In those rare, terrifying moments when I'm without Wi-Fi, I realize just how much of what I do on the computer is really what I do on the internet. Out of sheer habit I'll find myself trying to check email, read friends' Twitter feeds, or answer the question, "Did Kurtwood Smith have any major roles before he was in the original 1987 *RoboCop*?"[1]

Since so much work on a computer involves going on the internet, it'd be great if your programs could get online. *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.

bs4 Parses HTML, the format that web pages are written in.

selenium Launches and controls a web browser. The selenium module is able to fill in forms and simulate mouse clicks in this browser.

Project: mapIt.py with the webbrowser Module

The webbrowser module's open() function can launch a new browser to a specified URL. Enter the following into the interactive shell:

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

A web browser tab will open to the URL *https://inventwithpython.com/*. This is about the only thing the webbrowser module can do. Even so, the open() function does make some interesting things possible. For example, it's tedious to copy a street address to the clipboard and bring up a map of it on Google Maps. You could take a few steps out of this task by writing a simple script to automatically launch the map in your browser using the contents of your clipboard. This way, you only have to copy the address to a clipboard and run the script, and the map will be loaded for you.

This is what your program does:

1. Gets a street address from the command line arguments or clipboard
2. Opens the web browser to the Google Maps page for the address

This means your code will need to do the following:

1. Read the command line arguments from sys.argv.
2. Read the clipboard contents.
3. Call the webbrowser.open() function to open the web browser.

Open a new file editor tab and save it as *mapIt.py*.

Step 1: Figure Out the URL

Based on the instructions in Appendix B, set up *mapIt.py* so that when you run it from the command line, like so . . .

```
C:\> mapit 870 Valencia St, San Francisco, CA 94110
```

. . . the script will use the command line arguments instead of the clipboard. If there are no command line arguments, then the program will know to use the contents of the clipboard.

First you need to figure out what URL to use for a given street address. When you load *https://maps.google.com/* in the browser and search for an address, the URL in the address

bar looks something like this: *https://www.google.com/maps/place/870+Valencia+St/@37.7590311,-122.4215096,17z/data=!3m1!4b1!4m2!3m1!1s0x808f7e3dadc07a37:0xc86b0b2bb93b73d8*.

The address is in the URL, but there's a lot of additional text there as well. Websites often add extra data to URLs to help track visitors or customize sites. But if you try just going to *https://www.google.com/maps/place/870+Valencia+St+San+Francisco+CA/*, you'll find that it still brings up the correct page. So your program can be set to open a web browser to 'https://www.google.com/maps/place/*your_address_string*' (where *your_address_string* is the address you want to map).

Step 2: Handle the Command Line Arguments

Make your code look like this:

```
#! python3
# mapIt.py - Launches a map in the browser using an address from the
# command line or clipboard.

import webbrowser, sys
if len(sys.argv) > 1:
    # Get address from command line.
    address = ' '.join(sys.argv[1:])

# TODO: Get address from clipboard.
```

After the program's #! shebang line, you need to import the webbrowser module for launching the browser and import the sys module for reading the potential command line arguments. The sys.argv variable stores a list of the program's filename and command line arguments. If this list has more than just the filename in it, then len(sys.argv) evaluates to an integer greater than 1, meaning that command line arguments have indeed been provided.

Command line arguments are usually separated by spaces, but in this case, you want to

interpret all of the arguments as a single string. Since sys.argv is a list of strings, you can pass it to the join() method, which returns a single string value. You don't want the program name in this string, so instead of sys.argv, you should pass sys.argv[1:] to chop off the first element of the array. The final string that this expression evaluates to is stored in the address variable.

If you run the program by entering this into the command line . . .

```
mapit 870 Valencia St, San Francisco, CA 94110
```

. . . the sys.argv variable will contain this list value:

```
['mapIt.py', '870', 'Valencia', 'St, ', 'San', 'Francisco, ', 'CA', '94110']
```

The address variable will contain the string '870 Valencia St, San Francisco, CA 94110'.

Step 3: Handle the Clipboard Content and Launch the Browser

Make your code look like the following:

```
#! python3
# mapIt.py - Launches a map in the browser using an address from the
# command line or clipboard.
import webbrowser, sys, pyperclip
if len(sys.argv) > 1:
    # Get address from command line.
    address = ' '.join(sys.argv[1:])
else:
    # Get address from clipboard.
    address = pyperclip.paste()

webbrowser.open('https://www.google.com/maps/place/' + address)
```

If there are no command line arguments, the program will assume the address is stored on the clipboard. You can get the clipboard content with pyperclip.paste() and store it in a variable named address. Finally, to launch a web browser with the Google Maps URL, call webbrowser.open().

While some of the programs you write will perform huge tasks that save you hours, it can be just as satisfying to use a program that conveniently saves you a few seconds each time you perform a common task, such as getting a map of an address. Table 12-1 compares the steps needed to display a map with and without *mapIt.py*.

Table 12-1: Getting a Map with and Without *mapIt.py*

Manually getting a map	Using mapIt.py
1. Highlight the address. 2. Copy the address. 3. Open the web browser. 4. Go to *https://maps.google.com/*. 5. Click the address text field. 6. Paste the address. 7. Press enter.	1. Highlight the address. 2. Copy the address. 3. Run *mapIt.py*.

See how *mapIt.py* makes this task less tedious?

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 --user 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
❶ >>> res = requests.get('https://automatetheboringstuff.com/files/rj.txt')
```

```
>>> type(res)
<class 'requests.models.Response'>
❷ >>> 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*, provided on this book's site ❶. 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 ❷. (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.") You can find a complete list of HTTP status codes and their meanings at *https://en.wikipedia.org/wiki/List_of_HTTP_status_codes*.

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.

If the request failed and displayed an error message, like "Failed to establish a new connection" or "Max retries exceeded," then check your internet connection. Connecting to servers can be quite complicated, and I can't give a full list of possible problems here. You can find common causes of your error by doing a web search of the error message in quotes.

Checking for Errors

As you've seen, the Response object has a status_code attribute that can be checked against requests.codes.ok (a variable that has the integer value 200) 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('https://inventwithpython.com/page_that_does_not_exist')
>>> res.raise_for_status()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>

  File "C:\Users\Al\AppData\Local\Programs\Python\Python37\lib\site-packages\requests\models
.py", line 940, in raise_for_status
    raise HTTPError(http_error_msg, response=self)
requests.exceptions.HTTPError: 404 Client Error: Not Found for url: https://inventwithpython
.com/page_that_does_not_exist.html
```

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('https://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 for url: https://
inventwithpython.com/page_that_does_not_exist.html
```

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.

UNICODE ENCODINGS

Unicode encodings are beyond the scope of this book, but you can learn more about them from these web pages:

- Joel on Software: The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!): *https://www.joelonsoftware.com/articles/Unicode.html*
- Pragmatic Unicode: *https://nedbatchelder.com/text/unipain.html*

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 *https://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:

- *https://developer.mozilla.org/en-US/learn/html/*
- *https://htmldog.com/guides/html/beginner/*
- *https://www.codecademy.com/learn/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 12-1 in a browser.

Figure 12-1: Hello, world! *rendered in the browser*

The opening tag says that the enclosed text will appear in bold. The closing 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="https://inventwithpython.com">Python books</a>.
```

This HTML will look like Figure 12-2 in a browser.

Figure 12-2: 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 macOS) 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 12-3). This is the text your browser actually receives. The browser knows how to display, or *render*, the web page from this HTML.

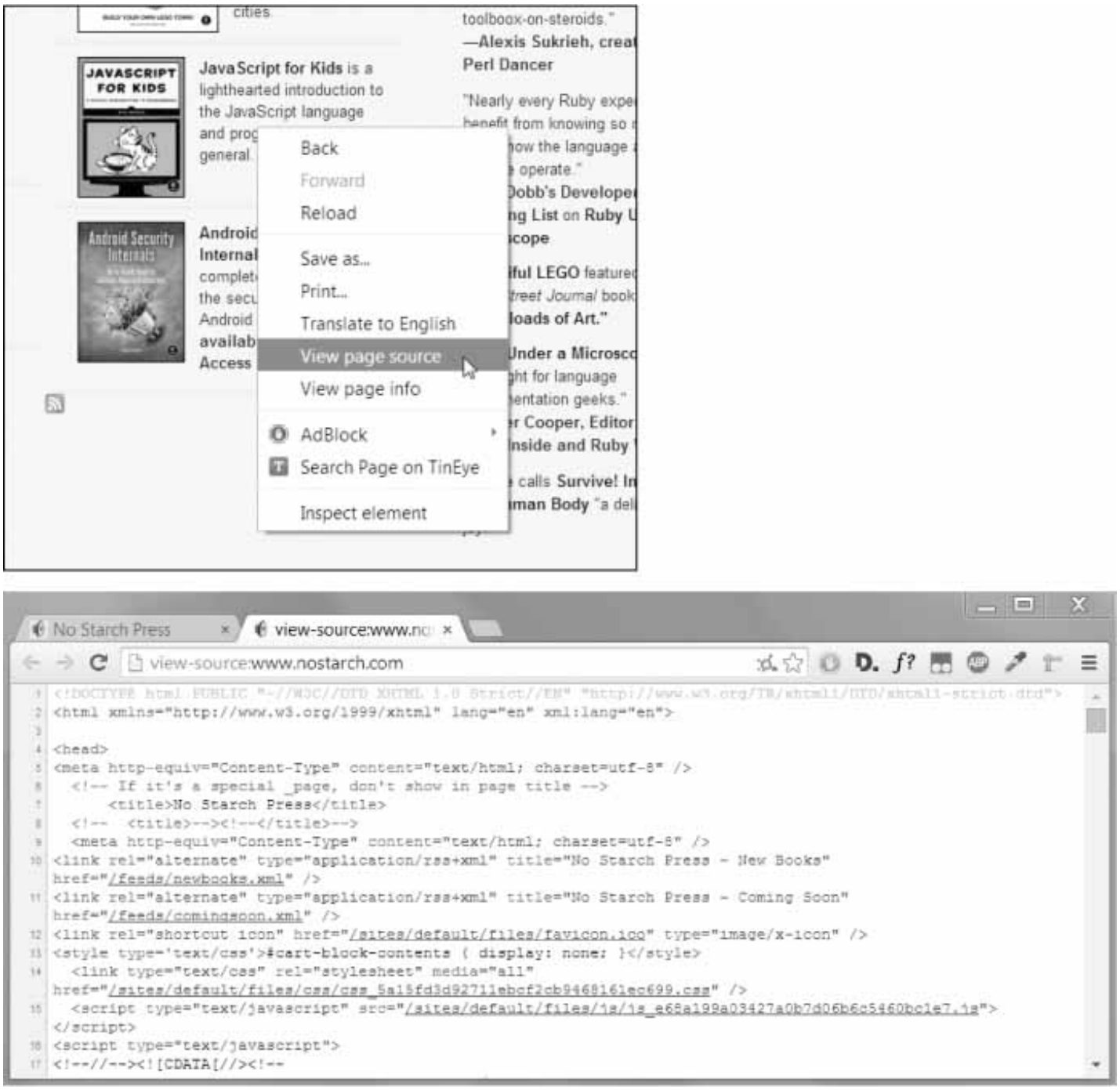


Figure 12-3: Viewing the source of a web page

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 12-4). 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 macOS, pressing ⌘-OPTION-I will open Chrome's Developer Tools.

Figure 12-4: 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 macOS. 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 bs4, will be less likely to result in bugs.

You can find an extended argument for why you shouldn't parse HTML with regular expressions at *https://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 *https://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 weather information for that ZIP code? Right-click where it is on the page (or CONTROL-click on macOS) 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 12-5 shows the developer tools open to the HTML of the nearest forecast. Note that if the *https://weather.gov/* site changes the design of its web pages, you'll need to repeat this process to inspect the new elements.

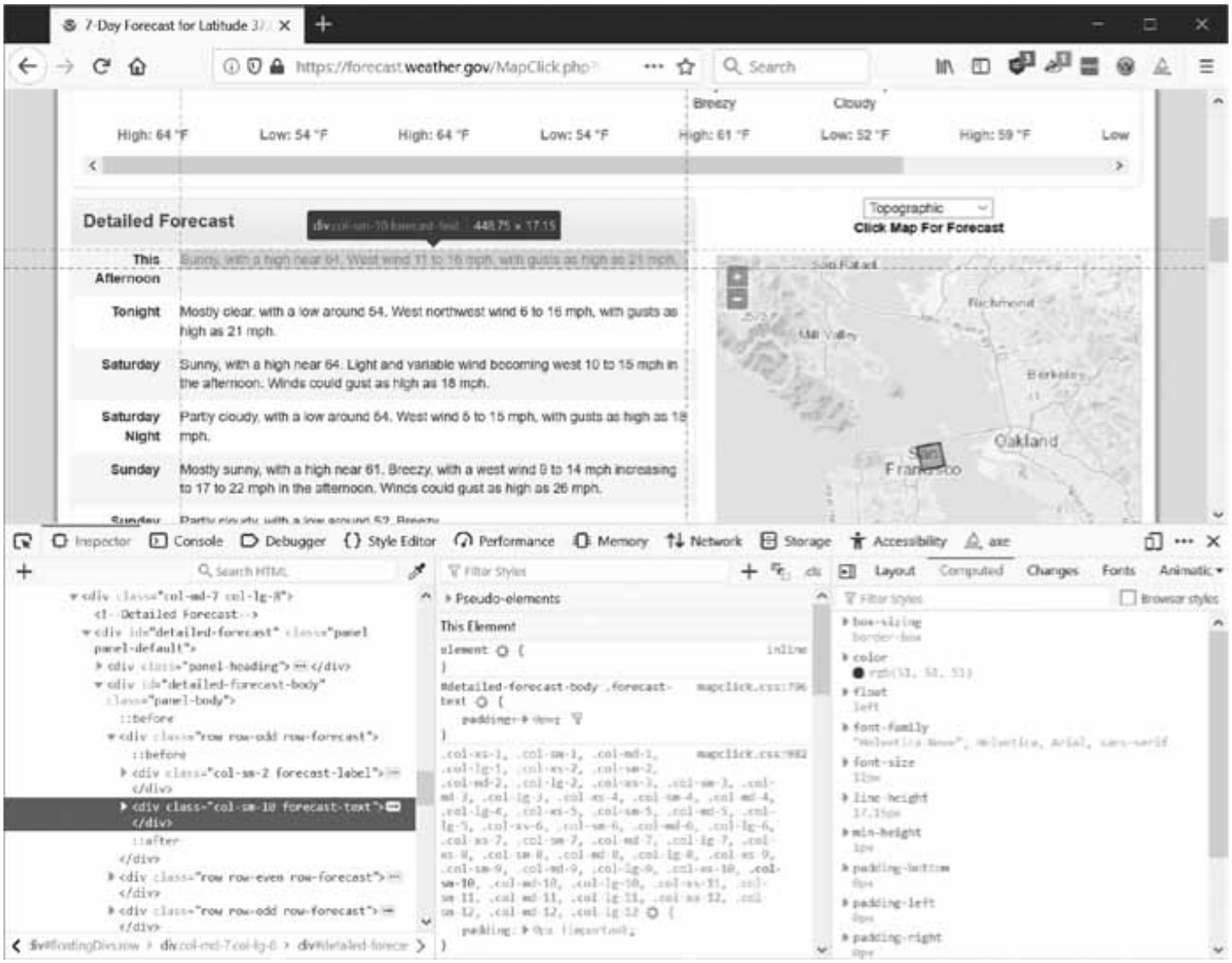


Figure 12-5: Inspecting the element that holds forecast text with the developer tools

From the developer tools, you can see that the HTML responsible for the forecast part of the web page is Sunny, with a high near 64. West wind 11 to 16 mph, with gusts as high as 21 mph. . This is exactly what you were looking for! It seems that the forecast information is contained inside a element with the forecast-text CSS class. Right-click on this element in the browser's developer console, and from the context menu that appears, select **Copy ▸ CSS Selector**. This will copy a string such as 'div.row-odd:nth-child(1) > div:nth-child(2)' to the clipboard. You can use this string for Beautiful Soup's select() or Selenium's find_element_by_css_selector() methods, as explained later in this chapter. Now that you know what you're looking for, the Beautiful Soup module will help you find it in the string.

Parsing HTML with the bs4 Module

Beautiful Soup is a module for extracting information from an HTML page (and is much better for this purpose than regular expressions). The Beautiful Soup module's name is `bs4` (for Beautiful Soup, version 4). To install it, you will need to run `pip install --user 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 tab in Mu, enter the following, and save it as *example.html*. Alternatively, download it from *https://nostarch.com/automatestuff2/*.

```
<!-- 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="https://
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 a `BeautifulSoup` object. Enter the following into the interactive shell while your computer is connected to the internet:

```
>>> import requests, bs4
>>> res = requests.get('https://nostarch.com')
>>> res.raise_for_status()
>>> noStarchSoup = bs4.BeautifulSoup(res.text, 'html.parser')
>>> 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() along with a second argument that tells Beautiful Soup which parser to use to analyze the HTML.

Enter the following into the interactive shell (after making sure the *example.html* file is in the working directory):

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

The 'html.parser' parser used here comes with Python. However, you can use the faster 'lxml' parser if you install the third-party lxml module. Follow the instructions in Appendix A to install this module by running pip install --user lxml. Forgetting to include this second argument will result in a UserWarning: No parser was explicitly specified warning.

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 *https://nostarch.com/automatestuff2/*), but here's a short introduction to selectors. Table 12-2 shows examples of the most common CSS selector patterns.

Table 12-2: Examples of CSS Selectors

Selector passed to the select() method	**Will match . . .**
soup.select('div')	All elements named
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 that are within an element named
soup.select('div > span')	All elements named that are *directly* within an element named , 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. Instead of writing the selector yourself, you can

also right-click on the element in your browser and select **Inspect Element**. When the browser's developer console opens, right-click on the element's HTML and select **Copy ▸ CSS Selector** to copy the selector string to the clipboard and paste it into your source code.

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(), 'html.parser')
>>> elems = exampleSoup.select('#author')
>>> type(elems) # elems is a list of Tag objects.
<class 'list'>
>>> len(elems)
1
>>> type(elems[0])
<class 'bs4.element.Tag'>
>>> str(elems[0]) # The Tag object as a string.
'<span id="author">Al Sweigart</span>'
>>> elems[0].getText()
'Al Sweigart'
>>> 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="https://
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'), 'html.parser')
>>> 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 elements and then store the first matched element in spanElem. Passing the attribute name 'id' to get() returns the attribute's value, 'author'.

Project: Opening All Search Results

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 with the search results page for the Python Package Index at *https://pypi.org/*. A program like this can be adapted to many other websites, although the Google and DuckDuckGo often employ measures that make

scraping their search results pages difficult.

This is what your program does:

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

This means your code will need to do the following:

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

Open a new file editor tab and save it as *searchpypi.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 search, you can see that the result page has a URL like *https://pypi.org/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
# searchpypi.py  - Opens several search results.

import requests, sys, webbrowser, bs4
```

```
print('Searching...')    # display text while downloading the search result page
res = requests.get('https://google.com/search?q=' 'https://pypi.org/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 search for *Beautiful Soup*, you can open the browser's developer tools and inspect some of the link elements on the page. They can look complicated, something like pages of this: <a class="package-snippet" href="HYPERLINK "view-source:https://pypi.org/project/xml-parser/"/project/xml-parser/">.

It doesn't matter that the element looks incredibly complicated. You just need to find the pattern that all the search result links have.

Make your code look like the following:

```
#! python3
# searchpypi.py - Opens several google results.
import requests, sys, webbrowser, bs4
```

```
--snip--
# Retrieve top search result links.
soup = bs4.BeautifulSoup(res.text, 'html.parser')
# Open a browser tab for each result.
linkElems = soup.select('.package-snippet')
```

If you look at the <a> elements, though, the search result links all have class="package-snippet". Looking through the rest of the HTML source, it looks like the package-snippet class is used only for search result links. You don't have to know what the CSS class package-snippet 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 '.package-snippet' to find all <a> elements that are within an element that has the package-snippet CSS class. Note that if the PyPI website changes its layout, you may need to update this program with a new CSS selector string to pass to soup.select(). The rest of the program will still be up to date.

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
# searchpypi.py - Opens several search results.
import requests, sys, webbrowser, bs4
--snip--
# Open a browser tab for each result.
linkElems = soup.select('.package-snippet')
numOpen = min(5, len(linkElems))
for i in range(numOpen):
    urlToOpen = 'https://pypi.org' + linkElems[i].get('href')
    print('Opening', urlToOpen)
```

webbrowser.open(urlToOpen)

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 '.package-snippet' 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 https://pypi.org part, so you have to concatenate that to the href attribute's string value.

Now you can instantly open the first five PyPI search results for, say, *boring stuff* by running searchpypi boring stuff 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 12-6). The front page at *https://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.

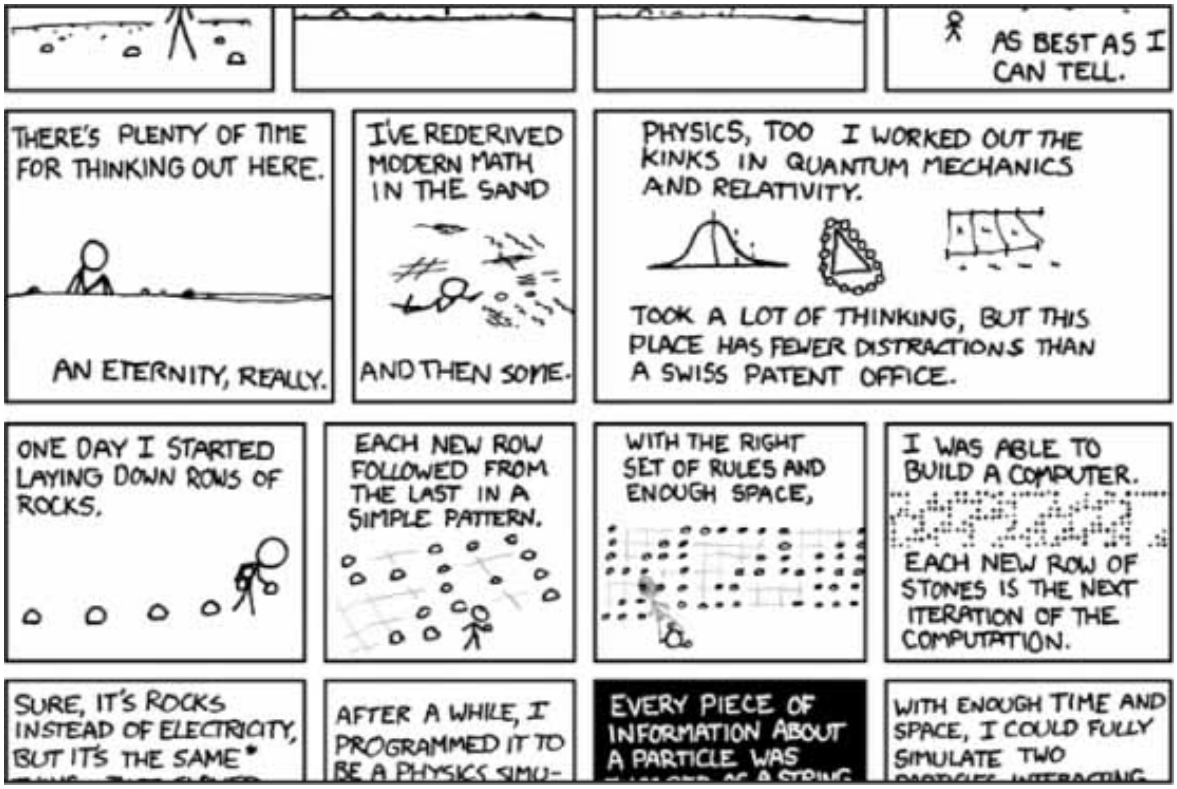


Figure 12-6: XKCD, "a webcomic of romance, sarcasm, math, and language"

Here's what your program does:

1. Loads the XKCD home page

2. Saves the comic image on that page
3. Follows the Previous Comic link
4. Repeats until it reaches the first comic

This means your code will need to do the following:

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

Open a new file editor tab 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 element.
- The Prev button has a rel HTML attribute with the value prev.
- The first comic's Prev button links to the *https://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 = 'https://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 `'https://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 remaining 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 = 'https://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, 'html.parser')

    # 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:
        comicUrl = 'https:' + comicElem[0].get('src')
        # Download the image.
        print('Downloading image %s...' % (comicUrl))
        res = requests.get(comicUrl)
        res.raise_for_status()

    # 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 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 = 'https://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 'https://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 macOS 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 https://xkcd.com...
Downloading image https://imgs.xkcd.com/comics/phone_alarm.png...
Downloading page https://xkcd.com/1358/...
Downloading image https://imgs.xkcd.com/comics/nro.png...
Downloading page https://xkcd.com/1357/...
Downloading image https://imgs.xkcd.com/comics/free_speech.png...
Downloading page https://xkcd.com/1356/...
Downloading image https://imgs.xkcd.com/comics/orbital_mechanics.png...
Downloading page https://xkcd.com/1355/...
Downloading image https://imgs.xkcd.com/comics/airplane_message.png...
Downloading page https://xkcd.com/1354/...
Downloading image https://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 *https://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 bs4 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 were a human user interacting with the page. Using selenium, you can interact with web pages in a much more advanced way than with requests and bs4; 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.

Still, if you need to interact with a web page in a way that, say, depends on the JavaScript code that updates the page, you'll need to use selenium instead of requests. That's because major ecommerce websites such as Amazon almost certainly have software

systems to recognize traffic that they suspect is a script harvesting their info or signing up for multiple free accounts. These sites may refuse to serve pages to you after a while, breaking any scripts you've made. The selenium module is much more likely to function on these sites long-term than requests.

A major "tell" to websites that you're using a script is the *user-agent* string, which identifies the web browser and is included in all HTTP requests. For example, the user-agent string for the requests module is something like 'python-requests/2.21.0'. You can visit a site such as *https://www.whatsmyua.info/* to see your user-agent string. Using selenium, you're much more likely to "pass for human" because not only is Selenium's user-agent is the same as a regular browser (for instance, 'Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:65.0) Gecko/20100101 Firefox/65.0'), but it has the same traffic patterns: a selenium-controlled browser will download images, advertisements, cookies, and privacy-invading trackers just like a regular browser. However, selenium can still be detected by websites, and major ticketing and ecommerce websites often block browsers controlled by selenium to prevent web scraping of their pages.

Starting a selenium-Controlled Browser

The following examples will show you how to control Firefox's web browser. If you don't already have Firefox, you can download it for free from *https://getfirefox.com/*. You can install selenium by running pip install --user selenium from a command line terminal. More information is available in Appendix A.

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('https://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('https://inventwithpython.com') directs the browser to *https://inventwithpython.com/*. Your browser should look something like Figure 12-7.

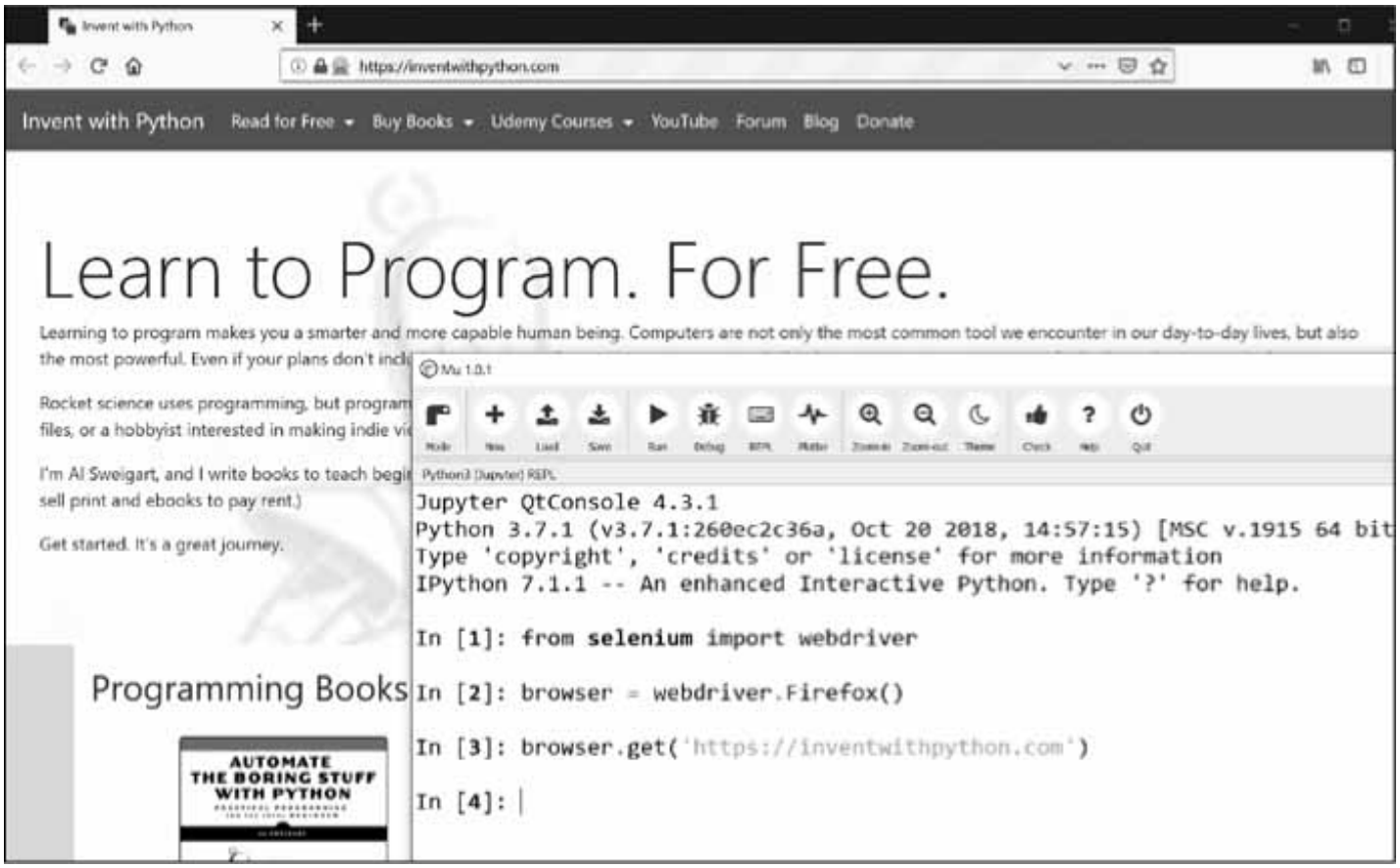


Figure 12-7: After we call webdriver.Firefox() and get() in Mu, the Firefox browser appears.

If you encounter the error message "'geckodriver' executable needs to be in PATH.", then you need to manually download the webdriver for Firefox before you can use selenium to control it. You can also control browsers other than Firefox if you install the webdriver for them.

For Firefox, go to *https://github.com/mozilla/geckodriver/releases* and download the geckodriver for your operating system. ("Gecko" is the name of the browser engine used in Firefox.) For example, on Windows you'll want to download the *geckodriver-v0.24.0-*

win64.zip link, and on macOS, you'll want the *geckodriver-v0.24.0-macos.tar.gz* link. Newer versions will have slightly different links. The downloaded ZIP file will contain a *geckodriver.exe* (on Windows) or *geckodriver* (on macOS and Linux) file that you can put on your system PATH. Appendix B has information about the system PATH, or you can learn more at *https://stackoverflow.com/q/40208051/1893164*.

For Chrome, go to *https://sites.google.com/a/chromium.org/chromedriver/downloads* and download the ZIP file for your operating system. This ZIP file will contain a *chromedriver.exe* (on Windows) or *chromedriver* (on macOS or Linux) file that you can put on your system PATH.

Other major web browsers also have webdrivers available, and you can often find these by performing an internet search for "<browser name> webdriver".

If you still have problems opening up a new browser under the control of selenium, it may be because the current version of the browser is incompatible with the selenium module. One workaround is to install an older version of the web browser—or, more simply, an older version of the selenium module. You can find the list of selenium version numbers at *https://pypi.org/project/selenium/#history*. Unfortunately, the compatibility between versions of selenium and a browser sometimes breaks, and you may need to search the web for possible solutions. Appendix A has more information about running pip to install a specific version of selenium. (For example, you might run pip install --user -U selenium==3.14.1.)

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 12-3 shows several examples of find_element_* and find_elements_* methods being called on a WebDriver object that's stored in the variable browser.

Table 12-3: 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*
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 12-4.

Table 12-4: 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 hello
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 tab and enter the following program:

```
from selenium import webdriver
browser = webdriver.Firefox()
browser.get('https://inventwithpython.com')
try:
    elem = browser.find_element_by_class_name(' cover-thumb')
    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('https://inventwithpython.com')
>>> linkElem = browser.find_element_by_link_text('Read Online for Free')
>>> type(linkElem)
<class 'selenium.webdriver.remote.webelement.FirefoxWebElement'>
>>> linkElem.click() # follows the "Read Online for Free" link
```

This opens Firefox to *https://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://login.metafilter.com')
>>> userElem = browser.find_element_by_id('user_name)
>>> userElem.send_keys('your_real_username_here')

>>> passwordElem = browser.find_element_by_id('user_pass')
>>> passwordElem.send_keys('your_real_password_here')
>>> passwordElem.submit()
```

As long as login page for MetaFilter 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.)

> **WARNING**
>
> *Avoid putting your passwords in source code whenever possible. It's easy to accidentally leak your passwords to others when they are left unencrypted on your hard drive. If possible, have your program prompt users to enter their passwords from the keyboard using the* pyinputplus.inputPassword() *function described in Chapter 8.*

Sending Special Keys

The selenium module 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 12-5 lists the commonly used Keys variables.

Table 12-5: 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
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('https://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

The selenium module 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 *https://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.

Practice Questions

1. Briefly describe the differences between the webbrowser, requests, bs4, and selenium modules.
2. What type of object is returned by requests.get()? How can you access the downloaded content as a string value?
3. What requests method checks that the download worked?
4. How can you get the HTTP status code of a requests response?
5. How do you save a requests response to a file?
6. What is the keyboard shortcut for opening a browser's developer tools?
7. How can you view (in the developer tools) the HTML of a specific element on a web page?
8. What is the CSS selector string that would find the element with an id attribute of main?
9. What is the CSS selector string that would find the elements with a CSS class of highlight?
10. What is the CSS selector string that would find all the elements inside another element?
11. What is the CSS selector string that would find the <button> element with a value attribute set to favorite?
12. Say you have a Beautiful Soup Tag object stored in the variable spam for the element

Hello, world! . How could you get a string 'Hello, world!' from the Tag object?

13. How would you store all the attributes of a Beautiful Soup Tag object in a variable named linkElem?
14. Running import selenium doesn't work. How do you properly import the selenium module?
15. What's the difference between the find_element_* and find_elements_* methods?
16. What methods do Selenium's WebElement objects have for simulating mouse clicks and keyboard keys?
17. You could call send_keys(Keys.ENTER) on the Submit button's WebElement object, but what is an easier way to submit a form with selenium?
18. How can you simulate clicking a browser's Forward, Back, and Refresh buttons with selenium?

Practice Projects

For practice, write programs to do the following tasks.

Command Line Emailer

Write a program that takes an email address and string of text on the command line and then, using selenium, logs in to your email account and sends an email of the string to the provided address. (You might want to set up a separate email account for this program.)

This would be a nice way to add a notification feature to your programs. You could also write a similar program to send messages from a Facebook or Twitter account.

Image Site Downloader

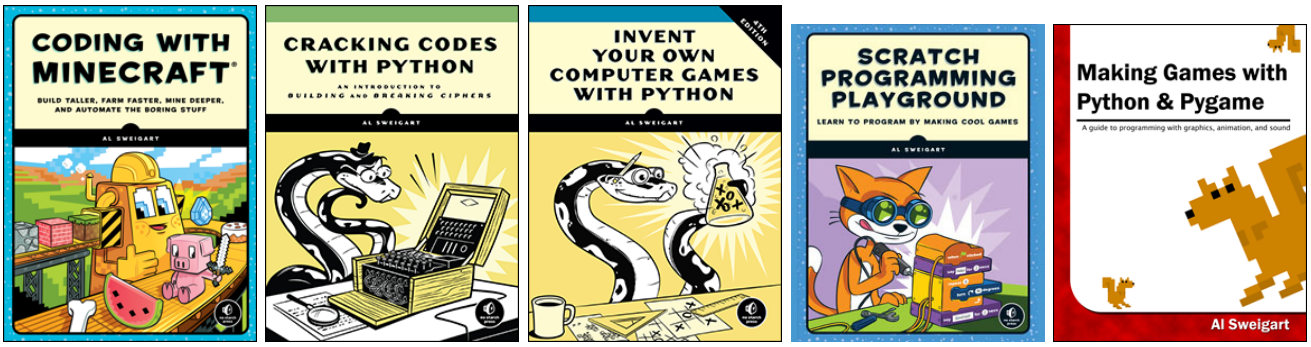
Write a program that goes to a photo-sharing site like Flickr or Imgur, searches for a category of photos, and then downloads all the resulting images. You could write a program that works with any photo site that has a search feature.

2048

2048 is a simple game where you combine tiles by sliding them up, down, left, or right with the arrow keys. You can actually get a fairly high score by repeatedly sliding in an up, right, down, and left pattern over and over again. Write a program that will open the game at *https://gabrielecirulli.github.io/2048/* and keep sending up, right, down, and left keystrokes to automatically play the game.

Link Verification

Write a program that, given the URL of a web page, will attempt to download every linked page on the page. The program should flag any pages that have a 404 "Not Found" status code and print them out as broken links.


CODING WITH MINECRAFT
BUILD TALLER, FARM FASTER, MINE DEEPER, AND AUTOMATE THE BORING STUFF
AL SWEIGART
CRACKING CODES WITH PYTHON
AN INTRODUCTION TO BUILDING AND BREAKING CIPHERS
AL SWEIGART
INVENT YOUR OWN COMPUTER GAMES WITH PYTHON
4TH EDITION
AL SWEIGART
SCRATCH PROGRAMMING PLAYGROUND
LEARN TO PROGRAM BY MAKING COOL GAMES
AL SWEIGART
Making Games with Python & Pygame
A guide to programming with graphics, animation, and sound
Al Sweigart

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 version 1.26.0. It's important that you install this version because future versions of PyPDF2 may be incompatible with the code. To install it, run `pip install --user PyPDF2==1.26.0` 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 a result, 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 15-1.

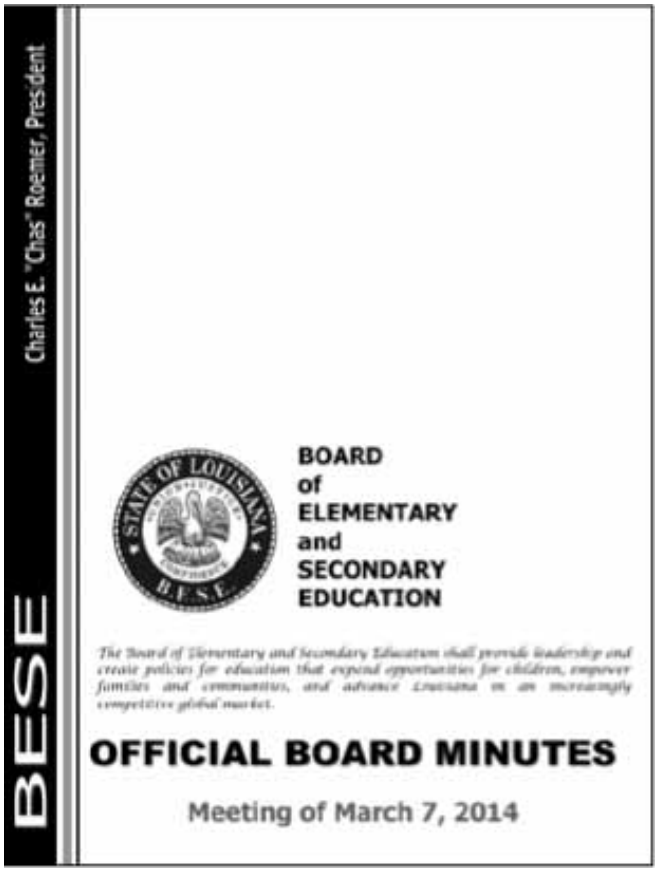


Figure 15-1: The PDF page that we will be extracting text from

Download this PDF from *https://nostarch.com/automatestuff2/* 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 '
>>> pdfFileObj.close()
```

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 page 1, 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
>>> pdfReader = PyPDF2.PdfFileReader(open('encrypted.pdf', 'rb'))
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.

> **NOTE**
>
> *Due to a bug in PyPDF2 version 1.26.0, calling getPage() on an encrypted PDF before calling decrypt() on it causes future getPage() calls to fail with the following error: IndexError: list index out of range. This is why our example reopened the file with a new PdfFileReader object.*

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 is PdfFileWriter, 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 *https://nostarch.com/automatestuff2/* 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):
        ❹ pageObj = pdf2Reader.getPage(pageNum)
        ❺ pdfWriter.addPage(pageObj)

❻ >>> 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 shown in

Figure 15-2. The return values from rotateClockwise() and rotateCounterClockwise() contain a lot of information that you can ignore.

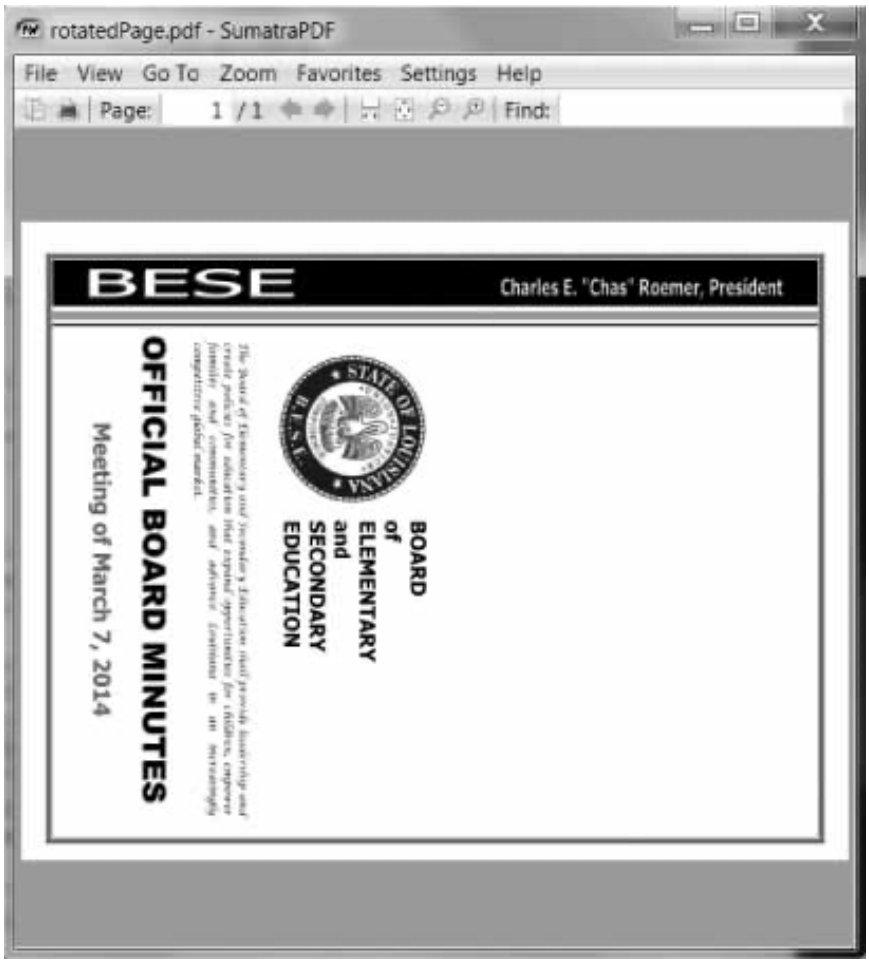


Figure 15-2: The rotatedPage.pdf *file with the page rotated 90 degrees clockwise*

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 *https://nostarch.com/automatestuff2/* 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')
❶ >>> 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 15-3 shows the results. Our new PDF, *watermarkedCover.pdf*, has all the contents of the *meetingminutes.pdf*, and the first page is watermarked.

Figure 15-3: 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:

1. Find all PDF files in the current working directory.
2. Sort the filenames so the PDFs are added in order.
3. 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:

1. Call os.listdir() to find all the files in the working directory and remove any non-PDF files.
2. Call Python's sort() list method to alphabetize the filenames.
3. Create a PdfFileWriter object for the output PDF.
4. Loop over each PDF file, creating a PdfFileReader object for it.

5. Loop over each page (except the first) in each PDF file.
6. Add the pages to the output PDF.
7. Write the output PDF to a file named *allminutes.pdf*.

For this project, open a new file editor tab 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 docx module. You can install the module by running pip install --user -U python-docx==0.8.10. (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 package name docx is for a different module that this book does not cover. However, when you are going to import the module from the python-docx package, you'll need to run import docx, not import python-docx.*

If you don't have Word, LibreOffice Writer and OpenOffice Writer are free alternative applications for Windows, macOS, and Linux that can be used to open *.docx* files. You can download them from *https://www.libreoffice.org/* and *https://openoffice.org/*, respectively. The full documentation for Python-Docx is available at *https://python-docx.readthedocs.io/*. Although there is a version of Word for macOS, 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 15-4 has four runs.

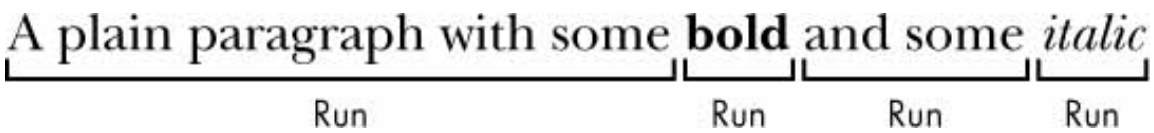


Figure 15-4: The Run objects identified in a Paragraph object

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 docx module. Download *demo.docx* from *https://nostarch.com/automatestuff2/* 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 changes 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 tab 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 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 15-5. On macOS, you can view the Styles pane by clicking the **View ▸ Styles** menu item.

Figure 15-5: 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'	'Heading 5'	'List Bullet'	'List Paragraph'
'Body Text'	'Heading 6'	'List Bullet 2'	'MacroText'
'Body Text 2'	'Heading 7'	'List Bullet 3'	'No Spacing'
'Body Text 3'	'Heading 8'	'List Continue'	'Quote'
'Caption'	'Heading 9'	'List Continue 2'	'Subtitle'
'Heading 1'	'Intense Quote'	'List Continue 3'	'TOC Heading'
'Heading 2'	'List'	'List Number '	'Title'
'Heading 3'	'List 2'	'List Number 2'	
'Heading 4'	'List 3'	'List Number 3'	

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 = 'Quote Char'.

In the current version of Python-Docx (0.8.10), 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 15-6 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.

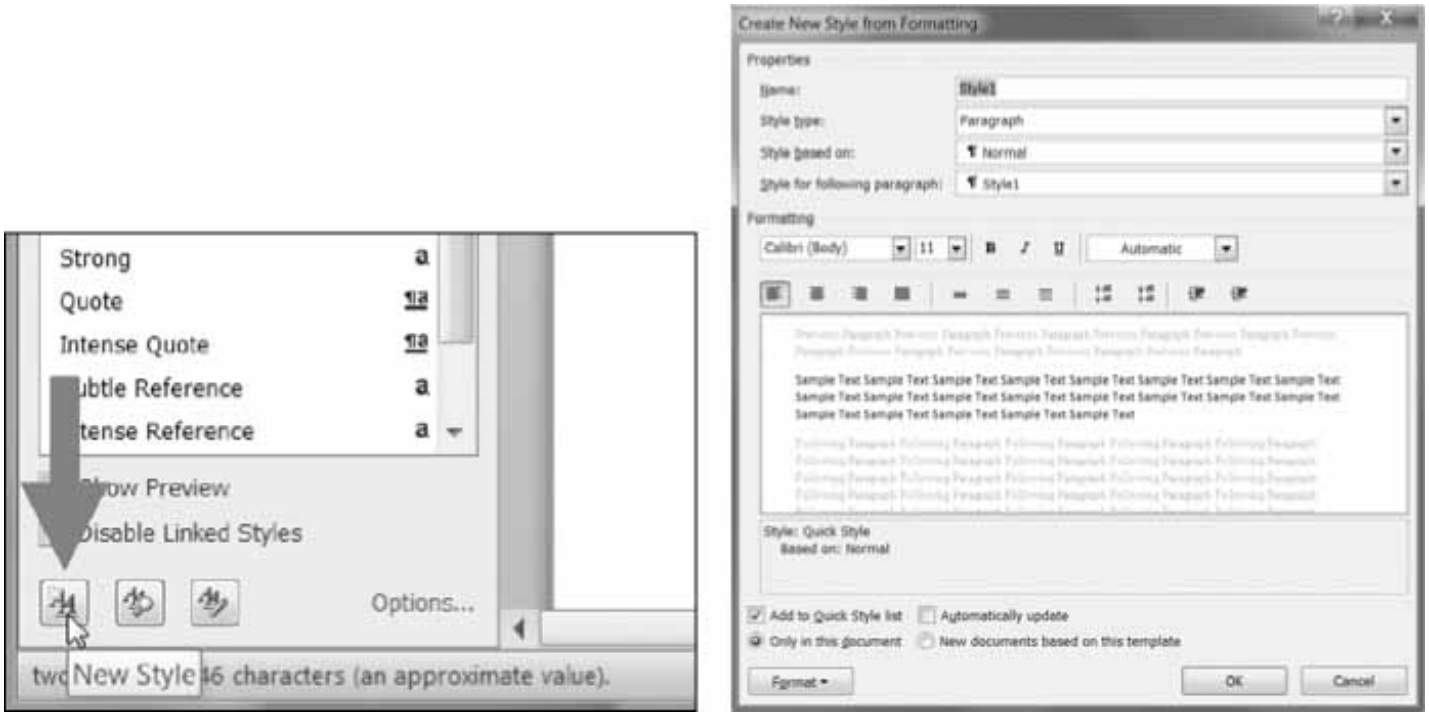


Figure 15-6: The New Style button (left) and the Create New Style from Formatting dialog (right)

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 15-1 lists the text attributes that can be set on Run objects.

Table 15-1: Run Object text Attributes

Attribute	Description
bold	The text appears in bold.
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:

```
>>> import docx
>>> doc = docx.Document('demo.docx')
>>> doc.paragraphs[0].text
'Document Title'
>>> doc.paragraphs[0].style # The exact id may be different:
_ParagraphStyle('Title') id: 3095631007984
```

```
>>> 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 individually. 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 15-7 shows how the styles of paragraphs and runs look in *restyled.docx*.

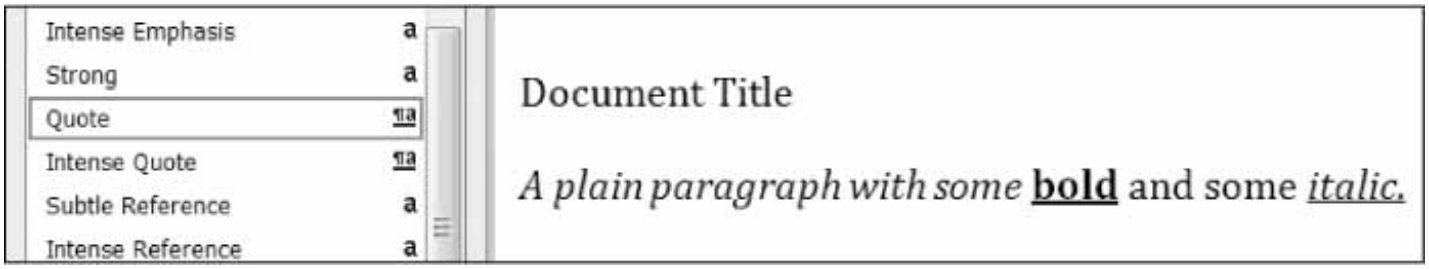


Figure 15-7: The restyled.docx *file*

You can find more complete documentation on Python-Docx's use of styles at *https://python-docx.readthedocs.io/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 15-8.

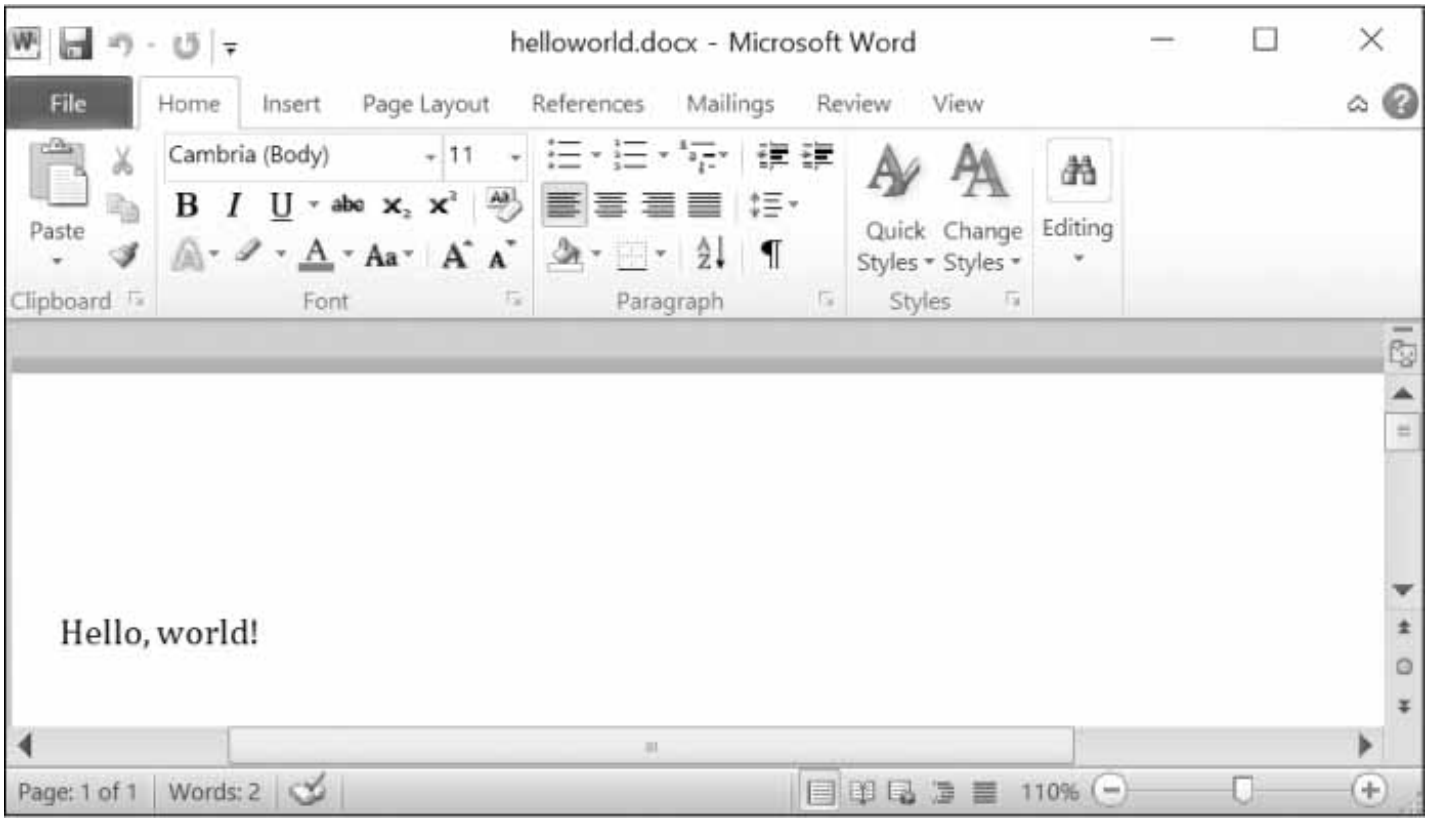


Figure 15-8: The Word document created using add_paragraph('Hello, world!')

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 15-9. 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 0.8.10, 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.

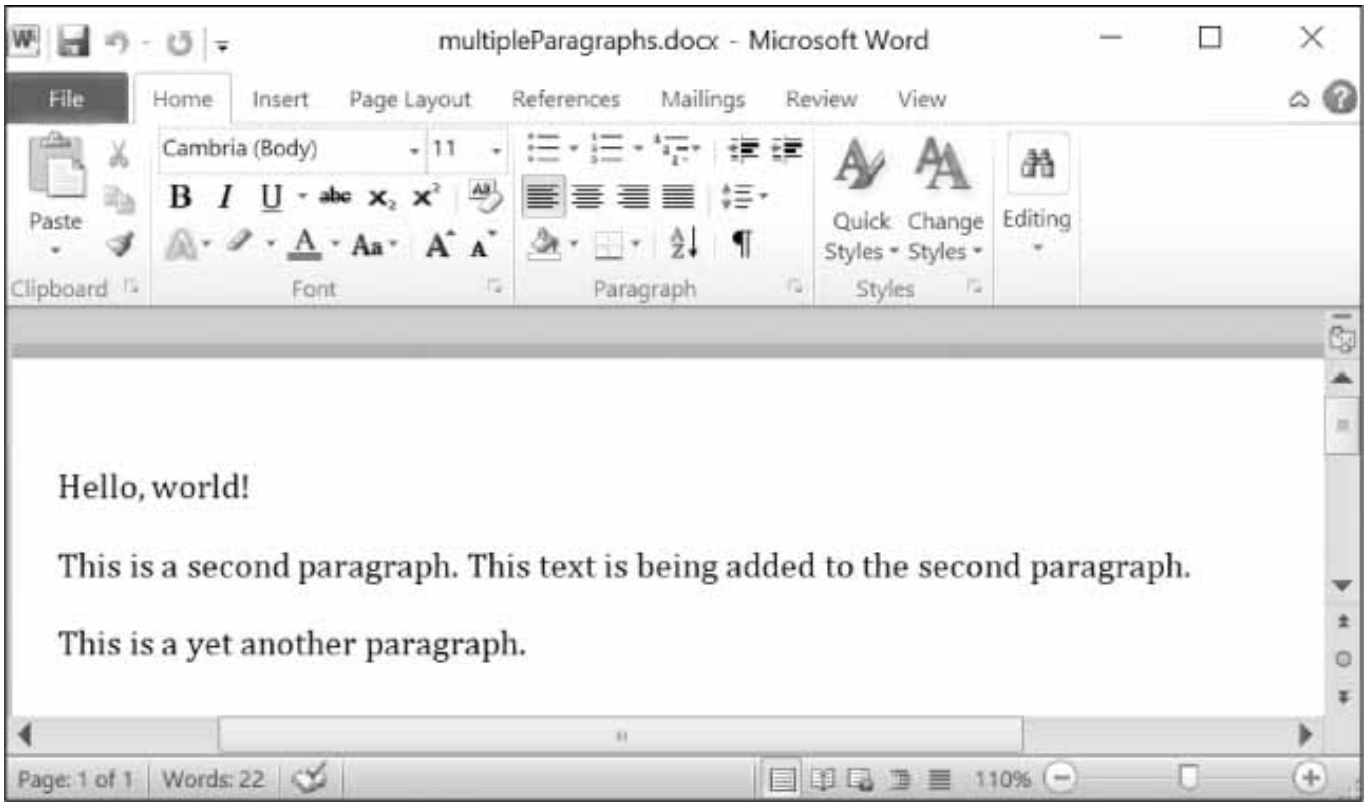


Figure 15-9: The document with multiple Paragraph and Run objects added

Both add_paragraph() and add_run() accept an optional second argument that is a string of the Paragraph or Run object's style. Here's an 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 15-10.

Figure 15-10: 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.enum.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.enum.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.

Creating PDFs from Word Documents

The PyPDF2 module doesn't allow you to create PDF documents directly, but there's a way to generate PDF files with Python if you're on Windows and have Microsoft Word installed. You'll need to install the Pywin32 package by running `pip install --user -U pywin32==224`. With this and the `docx` module, you can create Word documents and then convert them to PDFs with the following script.

Open a new file editor tab, enter the following code, and save it as *convertWordToPDF.py*:

```
# This script runs on Windows only, and you must have Word installed.
import win32com.client # install with "pip install pywin32==224"
import docx
wordFilename = 'your_word_document.docx'
pdfFilename = 'your_pdf_filename.pdf'

doc = docx.Document()
# Code to create Word document goes here.
doc.save(wordFilename)

wdFormatPDF = 17 # Word's numeric code for PDFs.
wordObj = win32com.client.Dispatch('Word.Application')

docObj = wordObj.Documents.Open(wordFilename)
docObj.SaveAs(pdfFilename, FileFormat=wdFormatPDF)
docObj.Close()
wordObj.Quit()
```

To write a program that produces PDFs with your own content, you must use the `docx` module to create a Word document, then use the Pywin32 package's `win32com.client` module to convert it to a PDF. Replace the `# Code to create Word document goes here.` comment

with docx function calls to create your own content for the PDF in a Word document.

This may seem like a convoluted way to produce PDFs, but as it turns out, professional software solutions are often just as complicated.

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 package's 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.

Practice Questions

1. A string value of the PDF filename is *not* passed to the PyPDF2.PdfFileReader() function. What do you pass to the function instead?
2. What modes do the File objects for PdfFileReader() and PdfFileWriter() need to be opened in?
3. How do you acquire a Page object for page 5 from a PdfFileReader object?

4. What PdfFileReader variable stores the number of pages in the PDF document?
5. If a PdfFileReader object's PDF is encrypted with the password swordfish, what must you do before you can obtain Page objects from it?
6. What methods do you use to rotate a page?
7. What method returns a Document object for a file named *demo.docx*?
8. What is the difference between a Paragraph object and a Run object?
9. How do you obtain a list of Paragraph objects for a Document object that's stored in a variable named doc?
10. What type of object has bold, underline, italic, strike, and outline variables?
11. What is the difference between setting the bold variable to True, False, or None?
12. How do you create a Document object for a new Word document?
13. How do you add a paragraph with the text 'Hello, there!' to a Document object stored in a variable named doc?
14. What integers represent the levels of headings available in Word documents?

Practice Projects

For practice, write programs that do the following.

PDF Paranoia

Using the os.walk() function from Chapter 10, write a script that will go through every PDF in a folder (and its subfolders) and encrypt the PDFs using a password provided on the command line. Save each encrypted PDF with an *_encrypted.pdf* suffix added to the original filename. Before deleting the original file, have the program attempt to read and decrypt the file to ensure that it was encrypted correctly.

Then, write a program that finds all encrypted PDFs in a folder (and its subfolders) and creates a decrypted copy of the PDF using a provided password. If the password is

incorrect, the program should print a message to the user and continue to the next PDF.

Custom Invitations as Word Documents

Say you have a text file of guest names. This *guests.txt* file has one name per line, as follows:

Prof. Plum

Miss Scarlet

Col. Mustard

Al Sweigart

RoboCop

Write a program that would generate a Word document with custom invitations that look like Figure 15-11.

Since Python-Docx can use only those styles that already exist in the Word document, you will have to first add these styles to a blank Word file and then open that file with Python-Docx. There should be one invitation per page in the resulting Word document, so call add_break() to add a page break after the last paragraph of each invitation. This way, you will need to open only one Word document to print all of the invitations at once.

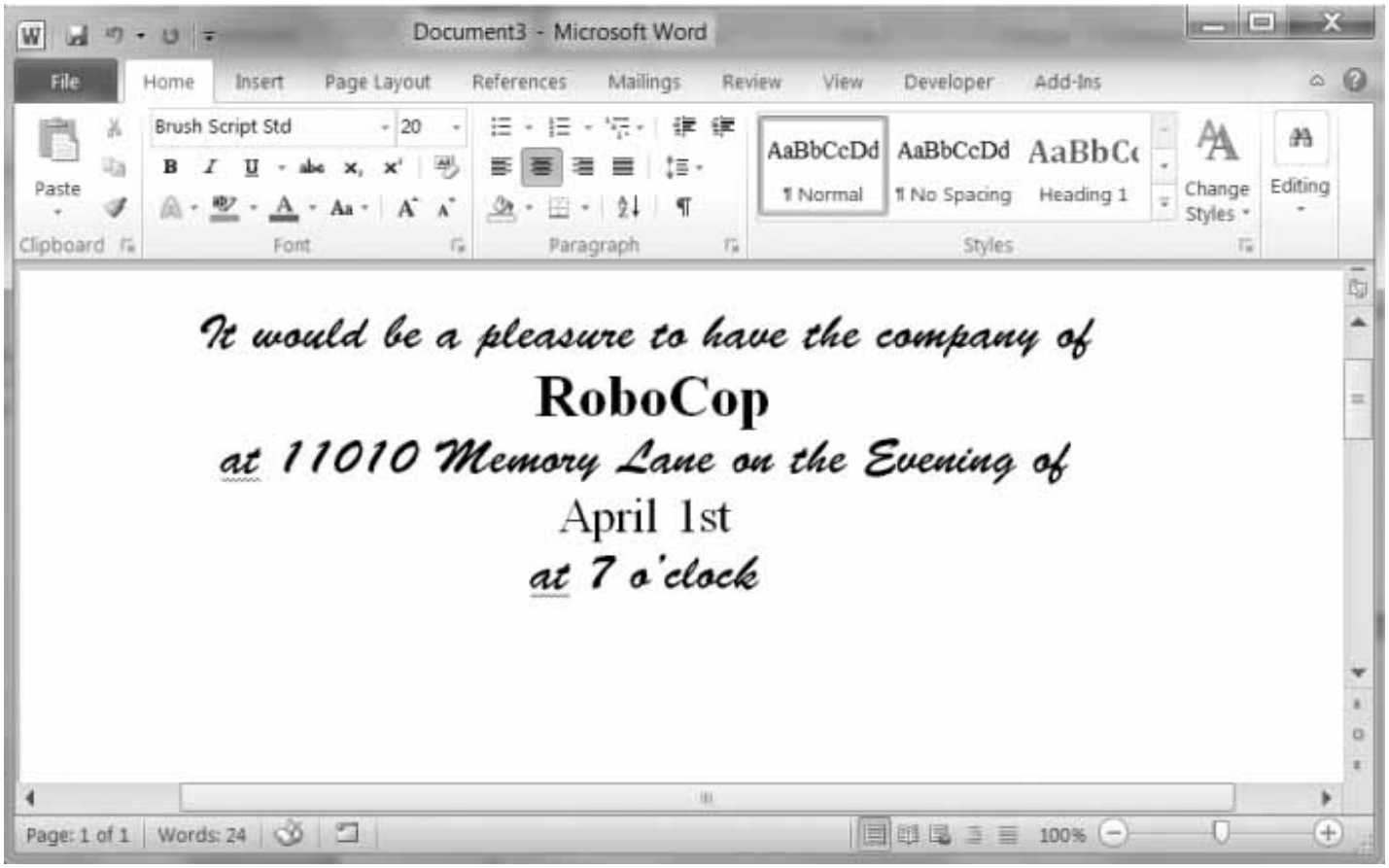


Figure 15-11: The Word document generated by your custom invite script

You can download a sample *guests.txt* file from *https://nostarch.com/automatestuff2/*.

Brute-Force PDF Password Breaker

Say you have an encrypted PDF that you have forgotten the password to, but you remember it was a single English word. Trying to guess your forgotten password is quite a boring task. Instead you can write a program that will decrypt the PDF by trying every possible English word until it finds one that works. This is called a *brute-force password attack*. Download the text file *dictionary.txt* from *https://nostarch.com/automatestuff2/*. This *dictionary file* contains over 44,000 English words with one word per line.

Using the file-reading skills you learned in Chapter 9, create a list of word strings by reading this file. Then loop over each word in this list, passing it to the decrypt() method. If this method returns the integer 0, the password was wrong and your program should continue to the next password. If decrypt() returns 1, then your program should break out of

the loop and print the hacked password. You should try both the uppercase and lowercase form of each word. (On my laptop, going through all 88,000 uppercase and lowercase words from the dictionary file takes a couple of minutes. This is why you shouldn't use a simple English word for your passwords.)

In Chapter 15, you learned how to extract text from PDF and Word documents. These files were in a binary format, which required special Python modules to access their data. CSV and JSON files, on the other hand, are just plaintext files. You can view them in a text editor, such as Mu. But Python also comes with the special csv and json modules, each providing functions to help you work with these file formats.

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 *https://nostarch.com/automatestuff2/* 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 *https://nostarch.com/automatestuff2/* 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 Mu), 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 9. For example, since each cell in a CSV file is separated by a comma, maybe you could just call `split(',')` on each line of text to get the comma-separated values as a list of strings. 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)
❺ >>> 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'
```

As you can see from the output, 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 16-1.

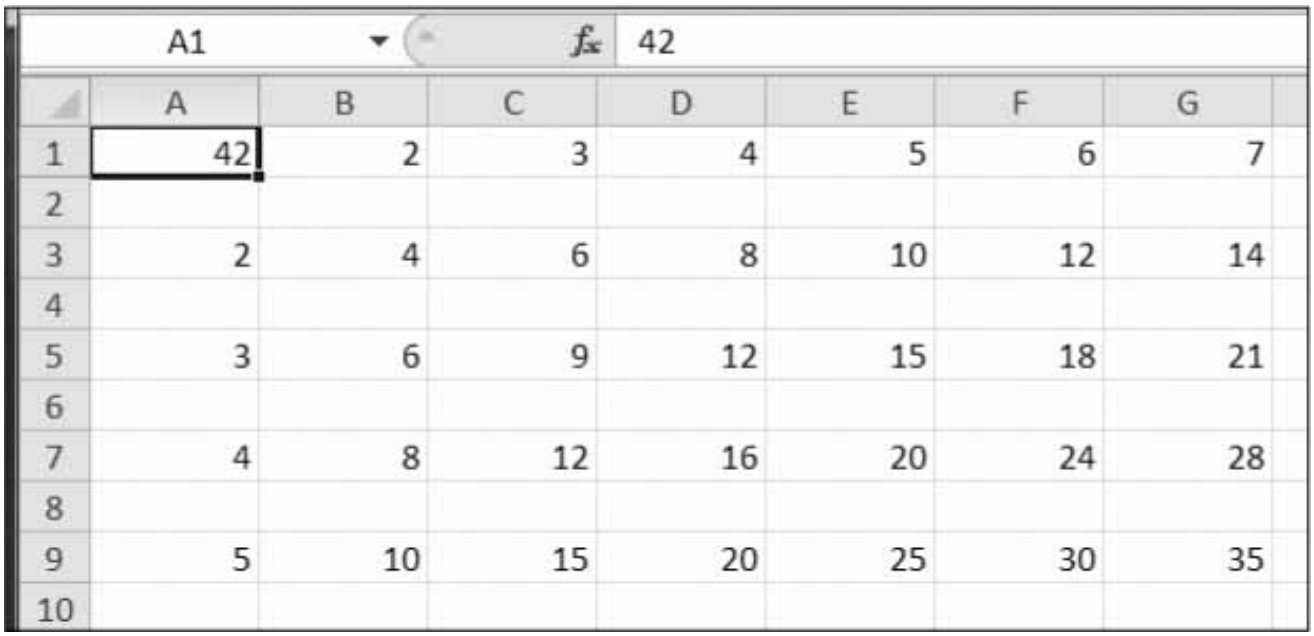


Figure 16-1: If you forget the newline='' keyword argument in open(), the CSV file will be double-spaced.

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 delimiter='\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.

DictReader and DictWriter CSV Objects

For CSV files that contain header rows, it's often more convenient to work with the DictReader and DictWriter objects, rather than the reader and writer objects.

The reader and writer objects read and write to CSV file rows by using lists. The DictReader and DictWriter CSV objects perform the same functions but use dictionaries instead, and they use the first row of the CSV file as the keys of these dictionaries.

Go to *https://nostarch.com/automatestuff2/* and download the *exampleWithHeader.csv* file. This file is the same as *example.csv* except it has Timestamp, Fruit, and Quantity as the column headers in the first row.

To read the file, enter the following into the interactive shell:

```
>>> import csv
>>> exampleFile = open('exampleWithHeader.csv')
>>> exampleDictReader = csv.DictReader(exampleFile)
>>> for row in exampleDictReader:
...     print(row['Timestamp'], row['Fruit'], row['Quantity'])
...
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
```

Inside the loop, DictReader object sets row to a dictionary object with keys derived from the headers in the first row. (Well, technically, it sets row to an OrderedDict object, which you can use in the same way as a dictionary; the difference between them is beyond the scope of this book.) Using a DictReader object means you don't need additional code to skip the first row's header information, since the DictReader object does this for you.

If you tried to use DictReader objects with *example.csv*, which doesn't have column headers in the first row, the DictReader object would use '4/5/2015 13:34', 'Apples', and '73' as the dictionary keys. To avoid this, you can supply the DictReader() function with a second argument containing made-up header names:

```
>>> import csv
>>> exampleFile = open('example.csv')
>>> exampleDictReader = csv.DictReader(exampleFile, ['time', 'name',
'amount'])
>>> for row in exampleDictReader:
...     print(row['time'], row['name'], row['amount'])
...
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
```

Because *example.csv*'s first row doesn't have any text for the heading of each column, we created our own: 'time', 'name', and 'amount'.

DictWriter objects use dictionaries to create CSV files.

```
>>> import csv
```

```
>>> outputFile = open('output.csv', 'w', newline='')
>>> outputDictWriter = csv.DictWriter(outputFile, ['Name', 'Pet', 'Phone'])
>>> outputDictWriter.writeheader()
>>> outputDictWriter.writerow({'Name': 'Alice', 'Pet': 'cat', 'Phone': '555-
1234'})
20
>>> outputDictWriter.writerow({'Name': 'Bob', 'Phone': '555-9999'})
15
>>> outputDictWriter.writerow({'Phone': '555-5555', 'Name': 'Carol', 'Pet':
'dog'})
20
>>> outputFile.close()
```

If you want your file to contain a header row, write that row by calling writeheader(). Otherwise, skip calling writeheader() to omit a header row from the file. You then write each row of the CSV file with a writerow() method call, passing a dictionary that uses the headers as keys and contains the data to write to the file.

The *output.csv* file this code creates looks like this:

```
Name,Pet,Phone
Alice,cat,555-1234
Bob,,555-9999
Carol,dog,555-5555
```

Notice that the order of the key-value pairs in the dictionaries you passed to writerow() doesn't matter: they're written in the order of the keys given to DictWriter(). For example, even though you passed the Phone key and value before the Name and Pet keys and values in the fourth row, the phone number still appeared last in the output.

Notice also that any missing keys, such as 'Pet' in {'Name': 'Bob', 'Phone': '555-9999'}, will simply be empty in the CSV file.

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.

> **WARNING**
>
> *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:

1. Find all the CSV files in the current working directory.
2. Read in the full contents of each file.
3. 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:

1. Loop over a list of files from os.listdir(), skipping the non-CSV files.
2. 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.
3. 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 *https://nostarch.com/automatestuff2/* 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 *https://nostarch.com/automatestuff2/*.

JSON isn’t the only way to format data into a human-readable string. There are many others, including XML (eXtensible Markup Language), TOML (Tom’s Obvious, Minimal Language), YML (Yet another Markup Language), INI (Initialization), or even the outdated ASN.1 (Abstract Syntax Notation One) formats, all of which provide a structure for representing data as human-readable text. This book won’t cover these, because JSON has quickly become the most widely used alternate format, but there are third-party Python modules that readily handle them.

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 12 to download data from the web.

Overall, the program does the following:

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

So the code will need to do the following:

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

For this project, open a new file editor window and save it as *getOpenWeather.py*. Then visit *https://openweathermap.org/api/* in your browser and sign up for a free account to obtain an *API key*, also called an app ID, which for the OpenWeatherMap service is a string code that looks something like '30144aba38018987d84710d0e319281e'. You don't need to pay for this service unless you plan on making more than 60 API calls per minute. Keep the API key secret; anyone who knows it can write scripts that use your account's usage quota.

Step 1: Get Location from the Command Line Argument

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

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

APPID = 'YOUR_APPID_HERE'
```

```
import json, requests, sys

# Compute location from command line arguments.
if len(sys.argv) < 2:
    print('Usage: getOpenWeather.py city_name, 2-letter_country_code')
    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. The `APPID` variable should be set to the API key for your account. Without this key, your requests to the weather service will fail. 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.

The OpenWeatherMap service requires that the query be formatted as the city name, a comma, and a two-letter country code (like "US" for the United States). You can find a list of these codes at *https://en.wikipedia.org/wiki/ISO_3166-1_alpha-2*. Our script displays the weather for the first city listed in the retrieved JSON text. Unfortunately, cities that share a name, like Portland, Oregon, and Portland, Maine, will both be included, though the JSON text will include longitude and latitude information to differentiate between the cities.

Command line arguments are split on spaces. The command line argument `San Francisco, US` would make `sys.argv` hold `['getOpenWeather.py', 'San', 'Francisco,', 'US']`. 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. First you must sign up for a free API key on the site. (This key is used to limit how frequently you make requests on their server, to keep their bandwidth costs down.) Your program simply has to download the page at *https://api.openweathermap.org/data/2.5/forecast/daily?q=<Location>&cnt=3&APPID=<APIkey>*, where *<Location>* is the name of the city whose weather you want and *<API key>* is your personal API key. Add the following to *getOpenWeather.py*.

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

--snip--

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

# Uncomment to see the raw JSON text:
#print(response.text)

# 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 *https://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 *getOpenWeather.py*.

```
! python3
# getOpenWeather.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 getOpenWeather.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 17 covers scheduling, and Chapter 18 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.

In Chapter 18, you'll break away from data formats and learn how to make your programs communicate with you by sending emails and text messages.

Practice Questions

1. What are some features Excel spreadsheets have that CSV spread-sheets don't?
2. What do you pass to csv.reader() and csv.writer() to create reader and writer objects?
3. What modes do File objects for reader and writer objects need to be opened in?
4. What method takes a list argument and writes it to a CSV file?
5. What do the delimiter and lineterminator keyword arguments do?
6. What function takes a string of JSON data and returns a Python data structure?
7. What function takes a Python data structure and returns a string of JSON data?

Practice Project

For practice, write a program that does the following.

Excel-to-CSV Converter

Excel can save a spreadsheet to a CSV file with a few mouse clicks, but if you had to

convert hundreds of Excel files to CSVs, it would take hours of clicking. Using the openpyxl module from Chapter 12, write a program that reads all the Excel files in the current working directory and outputs them as CSV files.

A single Excel file might contain multiple sheets; you'll have to create one CSV file per *sheet*. The filenames of the CSV files should be *<excel filename>_<sheet title>.csv*, where *<excel filename>* is the filename of the Excel file without the file extension (for example, 'spam_data', not 'spam_data.xlsx') and *<sheet title>* is the string from the Worksheet object's title variable.

This program will involve many nested for loops. The skeleton of the program will look something like this:

```
for excelFile in os.listdir('.'):
    # Skip non-xlsx files, load the workbook object.
    for sheetName in wb.get_sheet_names():
        # Loop through every sheet in the workbook.
        sheet = wb.get_sheet_by_name(sheetName)

        # Create the CSV filename from the Excel filename and sheet title.
        # Create the csv.writer object for this CSV file.

        # Loop through every row in the sheet.
        for rowNum in range(1, sheet.max_row + 1):
            rowData = []  # append each cell to this list
            # Loop through each cell in the row.
            for colNum in range(1, sheet.max_column + 1):
                # Append each cell's data to rowData.

            # Write the rowData list to the CSV file.

        csvFile.close()
```

Download the ZIP file *excelSpreadsheets.zip* from *https://nostarch.com/automatestuff2/* and unzip the spreadsheets into the same directory as your program. You can use these as the files to test the program on.