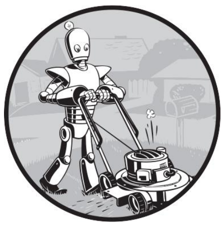
13

WEB SCRAPING



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 social media, or answer the question, "Did Kurtwood Smith have any major roles before he was in the original 1987 *RoboCop*?"[1]

Because so much work on a computer involves going on the internet, it'd be great if your programs could get online. *Web scraping* is a 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'll learn about the following modules, which make it easy to scrape web pages in Python:

`webbrowser` Comes with Python and opens a browser to a specific page

`requests` Downloads files and web pages from the internet

Beautiful Soup (`bs4`) Parses HTML, the format that web pages are written in, to extract the information you want

Selenium Launches and controls a web browser, such as by filling in forms and simulating mouse clicks

Playwright Launches and controls a web browser; newer than Selenium and has some additional features

HTTP and HTTPS

When you visit a website, its web address, such as [*https://autbor.com/example3.html*](https://autbor.com/example3.html), is known as a *uniform resource locator (URL)*. The *HTTPS* in the URL stands for *HyperText Transfer Protocol Secure*, the protocol that your web browser uses to access websites. The packages in this chapter allow your scripts to access web servers through this protocol.

More precisely, HTTPS is an encrypted version of HTTP, so it protects your privacy while you use the internet. If you were using HTTP, identity thieves, national intelligence agencies, and your internet service provider could view the content of the web pages you visited, including any passwords and credit card information you submit. Using a virtual private network (VPN) could keep your internet service provider from viewing your internet traffic; however, now the VPN provider would be able to view your traffic. An unscrupulous VPN provider could then sell information about what websites you visit to data brokers. (Tom Scott discusses what a VPN does and does not provide in his video "This Video Is Sponsored by VPN.")

By contrast, any web page content you view with HTTPS will be encrypted and hidden. Websites used to use HTTPS only for pages that sent passwords and credit card numbers, but nowadays, most websites encrypt all traffic. Keep in mind, though, that the identity of the website you visit can still be known; no one will be able to see exactly what you download from CatPhotos.com, but they will see that you were connecting to the CatPhotos.com website and can figure out that you were probably looking at photos of cats. The Tor Browser, which uses the Tor anonymization network, can provide true anonymous browsing, and you can download it from *https://www.torproject.org/download/*.

Project 6: Run a Program with the webbrowser Module

Let's learn about Python's `webbrowser` module by using it in a programming project. 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 every time you'd like to bring up a map of it on OpenStreetMap. You could eliminate a few steps from this task by writing a simple script to automatically launch the map in your browser using the contents of your clipboard. This way, you'd only have to copy the address to a clipboard and run the script for the map to load for you. We can put the address directly into the OpenStreetMap URL, so all we need is the `webbrowser.open()` function.

This is what your program does:

- Gets a street address from the command line arguments or clipboard
- Opens the web browser to the OpenStreetMap page for that address

This means your code needs to do the following:

- 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 tab and save it as *showmap.py*.

Step 1: Figure Out the URL

By following the instructions in Chapter 12, set up a *showmap.py* file so that when you run it from the command line, like so

```
C:\Users\al> showmap 777 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.

To do so, you need to figure out what URL to use for a given street address. When you load *https://www.openstreetmap.org* in the browser and search for an address, the URL in the address bar looks something like this: *https://www.openstreetmap.org/search?query=777%20Valencia%20St%2C%20San%20Francisco%2C%20CA%2094110#map=19/37.75897/-122.42142*.

We can test that the URL doesn't need the *#map* part by taking it out of the address bar and visiting that site to confirm it still loads properly. So, your program can be set to open a web browser to *https://www.openstreetmap.org/search?query=<your_address_string>* (where *<your_address_string>* is the address you want to map). Note that your browser automatically handles any necessary URL encoding, such as converting space characters in the URL to *%20*.

Step 2: Handle the Command Line Arguments

Make your code look like this:

```
# showmap.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.

# TODO: Open the web browser.
```

First, you 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 the program's filename and command line arguments as a list. 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'll want to interpret all of the arguments as a single string. Because `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 you should pass `sys.argv[1:]` instead of `sys.argv` 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

```
showmap 777 Valencia St, San Francisco, CA
94110
```

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

```
['showmap.py', '777', 'Valencia', 'St, ',
'San', 'Francisco, ', 'CA', '94110']
```

After you've joined `sys.argv[1:]` with a space character, the `address` variable will contain the string `'777 Valencia St, San Francisco, CA 94110'`.

Step 3: Retrieve the Clipboard Content

To fetch the URL from the clipboard, make your code look like the following:

```
# showmap.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()

# Open the web browser.
webbrowser.open('https://
www.openstreetmap.org/search?query=' +
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 OpenStreetMap 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 13-1 compares the steps needed to display a map with and without *showmap.py*.

Table 13-1: Getting a Map with and Without *showmap.py*

Manually getting a map	Using *showmap.py*
1. Highlight the address.	1. Highlight the address.
2. Copy the address.	2. Copy the address.
3. Open the web browser.	3. Run *showmap.py*.
4. Go to *https://www.openstreetmap.org*	
5. Click the address text field.	
6. Paste the address.	
7. Press ENTER.	

We're fortunate that the OpenStreetMap website doesn't require any interaction to get a map; we can just put the address information directly into the URL. The *showmap.py* script makes this task less tedious, especially if you do it frequently.

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 site.
- Open several social networking sites or bookmarked sites that you regularly check.
- Open a local *.html* file on your hard drive.

The last suggestion is useful for displaying help files. While your program could use `print()` to display a help page to the user, calling `webbrowser.open()` to open a *.html* file with help information allows the page to have different fonts, color, tables, and images. Instead of the *https://* prefix, use the *file://* prefix. For example, your *Desktop* folder should have a local *help.html* file at *file:///C:/Users/al/Desktop/ help.html* on Windows or *file:///Users/al/Desktop/ help.html* on macOS.

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 routing, and data compression. The module doesn't come with Python, so you'll have to install it before you can use it by following the instructions in Appendix A.

Downloading Web Pages

The `requests.get()` function takes a string representing a URL to download. By calling `type()` on the function'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 >>> response = requests.get('https://
   automatetheboringstuff.com/files/rj.txt')
   >>> type(response)
   <class 'requests.models.Response'>
❷ >>> response.status_code ==
   requests.codes.ok
   True
   >>> len(response.text)
   178978
   >>> print(response.text[:210])
   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
```

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

If the request succeeded, the downloaded web page is stored as a string in the Response object's text variable. This large string consists of the entire play; the call to len(response.text) shows you that it's more than 178,000 characters long. Finally, calling print(response.text[:210]) displays only the first 210 characters.

If the request failed and displayed an error message, like "Failed to establish a new connection" or "Max retries exceeded," 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. Also keep in mind that if you download a web page with requests, you'll get only the HTML content of the web page. You must download images and other media separately.

Checking for Errors

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

```
>>> response = requests.get('https://
inventwithpython.com/
page_that_does_not_exist')
>>> response.raise_for_status()
Traceback (most recent call last):
  File "<python-input-0>", line 1, in
<module>

  File "C:
```

```
\Users\Al\AppData\Local\Programs\Python\Pytho
nXX\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 an easy way to ensure that a program halts if a bad download occurs. Generally, you'll want your program to stop as soon as some unexpected error happens. If a failed download isn't a deal breaker, you can wrap the `raise_for_status()` line with `try` and `except` statements to handle this error case without crashing:

```
import requests
response = requests.get('https://
inventwithpython.com/
page_that_does_not_exist')
try:
    response.raise_for_status()
except Exception as exc:
    print(f'There was a problem: {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 should 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. However, 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
>>> response = requests.get('https://
automatetheboringstuff.com/files/rj.txt')
>>> response.raise_for_status()
>>> with open('RomeoAndJuliet.txt', 'wb') as
play_file:
...     for chunk in
response.iter_content(100000):
...         play_file.write(chunk)
...
100000
78978
```

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* now exists 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 `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 78,978 bytes.

A REVIEW OF FILE DOWNLOADING AND SAVING

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` object's `iter_content()` method.
- Call `write()` on each iteration to write the content to the file.

That's all there is to the `requests` module! You can learn about the module's other features at *https://requests.readthedocs.io/en/latest/*.

If you want to download video files from websites such as YouTube, Facebook, X (formerly Twitter), or other sites, use the `yt-dlp` module instead, covered in Chapter 24.

Accessing a Weather API

The apps you use are designed to interact with human users. However, you can write programs to interact with other programs through their *application programming interface (API)*, which is the specification that defines how one piece of software (such as your Python program) can communicate with another piece of software (such as the web server for a weather site). Online services often have APIs. For example, you could write a Python script to post to your social media accounts or download new photos. In this section, we'll write a script that accesses weather information from the free OpenWeather website.

Almost all online services require you to register an email address to use their API. Even if the API is free, they may have limits to how many API requests you can make per hour or day. If you're worried about receiving spam email, you can use a temporary, disposable email address service such as *https://10minutemail.com*. Keep in mind that you should use such services only to register for online accounts you don't care about, as an unscrupulous email service could take control of your online account by making a password reset request in your name.

To start, sign up for a free account at *https://openweathermap.org*. The free account tier limits you to making 60 API requests per minute.

This is more than enough for your small or medium-sized programming projects. If your program needs more than this limit (say, because it's processing requests from hundreds of simultaneous visitors to your website), you can purchase a paid account tier. Online services will give you an *API key*, which is sort of a password that identifies your account in your API requests. Keep this API key a secret! Anyone with this key can make API requests credited to your account. If you write a program that uses an API key, consider having the program read a text file that contains the key instead of including the API key directly in your source code. This way, you can share your program with others (who can sign up for their own API key) without worrying about exceeding the API request limits of your account.

Many HTTP APIs deliver their responses as one large string. This string is often formatted as JSON or XML. Chapter 18 covers JSON and XML in more detail, but for now, you just need to know that `json.loads(response.text)` returns a Python data structure of lists and dictionaries containing the JSON data in `response.text`. The examples in this chapter store this data in a variable named `response_data`, but this is an arbitrary choice, and you can use any variable name you'd like.

All online services document how to use their API. OpenWeather provides its documentation at *https://openweathermap.org/api*. After you've logged in to your account and obtained your API key from the *My API keys* page at *https://home.openweathermap.org/api_keys*, use it in the following interactive shell code. I'll use `'30ee784a80d81480dab1749d33980112'` as a fake API key in this example. Don't use this fake API key example in your code; it won't work.

First, you'll use OpenWeather to find the latitude and longitude of San Francisco:

```
>>> import requests
>>> city_name = 'San Francisco'
>>> state_code = 'CA'
>>> country_code = 'US'
>>> API_key =
'30ee784a80d81480dab1749d33980112'  # Not a
real API key
>>> response = requests.get(f'https://
api.openweathermap.org/geo/1.0/
direct?q={city_name},{state_code},
{country_code}&appid={API_key}')
>>> response.text  # This is a Python string.
'[{"name":"San Francisco","local_names":
{"id":"San Francisco",
--snip--
,"lat":37.7790262,"lon":-122.419906,"country"
:"US","state":"California"}]'
>>> import json
>>> response_data = json.loads(response.text)
>>> response_data  # This is a Python data
structure.
[{"name":"San Francisco","local_names":
{"id":"San Francisco",
--snip--
,"lat":37.7790262,"lon":-122.419906,"country"
:"US","state":"California"}]
```

To understand the data in the response, you should look at the online API documentation for OpenWeather or examine the dictionary in `response_data` in the interactive shell. You'll learn that the response is a list whose first item (at index `0`) is a dictionary with keys `'lat'` and `'lon'`. The values for these keys are float values of the latitude and longitude:

```
>>> response_data[0]['lat']
37.7790262
>>> response_data[0]['lon']
-122.419906
```

The specific URL used to make an API request is called the *endpoint*. The f-strings in this example replace the parts in curly brackets with the values of variables. The `direct?q={city_name},{state_code},{country_code}&appid={API_key}'` in the previous example becomes `direct?q=San Francisco,CA,US&appid=30ee784a80d81480dab1749d33980112'`.

Next, you can use this latitude and longitude information to find the current temperature of San Francisco:

```
>>> lat = json.loads(response.text)[0]['lat']
>>> lon = json.loads(response.text)[0]['lon']
>>> response = requests.get(f'https://
api.openweathermap.org/data/2.5/
weather?lat={lat}&lon={lon}&appid={API_key}')
>>> response_data = json.loads(response.text)
>>> response_data
{'coord': {'lon': -122.4199, 'lat': 37.779},
'weather': [{'id': 803,
--snip--
'timezone': -25200, 'id': 5391959, 'name':
'San Francisco', 'cod': 200}
>>> response_data['main']['temp']
285.44
>>> round(285.44 - 273.15, 1)  # Convert
Kelvin to Celsius.
12.3
>>> round(285.44 * (9 / 5) - 459.67, 1)  #
Convert Kelvin to Fahrenheit.
54.1
```

Notice that OpenWeather returns the temperature in Kelvin, so you'll need to do some math to get the temperature in Celsius or Fahrenheit.

Let's break down the full URL of the geolocation endpoint from the previous example:

https:// The *scheme* used to access the server, which is the protocol name (almost always HTTPS for online APIs) followed by a colon and two forward slashes.

api.openweathermap.org The domain name of the web server that handles the API request.

/geo/1.0/direct The path of the API.

?q={city_name},{state_code},{country_code}&appid={API_key} The URL's query string. The parts inside curly brackets need to be replaced by real values; you can think of them as parameters for a function call. In URL encoding, the parameter name and argument value are separated by an equal sign, and multiple parameter-argument pairs are separated by an ampersand.

You can take the endpoint URL (with the completed query string) and paste it into your web browser to view the response text directly. This is often a good practice when you're first learning how to use an API. The response text for web-based APIs is most often formatted in JSON or XML.

To avoid confusion when updating an API, most online services include a version number as part of the URL. Over time, a service may release new versions of the API and deprecate older versions. At this point, you'll have to update the code in your scripts to continue to make use of them.

The free tier of OpenWeather also provides five-day forecasts and information about precipitation, wind, and air pollution. The documentation web pages show you what URLs to access to get this data, as well as the structure of the JSON response for these API calls. The code in the next few sections assumes you've run `response_data = json.loads(response.text)` to convert the text returned from the website into a Python data structure.

Requesting a Latitude and Longitude

The endpoint to get the latitude and longitude coordinates of a city is *https://api.openweathermap.org/geo/1.0/direct?q={city_name},{state_code},{country_code}&appid={API_key}*. The state code refers to the state's abbreviation and is required only for cities in the United States. The country code is the two- or three-letter ISO 3166 code, listed at *https://en.wikipedia.org/wiki/List_of_ISO_3166_country_codes*. For example, use the code `'US'` for the United States or `'NZ'` for New Zealand. After converting the response JSON text into a Python data structure in a variable named `response_data`, you can retrieve the following information:

`response_data[0]['lat']` Holds the degrees latitude of the city as a float value

`response_data[0]['lon']` Holds the degrees longitude of the city as a float value

If the city name matches multiple responses, the list in `response_data` will contain different dictionaries at `response_data[0]`, `response_data[1]`, and so on. If

OpenWeather is unable to locate the city, `response_data` will be an empty list.

Fetching the Current Weather

The endpoint to get current weather information based on some latitude and longitude is *https://api.openweathermap.org/data/2.5/weather?lat={lat}&lon={lon}&appid={API_key}*. After converting the response JSON text into a Python data structure in a variable named `response_data`, you can retrieve the following information:

`response_data['weather'][0]['main']` Holds a string description, such as `'Clear'`, `'Rain'`, or `'Snow'`

`response_data['weather'][0]['description']` Holds a more descriptive string, such as `'light rain'`, `'moderate rain'`, or `'extreme rain'`

`response_data['main']['temp']` Holds the current temperature in Kelvin

`response_data['main']['feels_like']` Holds the human perception of the temperature in Kelvin

`response_data['main']['humidity']` Holds the humidity as a percentage

If you supplied an incorrect latitude or longitude argument, `response _data` will be a dictionary, like `{"cod":"400","message":"wrong latitude"}`.

Getting a Weather Forecast

The endpoint to get a five-day forecast based on some latitude and longitude is *https://api.openweathermap.org/data/2.5/forecast?lat={lat}&lon={lon}&appid={API_key}*. After converting the response JSON text into a Python data structure in a variable named `response_data`, you can retrieve the following information:

`response_data['list']` Holds a list of dictionaries containing the weather predictions for a given time.

`response_data['list'][0]['dt']` Holds a timestamp in the form of a Unix epoch float. Pass this value as an argument to `datetime.datetime.fromtimestamp()` to obtain the timestamp as a `datetime` object. Chapter 19 discusses Python's `datetime` module in more detail.

`response_data['list'][0]['main']` Holds a dictionary with keys like `'temp'`, `'feels_like'`, `'humidity'`, and others.

`response_data['list'][0]['weather'][0]` Holds a dictionary of descriptions with keys like `'main'`, `'description'`, and others.

The list in `response_data['list']` holds 40 dictionaries with forecasts at three-hour increments for the next five days, though this may change in future versions of the API.

Exploring APIs

Other websites, such as *https://weather.gov* and *https://www.weatherapi.com/*, provide their own free weather APIs. Every API works differently, but they're often accessed as requests over HTTPS, in which case you can use the Requests library and return responses formatted as JSON or XML text. However, someone may have created a third-party Python package to make using these APIs easier, with Python functions that handle accessing the endpoints and parsing the response for you. You can find these packages on *https://pypi.org*; read the package documentation to learn about their use.

Understanding HTML

Before you pick apart web pages, you must learn some *HyperText Markup Language (HTML)* basics. HTML is the format in which web pages are written, while *Cascading Style Sheets (CSS)* are a way to make categorical changes to the look of HTML elements in a web page. 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/docs/Learn/HTML*
- *https://www.freecodecamp.org/news/html-coding-introduction-course-for-beginners*
- *https://www.khanacademy.org/computing/computer-programming/html-css*

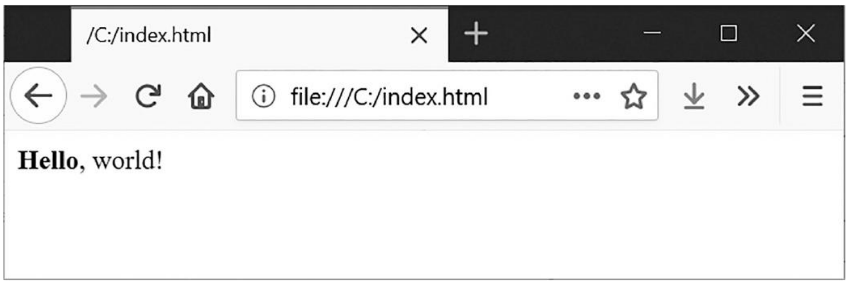
In this section, you'll also learn how to access your web browser's powerful Developer Tools, which make scraping information from the web much easier.

Exploring the Format

An HTML file is a plaintext file with the *.html* file extension. The text in these files is surrounded by HTML *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 HTML *element*. The text to display 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:

```
<b>Hello</b>, world!
```

In a browser, this HTML will look as shown in Figure 13-1.



Hello, world!

Figure 13-1: Hello, world! *rendered in the browser*

The opening `<b>` tag says that the enclosed text will appear in bold. The closing `</b>` tag tells the browser where the end of the bold text is. Together, they form an element: `<b>Hello</b>`.

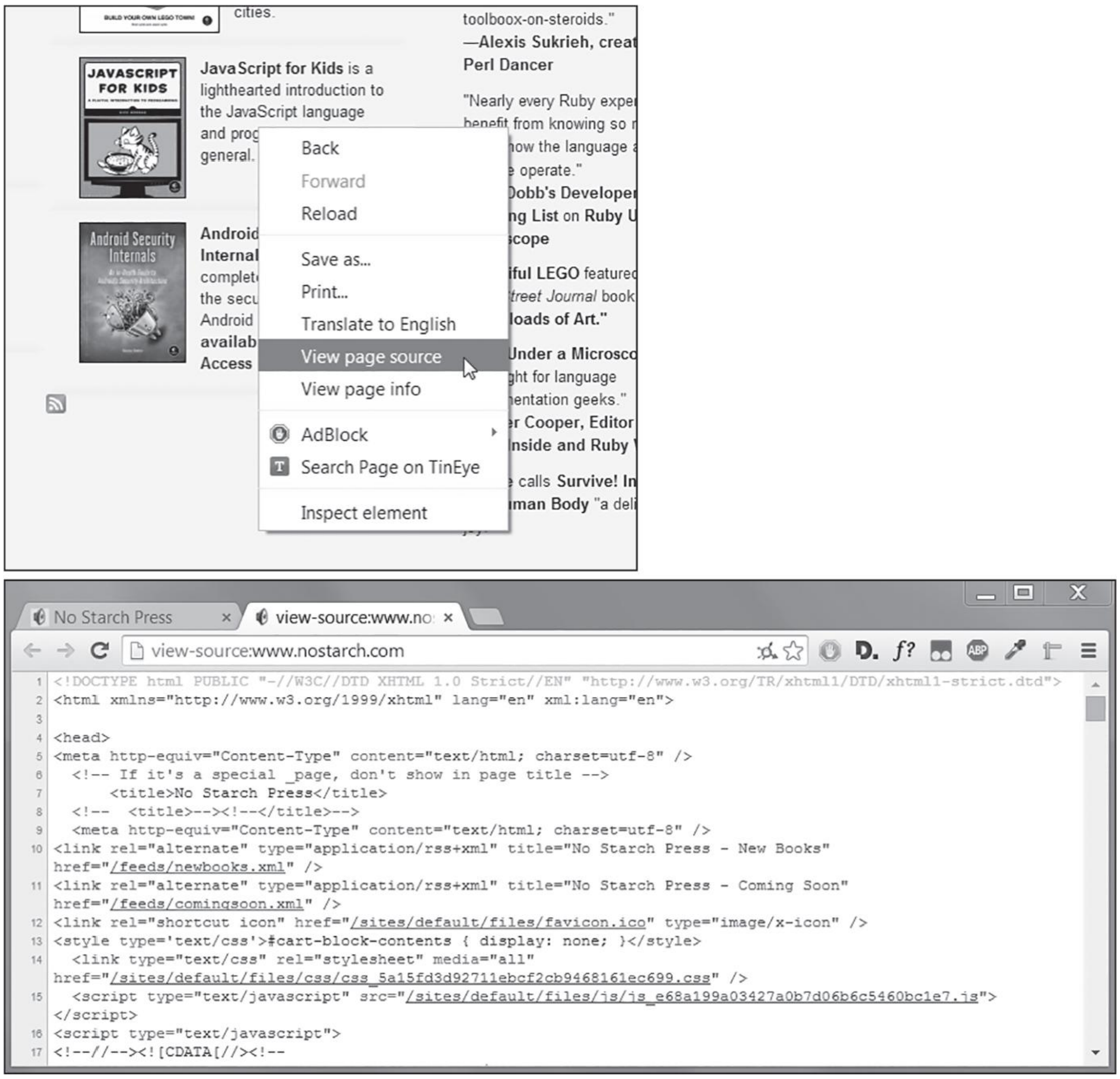
There are many different tags in HTML. Some of these tags have extra properties in the form of *attributes* within the angle brackets. For example, the `<a>` tag encloses text that should be a link, and the `href` attribute determines what URL to link to. Here's an example:

```
<a href="https://inventwithpython.com”>This
text is a link</a>
```

Some elements have an `id` attribute used to uniquely identify the element in the page. You'll often instruct your programs to seek out an element by its `id` attribute, so finding this attribute using the browser's Developer Tools is a common task when writing web scraping programs.

Viewing a Web Page's Source

You'll need to look at the HTML of the web pages your programs will work with, called the *source*. To do this, right-click any web page in your web browser (or CTRL-click it on macOS), and select **View Source** or **View page source** (Figure 13-2). The source is the text your browser actually receives. The browser knows how to display, or *render*, the web page from this HTML.



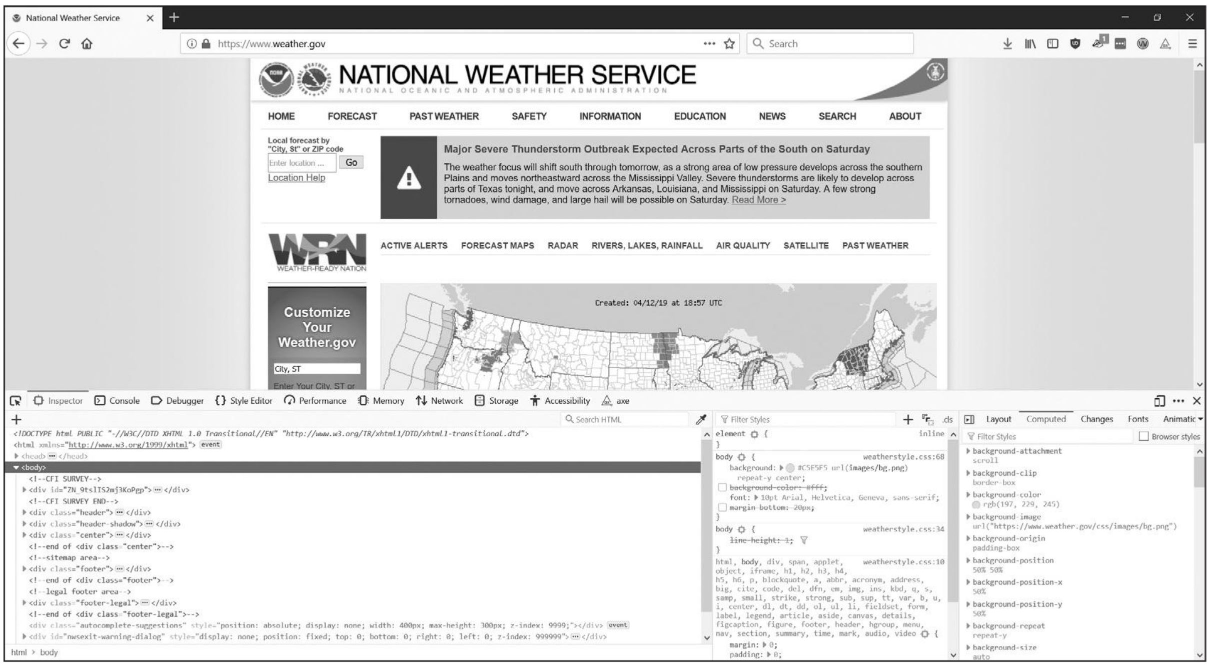
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">

<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <!-- If it's a special _page, don't show in page title -->
      <title>No Starch Press</title>
  <!--  <title>--><!--</title>-->
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<link rel="alternate" type="application/rss+xml" title="No Starch Press - New Books"
href="/feeds/newbooks.xml" />
<link rel="alternate" type="application/rss+xml" title="No Starch Press - Coming Soon"
href="/feeds/comingsoon.xml" />
<link rel="shortcut icon" href="/sites/default/files/favicon.ico" type="image/x-icon" />
<style type='text/css'>#cart-block-contents { display: none; }</style>
  <link type="text/css" rel="stylesheet" media="all"
href="/sites/default/files/css/css_5a15fd3d92711ebcf2cb9468161ec699.css" />
  <script type="text/javascript" src="/sites/default/files/js/js_e68a199a03427a0b7d06b6c5460bc1e7.js">
</script>
<script type="text/javascript">
<!--//--><![CDATA[//><!--
```

Figure 13-2: Viewing the source of a web page

Go ahead and view the source HTML of some of your favorite sites. It’s fine if you don’t fully understand what you’re seeing. You won’t need HTML mastery to write simple web scraping programs. 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 Firefox, Chrome, and Microsoft Edge, you can press F12 to make the tools appear (Figure 13-3). Pressing F12 again will make them disappear.



```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"> event
  <head> ··· </head>
  <body>
    <!--CFI SURVEY-->
    <div id="ZN_9tslIS2mj3KoPgp"> ··· </div>
    <!--CFI SURVEY END-->
    <div class="header"> ··· </div>
    <div class="header-shadow"> ··· </div>
    <div class="center"> ··· </div>
    <!--end of <div class="center">-->
    <!--sitemap area-->
    <div class="footer"> ··· </div>
    <!--end of <div class="footer">-->
    <!--legal footer area-->
    <div class="footer-legal"> ··· </div>
    <!--end of <div class="footer-legal">-->
    <div class="autocomplete-suggestions" style="position: absolute; display: none; width: 400px; max-height: 300px; z-index: 9999;"></div> event
    <div id="nwsexit-warning-dialog" style="display: none; position: fixed; top: 0; bottom: 0; right: 0; left: 0; z-index: 999999"> ··· </div>
html > body
```

```
element {                                              inline
}
body {                                     weatherstyle.css:68
  background: #C5E5F5 url(images/bg.png)
    repeat-y center;
  background-color: #fff;
  font: 10pt Arial, Helvetica, Geneva, sans-serif;
  margin-bottom: 20px;
}
body {                                     weatherstyle.css:34
  line-height: 1;
}
html, body, div, span, applet,             weatherstyle.css:10
object, iframe, h1, h2, h3, h4,
h5, h6, p, blockquote, a, abbr, acronym, address,
big, cite, code, del, dfn, em, img, ins, kbd, q, s,
samp, small, strike, strong, sub, sup, tt, var, b, u,
i, center, dl, dt, dd, ol, ul, li, fieldset, form,
label, legend, article, aside, canvas, details,
figcaption, figure, footer, header, hgroup, menu,
nav, section, summary, time, mark, audio, video {
  margin: 0;
  padding: 0;
```

```
background-attachment
  scroll
background-clip
  border-box
background-color
  rgb(197, 229, 245)
background-image
  url("https://www.weather.gov/css/images/bg.png")
background-origin
  padding-box
background-position
  50% 50%
background-position-x
  50%
background-position-y
  50%
background-repeat
  repeat-y
background-size
  auto
```

Figure 13-3: The Developer Tools window in the Chrome browser

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 help you parse HTML for your web scraping programs.

DON'T USE REGULAR EXPRESSIONS TO PARSE HTML

Locating a specific piece of HTML (or a piece of XML, JSON, TOML, or YAML) in a string seems like a perfect case for regular expressions. However, I advise you not to do this. HTML can be formatted in many ways and still be considered valid, but trying to capture all these possible variations in a regular expression is tedious and error prone. Using a module developed specifically for parsing HTML, such as `bs4`, is less likely to result in bugs.

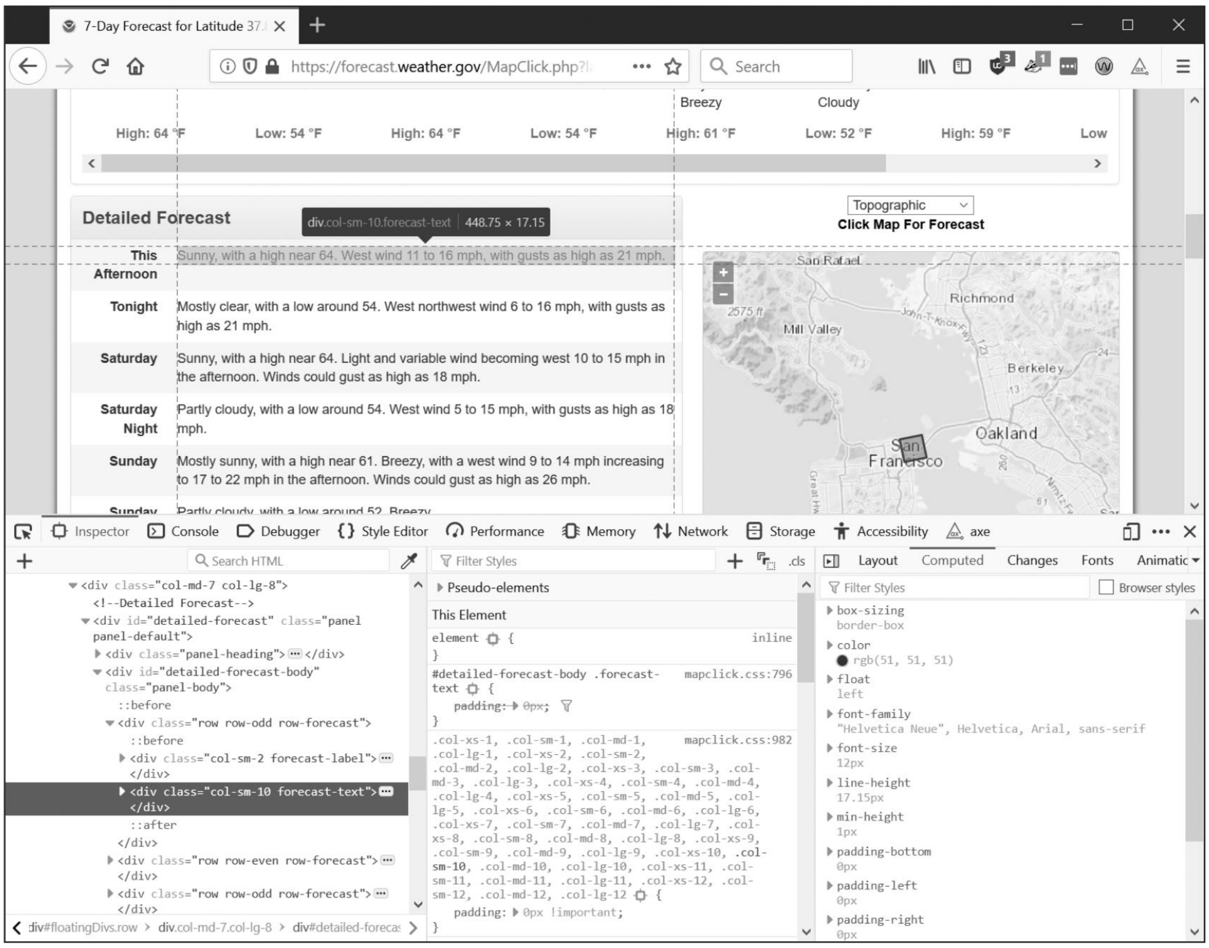
You can find an extended argument for why you shouldn't parse HTML with regular expressions at *https://stackoverflow.com/a/1732454/1893164*.

Finding HTML Elements

Once your program has downloaded a web page using the `requests` module, you'll 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, it should 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 that information on the page (or CTRL-click on macOS) and select **Inspect Element** from the context menu that appears. This brings up the Developer Tools window, which shows you the HTML that produces that particular part of the web page. Figure 13-4 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.



This Afternoon	Sunny, with a high near 64. West wind 11 to 16 mph, with gusts as high as 21 mph.
Tonight	Mostly clear, with a low around 54. West northwest wind 6 to 16 mph, with gusts as high as 21 mph.
Saturday	Sunny, with a high near 64. Light and variable wind becoming west 10 to 15 mph in the afternoon. Winds could gust as high as 18 mph.
Saturday Night	Partly cloudy, with a low around 54. West wind 5 to 15 mph, with gusts as high as 18 mph.
Sunday	Mostly sunny, with a high near 61. Breezy, with a west wind 9 to 14 mph increasing to 17 to 22 mph in the afternoon. Winds could gust as high as 26 mph.
Sunday	Partly cloudy, with a low around 52. Breezy.

```
▾<div class="col-md-7 col-lg-8">
   <!--Detailed Forecast-->
  ▾<div id="detailed-forecast" class="panel
   panel-default">
    ▸<div class="panel-heading">⋯</div>
    ▾<div id="detailed-forecast-body"
      class="panel-body">
       ::before
      ▾<div class="row row-odd row-forecast">
         ::before
        ▸<div class="col-sm-2 forecast-label">⋯
         </div>
        ▸<div class="col-sm-10 forecast-text">⋯
         </div>
         ::after
       </div>
      ▸<div class="row row-even row-forecast">⋯
       </div>
      ▸<div class="row row-odd row-forecast">⋯
       </div>
div#floatingDivs.row › div.col-md-7.col-lg-8 › div#detailed-forecas
```

```
▸ Pseudo-elements
This Element
element {                                   inline
}
#detailed-forecast-body .forecast-   mapclick.css:796
text {
   padding: ▸ 0px;
}
.col-xs-1, .col-sm-1, .col-md-1,     mapclick.css:982
.col-lg-1, .col-xs-2, .col-sm-2,
.col-md-2, .col-lg-2, .col-xs-3, .col-sm-3, .col-
md-3, .col-lg-3, .col-xs-4, .col-sm-4, .col-md-4,
.col-lg-4, .col-xs-5, .col-sm-5, .col-md-5, .col-
lg-5, .col-xs-6, .col-sm-6, .col-md-6, .col-lg-6,
.col-xs-7, .col-sm-7, .col-md-7, .col-lg-7, .col-
xs-8, .col-sm-8, .col-md-8, .col-lg-8, .col-xs-9,
.col-sm-9, .col-md-9, .col-lg-9, .col-xs-10, .col-
sm-10, .col-md-10, .col-lg-10, .col-xs-11, .col-
sm-11, .col-md-11, .col-lg-11, .col-xs-12, .col-
sm-12, .col-md-12, .col-lg-12 {
   padding: ▸ 0px !important;
}
```

```
▸ box-sizing
  border-box
▸ color
  ● rgb(51, 51, 51)
▸ float
  left
▸ font-family
  "Helvetica Neue", Helvetica, Arial, sans-serif
▸ font-size
  12px
▸ line-height
  17.15px
▸ min-height
  1px
▸ padding-bottom
  0px
▸ padding-left
  0px
▸ padding-right
  0px
```

Figure 13-4: Inspecting the element that holds forecast text

From the Developer Tools, you can see that the HTML responsible for the forecast part of the web page is this:

```
<div class="col-sm-10 forecast-text">Sunny,
with a high near 64.
West wind 11 to 16 mph, with gusts as high as
21 mph.</div>
```

This is exactly what you were looking for! It seems that the forecast information is contained inside a `<div>` element with the `forecast-text` CSS class.

Right-click this element in the browser's developer console and, from the context menu that appears, select **Copy ▸ CSS Selector**. This option copies a string such as `'div.row-odd:nth-child(1) > div:nth-child(2)'` to the clipboard. You can pass it to Beautiful Soup's `select()` method or Selenium's `find_element()` method, as explained later in this chapter, to find the element in the string.

The *CSS selector* syntax used in this string specifies which HTML elements to retrieve from a web page. The full selector syntax is beyond the scope of this book, but you can obtain the selector from the browser Developer Tools, as we did here. *XPath* is another syntax for selecting HTML elements, but is also beyond the scope of this book.

Keep in mind that when a website changes its layout, you'll need to update the HTML tags your scripts check. This can happen with little or no warning, so be sure to keep an eye on your program in case it suddenly displays errors about not being able to find elements. In general, it's better to use a website's API if it offers one, as it's much less likely to change than the website itself.

Parsing HTML with Beautiful Soup

Beautiful Soup is a package for extracting information from an HTML page. You'll use the name `beautifulsoup4` to install the package but the shorter module name `bs4` to import it. In this section, we'll use Beautiful Soup to *parse* (that is, analyze and extract the parts of) the HTML file at *https://autbor.com/example3.html*, which has the following content:

```
<!-- This is an HTML comment. -->

<html>
<head>
    <title>Example Website Title</title>
    <style>
        .slogan {
            color: gray;
            font-size: 2em;
        }
    </style>
</head>
<body>
    <h1>Example Website</h1>
    <p>This <p> tag puts <b>content</b> into
a <i>single</i> paragraph.</p>
    <p><a href="https://
inventwithpython.com”>This text is a link</a>
to books by <span id=
"author">Al Sweigart</span>.</p>
    <p><img src="wow_such_zophie_thumb.webp"
alt="Close-up of my cat Zophie." /></p>
    <p class="slogan">Learn to program in
Python!</p>
    <form>
        <p><label>Username: <input
id="login_user" placeholder="admin" /></
label></p>
        <p><label>Password: <input
id="login_pass" type="password"
placeholder="swordfish" />
    </form>
</label></p>
    <p><label>Agree to disagree: <input
type="checkbox" /></label><input
type="submit"
value="Fake Button" /></p>
</body>
</html>
```

Note that the login form on this page is fake and is included for cosmetic value.

Even a simple HTML file involves many different tags and attributes, and matters quickly get confusing when it comes to complex websites. Thankfully, Beautiful Soup makes working with HTML much easier.

Creating a Beautiful Soup Object

The `bs4.BeautifulSoup()` function accepts a string containing the HTML it will parse, then returns a `BeautifulSoup` object. For example, enter the following into the interactive shell while your computer is connected to the internet:

```
>>> import requests, bs4
>>> res = requests.get('https://autbor.com/
example3.html')
>>> res.raise_for_status()
>>> example_soup =
bs4.BeautifulSoup(res.text, 'html.parser')
>>> type(example_soup)
<class 'bs4.BeautifulSoup'>
```

This code uses `requests.get()` to download the main page of the Automate the Boring Stuff website and then passes the response's `text` attribute to `bs4.BeautifulSoup()`. Beautiful Soup can parse different formats, and the `'html.parser'` argument tells it that we are parsing HTML. Finally, the code stores the returned `BeautifulSoup` object in a variable named `example_soup`.

You can also load an HTML file from your hard drive by passing a `File` object to `bs4.BeautifulSoup()`. Enter the following into the interactive shell (after making sure the *example3.html* file is in the working directory):

```
>>> import bs4
>>> with open('example3.html') as
example_file:
...     example_soup =
bs4.BeautifulSoup(example_file,
'html.parser')
...
```

```
>>> type(example_soup)
<class 'bs4.BeautifulSoup'>
```

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

Finding an Element

You can retrieve a web page element from a `BeautifulSoup` object by calling its `select()` method and passing a CSS selector string for the element you're looking for. The method returns a list of `Tag` objects, which represent matching HTML elements. Table 13-2 shows examples of the most common CSS selector patterns using `select()`.

Table 13-2: Examples of CSS Selectors

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

You can combine the various selector patterns to make sophisticated matches. For example, `soup.select('p`

#author') matches any element that has an id attribute of author, as long as it's also inside a <p> element.

You can pass tag values to the str() function to show the HTML tags they represent. Tag values also have an attrs attribute containing all their HTML attributes as a dictionary. For example, download the https://autbor.com/example3.html page as *example3.html*, then enter the following into the interactive shell:

```
>>> import bs4
>>> example_file = open('example3.html')
>>> example_soup =
bs4.BeautifulSoup(example_file.read(),
'html.parser')
>>> elems = example_soup.select('#author')
>>> type(elems) # elems is a list of Tag
objects.
<class 'bs4.element.ResultSet'>
>>> 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()  # The inner text of
the element
'Al Sweigart'
>>> elems[0].attrs
{'id': 'author'}
```

This code finds the element with id="author" in our example HTML. We use select('#author') to return a list of all the elements with id="author". We then store this list of Tag objects in the variable elems. Running len(elems) tells us there is one Tag object in the list, meaning there was one match.

Passing the element to `str()` returns a string with the starting and closing tags and the element's text. Calling `gettext()` on the element returns the element's text, or the content between the opening and closing tags: in this case, `'Al Sweigart'`. 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:

```
>>> p_elems = example_soup.select('p')
>>> str(p_elems[0])
'<p>This <p> tag puts <b>content</b> into a
<i>single</i> paragraph.</p>'
>>> p_elems[0].gettext()
'This <p> tag puts content into a single
paragraph.'
>>> str(p_elems[1])
'<p> <a href="https://
inventwithpython.com/”>This text is a link</
a> to books by
<span id="author">Al Sweigart</span>.</p>'
>>> p_elems[1].gettext()
'This text is a link to books by Al
Sweigart.'
>>> str(p_elems[2])
'<p><img alt="Close-up of my cat Zophie."
src="wow_such_zophie_thumb.webp"/></p>'
>>> p_elems[2].gettext()
''
```

This time, `select()` gives us a list of three matches, which we store in `p_elems`. Using `str()` on `p_elems[0]`, `p_elems[1]`, and `p_elems[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 lets you access HTML attribute values from an element. You’ll pass the method an attribute name as a string and receive that attribute’s value. Using *example3.html* from *https://autbor.com/example3.html*, enter the following into the interactive shell:

```
>>> import bs4
>>> soup =
bs4.BeautifulSoup(open('example3.html'),
'html.parser')
>>> span_elem = soup.select('span')[0]
>>> str(span_elem)
'<span id="author">Al Sweigart</span>'
>>> span_elem.get('id')
'author'
>>>
span_elem.get('some_nonexistent_addr') ==
None
True
>>> span_elem.attrs
{'id': 'author'}
```

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

Project 7: Open All Search Results

When I look up a topic on a search engine, I don’t look at just one search result at a time. By *middle-clicking* a search result link (or clicking it while holding CTRL), I open the first several links in a bunch of new tabs to read later. I search the internet 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 enter a term on the command line and have my computer automatically open the top search results in new browser tabs.

Let's write a script to do this for the search results page of the Python Package Index at *https://pypi.org*. You could adapt a program like this to many other websites, although Google, DuckDuckGo, Amazon, and other large websites often employ measures that make scraping their search results pages difficult.

This is what the program should do:

- Get search keywords from the command line arguments
- Retrieve the search results page
- Open a browser tab for each result

This means your code needs to do the following:

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

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

Step 1: Get the Search Page

Before writing code, you first need to know the URL of the search results page. By looking at the browser's address bar after doing a search, you can see that the results page has a URL that looks like this: *https://pypi.org/search/?q=<SEARCH_TERM_HERE>*. The `requests` module can download this page; 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:

```
# searchpypi.py - Opens several search
results on pypi.org

import requests, sys, webbrowser, bs4

print('Searching...')  # Display text while
downloading the search results page.
res = requests.get('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 as command line arguments when launching the program, and the code stores these arguments as strings in a list in `sys.argv`.

Step 2: Find All 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 results 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 *pyautogui*, you can open the browser's Developer Tools and inspect some of the link elements on the page. They can look complicated, like pages of this: `<a class="package-snippet" href="/project/pyautogui" >`. But 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:

```
# searchpypi.py - Opens several search
results on pypi.org
import requests, sys, webbrowser, bs4
--snip--
# Retrieve top search result links.
soup = bs4.BeautifulSoup(res.text,
'parser.html')
# Open a browser tab for each result.
link_elems = soup.select('.package-snippet')
```

If you look at the <a> elements, you'll see that 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're 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 should remain up-to-date.

Step 3: Open Web Browsers for Each Result

Finally, you must tell the program to open web browser tabs for the results. Add the following to the end of your program:

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

By default, the program opens 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 `num_open`. Then, you can run through a `for` loop by calling `range(num_open)`.

On each iteration of the loop, the code uses `webbrowser.open()` to open a new tab in the web browser. Note that the `href` attribute's value in the returned `<a>` elements don't 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 Chapter 12 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 8: Download XKCD Comics

Blogs, web comics, 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. 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, shown in Figure 13-5, is a popular geek webcomic with a website that fits this structure. 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.

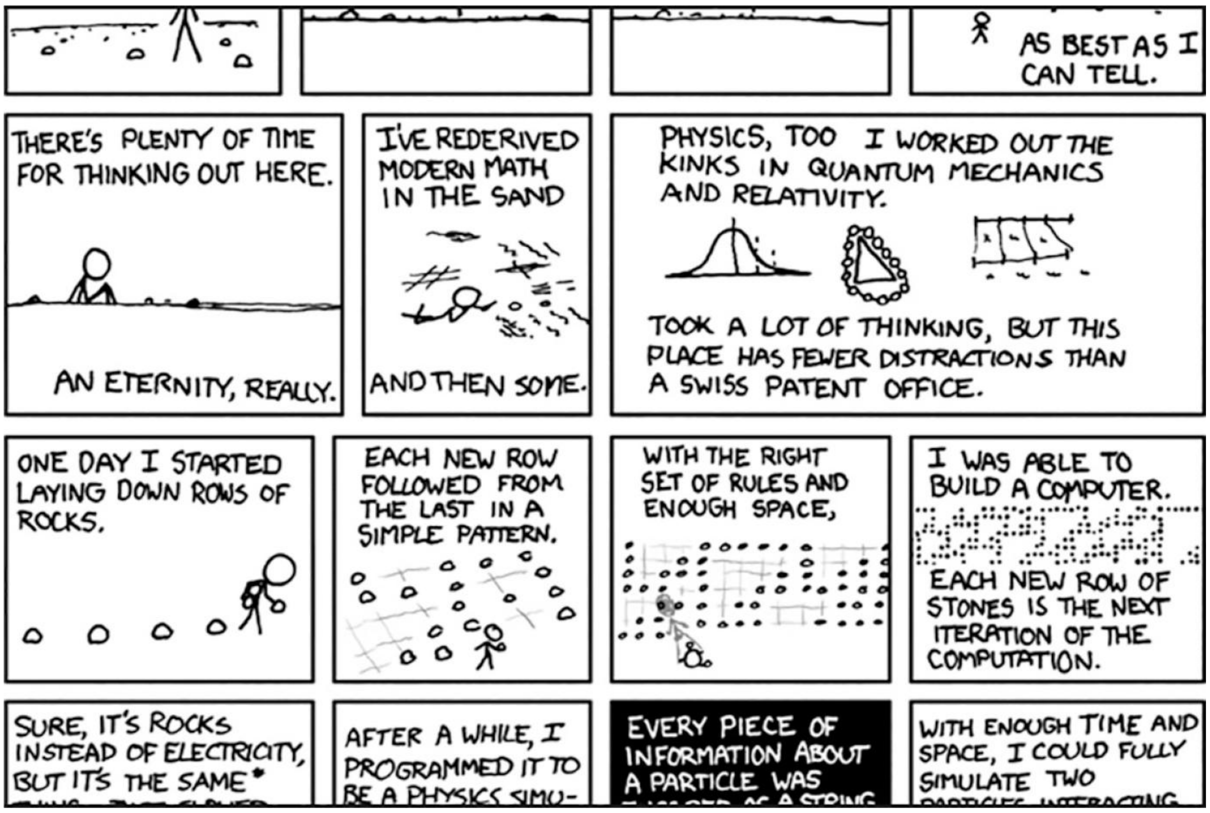

AS BEST AS I CAN TELL.
THERE'S PLENTY OF TIME FOR THINKING OUT HERE.
AN ETERNITY, REALLY.
I'VE REDERIVED MODERN MATH IN THE SAND
AND THEN SOME.
PHYSICS, TOO. I WORKED OUT THE KINKS IN QUANTUM MECHANICS AND RELATIVITY.
TOOK A LOT OF THINKING, BUT THIS PLACE HAS FEWER DISTRACTIONS THAN A SWISS PATENT OFFICE.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

Figure 13-5: XKCD, "a webcomic of romance, sarcasm, math, and language"

Here's what your program should do:

- Load the XKCD home page.
- Save the comic image on that page.
- Follow the Previous Comic link.
- Repeat until it reaches the first comic or the max download limit.

This means your code will need to do the following:

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

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

Step 1: Design the Program

If you open the browser's Developer Tools and inspect the elements on the page, you should find the following to be true:

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

- The oldest comic’s Prev button links to the https://xkcd.com/# URL, indicating that there are no more previous pages.

To prevent the readers of this book from eating up too much of the XKCD website’s bandwidth, let’s limit the number of downloads we make to 10 by default. Make your code look like the following:

```
# downloadXkcdComics.py - Downloads XKCD
comics

import requests, os, bs4, time

url = 'https://xkcd.com'  # Starting URL
os.makedirs('xkcd', exist_ok=True)  # Store
comics in ./xkcd
num_downloads = 0
MAX_DOWNLOADS = 10
while not url.endswith('#') and num_downloads
< MAX_DOWNLOADS:
    # 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.')
```

The program creates a `url` variable that starts with the value `'https://xkcd.com'` and repeatedly updates it (in a `while` loop) with the URL of the current page’s Prev link. At every step in the loop, you’ll download the comic at `url`. The loop stops when `url` ends with `'#'` or you have downloaded `MAX_DOWNLOADS` comics.

You'll 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 has already been created.

Step 2: Download the Web Page

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

```
# downloadXkcdComics.py - Downloads XKCD
comics

import requests, os, bs4, time

url = 'https://xkcd.com'  # Starting URL
os.makedirs('xkcd', exist_ok=True)  # Store
comics in ./xkcd
num_downloads = 0
MAX_DOWNLOADS = 10
while not url.endswith('#') and num_downloads
< MAX_DOWNLOADS:
    # Download the page.
    print(f'Downloading page {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 `requests.get()` function to download it. As always, you should 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, create a `BeautifulSoup` object from the text of the downloaded page.

Step 3: Find and Download the Comic Image

To download the comic on each page, make your code look like the following:

```
# downloadXkcdComics.py - Downloads XKCD
comics

import requests, os, bs4, time

--snip--

    # Find the URL of the comic image.
    comic_elem = soup.select('#comic img')
    if comic_elem == []:
        print('Could not find comic image.')
    else:
        comic_URL = 'https:' +
comic_elem[0].get('src')
        # Download the image.
        print(f'Downloading image
{comic_URL}...')
        res = requests.get(comic_URL)
        res.raise_for_status()
    # TODO: Save the image to ./xkcd.

    # TODO: Get the Prev button's url.

print('Done.')
```

Because you inspected the XKCD home page with your Developer Tools, you know that the `<img>` element for the comic image is inside another 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, `soup.select('#comic img')` will return a `ResultSet` object of 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

At this point, the comic's image file is stored in the `res` variable. You need to write this image data to a file on the hard drive. Make your code look like the following:

```
# downloadXkcdComics.py - Downloads XKCD
comics

import requests, os, bs4, time

--snip--

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

    # Get the Prev button's URL.
      prev_link =
soup.select('a[rel="prev"]')[0]
      url = 'https://xkcd.com' +
prev_link.get('href')
      num_downloads += 1
      time.sleep(1)  # Pause so we don't
hammer the web server.

print('Done.')
```

You'll also need a filename for the local image file to pass to `open()`. The `comic_URL` will have a value like `'https://imgs.xkcd.com/comics/heartbleed_explanation.png'`, which you might have noticed looks a lot like a filepath. In fact, you can call `os.path.basename()` with `comic_URL` to return just the last part of the URL, `'heartbleed_explanation.png'`, and use this as the filename when saving the image to your hard drive. 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'` 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 chunks of the image data to the file. Then, the code closes the file, saving the image to your hard drive.

Afterward, the selector `'a[rel="prev"]'` identifies the `<a>` element with the `rel` attribute set to `prev`. You can use this `<a>` element's `href` attribute to get the previous comic's URL, which gets stored in `url`.

The last part of the loop's code increments `num_downloads` by `1` so that it doesn't download all of the comics by default. It also introduces a one-second pause with `time.sleep(1)` to prevent the script from "hammering" the site (that is, impolitely downloading comics as fast as possible, which may cause performance issues for other website visitors). Then, the `while` loop begins the entire download process again.

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 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

Many web crawling programs involve downloading pages and following links. Similar programs could do the following:

- Back up an entire site by following all of its links.
- Copy all the messages on 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, this URL isn't always so easy to find. Or perhaps the website you want your program to navigate requires you to log in first. Selenium will give your programs the power to perform such sophisticated tasks.

Controlling the Browser with Selenium

Selenium lets Python directly control the browser by programmatically clicking links and filling in forms, just as a human user would. Using Selenium, you can interact with web pages in a much more advanced way than with `requests` and Beautiful Soup; but because it launches a web browser, it's 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, for instance, depends on the JavaScript code that updates the page, you'll need to use Selenium instead of `requests`. That's because major e-commerce 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. Selenium is much more likely than `requests` to function on these sites long term.

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/`*`X.XX.X`*`'`. 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 the same as a regular browser (for instance, `' Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:108.0) Gecko/20100101 Firefox/108.0'`), 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, websites can still find ways to detect Selenium, and major ticketing and e-commerce websites often block it to prevent the web scraping of their pages.

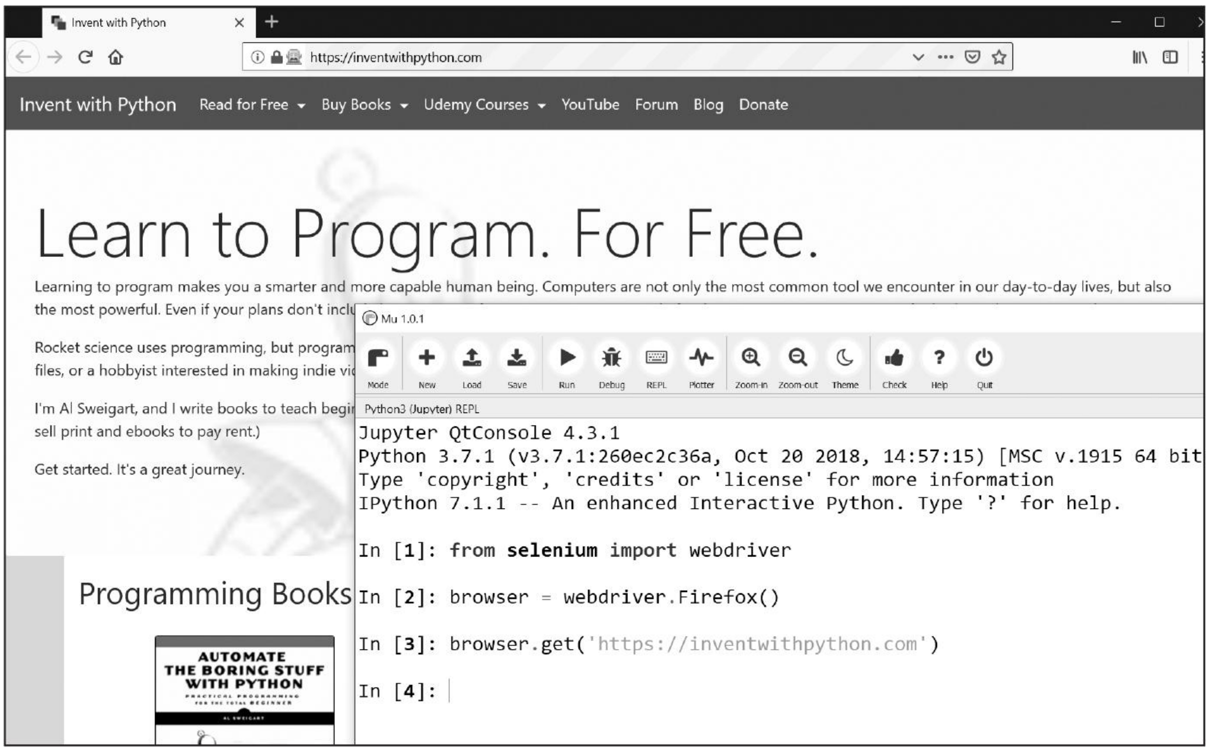
Starting a Selenium-Controlled Browser

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*.

Importing Selenium's modules is slightly tricky. Instead of `import selenium`, you must run **from selenium import webdriver**. (The exact reason why Selenium 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.WebDriv
er'>
>>> browser.get('https://
inventwithpython.com')
```

You'll notice that 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 13-6.



Learn to Program. For Free.

Learning to program makes you a smarter and more capable human being. Computers are not only the most common tool we encounter in our day-to-day lives, but also the most powerful. Even if your plans don't inclu

Rocket science uses programming, but program files, or a hobbyist interested in making indie vi

I'm Al Sweigart, and I write books to teach begi sell print and ebooks to pay rent.)

Get started. It's a great journey.

```
Jupyter QtConsole 4.3.1
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:57:15) [MSC v.1915 64 bit
Type 'copyright', 'credits' or 'license' for more information
IPython 7.1.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from selenium import webdriver

In [2]: browser = webdriver.Firefox()

In [3]: browser.get('https://inventwithpython.com')

In [4]:
```

Figure 13-6: 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," you need to manually download the web driver for Firefox before you can use Selenium to control it. You can also control browsers other than Firefox if you install the web driver for them, and instead of manually downloading browser web drivers, you can use the webdriver-manager package from *https://pypi.org/project/webdriver-manager/*.

Clicking Browser Buttons

Selenium can simulate clicks on various browser buttons 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

Finding Elements on the Page

A `WebDriver` object has the `find_element()` and `find_elements()` methods for finding elements on a web page. The `find_element()` method returns a single `WebElement` object, representing the first element on the page that matches your query. The `find_elements()` method returns a list of `WebElement` objects for *every* matching element on the page.

You can find elements through their class name, CSS selector, ID, or another means. First, run **`from selenium.webdriver.common.by import By`** to get the `By` object. The `By` object has several constants you can pass to the `find_element()` and `find_elements()` methods. Table 13-3 lists these constants.

Table 13-3: Selenium's `By` Constants for Finding Elements

Constant name	**`WebElement object/list returned`**
`By.CLASS_NAME`	Elements that use the CSS class *`name`*
`By.CSS_SELECTOR`	Elements that match the CSS *`selector`*
`By.ID`	Elements with a matching *`id`* attribute value
`By.LINK_TEXT`	`<a>` elements that completely match the *`text`* provided
`By.PARTIAL_LINK_TEXT`	`<a>` elements that contain the *`text`* provided
`By.NAME`	Elements with a matching *`name`* attribute value
`By.TAG_NAME`	Elements with a matching tag *`name`* (case-insensitive; an `<a>` element is matched by `'a'` and `'A'`)

If no elements exist on the page that match what the method is looking for, Selenium 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 learn more about it by reading the attributes or calling the methods in Table 13-4.

Table 13-4: `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, like `href` in an `<a>` element.
`get_property(`*`name`*`)`	The value for the element's property, which does not appear in the HTML code. Some examples of HTML properties are `innerHTML` and `innerText`.
`text`	

Attribute or method	Description
	The text within the element, such as `'hello'` in the following: `<span>hello </span>`
`clear()`	For text field or text area elements, clears the text entered 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.
`size`	A dictionary with keys `'width'` and `'height'` for the size of the element in the page.

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

```
from selenium import webdriver
from selenium.webdriver.common.by import By
browser = webdriver.Firefox()
browser.get('https://autbor.com/
example3.html')
elems =
browser.find_elements(By.CSS_SELECTOR, 'p')
print(elems[0].text)
print(elems[0].get_property('innerHTML'))
```

Here, we open Firefox and direct it to a URL. On this page, we get a list of the `<p>` elements, look at the first one at index `0`, and then get the string of the text inside that `<p>` element. Next, we get the string of its `innerHTML` property. This program outputs the following:

```
This <p> tag puts content into a single
paragraph.
This <p> tag puts <b>content</b> into a
<i>single</i> paragraph.
```

The element's `text` attribute shows the text as we'd see it in the web browser: "This `<p>` tag puts content into a single paragraph." We can also examine the element's `innerHTML` property by calling the `get_property()` method, which is the HTML source code that includes tags and HTML entities. (The < and > are HTML escape characters that represent the less than [<] and greater than [>] characters.)

Note that the `text` attribute is just a shortcut for calling `get_property('innerText')`. The names *innerHTML* and *innerText* are standard names of properties for HTML elements. In short, these element properties are accessed by JavaScript code and web drivers, while element attributes are part of the HTML source code, like the `href` in `<a href="https://inventwithpython.com">`.

Clicking Elements on the Page

The `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 a mouse clicks the element. For example, enter the following into the interactive shell:

```
>>> from selenium import webdriver
>>> from selenium.webdriver.common.by import
By
>>> browser = webdriver.Firefox()
>>> browser.get('https://autbor.com/
example3.html')
>>> link_elem =
browser.find_element(By.LINK_TEXT, 'This text
is a link')
>>> type(link_elem)
<class
```

```
'selenium.webdriver.remote.webelement.WebElement'>
>>> link_elem.click()  # Follows the "This text is a link" link
```

This code opens Firefox to https://autbor.com/example3.html, gets the `WebElement` object for the `<a>` element with the text *This is a link*, and then simulates clicking that `<a>` element as if you'd 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
>>> from selenium.webdriver.common.by import By
>>> browser = webdriver.Firefox()
>>> browser.get('https://autbor.com/example3.html')
>>> user_elem = browser.find_element(By.ID, 'login_user')
>>> user_elem.send_keys('your_real_username_here')
>>> password_elem = browser.find_element(By.ID, 'login_pass')
>>> password_elem.send_keys('your_real_password_here')
>>> password_elem.submit()
```

As long as the login page hasn't changed the `id` of the username and password `<input>` elements, 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 `user_elem.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.

Sending Special Keys

Selenium has a module, `selenium.webdriver.common.keys`, to represent keyboard keys, which it stores in attributes. Because the module has such a long name, it's much easier to run `from selenium.webdriver.common.keys import Keys` at the top of your program; if you do, you can simply write `Keys` anywhere you'd normally have to write `selenium.webdriver.common.keys`.

You can pass `send_keys()` any of the following constants:

```
Keys.ENTER  Keys.PAGE_UP       Keys.DOWN
Keys.RETURN  Keys.ESCAPE        Keys.LEFT
Keys.HOME    Keys.BACK_SPACE    Keys.RIGHT
Keys.END     Keys.DELETE        Keys.TAB
Keys.PAGE_DOWN   Keys.UP        Keys.F1 to
Keys.F12
```

You can also pass the method a string, such as `'hello'` or `'?'`.

For example, if the cursor isn't 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.by import
By
>>> from selenium.webdriver.common.keys
import Keys
```

```
>>> browser = webdriver.Firefox()
>>> browser.get('https://nostarch.com')
>>> html_elem =
browser.find_element(By.TAG_NAME, 'html')
>>> html_elem.send_keys(Keys.END)  # Scrolls
to bottom
>>>
html_elem.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 via the main `<html>` tag. This would be useful if, for example, new content is loaded once you've scrolled to the bottom of the page.

Selenium can do much more than 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.io*. You can also find Python conference talks on Selenium by searching the website *https://pyvideo.org*.

Controlling the Browser with Playwright

Playwright is a browser-controlling library similar to Selenium, but it's newer. While it might not currently have the wide audience that Selenium has, it does offer some features that merit learning. Chief among these new features is the ability to run in *headless mode*, meaning you can simulate a browser without actually having the browser window open on your screen. This makes it useful for running automated tests or web scraping jobs in the background. Playwright's full documentation is at *https://playwright.dev/python/docs/intro*.

Also, installing web drivers for individual browsers is easier to do with Playwright compared to Selenium: just run **`python -m playwright install`** on Windows and **`python3 -m playwright install`** on macOS and Linux from a terminal window to install the web drivers for Firefox, Chrome, and Safari. Because Playwright is otherwise similar to Selenium, I won't cover the general web scraping and CSS selector information in this section.

Starting a Playwright-Controlled Browser

Once Playwright is installed, you can test it with the following program:

```
from playwright.sync_api import
sync_playwright
with sync_playwright() as playwright:
    browser = playwright.firefox.launch()
    page = browser.new_page()
    page.goto('https://autbor.com/
example3.html')
    print(page.title())
    browser.close()
```

When run, this program pauses while it loads the Firefox browser and the *https://autbor.com/example3.html* website, and then prints its title, "Example Website." You can also use `playwright.chromium.launch()` or `playwright.webkit.launch()` to use the Chrome and Safari browsers, respectively.

Playwright automatically calls the `start()` and `stop()` methods when the execution enters and exits the `with` statement's block. Playwright has a synchronous mode, where its functions don't return until the operation is complete. This way, you don't accidentally tell the browser to find an element before the page has finished loading. Playwright's asynchronous features are beyond the scope of this book.

You may have noticed that no browser window appeared at all, because, by default, Playwright runs in headless mode. This, along with how Playwright puts its code inside a `with` statement, can make debugging tricky. To run Playwright one step at a time, enter the following into the interactive shell:

```
>>> from playwright.sync_api import
sync_playwright
>>> playwright = sync_playwright().start()
>>> browser =
playwright.firefox.launch(headless=False,
slow_mo=50)
>>> page = browser.new_page()
>>> page.goto('https://autbor.com/
example3.html')
<Response url='https://autbor.com/
example3.html' request=<Request
 url='https://autbor.com/example3.html'
method='GET'>>
>>> browser.close()
>>> playwright.stop()
```

The `headless=False` and `slow_mo=50` keyword arguments to `playwright.firefox.launch()` make the browser window appear on your screen and add a 50 ms delay to its operations so that it's easier for you to see what is happening. You don't have to worry about adding pauses to give web pages time to load: Playwright is much better than Selenium about not moving on to new operations before the previous one has finished.

The `Page` object returned by the `new_page()` `Browser` method represents a new tab in a new browser window. You can have multiple browser windows open at the same time when using Playwright.

Clicking Browser Buttons

Playwright can simulate clicking the browser buttons by calling the following `Page` methods on the `Page` object returned by `browser.new_page()`:

`page.go_back()`	Clicks the Back button
`page.go_forward()`	Clicks the Forward button
`page.reload()`	Clicks the Refresh/Reload button
`page.close()`	Clicks the Close Window button

Finding Elements on the Page

Playwright has `Page` object methods colloquially called *locators* that return `Locator` objects, which represent possible HTML elements on a web page. I say *possible* because, while Selenium immediately raises an error if it can't find the element you ask for, Playwright understands that the page might dynamically create the element later. This is useful but has a slightly unfortunate side effect: if the element you specified doesn't exist, Playwright pauses for 30 seconds while it waits for the element to appear.

But this 30-second pause is tedious if you've simply made a typo. To immediately check whether an element exists and is visible on the page, call the `is_visible()` method on the `Locator` object returned by the locator. You can also call `page.query_selector('`*`selector`*`')` where *`selector`* is a string of the element's CSS or XPath selector. The `page.query_selector()` method immediately returns, and if it returns `None`, the element doesn't currently exist on the page. A `Locator` object may match one or more HTML elements on the web page. Table 13-5 contains Playwright's locators.

Table 13-5: Playwright's Locators for Finding Elements

Locator	**Locator object returned**
`page.get_by_role(`*`role`*`,` *`name=label`*`)`	Elements by their role and optionally their *`label`*
`page.get_by_text(`*`text`*`)`	Elements that contain *`text`* as part of their inner text
`page.get_by_label(`*`label`*`)`	Elements with matching *`<label>`* text as *`label`*
`page.get_by_placeholder(`*`text`*`)`	`<input>` and `<textarea>` elements with matching `placeholder` attribute values as the *`text`* provided
`page.get_by_alt_text(`*`text`*`)`	`<img>` elements with matching `alt` attribute values as the *`text`* provided
`page.locator(`*`selector`*`)`	Elements with a matching CSS or XPath *`selector`*

The `get_by_role()` method makes use of *Accessible Rich Internet Applications (ARIA)* roles, a set of standards that enable software to identify web page content to adapt it for users with vision or other disabilities. For example, the "heading" role applies to the `<h1>` through `<h6>` tags, with the text in between `<h1>` and `</h1>` being the text you can identify with the `get_by_role()` method's `name` keyword parameter. (There is much more to ARIA roles than this, but the topic is beyond the scope of this book.)

You can use the text between the starting and ending tags to locate elements. Calling `page.get_by_text('is a link')` would locate the `<a>` element in `<a href="https://inventwithpython.com">This text is a link</a>`. A partial, case-insensitive text match is generally good enough to locate the element.

The `page.get_by_label()` method locates elements using the text between `<label>` and `</label>` tags. For example, `page.get_by_label('Agree')` would locate the `<input>` checkbox element in `<label>Agree to disagree: <input type="checkbox" /></label>`.

The `<input>` and `<textarea>` tags can have a `placeholder` attribute to show placeholder text until the user enters real text. For example, `page.get_by_placeholder('admin')` would locate the `<input>` element for `<input id="login_user" placeholder="admin" />`.

Images on web pages can have alt text in their `alt` attribute to describe the image contents to sight-impaired users. Some browsers show the alt text as a tool tip if you hover the mouse cursor over the image. The `page.get_by_alt_text('Zophie')` call would return the `<img>` element in `<img src="wow_such_zophie_thumb.webp" alt="Close-up of my cat Zophie." />`.

If you just need to obtain a `Locator` object via a CSS selector, call the `locator()` locator and pass it the selector string. This is similar to Selenium's `find_elements()` method with the `By.CSS_SELECTOR` constant.

Table 13-6: `Locator` Methods

Method	Description
`get_attribute(`*`name`*`)`	Returns the value for the element's *`name`* attribute, such as `'https://nostarch.com'` for the `href` attribute in an `<a href="https://nostarch.com">` element.
`count()`	Returns an integer of the number of matching elements in this `Locator` object.
`nth(`*`index`*`)`	Returns a `Locator` object of the matching element given by the index. For example, `nth(3)` returns the fourth matching element since index `0` is the first matching element.
`first`	The `Locator` object of the first matching element. This is the same as `nth(0)`.
`last`	The `Locator` object of the last matching element. If there are, say, five match elements, this is the same as `nth(4)`.
`all()`	Returns a list of `Locator` objects for each individual matching element.

Method	Description
`inner_text()`	Returns the text within the element, such as `'hello'` in `<b>hello</b>`.
`inner_html()`	Returns the HTML source within the element, such as `'<b>hello</b>'` in `<b>hello</b>`.
`click()`	Simulates a click on the element, which is useful for link, checkbox, and button elements.
`is_visible()`	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_checked()`	For checkbox or radio button elements, returns `True` if the element is selected; otherwise, returns `False`.
`bounding_box()`	Returns a dictionary with keys `'x'` and `'y'` for the position of the element's top-left corner in the page, along with keys `'width'` and `'height'` for the element's size.

Since `Locator` objects can represent multiple elements, you can obtain a `Locator` object for an individual element with the `nth()` method, passing the zero-based index. For example, open a new file editor tab and enter the following program:

```
from playwright.sync_api import
sync_playwright
with sync_playwright() as playwright:
    browser =
playwright.firefox.launch(headless=False,
slow_mo=50)
    page = browser.new_page()
    page.goto('https://autbor.com/
example3.html')
    elems = page.locator('p')
    print(elems.nth(0).inner_text())
    print(elems.nth(0).inner_html())
```

Like the Selenium example, this program outputs the following:

```
This <p> tag puts content into a single
paragraph.
This <p> tag puts <b>content</b> into a
<i>single</i> paragraph.
```

The `page.locator('p')` code returns a `Locator` object that matches all `<p>` elements in the web page, and the `nth(0)` method call returns a `Locator` object for just the first `<p>` element. The `Locator` objects also have a `count()` method for returning the number of matching elements in the locator (similar to the `len()` function for Python lists). There are also `first` and `last` attributes that contain a locator that matches the first or last element. If you want a list of `Locator` objects for each individual matching element, call the `all()` method.

Once you have `Locator` objects for elements, you can perform mouse clicks and key presses on them, as described in the next few sections.

Clicking Elements on the Page

The `Page` object has `click()`, `check()`, `uncheck()`, and `set_checked()` methods for simulating clicks on link, button, and checkbox elements. You can call these methods and pass the string of a CSS or XPath selector of the element, or you can use Playwright's `Locator` functions in Table 13-6. Enter the following into the interactive shell:

```
>>> from playwright.sync_api import
sync_playwright
>>> playwright = sync_playwright().start()
>>> browser =
playwright.firefox.launch(headless=False,
slow_mo=50)
>>> page = browser.new_page()
>>> page.goto('https://autbor.com/
example3.html')
<Response url='https://autbor.com/
```

```
example3.html' request=<Request
 url='https://autbor.com/example3.html'
method='GET'>>
>>> page.click('input[type=checkbox]')  #
Checks the checkbox
>>> page.click('input[type=checkbox]')  #
Unchecks the checkbox
>>> page.click('a')  # Clicks the link
>>> page.go_back()
>>> checkbox_elem =
page.get_by_role('checkbox')  # Calls a
Locator method
>>> checkbox_elem.check()  # Checks the
checkbox
>>> checkbox_elem.uncheck()  # Unchecks the
checkbox
>>>
checkbox_elem.set_checked(True)  # Checks the
checkbox
>>> checkbox_elem.set_checked(False)  #
Unchecks the checkbox
>>> page.get_by_text('is a link').click()  #
Uses a Locator method
>>> browser.close()
>>> playwright.stop()
```

The `check()` and `uncheck()` methods are more reliable than `click()` for checkboxes. The `click()` method toggles the checkbox to the opposite state, while `check()` and `uncheck()` leave them checked or unchecked no matter what state they were in before. Similarly, the `set_checked()` method allows you to pass `True` to check the checkbox or `False` to uncheck it.

Filling Out and Submitting Forms

`Locator` objects have a `fill()` method that takes a string and fills in the `<input>` or `<textarea>` element with the text. This is useful for filling out online forms, such as the login form in our *example3.html* web page:

```
>>> from playwright.sync_api import
sync_playwright
>>> playwright = sync_playwright().start()
>>> browser =
playwright.firefox.launch(headless=False,
slow_mo=50)
>>> page = browser.new_page()
>>> page.goto('https://autbor.com/
example3.html')
<Response url='https://autbor.com/
example3.html' request=<Request
url='https://autbor.com/example3.html'
method='GET'>>
>>> page.locator('#login_user').fill('your_re
al_username_here')
>>> page.locator('#login_pass').fill('your_re
al_password_here')
>>> page.locator('input[type=submit]').click(
)
>>> browser.close()
>>> playwright.stop()
```

There's also a `clear()` method, which erases all of the text currently in the element. Unlike in Selenium, there's no `submit()` method in Playwright, and you'll have to call `click()` on its `Locator` object matching the Submit button's element.

Sending Special Keys

You can also simulate keyboard key presses on elements in the web page with the `press()` method for `Locator` objects. For example, if the cursor isn't 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 `press()` calls scroll the page:

```
>>> from playwright.sync_api import
sync_playwright
>>> playwright = sync_playwright().start()
>>> browser =
playwright.firefox.launch(headless=False,
slow_mo=50)
>>> page = browser.new_page()
>>> page.goto('https://autbor.com/
example3.html')
<Response url='https://autbor.com/
example3.html' request=<Request
url='https://autbor.com/example3.html'
method='GET'>>
>>> page.locator('html').press('End')  #
Scrolls to bottom
>>> page.locator('html').press('Home')  #
Scrolls to top
>>> browser.close()
>>> playwright.stop()
```

The strings you pass to `press()` can include single character strings (such as `'a'` or `'?'`); the modification keys `'Shift'`, `'Control'`, `'Alt'`, or `'Meta'` (as in `'Control+A'`, for CTRL-A); and any of the following:

`'Backquote'`	`'Escape'`	`'ArrowDown'`
`'Minus'`	`'End'`	`'ArrowRight'`
`'Equal'`	`'Enter'`	`'ArrowUp'`
`'Backslash'`	`'Home'`	`'F1'` to `'F12'`
`'Backspace'`	`'Insert'`	`'Digit0'` to `'Digit9'`
`'Tab'`	`'PageUp'`	`'KeyA'` to `'KeyZ'`
`'Delete'`	`'PageDown'`	

Playwright can do much more beyond the functions described here. To learn more about these features, you can visit the Playwright documentation at *https://playwright.dev*. You can also find Python conference talks on Playwright by searching *https://pyvideo.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 task, you need direct control of your web browser through the Selenium and Playwright packages. These packages will allow you to log in to websites and fill out forms automatically. Because 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`, and `bs4` 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 two formats do most online APIs return their responses in?
7. What is the keyboard shortcut for opening a browser's Developer Tools?
8. How can you view (in the Developer Tools) the HTML of a specific element on a web page?
9. What CSS selector string would find the element with an `id` attribute of `main`?
10. What CSS selector string would find the elements with an `id` attribute of `highlight`?
11. Say you have a Beautiful Soup `Tag` object stored in the variable `spam` for the element `<div>Hello, world!</div>`. How could you get a string `'Hello, world!'` from the `Tag` object?
12. How would you store all the attributes of a Beautiful Soup `Tag` object in a variable named `link_elem`?

13. Running `import selenium` doesn't work. How do you properly import Selenium?
14. What's the difference between the `find_element()` and `find_elements()` methods in Selenium?
15. What methods do Selenium's `WebElement` objects have for simulating mouse clicks and keyboard keys?
16. In Playwright, what locator method call simulates pressing CTRL-A to select all the text on the page?
17. How can you simulate clicking a browser's Forward, Back, and Refresh buttons with Selenium?
18. How can you simulate clicking a browser's Forward, Back, and Refresh buttons with Playwright?

Practice Programs

For practice, write programs to do the following tasks.

Image Site Downloader

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

2048

The game 2048 is a simple game in which you combine tiles by sliding them up, down, left, or right with the arrow keys. You can actually get a fairly high score by sliding tiles in random directions. Write a program that will open the game at *https://play2048.co* 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 find every `<a>` link on the page and test whether the linked URL results in a "404 Not Found" status code. The program should print out any broken links.

[1] The answer is no.