11

Web Scraping

In those rare, terrifying moments when I'm without wifi, 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 our programs could go on the Internet as well. *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, we will learn about several modules that make it easy to scrape web pages in Python:

webbrowser comes with Python and can open a browser to a specific page.

Requests is a module for downloading files and web pages from the Internet.

Beautiful Soup is a module for parsing HTML, the format that web pages are written in.

Selenium is a module for launching and controlling a web browser. Selenium 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. Try entering the following into the interactive shell:

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

A web browser tab will open to the URL *http://inventwithpython.com*.

This is about the only thing that the `webbrowser` module can do. Even so, the `open()` function does make some interesting things possible. For example, it's tedious to copy an 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, run the script, and the map will be loaded for us.

What your program does:

[1] The answer is: no.

Gets a street address from the command-line arguments or clipboard.

Open the web browser to the Google Maps page for the address.

This means your code will need to:

Read the command-line arguments from `sys.argv`.

Read the clipboard contents.

Call the `webbrowser.open()` function to open the web browser.

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

Step 1: Figure Out the URL

Name this script *mapIt.py*. Based on the instructions in Appendix B, set it up 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, we need to figure out what URL to use for a given street address. When you load *http://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 a lot of additional 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 just 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

After the program's `#!` shebang line, you'll need to import the `webbrowser` module for launching the browser, and 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)` will evaluate 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 will return 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.

So 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 the following list value:

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

Make your code look like this:

```
#! python3
import webbrowser, sys
if len(sys.argv) > 1:
    # get address from command line
    address = ' '.join(sys.argv[1:])
```

Step 3: Handle the Clipboard Content, Launch the Browser

If there were 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()`. Make your code look like the following (the new code is in bold):

```
#! python3
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)
```

While some of the programs we write will perform huge tasks that save us hours, sometimes we can use a program that conveniently saves us a few seconds each time we perform a common task, such as getting a map of an address. Consider how much quicker it is to display a map now that we have *mapIt.py*:

INSERT TABLE 10-3 HERE

Table 10-1: Getting a map with and without mapit.py.

Manually Getting a Map	Using mapIt.py
Highlight the address. Copy the address. Open the web browser. Go to *http://maps.google.com* Click on the address text field. Paste the address. Press enter.	Highlight the address. Copy the address. Run *mapIt.py*

Ideas for Similar Programs

As long as you have a URL, the `webbrowser` module is perfect for saving the user the step of opening the browser and directing themselves to a website. Other programs that could use this are:

Opening all links on a page in separate browser tabs.

Opening the browser to the URL for your local weather.

Opening 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 like network errors, connection retrying, and data compression. The Requests

module doesn't come with Python, so you'll have to install it first. From the command line, run `pip install requests`. Appendix A has additional details on how to install third-party modules.

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

Next do a simple test to make sure Requests installed itself correctly. Enter the following into the interactive shell:

```
>>> import requests
```

If no error messages show up, then Requests has been successfully installed.

@Al, Greg: Centering this comment because it's kind of a big blocker: the user might get tripped up here (and in previous chapters, using pyperclip) if they have multiple Pythons installed and IDLE is not using the same Python as the system default.

Apple ships Python 2.7 as system Python, bless their hearts, and I have installed Python 3.3 in a virtualenv for this book and am using IDLE with a bundled copy of 3.3 that's separate from the venv.

When I run "pip install –U requests" outside of my venv (since IDLE can't see the venv packages anyway) it gets installed for 2.7 and IDLE cannot see it. As a fallback I can always install Requests to the 3.3 venv and just use the Python interactive shell on the command line from that env. But it means that I can't stay within IDLE as the book context suggests.

@Al so, if you have any suggestions for how to install packages specifically to IDLE's Python location so that IDLE can load them, that might be required reading for anyone else using a Mac. I googled around for a way to get IDLE's Python path, but I'm not seeing it. Thanks :D

Download 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 value*, which contains the response that the web server gave for your request. I'll explain the response value 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('http://www.gutenberg.org/cache/epub/1112/pg1112.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 by Project Gutenberg ❶. You can tell that the request for this web page succeeded by checking the `status_code` attribute of the response value. 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".)

If the request succeeded, the downloaded web page is stored as a string in the response value's `text` variable. This variable holds a very large string of the entire play; the call to `len(res.text)` shows us that it is over 178,000 characters long. The above example uses the `[:250]` slice to only display the first 250 characters.

Let's download another website: the home page of No Starch Press. Enter the following into the interactive shell:

```
>>> res = requests.get('http://www.nostarch.com')
>>> res.status_code == requests.codes.ok
True
>>> res.text[:250]
'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">\n<html
xmlns="http://www.w3.org/1999/xhtml" lang="en"
xml:lang="en">\n\n<head>\n<meta http-equiv="Content-Type" content="text/html;
charset=utf-'
```

Whoa! If you don't know HTML, this looks strange. The words "DOCTYPE" and "w3.org" don't appear anywhere in the web browser when we look at the No Starch Press website, so why is it showing up here? That's because, unlike the plaintext file of *Romeo and Juliet* we downloaded before, the No Starch Press page is formatted with HTML. I'll explain HTML later in this chapter.

Checking for Errors

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

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

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

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

The `raise_for_status()` method call above causes this program to output:

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

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

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 was 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.

BEGIN BOX

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!) *http://autbor.com/joelunicode*

Pragmatic Unicode *http://autbor.com/pragmaticunicode*

END BOX

To write the web page to a file, you can use a `for` loop with the response value's `iter_content()` method. The `iter_content()` method will return "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_context()`:

```
>>> import requests
>>> res = requests.get('http://www.gutenberg.org/cache/epub/1112/pg1112.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 file *RomeoAndJuliet.txt* will now exist in the current working directory. Note that while the filename on the website was *pg1112.txt*, the file on your hard drive has a different filename. The Requests module simply handles downloading the contents of web pages. Once the page is downloaded, it is simply data in your program. Even if you were to lose your Internet connection after downloading the web page, all the page data would still be on your computer.

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

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

Call `requests.get()` to download the file.

Call `open()` with `'wb'` to create a new file in write binary mode.

Loop over the response value's `iter_content()` method.

Call `write()` on each iteration to write the content to the file.

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 Requests doesn't eat up too much memory even if you download massive files. You can learn about the Requests module's other features from *http://requests.readthedocs.org*.

HTML

Before we start picking apart web pages, first let's cover some HTML basics. I'll also show you how to access your web browser's powerful Developer Tools, which will make the job of scraping information from the web much easier.

Resources for Learning HTML

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

http://htmldog.com/guides/html/beginner/

http://www.codecademy.com/tracks/web

https://developer.mozilla.org/en-US/learn/html

A Quick Refresher

In case it's been a while since you've looked at any HTML, here's a quick overview of the basics. An HTML file is a plaintext file (much like a Python script) 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 in 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!
```

The above HTML will look like this in a browser:

INSERT AUTOMATE10_HTMLHELLOWORLD1.PNG (AND ADD BLACK OUTLINE)

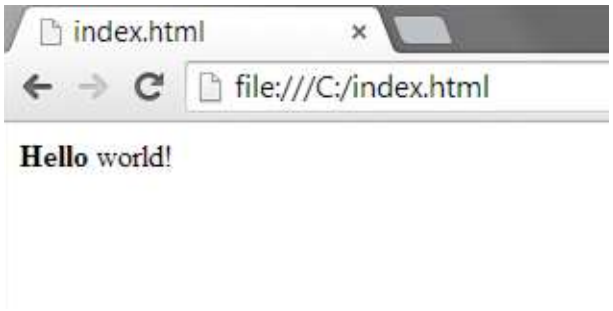

Figure 10-1: The "Hello world!" HTML rendered in the browser.

The opening `<strong>` tag says that the enclosed text will appear bolded and strongly-emphasized. The closing `</strong>` 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. For example:

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

The above HTML will look like this in a browser:

INSERT AUTOMATE10_HTMLHELLOWORLD2.PNG (AND ADD BLACK OUTLINE)

Figure 10-2: The link HTML rendered in the browser.

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

Viewing the Source HTML of a Web Page

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

INSERT AUTOMATE10_VIEWSOURCEMENU.PNG AND AUTOMATE10_VIEWSOURCE.PNG

Figure 10-1: 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, your browser also has *Developer Tools* that make it easy to look through a page's HTML. On Chrome and Internet Explorer, the Developer Tools are already installed and you can press F12 to make it appear. Pressing F12 again will make the Developer Tools disappear.

On Firefox, the Web Developer Tools's Inspector is brought up with Ctrl-Shift-C on Windows and Linux, or Command-Option-C on OS X. The layout of Firebug is almost identical to Chrome's Developer Tools.

On 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 then right-click on 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 we begin to parse HTML for our web scraping programs.

INSERT AUTOMATE10_DEVTOOLS.PNG

Figure 10-2: The Developer Tools window in the Chrome browser.

BEGIN BOX

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 would advise you against it. There are many different ways that HTML can be formatted and still be considered valid HTML, but trying to capture all these possible variations

in a regular expression can be tedious and error-prone. A module developed specifically for parsing HTML, such as Beautiful Soup, will be less likely to result in bugs.

An extended argument for why not to parse HTML with regular expressions can be found at *http://autbor.com/htmlregex*.

END BOX

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 that you're interested in.

This is where the browser's Developer Tools can help. Say you want to write a program to pull weather forecast data from *http://weather.gov*. The first thing to do is a little research, before writing any code. If you visit the site and search for the 94105 zip code, it'll take you to a page showing the forecast for that area.

What if you're interested in scraping the temperature information? Right-click on where it is on the page (or CTRL-click on OS X) and select **Inspect Element** from the context menu that appears. This will bring up the Developer Tools window, and shows you the HTML that produces this particular part of the web page. Figure 10-3 shows the Developer Tools open to the HTML of the temperature:

INSERT AUTOMATE10_WEATHERGOV.PNG

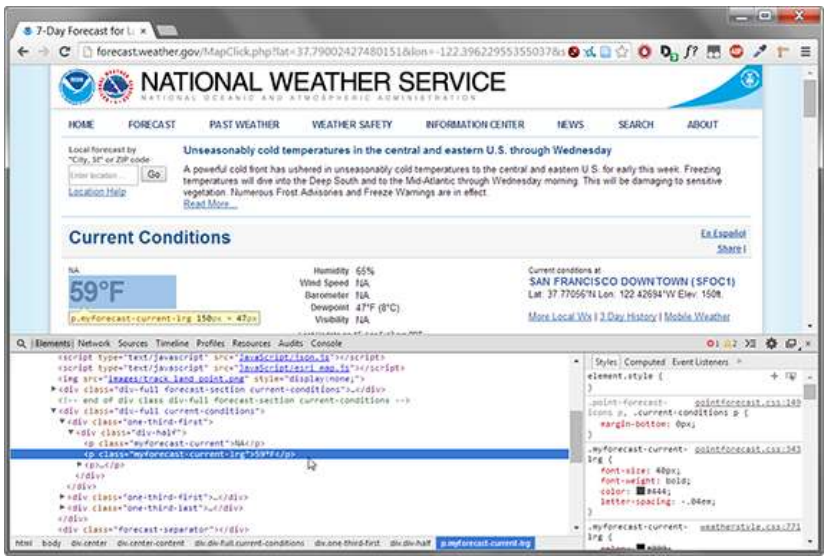

Figure 10-3: Inspecting the element that holds the temperature text with Developer Tools.

From the Developer Tools, you can see that the HTML responsible for the part of the web page with the temperature is `<p class="myforecast-current-lrg">57°F</p>`. This is exactly what you were looking for! It seems that the temperature information in contained inside a `<p>` element with the `myforecast-current-lrg` class. Now that you know what you're looking for, the Beautiful Soup module will help you find it in the string.

The Beautiful Soup 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*), so to install it you will need to run `pip install beautifulsoup4` from the command line. Check out Appendix A for instructions on installing third-party modules. While `beautifulsoup4` is the name used for installation, to import Beautiful Soup you run `import bs4`.

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

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

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

Creating a Soup Value from HTML

First, the `bs4.BeautifulSoup()` function needs to be called with a string containing the HTML it will parse. The value that `bs4.BeautifulSoup()` returns is of the *BeautifulSoup data type*, which we will call a *soup value* for short in this book. Enter the following into the interactive shell while your computer is connected to the Internet:

```
>>> import requests, bs4
```

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

<class 'bs4.BeautifulSoup'>The above 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 soup value that it returns is stored in a variable named `noStarchSoup`.

You can also load an HTML file off of your hard drive by passing a File object to `bs4.BeautifulSoup()`. Enter the following into the interactive shell (you can obtain a copy of the *example.html* file from *http://autbor.com/example.html*, and it must be in the working directory):

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

Once you have a soup value, 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 soup value 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 at *http://autbot.com/selectors*), but in the meantime, here's an incredibly short introduction to selectors. The names `spam`, `eggs`, `bacon`, and `ham` below aren't real HTML elements—they're just placeholder names for these examples.

PROD: WE'LL NEED TO FILL IN THE URL ABOVE.

INSERT TABLE 10-2 HERE

Table 10-2: Examples of CSS Selectors

Selector Passed to select() Method	Will Match
`soup.select('div')`	All elements named `<div>`.
`soup.select('#author)`	The element with an `id` attribute of `author`.
`soup.select('.notice')`	All elements that use a CSS class named `notice`.
`soup.select('div span')`	All elements named `<span>` that are within an element named `<div>`.
`soup.select('div > span')`	All elements named `<span>` that are *directly* within an element named `<div>`, with no other element in between.
`soup.select('input[name]')`	All elements named `<input>` that have a `name` attribute with any value.
`soup.select('input[type="button"]')`	All elements named `<input>` that have an attribute named `type` with value `button`.

The various selector patterns can be combined together to make very 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 of a `<p>` element. There are a few other CSS selector patterns, but the ones in Table 10-2 are the most common.

The `select()` method will return a list of *tag values*, which is how Beautiful Soup represents an HTML element. The list will contain one tag value for every match in the soup value'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 which shows all the HTML attributes of the tag as a dictionary. Using the *example.html* example file from before, try entering the following example into the interactive shell, which will pull the element with `id="author"` out of the HTML:

```
>>> import bs4
>>> exampleFile = open('example.html')
>>> exampleSoup = bs4.BeautifulSoup(exampleFile.read())
>>> elems = exampleSoup.select('#author')
>>> type(elems)
```

```
<class 'list'>
>>> len(elems)
1
>>> type(elems[0])
<class 'bs4.element.Tag'>
>>> elems[0].getText()
'Al Sweigart'
>>> str(elems[0])
'<span id="author">Al Sweigart</span>'
>>> elems[0].attrs
{'id': 'author'}
```

Recall that the text (also called inner HTML) is the content in between starting and closing tags. The `getText()` method will return the element's text, while passing it to `str()` will return a string with the starting and closing tags along with the element's text.

We can also pull all the `<p>` elements from the soup value:

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

Getting Data from an Element's Attributes

The `get()` method for tag values 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. Tag values also have an `attrs` attribute, which contains a dictionary of all the element's attribute names and values. Using *example.html*, try entering the following into the interactive shell:

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

Prettifying HTML

Beautiful Soup's `prettify()` method on soup values will return a string of "prettified" HTML. The prettified text will have elements' opening and closing tags on separate lines with proper indentation. This is useful if you'd like to make your HTML a bit more readable. Consider the following *prettifyAllHtml.py* program, which will open every *.html* file in the current working folder and create a prettified version of it:

```
#! python3
# Prettifies all the .html files in the cwd
import os, bs4

for filename in os.listdir('.'):
    if not filename.endswith('.html'):
        continue # skip if not an .html file

    # Read in the contents of the file.
    htmlFile = open(filename)
    content = htmlFile.read()
    htmlFile.close()

    # Create a new file.
    prettyFile = open(filename[:-5] + '_pretty.html', 'w')
    soup = bs4.BeautifulSoup(open(filename))
    prettyFile.write(soup.prettify())
    prettyFile.close()
```

Project: "I'm Feeling Lucky" Google Search

Whenever I search a topic on Google, I don't just look at one search result at a time. By middle-clicking a search result link (or clicking while holding CTRL), I like to open the first several links in a bunch of new tabs to read later. I search Google often enough that this workflow—opening up 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 up a browser with all the top search results in new tabs. Let's write a script to do this.

What your program does:

Gets search keywords from the command-line arguments.

Retrieves the search results page.

Opens a browser tab for each result.

This means your code will need to:

Read the command-line arguments from `sys.argv`.

Fetch the search result page with Requests.

Find the links to each search result.

Call the `webbrowser.open()` function to open the web browser.

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

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

Before coding anything, we first need to know the URL of the search result page. By looking at the browser's address bar after doing a Google search, we can see that the result page has a URL like *https://www.google.com/search?q=SEARCH_TERM_HERE*. The Requests module can download this page, and then we can use Beautiful Soup to find the search result links in the HTML. Finally, we'll use the `webbrowser` module to open those links in browser tabs.

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`. Make your code look like the following:

```
#! python3
# Open several Google search results.
```

```
import requests, sys, webbrowser, bs4

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

Step 2: Find All the Results

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

After doing a Google search for "Beautiful Soup", we can open the browser's Developer Tools and inspect some of the link elements on the page. They look incredibly complicated, something like this: `Beautiful Soup: We called him Tortoise because he taught us.`

PROD: THE ABOVE LINK IS MEANT TO BE LONG AND COMPLICATED, BUT IF IT'S GOING TO LOOK WEIRD IN PRINT AS PRESENTLY FORMATTED, WE COULD PROBABLY PUT IT IN A CODE LISTING.

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

If we look up a little from the `<a>` element, though, there is an element like this: `<h3 class="r">`. Looking through the rest of the HTML source, it looks like the `r` class is only used for search result links. We don't have to know what the CSS class `r` is or what it does. We're just going to use it as a marker for the `<a>` element we are looking for. We can create a soup value

from the downloaded page's HTML text, and then use the selector `'.r a'` to find all `<a>` elements that are within an element that has the `r` CSS class.

Make your code look like the following (the new code is in bold):

```
#! python3
# Open several google search results.

import requests, sys, webbrowser, bs4

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

soup = bs4.BeautifulSoup(res.text)
linkElems = soup.select('.r a')
```

Step 3: Open Web Browsers for Each Result

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

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

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

Make your code look like the following (the new code is in bold):

```
#! python3
# Open several google search results.
```

```
import requests, sys, webbrowser, bs4

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

soup = bs4.BeautifulSoup(res.text)
linkElems = soup.select('.r a')
numOpen = min(5, len(linkElems))
for i in range(numOpen):
    webbrowser.open('http://google.com' + linkElems[i].get('href'))
```

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

Ideas for Similar Programs

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

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

Project: Download all XKCD Comics

Blogs and other regularly updating websites usually have a front page with the most recent post, and 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 very 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: the front page at *xkcd.com* has a "Prev" button which guides the user back through prior comics. Downloading

each comic by hand would take forever, but we can write a script to do this in a couple of minutes.

INSERT AUTOMATE10_XKCD.PNG HERE

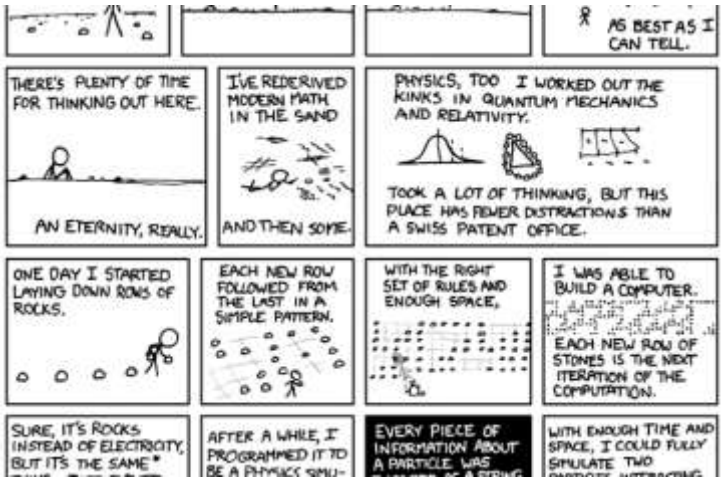

Figure 10-4: XKCD, "a webcomic of romance, sarcasm, math, and language"

What your program does:

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

This means your code will need to:

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

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

Step 1: Designing the Program

If you open the browser's Developer Tools and inspect the elements on the page, you'll find:

- The URL of the comic's image file is given by the `href` attribute of an `<img>` element.
- The `<img>` element is inside a `<div id="comic">` element.
- The "Prev" button has a `rel` HTML attribute with the value `prev`.

The very first comic's "Prev" button links to the *http://xkcd.com/#* URL, indicating that there are no more previous pages.

These observations suggest a structure for our program: we'll have a `url` variable that starts out with the value `'http://xkcd.com'`, and repeatedly update it (in a `for` loop) with the URL of the current page's "Prev" link. At every step in the loop, we'll download the comic at `url`. We'll know to end the loop when `url` ends with `'#'`.

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

Make your code look like the following:

```
#! python3
# Downloads every single XKCD comic.

import requests, os, bs4

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

    # Find the URL of the comic image.

    # Download the image.

    # Save the image to ./xkcd

    # Get the "Prev" button's url.

print('Done.')
```

Step 2: Download the Web Page

Let's implement the code for the "Download the page" part. First we'll print out `url` so that the user knows which URL the program is about to download; then we'll use the Requests module's

`request.get()` function to download it. As always, we immediately call the response value's `raise_for_status()` method to throw an exception and end our program if something went wrong with the download. Otherwise, we create a soup value from the text of the downloaded page.

Make your code look like the following (the new code is in bold):

```
#! python3
# Downloads every single XKCD comic.

import requests, os, bs4

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

    soup = bs4.BeautifulSoup(res.text)

    # Find the URL of the comic image.

    # Download the image.

    # Save the image to ./xkcd

    # Get the "Prev" button's url.

print('Done.')
```

Step 3: Find and Download the Comic Image

From inspecting *xkcd.com* with our Developer Tools, we know that the `<img>` element for the comic image is inside a `<div>` element with the `id` attribute set to `comic`, so the selector `'#comic img'` will get us the correct `<img>` element from the soup value.

A few XKCD pages have a special content that isn't a simple image file. That's fine: we'll just skip those. If our selector doesn't find any elements, then `soup.select('#comic img')` will return a blank list. In that case, our 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. We can get the `src` attribute from this `<img>` element, and pass it to `requests.get()` to download the comic's image file.

Make your code look like the following (the new code is in bold):

```
#! python3
# Downloads every single XKCD comic.

import requests, os, bs4

...cut for brevity...

    # Find the URL of the comic image.
    comicElem = soup.select('#comic img')
    if comicElem == []:
        print('Could not find comic image.')
    else:
        comicUrl = comicElem[0].get('src')
        # Download the image.
        print('Downloading image %s...' % (comicUrl))
        res = requests.get(comicUrl)
        res.raise_for_status()

    # Save the image to ./xkcd

    # Get the "Prev" button's url.

print('Done.')
```

Step 4: Save the Image and Find the "Prev" Comic

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

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

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

Afterwards, the selector `'a[rel="prev"]'` will identify the `<a>` element with the `rel` attribute set to `prev`, and we 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.

Make your code look like the following (the new code is in bold):

```
#! python3
# Downloads every single XKCD comic.

import requests, os, bs4

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

    # Get the "Prev" button's url.
    prevLink = soup.select('a[rel="prev"]')[0]
```

```
    url = 'http://xkcd.com' + prevLink.get('href')

print('Done.')
```

The output of this program will look like this:

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

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

Ideas for Similar Programs

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

- 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 Beautiful Soup modules are great as long as we can figure out the URL we need to pass to `requests.get()`. However, sometimes this isn't so easy to find. Or perhaps the website we want our program to navigate requires us to log in first. At that point, the Selenium module will give our programs the power to perform such sophisticated tasks.

Controlling the Browser with Selenium

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

The Selenium module can be installed by running `pip install -U selenium` from the command line. Appendix A has more detailed steps on installing third party modules.

Starting up a Selenium-Controlled Browser

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

Importing the modules for Selenium is slightly tricky. Instead of `import selenium`, you will 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. Try entering the following into the interactive shell:

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

You'll notice when `webdriver.Firefox()` gets called, the Firefox web browser will start up. Calling `type()` on the value that `webdriver.Firefox()` reveals it's of the *WebDriver data type*. And calling `browser.get('http://inventwithpython.com')` will direct the browser to *http://inventwithpython.com*. Your browser should look something like Figure 10-5:

Figure 10-5: After calling webdriver.Firefox() and get() in IDLE, the Firefox browser appears.

Finding Elements on the Page

WebDriver values have quite a few methods for finding elements in a page. They are divided into the `find_element` and `find_elements` methods. The `find_element` methods will return a single *WebElement value*, representing the first element on the page that matches your query. The `find_elements` methods will return a list of WebElement values for *every* matching element on the page.

Table 10-3 shows several examples of *find_element* and *find_elements* methods being called on a WebDriver value that's stored in the variable `browser`.

INSERT TABLE 10-3 HERE

Table 10-3: Selenium's WebDriver methods for finding elements.

Method Name	WebElement value (or 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_selectors(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 will raise 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 value, you can find out more information about it by reading the attributes or calling the methods in Table 10-4.

Table 10-4: WebElement attributes and methods.

Attribute or Method	**Description**
`tag_name`	The tag name, such as `'a'` for an `<a>` element.
`get_attribute(`*`Name`*`)`	The value for the element's *Name* attribute.
`text`	The text within the element, such as `'hello'` in `<span>hello</span>`.
`clear()`	For text field or text area elements, clears the text typed in it.
`is_displayed()`	Returns `True` if the element is visible, otherwise

	returns `False`.
`is_enabled()`	For input elements, returns `True` if the element is enabled, otherwise returns `False`.
`is_selected()`	For checkbox or radio button elements, returns `True` if the element is selected, otherwise returns `False`.
`location`	A dictionary with keys `'x'` and `'y'` for the position of the element in the page.

For example, open a new file editor and try entering and running the following program:

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

This program will output:

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

Clicking on the Page

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

```
>>> from selenium import webdriver
>>> browser = webdriver.Firefox()
>>> browser.get('http://inventwithpython.com')
```

```
>>> linkElem = browser.find_element_by_link_text('Read It Online')
>>> type(linkElem)
<class 'selenium.webdriver.remote.webelement.WebElement'>
>>> linkElem.click() # follows the "Read It Online" link
```

The above code opens Firefox to *http://inventwithpython.com*, gets the WebElement value for the `<a>` element with the text "Read It Online", and then simulates clicking on that `<a>` element. Just like if you clicked on the link yourself, the browser will then follow 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('http://gmail.com')
>>> emailElem = browser.find_element_by_id('Email')
>>> emailElem.send_keys('not_my_real_email@gmail.com')
>>> passwordElem = browser.find_element_by_id('Passwd')
>>> passwordElem.send_keys('12345')
>>> passwordElem.submit()
```

As long as Gmail hasn't changed the `id` of the username and password text fields since this book was published, the above 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. (We could have just as easily called `emailElem.submit()` and the code would have done the same thing.)

Sending Special Keys

Just like how escape characters are used for characters that are impossible to type into a string value (such as `\n` for newlines), Selenium has a module for keyboard keys that are impossible to type into a string value. 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 just 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 10-5 gives you a list of the commonly used `Keys` variables.

INSERT TABLE 10-5 HERE.

Table 10-5: Commonly used variables in the `selenium.webdriver.common.keys` module.

Attributes	Meanings
`Keys.DOWN`, `Keys.UP`, `Keys.LEFT`, `Keys.RIGHT`	Keyboard arrow keys.
`Keys.ENTER`, `Keys.RETURN`	The ENTER or RETURN key.
`Keys.HOME`, `Keys.END`, `Keys.PAGE_DOWN`, `Kyes.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. Try entering the following into the interactive shell, and notice how the `send_keys()` calls scroll the page:

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

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

Clicking on Browser Buttons

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

`browser.back()` clicks the Back button.

`browser.forward()` clicks the Forward button.

`browser.refresh()` clicks the Refresh/Reload button.

`browser.quit()` clicks the close window button.

More Information on Selenium

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

Summary

Most boring tasks aren't limited to the files on your computer. Being able to programmatically download web pages will extend your programs to the internet. The Requests module makes downloading straightforward, while knowing about basic HTML concepts and selectors will let you utilize the Beautiful Soup module to its full extent.

But to fully extend the reach of your programs, directly controlling a web browser through the Selenium module will let you automate any web-based tasks. 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

Briefly describe the differences between the `webbrowser`, Requests, Beautiful Soup, and Seleniummodules.

What type of value is returned by `requests.get()`? How can you access the downloaded content as a string value?

What Requests method checks that the download worked?

How can you get the HTTP status code of a Requests response?

How do you save a Requests response to a file?

What is the keyboard shortcut for opening a browser's Developer Tools?

How can you view (in the Developer Tools) the HTML of a specific element on a web page?

What is the CSS selector string that would find the element with an `id` attribute of `main`?

What is the CSS selector string that would find the elements with a CSS class of `highlight`?

What is the CSS selector string that would find all the `<div>` elements inside another `<div>` element?

What is the CSS selector string that would find the `<button>` element with a `value` attribute set to `favorite`?

Say you have a Beautiful Soup tag value stored in the variable `spam`, for the element `<div>Hello world!</div>`. How could you get a string `'Hello world!'` from the tag value?

How can you get all the attributes of a Beautiful Soup tag value stored in a variable named `linkElem`?

What does Beautiful Soup's `prettify()` soup value method do?

Running `import selenium` doesn't work. How do you properly import the Selenium module?

What's the difference between the `find_element` and `find_elements` methods?

What methods do Selenium's WebElement values have for simulating mouse clicks and keyboard keys?

You could call `send_keys(Keys.ENTER)` on the Submit button's WebElement value, but what is an easier way to submit a form with Selenium?

How can you simulate clicking on a browser's Forward, Back, and Refresh buttons with Selenium?

Practice Question Answers

The `webbrowser` module has an `open()` method that will launch a web browser to a specific URL, and that's it. The `requests` module can download files and pages from the web. The `bs4` module parses HTML. Finally, the `selenium` module can launch and control a browser.

The `requests.get()` function returns a response value, which has a `text` attribute that contains the downloaded content as a string.

The `raise_for_status()` method will raise an exception if the download had problems, and does nothing if the download succeeded.

The `status_code` attribute of the response value will contain the HTTP status code.

After opening the new file on your computer in `'wb'` "write binary" mode, use a `for` loop that iterates over the Response value's `iter_content()` method to write out chunks to the file. For example:

```
saveFile = open('filename.html', 'wb')
for chunk in res.iter_content(100000):
    saveFile.write(chunk)
```

F12 will bring up the Developer Tools in Chrome. Ctrl-Shift-C (on Windows and Linux) or Command-Option-C (on OS X) will bring up Developer Tools in Firefox.

Right-click on the element in the page, and select Inspect Element from the menu.

```
'#main'
'.highlight'
'div div'
'button[value="favorite"]'
spam.text
linkElem.attrs
```

The `prettify()` method will return a string of the HTML formatted for readability.

The Selenium module is imported with: `from selenium import webdriver`.

The `find_element` methods will return the first matching element as a WebElement value. The `find_elements` methods will return a list of all matching elements as WebElement values.

The `click()` and `send_keys()` methods will simulate mouse clicks and keyboard keys.

Calling the `submit()` method on any element within a form will submit the form.

The `forward()`, `back()`, and `refresh()` WebDriver value methods simulate these browser buttons.

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

Clipboard Prettifier

Write a program that will get the HTML text currently on the clipboard and replace it with the prettified form of it. This could be useful if you are editing some HTML pages and want a quick way to prettify the HTML. You could copy all the HTML text, run the clipboard prettifier program, and then replace the HTML in your editor by pasting the prettified HTML.

2048

2048 is a simple game where you slide tiles up, down, left, or right with the arrow keys to combine them together. You can actually get a fairly high score by repeatedly sliding up, right, down, and left 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 the every linked page on the page. The program should note any pages that have a 404 “not found” status code and print them out to the user as broken links.