergePage(pdfWatermarkReader.getPage(0))
❺ >>> pdfWriter = PyPDF2.PdfFileWriter()
❻ >>> pdfWriter.addPage(minutesFirstPage)

❼ >>> for pageNum in range(1, pdfReader.numPages):
          pageObj = pdfReader.getPage(pageNum)
          pdfWriter.addPage(pageObj)
  >>> resultPdfFile = open('watermarkedCover.pdf', 'wb')
  >>> pdfWriter.write(resultPdfFile)
  >>> minutesFile.close()
  >>> resultPdfFile.close()
```

Here we make a `PdfFileReader` object of *meetingminutes.pdf* ❶. We call `getPage(0)` to get a `Page` object for the first page and store this object in `minutesFirstPage` ❷. We then make a `PdfFileReader` object for *watermark.pdf* ❸ and call `mergePage()` on `minutesFirstPage` ❹. The argument we pass to `mergePage()` is a `Page` object for the first page of *watermark.pdf*.

Now that we’ve called `mergePage()` on `minutesFirstPage`, `minutesFirstPage` represents the watermarked first page. We make a `PdfFileWriter` object ❺ and add the watermarked first page ❻. Then we loop through the rest of the pages in *meetingminutes.pdf* and add them to the `PdfFileWriter` object ❼. Finally, we open a new PDF called *watermarkedCover.pdf* and write the contents of the PdfFileWriter to the new PDF.

Figure 13-3 shows the results. Our new PDF, *watermarkedCover.pdf*, has all the contents of the *meetingminutes.pdf*, and the first page is watermarked.

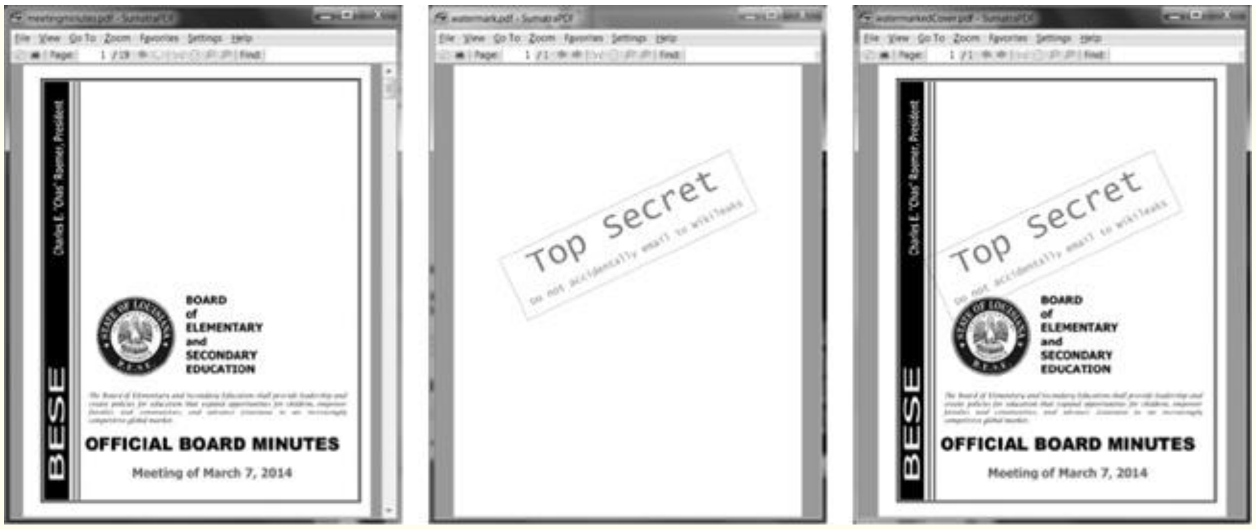

BESE
BOARD of ELEMENTARY and SECONDARY EDUCATION
OFFICIAL BOARD MINUTES
Meeting of March 7, 2014
Top Secret
Do not accidentally email to wikileaks

Figure 13-3. The original PDF (left), the watermark PDF (center), and the merged PDF (right)

Encrypting PDFs

A `PdfFileWriter` object can also add encryption to a PDF document. Enter the following into the interactive shell:

```
   >>> import PyPDF2
   >>> pdfFile = open('meetingminutes.pdf', 'rb')
   >>> pdfReader = PyPDF2.PdfFileReader(pdfFile)
   >>> pdfWriter = PyPDF2.PdfFileWriter()
   >>> for pageNum in range(pdfReader.numPages):
           pdfWriter.addPage(pdfReader.getPage(pageNum))

❶ >>> pdfWriter.encrypt('swordfish')
   >>> resultPdf = open('encryptedminutes.pdf', 'wb')
   >>> pdfWriter.write(resultPdf)
   >>> resultPdf.close()
```

Before calling the `write()` method to save to a file, call the `encrypt()` method and pass it a password string ❶. PDFs can have a *user password* (allowing you to view the PDF) and an *owner password* (allowing you to set permissions for printing, commenting, extracting text, and other features). The user password and owner password are the first and second arguments to `encrypt()`, respectively. If only one string argument is passed to `encrypt()`, it will be used for both passwords.

In this example, we copied the pages of *meetingminutes.pdf* to a `PdfFileWriter` object. We encrypted the PdfFileWriter with the password *swordfish*, opened a new PDF called *encryptedminutes.pdf*, and wrote the contents of the PdfFileWriter to the new PDF. Before anyone can view *encryptedminutes.pdf*, they'll have to enter this password. You may want to delete the original, unencrypted *meetingminutes.pdf* file after ensuring its copy was correctly encrypted.

Project: Combining Select Pages from Many PDFs

Say you have the boring job of merging several dozen PDF documents into a single PDF file. Each of them has a cover sheet as the first page, but you don't want the cover sheet repeated in the final result. Even though there are lots of free programs for combining PDFs, many of them simply merge entire files together. Let's write a Python program to customize which pages you want in the combined PDF.

At a high level, here's what the program will do:

- Find all PDF files in the current working directory.
- Sort the filenames so the PDFs are added in order.
- Write each page, excluding the first page, of each PDF to the output file.

  In terms of implementation, your code will need to do the following:

- Call `os.listdir()` to find all the files in the working directory and remove any non-PDF files.
- Call Python's `sort()` list method to alphabetize the filenames.
- Create a `PdfFileWriter` object for the output PDF.
- Loop over each PDF file, creating a `PdfFileReader` object for it.
- Loop over each page (except the first) in each PDF file.
- Add the pages to the output PDF.
- Write the output PDF to a file named *allminutes.pdf*.

For this project, open a new file editor window and save it as *combinePdfs.py*.

Step 1: Find All PDF Files

First, your program needs to get a list of all files with the *.pdf* extension in the current working directory and sort them. Make your code look like the following:

```
   #! python3
   # combinePdfs.py - Combines all the PDFs in the current working directory
into
   # into a single PDF.

❶ import PyPDF2, os

   # Get all the PDF filenames.
   pdfFiles = []
   for filename in os.listdir('.'):
       if filename.endswith('.pdf'):
❷          pdfFiles.append(filename)
❸ pdfFiles.sort(key=str.lower)

❹ pdfWriter = PyPDF2.PdfFileWriter()

   # TODO: Loop through all the PDF files.

   # TODO: Loop through all the pages (except the first) and add them.

   # TODO: Save the resulting PDF to a file.
```

After the shebang line and the descriptive comment about what the program does, this code imports the `os` and `PyPDF2` modules ❶. The `os.listdir('.')` call will return a list of every file in the current working directory. The code loops over this list and adds only those files with the *.pdf* extension to `pdfFiles` ❷. Afterward, this list is sorted in alphabetical order with the `key=str.lower` keyword argument to `sort()` ❸.

A `PdfFileWriter` object is created to hold the combined PDF pages ❹. Finally, a few comments outline the rest of the program.

Step 2: Open Each PDF

Now the program must read each PDF file in `pdfFiles`. Add the following to your program:

```
#! python3
# combinePdfs.py - Combines all the PDFs in the current working directory
into
# a single PDF.
```

```
import PyPDF2, os

# Get all the PDF filenames.
pdfFiles = []
--snip--
# Loop through all the PDF files.
for filename in pdfFiles:
    pdfFileObj = open(filename, 'rb')
    pdfReader = PyPDF2.PdfFileReader(pdfFileObj)
    # TODO: Loop through all the pages (except the first) and add them.

# TODO: Save the resulting PDF to a file.
```

For each PDF, the loop opens a filename in read-binary mode by calling `open()` with `'rb'` as the second argument. The `open()` call returns a `File` object, which gets passed to `PyPDF2.PdfFileReader()` to create a `PdfFileReader` object for that PDF file.

STEP 3: ADD EACH PAGE

For each PDF, you'll want to loop over every page except the first. Add this code to your program:

```
   #! python3
   # combinePdfs.py - Combines all the PDFs in the current working directory
into
   # a single PDF.

   import PyPDF2, os

   --snip--

   # Loop through all the PDF files.
   for filename in pdfFiles:
   --snip--
       # Loop through all the pages (except the first) and add them.
❶      for pageNum in range(1, pdfReader.numPages):
           pageObj = pdfReader.getPage(pageNum)
           pdfWriter.addPage(pageObj)

   # TODO: Save the resulting PDF to a file.
```

The code inside the `for` loop copies each `Page` object individually to the `PdfFileWriter` object. Remember, you want to skip the first page. Since PyPDF2 considers `0` to be the first page, your loop should start at `1` ❶ and then go up to, but not include, the integer in `pdfReader.numPages`.

STEP 4: SAVE THE RESULTS

After these nested `for` loops are done looping, the `pdfWriter` variable will contain a `PdfFileWriter` object with the pages for all the PDFs combined. The last step is to write this content to a file on the hard drive. Add this code to your program:

```
#! python3
# combinePdfs.py - Combines all the PDFs in the current working directory
into
# a single PDF.
import PyPDF2, os

--snip--

# Loop through all the PDF files.
for filename in pdfFiles:
--snip--
    # Loop through all the pages (except the first) and add them.
    for pageNum in range(1, pdfReader.numPages):
    --snip--

# Save the resulting PDF to a file.
pdfOutput = open('allminutes.pdf', 'wb')
pdfWriter.write(pdfOutput)
pdfOutput.close()
```

Passing `'wb'` to `open()` opens the output PDF file, *allminutes.pdf*, in write-binary mode. Then, passing the resulting `File` object to the `write()` method creates the actual PDF file. A call to the `close()` method finishes the program.

IDEAS FOR SIMILAR PROGRAMS

Being able to create PDFs from the pages of other PDFs will let you make programs that can do the following:

- Cut out specific pages from PDFs.
- Reorder pages in a PDF.
- Create a PDF from only those pages that have some specific text, identified by `extractText()`.

WORD DOCUMENTS

Python can create and modify Word documents, which have the *.docx* file extension, with the `python-docx` module. You can install the module by running `pip install python-docx`. (Appendix A has full details on installing third-party modules.)

NOTE

When using pip to first install Python-Docx, be sure to install `python-docx`*, not* `docx`*. The installation name* `docx` *is for a different module that this book does not cover. However, when you are going to import the* `python-docx` *module, you'll need to run* `import docx`*, not* `import python-docx`*.*

If you don't have Word, LibreOffice Writer and OpenOffice Writer are both free alternative applications for Windows, OS X, and Linux that can be used to open *.docx* files. You can download them from *https://www.libreoffice.org* and *http://openoffice.org*, respectively. The full documentation for Python-Docx is available at *https://python-docx.readthedocs.org/*. Although there is a version of Word for OS X, this chapter will focus on Word for Windows.

Compared to plaintext, *.docx* files have a lot of structure. This structure is represented by three different data types in Python-Docx. At the highest level, a `Document` object represents the entire document. The `Document` object contains a list of `Paragraph` objects for the paragraphs in the document. (A new paragraph begins whenever the user presses ENTER or RETURN while typing in a Word document.) Each of these `Paragraph` objects contains a list of one or more `Run` objects. The single-sentence paragraph in Figure 13-4 has four runs.

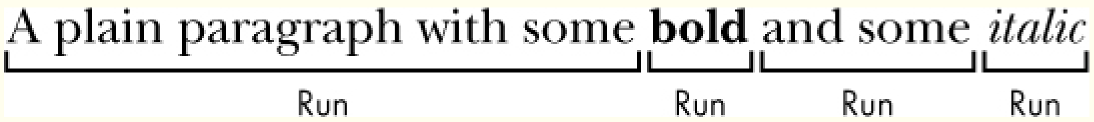


Figure 13-4. The `Run` objects identified in a `Paragraph` object

The text in a Word document is more than just a string. It has font, size, color, and other styling information associated with it. A *style* in Word is a collection of these attributes. A `Run` object is a contiguous run of text with the same style. A new `Run` object is needed whenever the text style changes.

READING WORD DOCUMENTS

Let's experiment with the `python-docx` module. Download *demo.docx* from *http://nostarch.com/automatestuff/* and save the document to the working directory. Then enter the following into the interactive shell:

```
  >>> import docx
❶ >>> doc = docx.Document('demo.docx')
❷ >>> len(doc.paragraphs)
  7
❸ >>> doc.paragraphs[0].text
  'Document Title'
❹ >>> doc.paragraphs[1].text
  'A plain paragraph with some bold and some italic'
❺ >>> len(doc.paragraphs[1].runs)
  4
❻ >>> doc.paragraphs[1].runs[0].text
  'A plain paragraph with some '
❼ >>> doc.paragraphs[1].runs[1].text
  'bold'
❽ >>> doc.paragraphs[1].runs[2].text
  ' and some '
❾ >>> doc.paragraphs[1].runs[3].text
  'italic'
```

At ❶, we open a *.docx* file in Python, call `docx.Document()`, and pass the filename *demo.docx*. This will return a `Document` object, which has a `paragraphs` attribute that is a list of `Paragraph` objects. When we call `len()` on `doc.paragraphs`, it returns `7`, which tells us that there are seven `Paragraph` objects in this document ❷. Each of these `Paragraph` objects has a `text` attribute that contains a string of the text in that paragraph (without the style information). Here, the first `text` attribute contains `'DocumentTitle'` ❸, and the second contains `'A plain paragraph with some bold and some italic'` ❹.

Each `Paragraph` object also has a `runs` attribute that is a list of `Run` objects. `Run` objects also have a `text` attribute, containing just the text in that particular run. Let's look at the `text` attributes in the second `Paragraph` object, `'A plain paragraph with some bold and some italic'`. Calling `len()` on this `Paragraph` object tells us that there are four `Run` objects ❺. The first run object contains `'A plain paragraph with some '` ❻. Then, the text change to a bold style, so `'bold'` starts a new `Run` object ❼. The text returns to an unbolded style after that, which results in a third `Run` object, `' and some '` ❽. Finally, the fourth and last `Run` object contains `'italic'` in an italic style ❾.

With Python-Docx, your Python programs will now be able to read the text from a *.docx* file and use it just like any other string value.

GETTING THE FULL TEXT FROM A .DOCX FILE

If you care only about the text, not the styling information, in the Word document, you can use the `getText()` function. It accepts a filename of a *.docx* file and returns a single string value of its text. Open a new file editor window and enter the following code, saving it as *readDocx.py*:

```
#! python3

import docx

def getText(filename):
    doc = docx.Document(filename)
    fullText = []
    for para in doc.paragraphs:
        fullText.append(para.text)
    return '\n'.join(fullText)
```

The `getText()` function opens the Word document, loops over all the `Paragraph` objects in the `paragraphs` list, and then appends their text to the list in `fullText`. After the loop, the strings in `fullText` are joined together with newline characters.

The *readDocx.py* program can be imported like any other module. Now if you just need the text from a Word document, you can enter the following:

```
>>> import readDocx
>>> print(readDocx.getText('demo.docx'))
Document Title
A plain paragraph with some bold and some italic
Heading, level 1
Intense quote
first item in unordered list
first item in ordered list
```

You can also adjust `getText()` to modify the string before returning it. For example, to indent each paragraph, replace the `append()` call in *readDocx.py* with this:

```
fullText.append(' ' + para.text)
```

To add a double space in between paragraphs, change the `join()` call code to this:

```
return '\n\n'.join(fullText)
```

As you can see, it takes only a few lines of code to write functions that will read a *.docx* file and return a string of its content to your liking.

STYLING PARAGRAPH AND RUN OBJECTS

In Word for Windows, you can see the styles by pressing CTRL-ALT-SHIFT-S to display the Styles pane, which looks like Figure 13-5. On OS X, you can view the Styles pane by clicking the **View ▸ Styles** menu item.

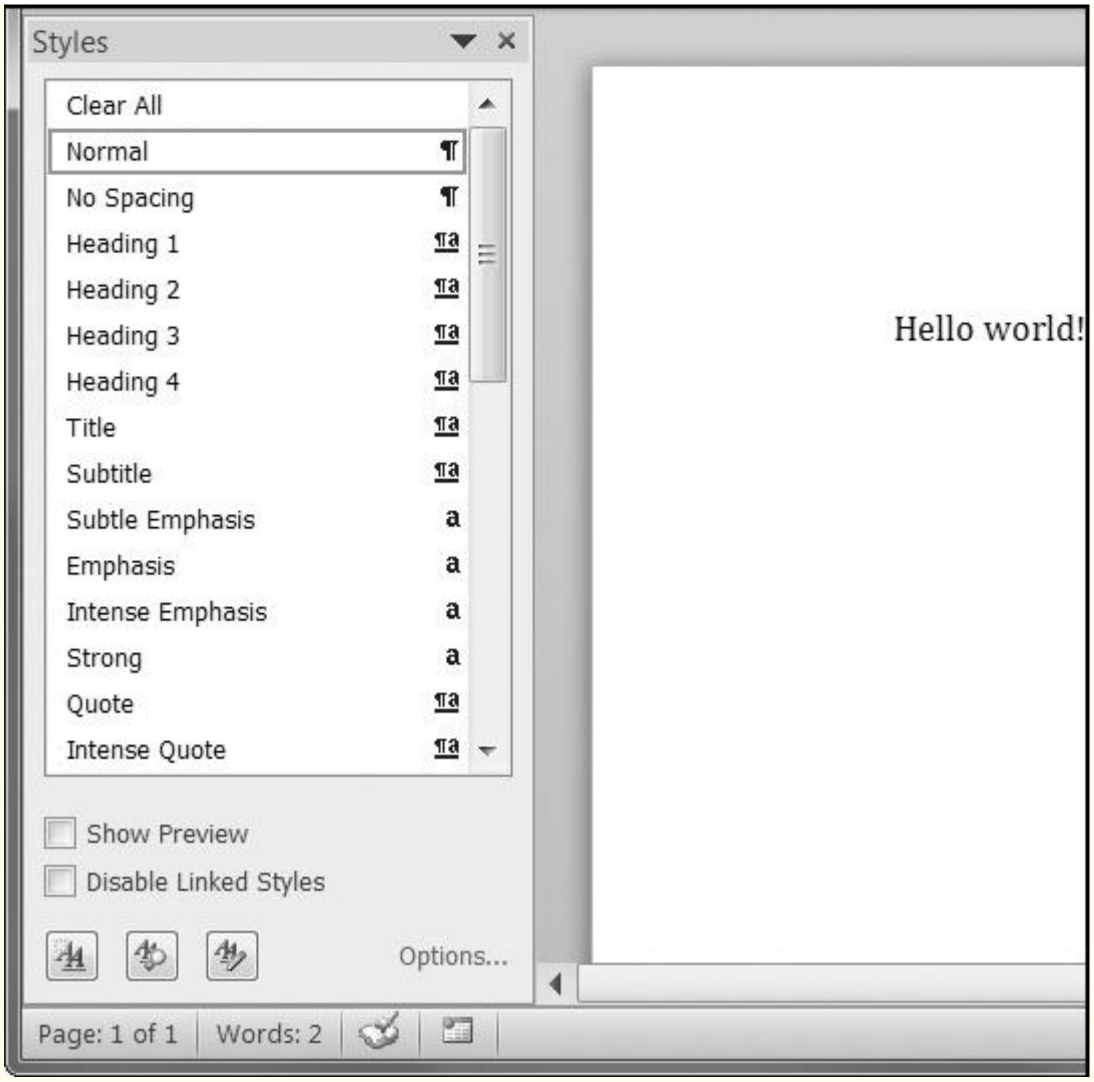


Figure 13-5. Display the Styles pane by pressing `CTRL-ALT-SHIFT`-S on Windows.

Word and other word processors use styles to keep the visual presentation of similar types of text consistent and easy to change. For example, perhaps you want to set body paragraphs in 11-point, Times New Roman, left-justified, ragged-right text. You can create a style with these settings and assign it to all body paragraphs. Then, if you later want to change the presentation of all body paragraphs in the document, you can just change the style, and all those paragraphs will be automatically updated.

For Word documents, there are three types of styles: *Paragraph styles* can be applied to `Paragraph` objects, *character styles* can be applied to `Run` objects, and *linked styles* can be applied to both kinds of objects. You can give both `Paragraph` and `Run` objects styles by setting their `style` attribute to a string. This string should be the name of a style. If `style` is set to `None`, then there will be no style associated with the `Paragraph` or `Run` object.

The string values for the default Word styles are as follows:

```
'Normal'    'Heading5'     'ListBullet'     'ListParagraph'
'BodyText'  'Heading6'     'ListBullet2'    'MacroText'
'BodyText2' 'Heading7'     'ListBullet3'    'NoSpacing'
'BodyText3' 'Heading8'     'ListContinue'   'Quote'
'Caption'   'Heading9'     'ListContinue2'  'Subtitle'
'Heading1'  'IntenseQuote' 'ListContinue3'  'TOCHeading'
'Heading2'  'List'         'ListNumber'     'Title'
'Heading3'  'List2'        'ListNumber2'

'Heading4'  'List3'        'ListNumber3'
```

When setting the `style` attribute, do not use spaces in the style name. For example, while the style name may be Subtle Emphasis, you should set the `style` attribute to the string value `'SubtleEmphasis'` instead of `'Subtle Emphasis'`. Including spaces will cause Word to misread the style name and not apply it.

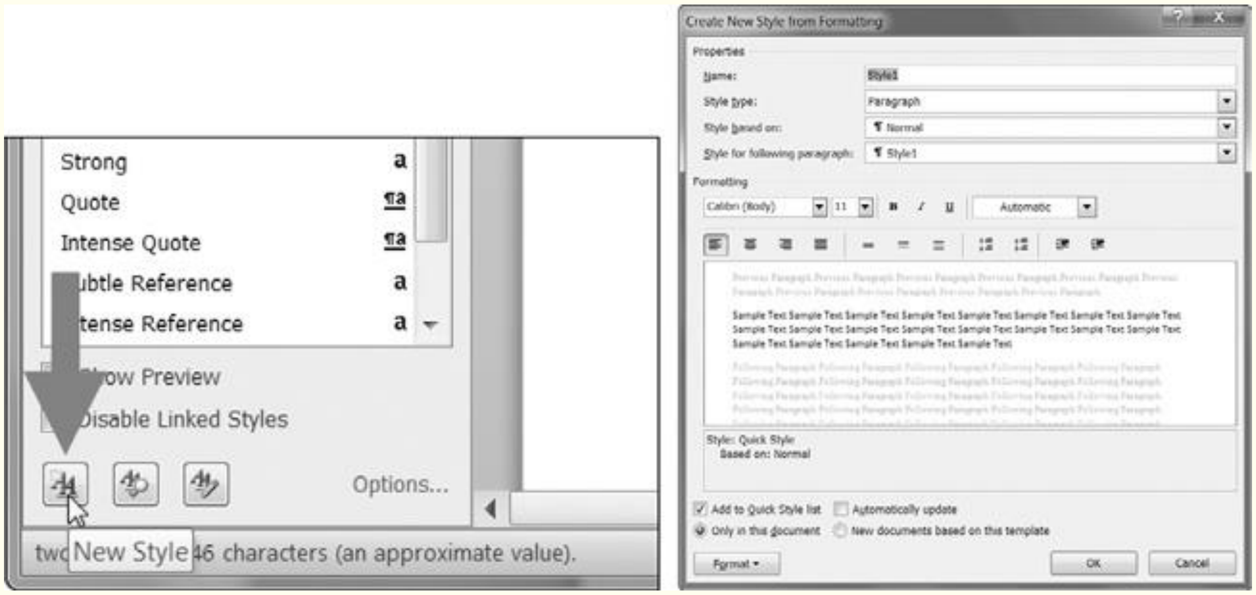
When using a linked style for a `Run` object, you will need to add `'Char'` to the end of its name. For example, to set the Quote linked style for a `Paragraph` object, you would use `paragraphObj.style = 'Quote'`, but for a `Run` object, you would use `runObj.style = 'QuoteChar'`.

In the current version of Python-Docx (0.7.4), the only styles that can be used are the default Word styles and the styles in the opened *.docx*. New styles cannot be created—though this may change in future versions of Python-Docx.

Creating Word Documents with Nondefault Styles

If you want to create Word documents that use styles beyond the default ones, you will need to open Word to a blank Word document and create the styles yourself by clicking the **New Style** button at the bottom of the Styles pane (Figure 13-6 shows this on Windows).

This will open the Create New Style from Formatting dialog, where you can enter the new style. Then, go back into the interactive shell and open this blank Word document with `docx.Document()`, using it as the base for your Word document. The name you gave this style will now be available to use with Python-Docx.



Name: Style1
Style type: Paragraph
Style based on: ¶ Normal
Style for following paragraph: ¶ Style1
Calibri (Body) 11 Automatic

Sample Text Sample Text

Style: Quick Style
Based on: Normal

[✓] Add to Quick Style list [ ] Automatically update
[✓ Only in this document] New documents based on this template

Figure 13-6. The New Style button (left) and the Create New Style from Formatting dialog (right)

Run Attributes

Runs can be further styled using `text` attributes. Each attribute can be set to one of three values: `True` (the attribute is always enabled, no matter what other styles are applied to the run), `False` (the attribute is always disabled), or `None` (defaults to whatever the run's style is set to).

Table 13-1 lists the `text` attributes that can be set on `Run` objects.

Table 13-1. `Run` Object `text` Attributes

Attribute	Description
`bold`	The text appears in bold.

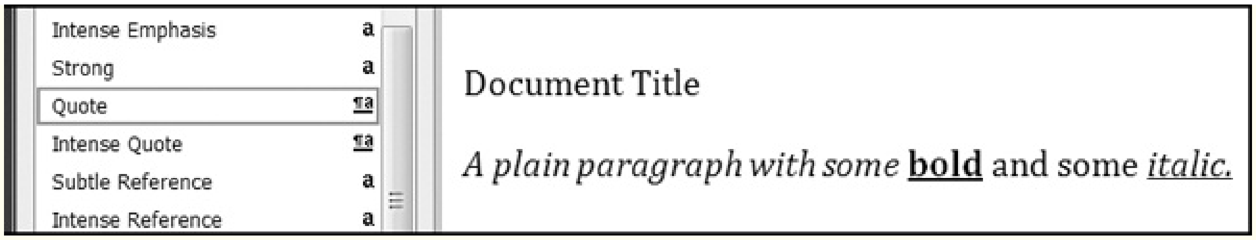
Attribute	Description
`italic`	The text appears in italic.
`underline`	The text is underlined.
`strike`	The text appears with strikethrough.
`double_strike`	The text appears with double strikethrough.
`all_caps`	The text appears in capital letters.
`small_caps`	The text appears in capital letters, with lowercase letters two points smaller.
`shadow`	The text appears with a shadow.
`outline`	The text appears outlined rather than solid.
`rtl`	The text is written right-to-left.
`imprint`	The text appears pressed into the page.
`emboss`	The text appears raised off the page in relief.

For example, to change the styles of *demo.docx*, enter the following into the interactive shell:

```
>>> doc = docx.Document('demo.docx')
>>> doc.paragraphs[0].text
'Document Title'
>>> doc.paragraphs[0].style
'Title'
>>> doc.paragraphs[0].style = 'Normal'
>>> doc.paragraphs[1].text
'A plain paragraph with some bold and some italic'
>>> (doc.paragraphs[1].runs[0].text, doc.paragraphs[1].runs[1].text, doc.
paragraphs[1].runs[2].text, doc.paragraphs[1].runs[3].text)
('A plain paragraph with some ', 'bold', ' and some ', 'italic')
>>> doc.paragraphs[1].runs[0].style = 'QuoteChar'
>>> doc.paragraphs[1].runs[1].underline = True
>>> doc.paragraphs[1].runs[3].underline = True
>>> doc.save('restyled.docx')
```

Here, we use the `text` and `style` attributes to easily see what's in the paragraphs in our document. We can see that it's simple to divide a paragraph into runs and access each run individiaully. So we get the first, second, and fourth runs in the second paragraph, style each run, and save the results to a new document.

The words *Document Title* at the top of *restyled.docx* will have the Normal style instead of the Title style, the `Run` object for the text *A plain paragraph with some* will have the QuoteChar style, and the two `Run` objects for the words *bold* and *italic* will have their `underline` attributes set to `True`. Figure 13-7 shows how the styles of paragraphs and runs look in *restyled.docx*.



Document Title

A plain paragraph with some **bold** and some *italic.*

Figure 13-7. The *restyled.docx* file

You can find more complete documentation on Python-Docx's use of styles at https://python-docx.readthedocs.org/en/latest/user/styles.html.

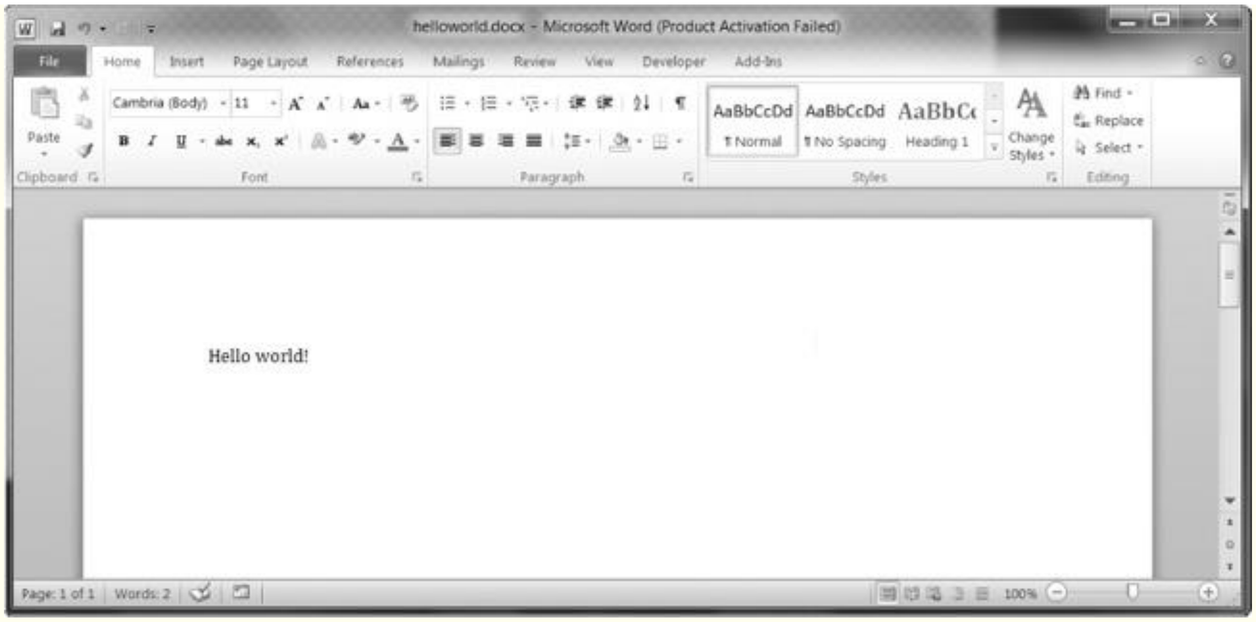
WRITING WORD DOCUMENTS

Enter the following into the interactive shell:

```
>>> import docx
>>> doc = docx.Document()
>>> doc.add_paragraph('Hello world!')
<docx.text.Paragraph object at 0x0000000003B56F60>
>>> doc.save('helloworld.docx')
```

To create your own *.docx* file, call `docx.Document()` to return a new, blank Word `Document` object. The `add_paragraph()` document method adds a new paragraph of text to the document and returns a reference to the `Paragraph` object that was added. When you're done adding text, pass a filename string to the `save()` document method to save the `Document` object to a file.

This will create a file named *helloworld.docx* in the current working directory that, when opened, looks like Figure 13-8.



Hello world!

Figure 13-8. The Word document created using `add_paragraph('Hello world!')`

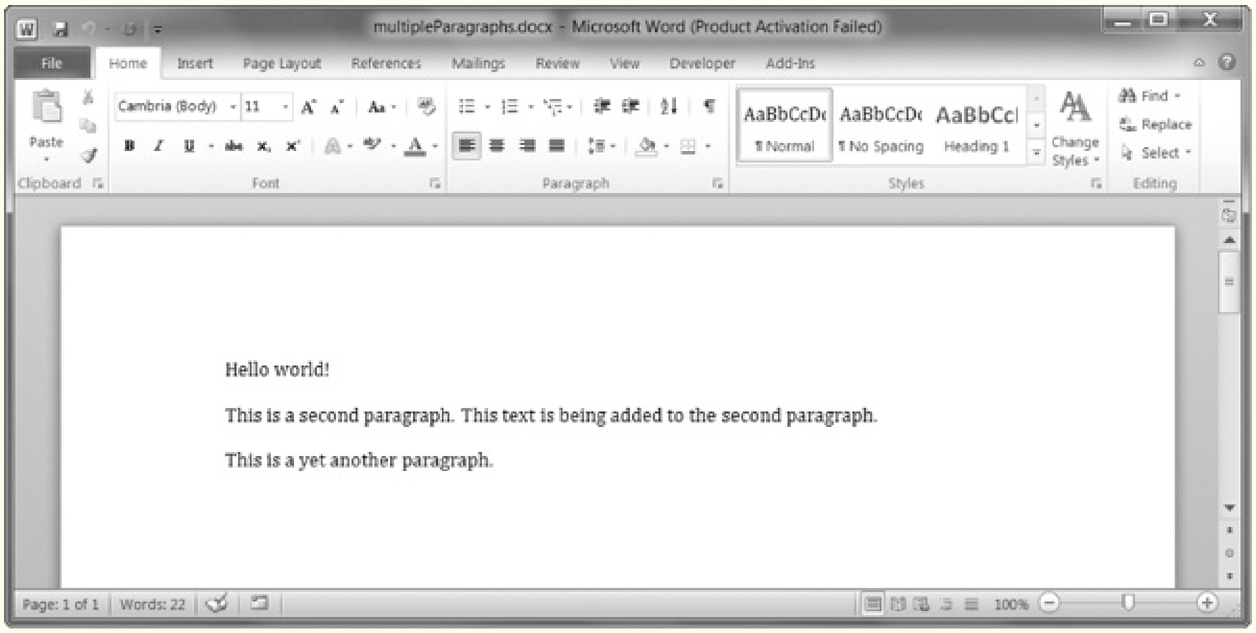
You can add paragraphs by calling the `add_paragraph()` method again with the new paragraph's text. Or to add text to the end of an existing paragraph, you can call the paragraph's `add_run()` method and pass it a string. Enter the following into the interactive shell:

```
>>> import docx
>>> doc = docx.Document()
>>> doc.add_paragraph('Hello world!')
<docx.text.Paragraph object at 0x000000000366AD30>
>>> paraObj1 = doc.add_paragraph('This is a second paragraph.')
>>> paraObj2 = doc.add_paragraph('This is a yet another paragraph.')
>>> paraObj1.add_run(' This text is being added to the second paragraph.')
<docx.text.Run object at 0x0000000003A2C860>
>>> doc.save('multipleParagraphs.docx')
```

The resulting document will look like Figure 13-9. Note that the text *This text is being added to the second paragraph.* was added to the `Paragraph` object in `paraObj1`, which was the second paragraph added to `doc`. The `add_paragraph()` and `add_run()` functions return paragraph and `Run` objects, respectively, to save you the trouble of extracting them as a separate step.

Keep in mind that as of Python-Docx version 0.5.3, new `Paragraph` objects can be added only to the end of the document, and new `Run` objects can be added only to the end of a `Paragraph` object.

The `save()` method can be called again to save the additional changes you've made.



Hello world!

This is a second paragraph. This text is being added to the second paragraph.

This is a yet another paragraph.

Figure 13-9. The document with multiple `Paragraph` and `Run` objects added

Both `add_paragraph()` and `add_run()` accept an optional second argument that is a string of the `Paragraph` or `Run` object's style. For example:

```
>>> doc.add_paragraph('Hello world!', 'Title')
```

This line adds a paragraph with the text *Hello world!* in the Title style.

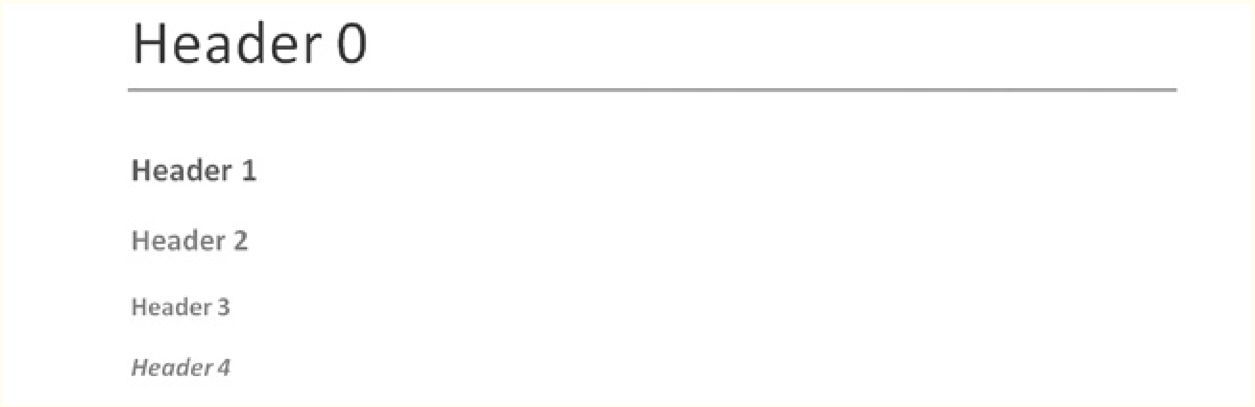
ADDING HEADINGS

Calling `add_heading()` adds a paragraph with one of the heading styles. Enter the following into the interactive shell:

```
>>> doc = docx.Document()
>>> doc.add_heading('Header 0', 0)
<docx.text.Paragraph object at 0x00000000036CB3C8>
>>> doc.add_heading('Header 1', 1)
<docx.text.Paragraph object at 0x00000000036CB630>
>>> doc.add_heading('Header 2', 2)
<docx.text.Paragraph object at 0x00000000036CB828>
>>> doc.add_heading('Header 3', 3)
<docx.text.Paragraph object at 0x00000000036CB2E8>
>>> doc.add_heading('Header 4', 4)
<docx.text.Paragraph object at 0x00000000036CB3C8>
>>> doc.save('headings.docx')
```

The arguments to `add_heading()` are a string of the heading text and an integer from `0` to `4`. The integer `0` makes the heading the Title style, which is used for the top of the document. Integers `1` to `4` are for various heading levels, with `1` being the main heading and `4` the lowest subheading. The `add_heading()` function returns a `Paragraph` object to save you the step of extracting it from the `Document` object as a separate step.

The resulting *headings.docx* file will look like Figure 13-10.



Header 0

Header 1

Header 2

Header 3

Header 4

Figure 13-10. The *headings.docx* document with headings 0 to 4

ADDING LINE AND PAGE BREAKS

To add a line break (rather than starting a whole new paragraph), you can call the `add_break()` method on the `Run` object you want to have the break appear after. If you want to add a page break instead, you need to pass the value `docx.text.WD_BREAK.PAGE` as a lone argument to `add_break()`, as is done in the middle of the following example:

```
   >>> doc = docx.Document()
   >>> doc.add_paragraph('This is on the first page!')
   <docx.text.Paragraph object at 0x0000000003785518>
❶ >>> doc.paragraphs[0].runs[0].add_break(docx.text.WD_BREAK.PAGE)
   >>> doc.add_paragraph('This is on the second page!')
   <docx.text.Paragraph object at 0x00000000037855F8>
   >>> doc.save('twoPage.docx')
```

This creates a two-page Word document with *This is on the first page!* on the first page and *This is on the second page!* on the second. Even though there was still plenty of space on the first page after the text *This is on the first page!*, we forced the next paragraph to begin on a new page by inserting a page break after the first run of the first paragraph ❶.

ADDING PICTURES

`Document` objects have an `add_picture()` method that will let you add an image to the end of the document. Say you have a file *zophie.png* in the current working directory. You can add *zophie.png* to the end of your document with a width of 1 inch and height of 4 centimeters (Word can use both imperial and metric units) by entering the following:

```
>>> doc.add_picture('zophie.png', width=docx.shared.Inches(1),
height=docx.shared.Cm(4))
<docx.shape.InlineShape object at 0x00000000036C7D30>
```

The first argument is a string of the image's filename. The optional `width` and `height` keyword arguments will set the width and height of the image in the document. If left out, the width and height will default to the normal size of the image.

You'll probably prefer to specify an image's height and width in familiar units such as inches and centimeters, so you can use the `docx.shared.Inches()` and `docx.shared.Cm()` functions when you're specifying the `width` and `height` keyword arguments.

SUMMARY

Text information isn't just for plaintext files; in fact, it's pretty likely that you deal with PDFs and Word documents much more often. You can use the `PyPDF2` module to read and write PDF documents. Unfortunately, reading text from PDF documents might not always result in a perfect translation to a string because of the complicated PDF file format, and some PDFs might not be readable at all. In these cases, you're out of luck unless future updates to PyPDF2 support additional PDF features.

Word documents are more reliable, and you can read them with the `python-docx` module. You can manipulate text in Word documents via `Paragraph` and `Run` objects. These objects can also be given styles, though they must be from the default set of styles or styles already in the document. You can add new paragraphs, headings, breaks, and pictures to the document, though only to the end.

Many of the limitations that come with working with PDFs and Word documents are because these formats are meant to be nicely displayed for human readers, rather than easy to parse by software. The next chapter takes a look at two other common formats for storing information: JSON and CSV files. These formats are designed to be used by computers, and you'll see that Python can work with these formats much more easily.

WORKING WITH CSV FILES AND JSON DATA

CSV stands for “comma-separated values,” and CSV files are simplified spreadsheets stored as plaintext files. Python’s `csv` module makes it easy to parse CSV files.

JSON (pronounced “JAY-sawn” or “Jason”—it doesn’t matter how because either way people will say you’re pronouncing it wrong) is a format that stores information as JavaScript source code in plaintext files.

(JSON is short for JavaScript Object Notation.) You don’t need to know the JavaScript programming language to use JSON files, but the JSON format is useful to know because it’s used in many web applications.

THE CSV MODULE

Each line in a CSV file represents a row in the spreadsheet, and commas separate the cells in the row. For example, the spreadsheet *example.xlsx* from http://nostarch.com/automatestuff/ would look like this in a CSV file:

```
4/5/2015 13:34,Apples,73
4/5/2015 3:41,Cherries,85
4/6/2015 12:46,Pears,14
4/8/2015 8:59,Oranges,52
4/10/2015 2:07,Apples,152
4/10/2015 18:10,Bananas,23
4/10/2015 2:40,Strawberries,98
```

I will use this file for this chapter’s interactive shell examples. You can download *example.csv* from http://nostarch.com/automatestuff/ or enter the text into a text editor and save it as *example.csv*.

CSV files are simple, lacking many of the features of an Excel spreadsheet. For example, CSV files

- Don’t have types for their values—everything is a string
- Don’t have settings for font size or color
- Don’t have multiple worksheets
- Can’t specify cell widths and heights
- Can’t have merged cells

- Can’t have images or charts embedded in them

The advantage of CSV files is simplicity. CSV files are widely supported by many types of programs, can be viewed in text editors (including IDLE’s file editor), and are a straightforward way to represent spreadsheet data. The CSV format is exactly as advertised: It’s just a text file of comma-separated values.

Since CSV files are just text files, you might be tempted to read them in as a string and then process that string using the techniques you learned in Chapter 8. For example, since each cell in a CSV file is separated by a comma, maybe you could just call the `split()` method on each line of text to get the values. But not every comma in a CSV file represents the boundary between two cells. CSV files also have their own set of escape characters to allow commas and other characters to be included *as part of the values*. The `split()` method doesn’t handle these escape characters. Because of these potential pitfalls, you should always use the `csv` module for reading and writing CSV files.

Reader Objects

To read data from a CSV file with the `csv` module, you need to create a `Reader` object. A `Reader` object lets you iterate over lines in the CSV file. Enter the following into the interactive shell, with *example.csv* in the current working directory:

```
❶ >>> import csv
❷ >>> exampleFile = open('example.csv')
❸ >>> exampleReader = csv.reader(exampleFile)
❹ >>> exampleData = list(exampleReader)
❹ >>> exampleData
   [['4/5/2015 13:34', 'Apples', '73'], ['4/5/2015 3:41', 'Cherries', '85'],
   ['4/6/2015 12:46', 'Pears', '14'], ['4/8/2015 8:59', 'Oranges', '52'],
   ['4/10/2015 2:07', 'Apples', '152'], ['4/10/2015 18:10', 'Bananas', '23'],
   ['4/10/2015 2:40', 'Strawberries', '98']]
```

The `csv` module comes with Python, so we can import it ❶ without having to install it first.

To read a CSV file with the `csv` module, first open it using the `open()` function ❷, just as you would any other text file. But instead of calling the `read()` or `readlines()` method on the `File` object that `open()` returns, pass it to the `csv.reader()` function ❸. This will return a `Reader` object for you to use. Note that you don't pass a filename string directly to the `csv.reader()` function.

The most direct way to access the values in the `Reader` object is to convert it to a plain Python list by passing it to `list()` ❹. Using `list()` on this `Reader` object returns a list of lists, which you can store in a variable like `exampleData`. Entering `exampleData` in the shell displays the list of lists ❺.

Now that you have the CSV file as a list of lists, you can access the value at a particular row and column with the expression `exampleData[row][col]`, where `row` is the index of one of the lists in `exampleData`, and `col` is the index of the item you want from that list. Enter the following into the interactive shell:

```
>>> exampleData[0][0]
'4/5/2015 13:34'
>>> exampleData[0][1]
'Apples'
>>> exampleData[0][2]
'73'
>>> exampleData[1][1]
'Cherries'
>>> exampleData[6][1]
'Strawberries'
```

`exampleData[0][0]` goes into the first list and gives us the first string, `exampleData[0][2]` goes into the first list and gives us the third string, and so on.

READING DATA FROM READER OBJECTS IN A FOR LOOP

For large CSV files, you'll want to use the `Reader` object in a `for` loop. This avoids loading the entire file into memory at once. For example, enter the following into the interactive shell:

```
>>> import csv
>>> exampleFile = open('example.csv')
>>> exampleReader = csv.reader(exampleFile)
>>> for row in exampleReader:
        print('Row #' + str(exampleReader.line_num) + ' ' + str(row))

Row #1 ['4/5/2015 13:34', 'Apples', '73']
Row #2 ['4/5/2015 3:41', 'Cherries', '85']
Row #3 ['4/6/2015 12:46', 'Pears', '14']
Row #4 ['4/8/2015 8:59', 'Oranges', '52']
Row #5 ['4/10/2015 2:07', 'Apples', '152']
Row #6 ['4/10/2015 18:10', 'Bananas', '23']
Row #7 ['4/10/2015 2:40', 'Strawberries', '98']
```

After you import the `csv` module and make a `Reader` object from the CSV file, you can loop through the rows in the `Reader` object. Each row is a list of values, with each value representing a cell.

The `print()` function call prints the number of the current row and the contents of the row. To get the row number, use the `Reader` object's `line_num` variable, which contains the number of the current line.

The `Reader` object can be looped over only once. To reread the CSV file, you must call `csv.reader` to create a `Reader` object.

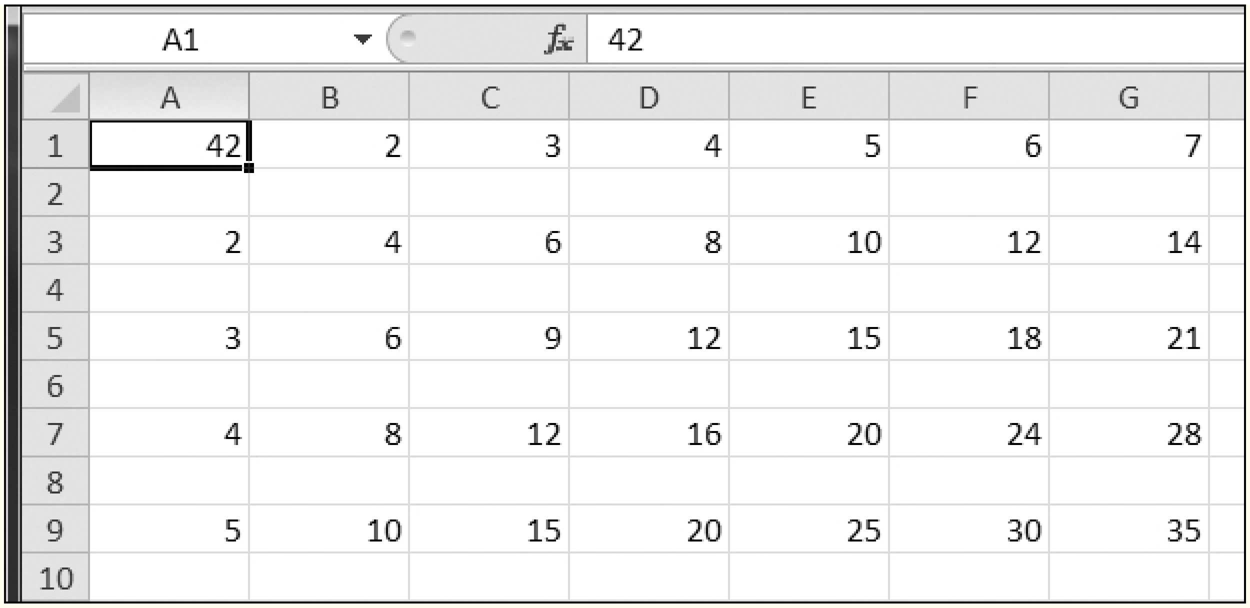
WRITER OBJECTS

A `Writer` object lets you write data to a CSV file. To create a `Writer` object, you use the `csv.writer()` function. Enter the following into the interactive shell:

```
  >>> import csv
❶ >>> outputFile = open('output.csv', 'w', newline='')
❷ >>> outputWriter = csv.writer(outputFile)
  >>> outputWriter.writerow(['spam', 'eggs', 'bacon', 'ham'])
  21
  >>> outputWriter.writerow(['Hello, world!', 'eggs', 'bacon', 'ham'])
  32
  >>> outputWriter.writerow([1, 2, 3.141592, 4])
  16
  >>> outputFile.close()
```

First, call `open()` and pass it `'w'` to open a file in write mode ❶. This will create the object you can then pass to `csv.writer()` ❷ to create a `Writer` object.

On Windows, you'll also need to pass a blank string for the `open()` function's `newline` keyword argument. For technical reasons beyond the scope of this book, if you forget to set the `newline` argument, the rows in *output.csv* will be double-spaced, as shown in Figure 14-1.


A1 42

	A	B	C	D	E	F	G
1	42	2	3	4	5	6	7
2							
3	2	4	6	8	10	12	14
4							
5	3	6	9	12	15	18	21
6							
7	4	8	12	16	20	24	28
8							
9	5	10	15	20	25	30	35
10							

Figure 14-1. If you forget the `newline=''` keyword argument in `open()`, the CSV file will be double-spaced.

The `writerow()` method for `Writer` objects takes a list argument. Each value in the list is placed in its own cell in the output CSV file. The return value of `writerow()` is the number of characters written to the file for that row (including newline characters).

This code produces an *output.csv* file that looks like this:

```
spam,eggs,bacon,ham
"Hello, world!",eggs,bacon,ham
1,2,3.141592,4
```

Notice how the `Writer` object automatically escapes the comma in the value `'Hello, world!'` with double quotes in the CSV file. The `csv` module saves you from having to handle these special cases yourself.

THE DELIMITER AND LINETERMINATOR KEYWORD ARGUMENTS

Say you want to separate cells with a tab character instead of a comma and you want the rows to be double-spaced. You could enter something like the following into the interactive shell:

```
   >>> import csv
   >>> csvFile = open('example.tsv', 'w', newline='')
❶ >>> csvWriter = csv.writer(csvFile, delimiter='\t', lineterminator='\n\n')
```

```
>>> csvWriter.writerow(['apples', 'oranges', 'grapes'])
24
>>> csvWriter.writerow(['eggs', 'bacon', 'ham'])
17
>>> csvWriter.writerow(['spam', 'spam', 'spam', 'spam', 'spam', 'spam'])
32
>>> csvFile.close()
```

This changes the delimiter and line terminator characters in your file. The *delimiter* is the character that appears between cells on a row. By default, the delimiter for a CSV file is a comma. The *line terminator* is the character that comes at the end of a row. By default, the line terminator is a newline. You can change characters to different values by using the `delimiter` and `lineterminator` keyword arguments with `csv.writer()`.

Passing `delimeter='\t'` and `lineterminator='\n\n'` ❶ changes the character between cells to a tab and the character between rows to two newlines. We then call `writerow()` three times to give us three rows.

This produces a file named *example.tsv* with the following contents:

```
apples  oranges grapes

eggs    bacon   ham
spam    spam    spam    spam    spam    spam
```

Now that our cells are separated by tabs, we're using the file extension *.tsv*, for tab-separated values.

PROJECT: REMOVING THE HEADER FROM CSV FILES

Say you have the boring job of removing the first line from several hundred CSV files. Maybe you'll be feeding them into an automated process that requires just the data and not the headers at the top of the columns. You *could* open each file in Excel, delete the first row, and resave the file—but that would take hours. Let's write a program to do it instead.

The program will need to open every file with the *.csv* extension in the current working directory, read in the contents of the CSV file, and rewrite the contents without the first row to a file of the same name. This will replace the old contents of the CSV file with the new, headless contents.

NOTE

As always, whenever you write a program that modifies files, be sure to back up the files, first just in case your program does not work the way you expect it to. You don't want to accidentally erase your original files.

At a high level, the program must do the following:

- Find all the CSV files in the current working directory.
- Read in the full contents of each file.
- Write out the contents, skipping the first line, to a new CSV file.

  At the code level, this means the program will need to do the following:

- Loop over a list of files from `os.listdir()`, skipping the non-CSV files.
- Create a CSV `Reader` object and read in the contents of the file, using the `line_num` attribute to figure out which line to skip.
- Create a CSV `Writer` object and write out the read-in data to the new file.

For this project, open a new file editor window and save it as *removeCsvHeader.py*.

STEP 1: LOOP THROUGH EACH CSV FILE

The first thing your program needs to do is loop over a list of all CSV filenames for the current working directory. Make your *removeCsvHeader.py* look like this:

```
   #! python3
   # removeCsvHeader.py - Removes the header from all CSV files in the
current
   # working directory.

   import csv, os

   os.makedirs('headerRemoved', exist_ok=True)
   # Loop through every file in the current working directory.
   for csvFilename in os.listdir('.'):
       if not csvFilename.endswith('.csv'):
❶          continue    # skip non-csv files

       print('Removing header from ' + csvFilename + '...')

       # TODO: Read the CSV file in (skipping first row).

       # TODO: Write out the CSV file.
```

The `os.makedirs()` call will create a `headerRemoved` folder where all the headless CSV files will be written. A `for` loop on `os.listdir('.')` gets you partway there, but it will loop over *all* files in the working directory, so you'll need to add some code at the start of the loop that skips filenames that don't end with `.csv`. The `continue` statement ❶ makes the `for` loop move on to the next filename when it comes across a non-CSV file.

Just so there's *some* output as the program runs, print out a message saying which CSV file the program is working on. Then, add some `TODO` comments for what the rest of the program should do.

STEP 2: READ IN THE CSV FILE

The program doesn't remove the first line from the CSV file. Rather, it creates a new copy of the CSV file without the first line. Since the copy's filename is the same as the original filename, the copy will overwrite the original.

The program will need a way to track whether it is currently looping on the first row. Add the following to *removeCsvHeader.py*.

```
#! python3
# removeCsvHeader.py - Removes the header from all CSV files in the current
# working directory.

--snip--
# Read the CSV file in (skipping first row).
csvRows = []
csvFileObj = open(csvFilename)
readerObj = csv.reader(csvFileObj)
for row in readerObj:
    if readerObj.line_num == 1:
        continue    # skip first row
    csvRows.append(row)
csvFileObj.close()
```

```
# TODO: Write out the CSV file.
```

The `Reader` object's `line_num` attribute can be used to determine which line in the CSV file it is currently reading. Another `for` loop will loop over the rows returned from the CSV `Reader` object, and all rows but the first will be appended to `csvRows`.

As the `for` loop iterates over each row, the code checks whether `readerObj.line_num` is set to `1`. If so, it executes a `continue` to move on to the next row without appending it to `csvRows`. For every row afterward, the condition will be always be `False`, and the row will be appended to `csvRows`.

STEP 3: WRITE OUT THE CSV FILE WITHOUT THE FIRST ROW

Now that `csvRows` contains all rows but the first row, the list needs to be written out to a CSV file in the *headerRemoved* folder. Add the following to *removeCsvHeader.py*:

```
   #! python3
   # removeCsvHeader.py - Removes the header from all CSV files in the
current
   # working directory.
   --snip--

   # Loop through every file in the current working directory.
❶ for csvFilename in os.listdir('.'):
       if not csvFilename.endswith('.csv'):
           continue    # skip non-CSV files

       --snip--

       # Write out the CSV file.
       csvFileObj = open(os.path.join('headerRemoved', csvFilename), 'w',
                     newline='')
       csvWriter = csv.writer(csvFileObj)
       for row in csvRows:
           csvWriter.writerow(row)
       csvFileObj.close()
```

The CSV `Writer` object will write the list to a CSV file in `headerRemoved` using `csvFilename` (which we also used in the CSV reader). This will overwrite the original file.

Once we create the `Writer` object, we loop over the sublists stored in `csvRows` and write each sublist to the file.

After the code is executed, the outer `for` loop ❶ will loop to the next filename from `os.listdir('.')`. When that loop is finished, the program will be complete.

To test your program, download *removeCsvHeader.zip* from *http://nostarch.com/automatestuff/* and unzip it to a folder. Run the *removeCsvHeader.py* program in that folder. The output will look like this:

```
Removing header from NAICS_data_1048.csv...
Removing header from NAICS_data_1218.csv...
--snip--
Removing header from NAICS_data_9834.csv...
Removing header from NAICS_data_9986.csv...
```

This program should print a filename each time it strips the first line from a CSV file.

IDEAS FOR SIMILAR PROGRAMS

The programs that you could write for CSV files are similar to the kinds you could write for Excel files, since they're both spreadsheet files. You could write programs to do the following:

- Compare data between different rows in a CSV file or between multiple CSV files.
- Copy specific data from a CSV file to an Excel file, or vice versa.
- Check for invalid data or formatting mistakes in CSV files and alert the user to these errors.
- Read data from a CSV file as input for your Python programs.

JSON AND APIS

JavaScript Object Notation is a popular way to format data as a single human-readable string. JSON is the native way that JavaScript programs write their data structures and usually resembles what Python's `pprint()` function would produce. You don't need to know JavaScript in order to work with JSON-formatted data.

Here's an example of data formatted as JSON:

```
{"name": "Zophie", "isCat": true,
"miceCaught": 0, "napsTaken": 37.5,
"felineIQ": null}
```

JSON is useful to know, because many websites offer JSON content as a way for programs to interact with the website. This is known as providing an *application programming interface (API)*. Accessing an API is the same as accessing any other web page via a URL. The difference is that the data returned by an API is formatted (with JSON, for example) for machines; APIs aren't easy for people to read.

Many websites make their data available in JSON format. Facebook, Twitter, Yahoo, Google, Tumblr, Wikipedia, Flickr, Data.gov, Reddit, IMDb, Rotten Tomatoes, LinkedIn, and many other popular sites offer APIs for programs to use. Some of these sites require registration, which is almost always free. You'll have to find documentation for what URLs your program needs to request in order to get the data you want, as well as the general format of the JSON data structures that are returned. This documentation should be provided by whatever site is offering the API; if they have a "Developers" page, look for the documentation there.

Using APIs, you could write programs that do the following:

- Scrape raw data from websites. (Accessing APIs is often more convenient than downloading web pages and parsing HTML with Beautiful Soup.)
- Automatically download new posts from one of your social network accounts and post them to another account. For example, you could take your Tumblr posts and post them to Facebook.
- Create a "movie encyclopedia" for your personal movie collection by pulling data from IMDb, Rotten Tomatoes, and Wikipedia and putting it into a single text file on your computer.

You can see some examples of JSON APIs in the resources at *http://nostarch.com/automatestuff/*.

THE JSON MODULE

Python's `json` module handles all the details of translating between a string with JSON data and Python values for the `json.loads()` and `json.dumps()` functions. JSON can't store *every* kind of

Python value. It can contain values of only the following data types: strings, integers, floats, Booleans, lists, dictionaries, and `NoneType`. JSON cannot represent Python-specific objects, such as `File` objects, CSV `Reader` or `Writer` objects, `Regex` objects, or Selenium `WebElement` objects.

READING JSON WITH THE LOADS() FUNCTION

To translate a string containing JSON data into a Python value, pass it to the `json.loads()` function. (The name means “load string,” not “loads.”) Enter the following into the interactive shell:

```
>>> stringOfJsonData = '{"name": "Zophie", "isCat": true, "miceCaught": 0,
"felineIQ": null}'
>>> import json
>>> jsonDataAsPythonValue = json.loads(stringOfJsonData)
>>> jsonDataAsPythonValue
{'isCat': True, 'miceCaught': 0, 'name': 'Zophie', 'felineIQ': None}
```

After you import the `json` module, you can call `loads()` and pass it a string of JSON data. Note that JSON strings always use double quotes. It will return that data as a Python dictionary. Python dictionaries are not ordered, so the key-value pairs may appear in a different order when you print `jsonDataAsPythonValue`.

WRITING JSON WITH THE DUMPS() FUNCTION

The `json.dumps()` function (which means “dump string,” not “dumps”) will translate a Python value into a string of JSON-formatted data. Enter the following into the interactive shell:

```
>>> pythonValue = {'isCat': True, 'miceCaught': 0, 'name': 'Zophie',
'felineIQ': None}
>>> import json
>>> stringOfJsonData = json.dumps(pythonValue)
>>> stringOfJsonData
'{"isCat": true, "felineIQ": null, "miceCaught": 0, "name": "Zophie" }'
```

The value can only be one of the following basic Python data types: dictionary, list, integer, float, string, Boolean, or `None`.

PROJECT: FETCHING CURRENT WEATHER DATA

Checking the weather seems fairly trivial: Open your web browser, click the address bar, type the URL to a weather website (or search for one and then click the link), wait for the page to load, look past all the ads, and so on.

Actually, there are a lot of boring steps you could skip if you had a program that downloaded the weather forecast for the next few days and printed it as plaintext. This program uses the `requests` module from Chapter 11 to download data from the Web.

Overall, the program does the following:

- Reads the requested location from the command line.
- Downloads JSON weather data from OpenWeatherMap.org.
- Converts the string of JSON data to a Python data structure.
- Prints the weather for today and the next two days.

So the code will need to do the following:

- Join strings in `sys.argv` to get the location.
- Call `requests.get()` to download the weather data.
- Call `json.loads()` to convert the JSON data to a Python data structure.
- Print the weather forecast.

For this project, open a new file editor window and save it as *quickWeather.py*.

STEP 1: GET LOCATION FROM THE COMMAND LINE ARGUMENT

The input for this program will come from the command line. Make *quickWeather.py* look like this:

```
#! python3
# quickWeather.py - Prints the weather for a location from the command line.

import json, requests, sys
```

```
# Compute location from command line arguments.
if len(sys.argv) < 2:
    print('Usage: quickWeather.py location')
    sys.exit()
location = ' '.join(sys.argv[1:])

# TODO: Download the JSON data from OpenWeatherMap.org's API.

# TODO: Load JSON data into a Python variable.
```

In Python, command line arguments are stored in the `sys.argv` list. After the `#!` shebang line and `import` statements, the program will check that there is more than one command line argument. (Recall that `sys.argv` will always have at least one element, `sys.argv[0]`, which contains the Python script's filename.) If there is only one element in the list, then the user didn't provide a location on the command line, and a "usage" message will be provided to the user before the program ends.

Command line arguments are split on spaces. The command line argument `San Francisco, CA` would make `sys.argv` hold `['quickWeather.py', 'San', 'Francisco,', 'CA']`. Therefore, call the `join()` method to join all the strings except for the first in `sys.argv`. Store this joined string in a variable named `location`.

STEP 2: DOWNLOAD THE JSON DATA

OpenWeatherMap.org provides real-time weather information in JSON format. Your program simply has to download the page at *http://api.openweathermap.org/data/2.5/forecast/daily?q=<Location>&cnt=3*, where *<Location>* is the name of the city whose weather you want. Add the following to *quickWeather.py*.

```
#! python3
# quickWeather.py - Prints the weather for a location from the command line.

--snip--

# Download the JSON data from OpenWeatherMap.org's API.
url ='http://api.openweathermap.org/data/2.5/forecast/daily?q=%s&cnt=3' %
(location)
response = requests.get(url)
response.raise_for_status()

# TODO: Load JSON data into a Python variable.
```

We have `location` from our command line arguments. To make the URL we want to access, we use the `%s` placeholder and insert whatever string is stored in `location` into that spot in the URL string. We store the result in `url` and pass `url` to `requests.get()`. The `requests.get()` call returns a `Response` object, which you can check for errors by calling `raise_for_status()`. If no exception is raised, the downloaded text will be in `response.text`.

STEP 3: LOAD JSON DATA AND PRINT WEATHER

The `response.text` member variable holds a large string of JSON-formatted data. To convert this to a Python value, call the `json.loads()` function. The JSON data will look something like this:

```
{'city': {'coord': {'lat': 37.7771, 'lon': -122.42},
          'country': 'United States of America',
          'id': '5391959',
          'name': 'San Francisco',
          'population': 0},
'cnt': 3,
'cod': '200',
'list': [{'clouds': 0,
          'deg': 233,
          'dt': 1402344000,
          'humidity': 58,
          'pressure': 1012.23,
          'speed': 1.96,
          'temp': {'day': 302.29,
                   'eve': 296.46,
                   'max': 302.29,
                   'min': 289.77,
                   'morn': 294.59,
                   'night': 289.77},
          'weather': [{'description': 'sky is clear',
                       'icon': '01d',
--snip--
```

You can see this data by passing `weatherData` to `pprint.pprint()`. You may want to check *http://openweathermap.org/* for more documentation on what these fields mean. For example, the online documentation will tell you that the `302.29` after `'day'` is the daytime temperature in Kelvin, not Celsius or Fahrenheit.

The weather descriptions you want are after `'main'` and `'description'`. To neatly print them out, add the following to *quickWeather.py*.

```
   ! python3
   # quickWeather.py - Prints the weather for a location from the command
line.

   --snip--

   # Load JSON data into a Python variable.
   weatherData = json.loads(response.text)
   # Print weather descriptions.
❶ w = weatherData['list']
   print('Current weather in %s:' % (location))
   print(w[0]['weather'][0]['main'], '-', w[0]['weather'][0]['description'])
   print()
   print('Tomorrow:')
   print(w[1]['weather'][0]['main'], '-', w[1]['weather'][0]['description'])
   print()
   print('Day after tomorrow:')
   print(w[2]['weather'][0]['main'], '-', w[2]['weather'][0]['description'])
```

Notice how the code stores `weatherData['list']` in the variable `w` to save you some typing ❶. You use `w[0]`, `w[1]`, and `w[2]` to retrieve the dictionaries for today, tomorrow, and the day after tomorrow's weather, respectively. Each of these dictionaries has a `'weather'` key, which contains a list value. You're interested in the first list item, a nested dictionary with several more keys, at index 0. Here, we print the values stored in the `'main'` and `'description'` keys, separated by a hyphen.

When this program is run with the command line argument `quickWeather.py San Francisco, CA`, the output looks something like this:

```
Current weather in San Francisco, CA:
Clear - sky is clear

Tomorrow:
Clouds - few clouds

Day after tomorrow:
Clear - sky is clear
```

(The weather is one of the reasons I like living in San Francisco!)

Ideas for Similar Programs

Accessing weather data can form the basis for many types of programs. You can create similar programs to do the following:

- Collect weather forecasts for several campsites or hiking trails to see which one will have the best weather.
- Schedule a program to regularly check the weather and send you a frost alert if you need to move your plants indoors. (Chapter 15 covers scheduling, and Chapter 16 explains how to send email.)
- Pull weather data from multiple sites to show all at once, or calculate and show the average of the multiple weather predictions.

SUMMARY

CSV and JSON are common plaintext formats for storing data. They are easy for programs to parse while still being human readable, so they are often used for simple spreadsheets or web app data.
The `csv` and `json` modules greatly simplify the process of reading and writing to CSV and JSON files.

The last few chapters have taught you how to use Python to parse information from a wide variety of file formats. One common task is taking data from a variety of formats and parsing it for the particular information you need. These tasks are often specific to the point that commercial software is not optimally helpful. By writing your own scripts, you can make the computer handle large amounts of data presented in these formats.