WEBSCRAPPING

Web scraping is the term for using a program to download and process content from the web. For example, Google runs many web scraping programs to index web pages for its search engine.

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

requests Downloads files and web pages from the internet.

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

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

Project: mapIt.py with the webbrowser Module

The webbrowser module’s open() function can launch a new browser to a specified URL.

In [ ]:

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

A web browser tab will open to the URL https://inventwithpython.com/ (https://inventwithpython.com/). This is about the only thing the webbrowser module can do.

the open() function does make some interesting things possible. For example, it’s tedious to copy a street address to the clipboard and bring up a map of it on Google Maps. You could take a few steps out of this task by writing a simple script to automatically launch the map in your browser using the contents of your clipboard. This way, you only have to copy the address to a clipboard and run the script, and the map will be loaded for you.

This is what your program does:

Gets a street address from the command line arguments or clipboard

Opens the web browser to the Google Maps page for the address

This means your code will need 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 mapIt.py.

Step 1: Figure Out the URL

```
set up mapIt.py so that when you run it from the command line, like so . . .

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

Step 2: Handle the Command Line Arguments

In [ ]:

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

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

#Command line arguments are usually separated by spaces,
#but in this case, you want to interpret all of the arguments as a single string.
#ince sys.argv is a list of strings, you can pass it to the join() method, which returns a
#ou don’t want the program name in this string, so instead of sys.argv, you should pass sys
#to chop off the first element of the array.
#The final string that this expression evaluates to is stored in the address variable.
```

Step 3: Handle the Clipboard Content and Launch the Browser

In [ ]:

```
import webbrowser, sys, pyperclip
if len(sys.argv) > 1:
    # Get address from command line.
    address = ' '.join(sys.argv[1:])
else:
    # Get address from clipboard.
    address = pyperclip.paste()
webbrowser.open('https://www.google.com/maps/place/' + address)
# If there are no command line arguments, the program will assume the address is stored on
```

Downloading Files from the Web with the requests Module

The requests module lets you easily download files from the web without having to worry about complicated issues such as network errors, connection problems, and data compression.

The requests module doesn’t come with Python, so you’ll have to install it first. From the command line,

pip install --user requests.

In [ ]:

```
import requests
```

Downloading a Web Page with the requests.get() Function

The requests.get() function takes a string of a URL to download.

By calling type() on requests.get()’s return value,it returns a Response object, which contains the response that the web server gave for the request.

In [ ]:

```
import requests
res = requests.get('https://automatetheboringstuff.com/files/rj.txt')
type(res)
#the request for this web page is succeeded can be done by checking the status_code attribu
#of the Response object.
#If it is equal to the value of requests.codes.ok, then everything went fine
res.status_code == requests.codes.ok

#If the request succeeded,
#the downloaded web page is stored as a string in the Response object’s text variable.
#the call to len(res.text) shows you that it is more than 178,000 characters long.
len(res.text)

#calling print(res.text[:250]) displays only the first 250 characters.
print(res.text[:250])
```

Checking for Errors

A simpler way to check for success is to call the raise_for_status() method on the Response object. This will raise an exception if there was an error downloading the file and will do nothing if the download succeeded.

In [ ]:

```
res = requests.get('https://inventwithpython.com/page_that_does_not_exist')
>>> res.raise_for_status()
```

In [ ]:

```
#If a failed download isn’t a deal breaker for your program,
# then raise_for_status() linecan be wrapped with try and except statements to handle this
#case without crashing.
import requests
res = requests.get('https://inventwithpython.com/page_that_does_not_exist')
try:
    res.raise_for_status()
except Exception as exc:
    print('There was a problem: %s' % (exc))
```

Saving Downloaded Files to the Hard Drive

the web page can be saved to a file on hard drive with the standard open() function and write() method.

First, open the file in write binary mode by passing the string 'wb' as the second argument to open().

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

In [ ]:

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

The iter_content() method returns “chunks” of the content on each iteration through the loop. Each chunk is of the bytes data type, and you get to specify how many bytes each chunk will contain.The file RomeoAndJuliet.txt will now exist in the current working directory.The requests module simply handles downloading the contents of web pages.The write() method returns the number of bytes written to the file.

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.

Call close() to close the file.

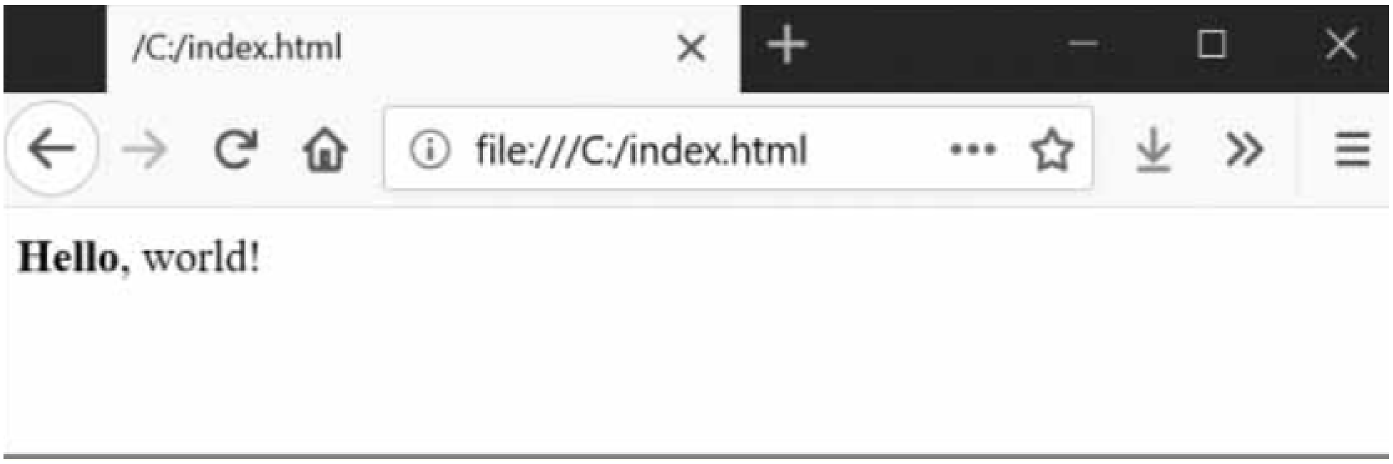
In [ ]:

```
import os
os.getcwd()
```

HTML

Hypertext Markup Language (HTML) is the format that web pages are written in. An HTML file is a plaintext file with the .html file extension. The text in these files is surrounded by tags, which are words enclosed in angle brackets. The tags tell the browser how to format the web page. A starting tag and closing tag can enclose some text to form an element. The text (or inner HTML) is the content between the starting and closing tags. For example, the following HTML will display Hello, world! in the browser, with Hello in bold:**Hello**, world!

Hello, world!



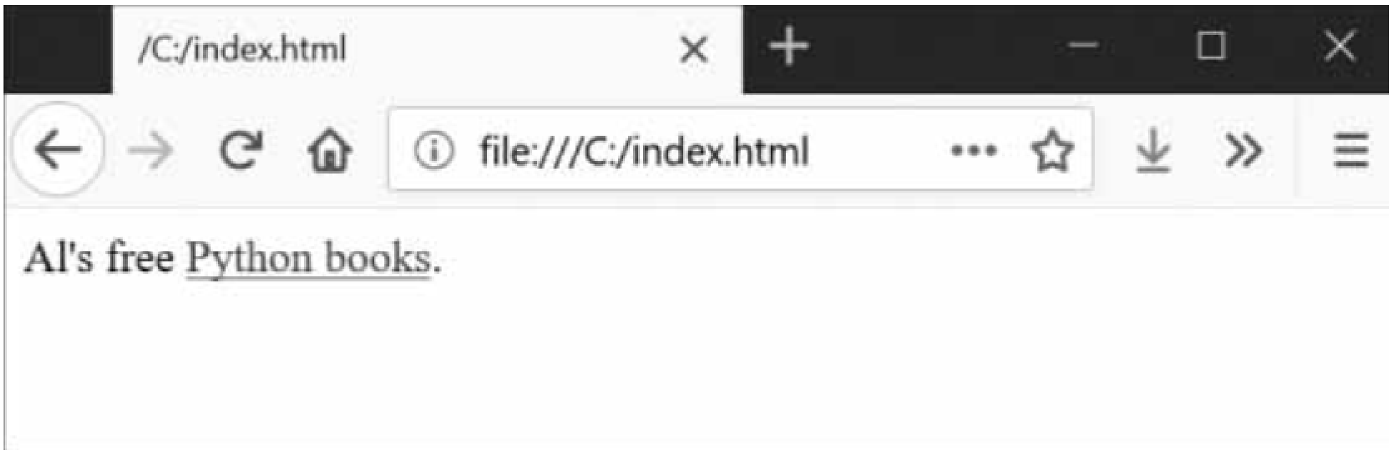
Hello, world!

In [ ]:

```
The opening <strong> tag says that the enclosed text will appear in bold. The closing </str

There are many different tags in HTML. Some of these tags have extra properties in the form

  <a href="https://inventwithpython.com">Python books</a>
```

In [ ]:

```

```

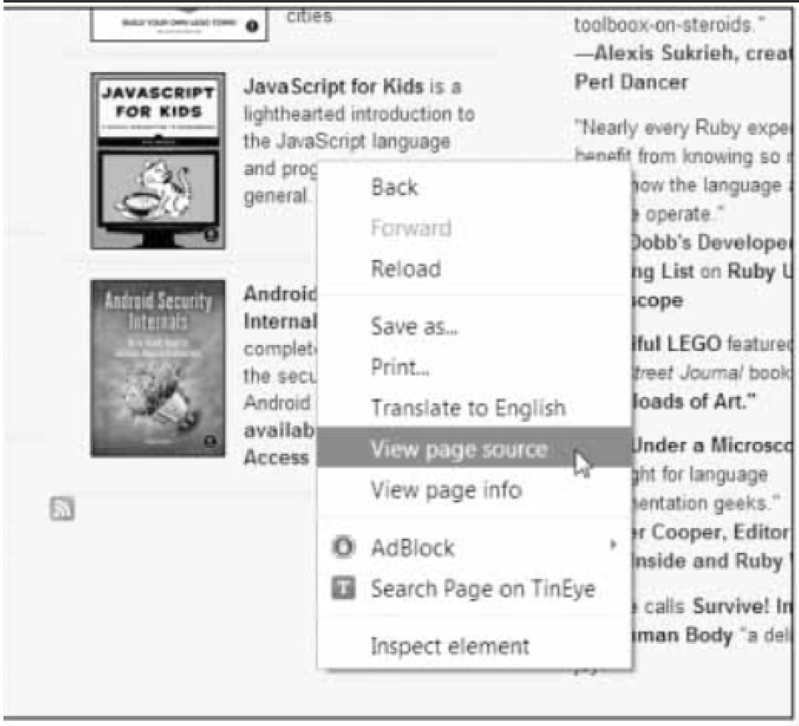


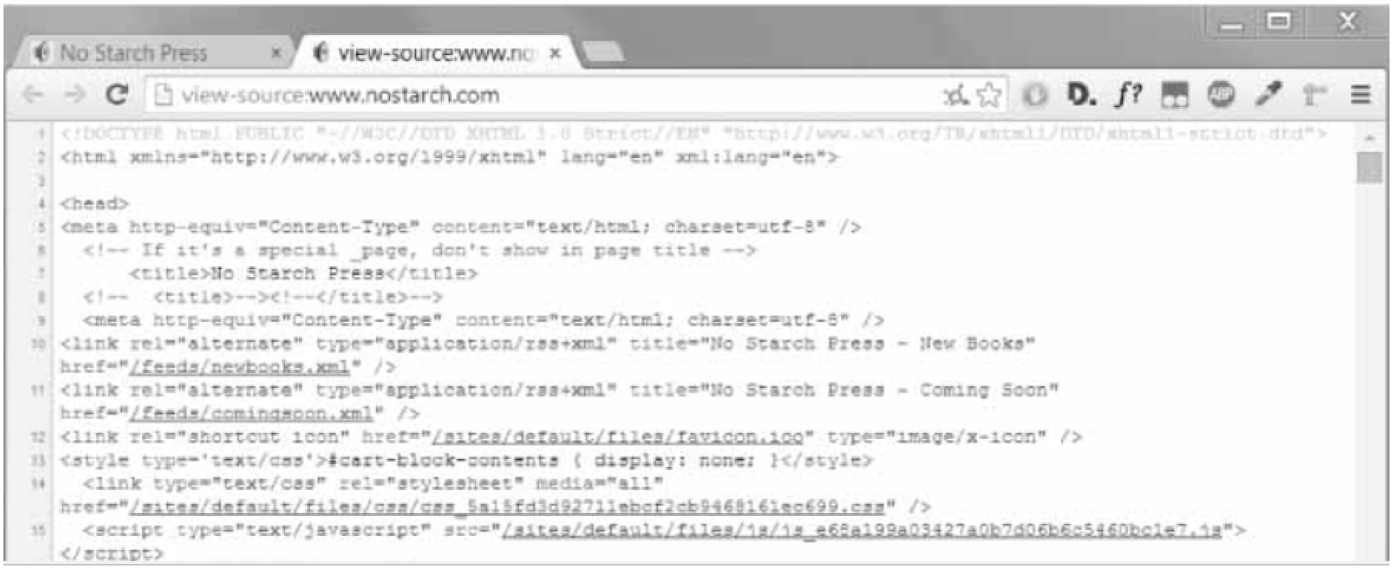
Al's free Python books.

Some elements have an id attribute that is used to uniquely identify the element in the page.

Viewing the Source HTML of a Web Page

To look at the HTML source of the web pages that the programs will work with, right-click (or CTRL-click on macOS) any web page in the web browser, and select View Source or View page source to see the HTML text of the page see fig below.





```
<!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>
```

Opening Your Browser’s Developer Tools

page’s in HTML can be looked through using browser’s developer tools.

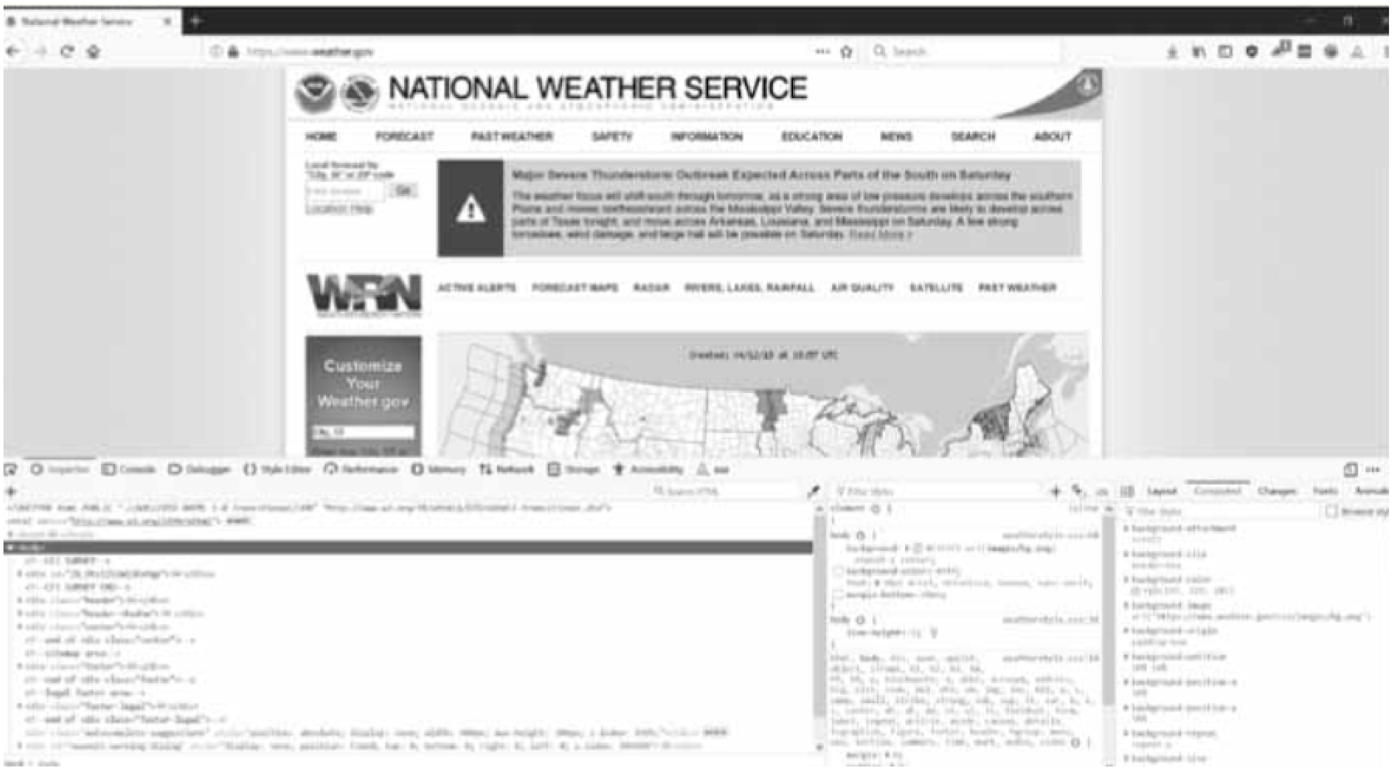
In Chrome and Internet Explorer for Windows, the developer tools are already installed, and you can press F12 to make them appear

In Chrome, you can also bring up the developer tools by selecting View ▸ Developer ▸ Developer Tools. In macOS, pressing image-OPTION-I will open Chrome’s Developer Tools.

In Firefox, you can bring up the Web Developer Tools Inspector by pressing CTRL-SHIFT-C on Windows and Linux or by pressing image-OPTION-C on macOS. The layout is almost identical to Chrome’s developer tools.

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

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

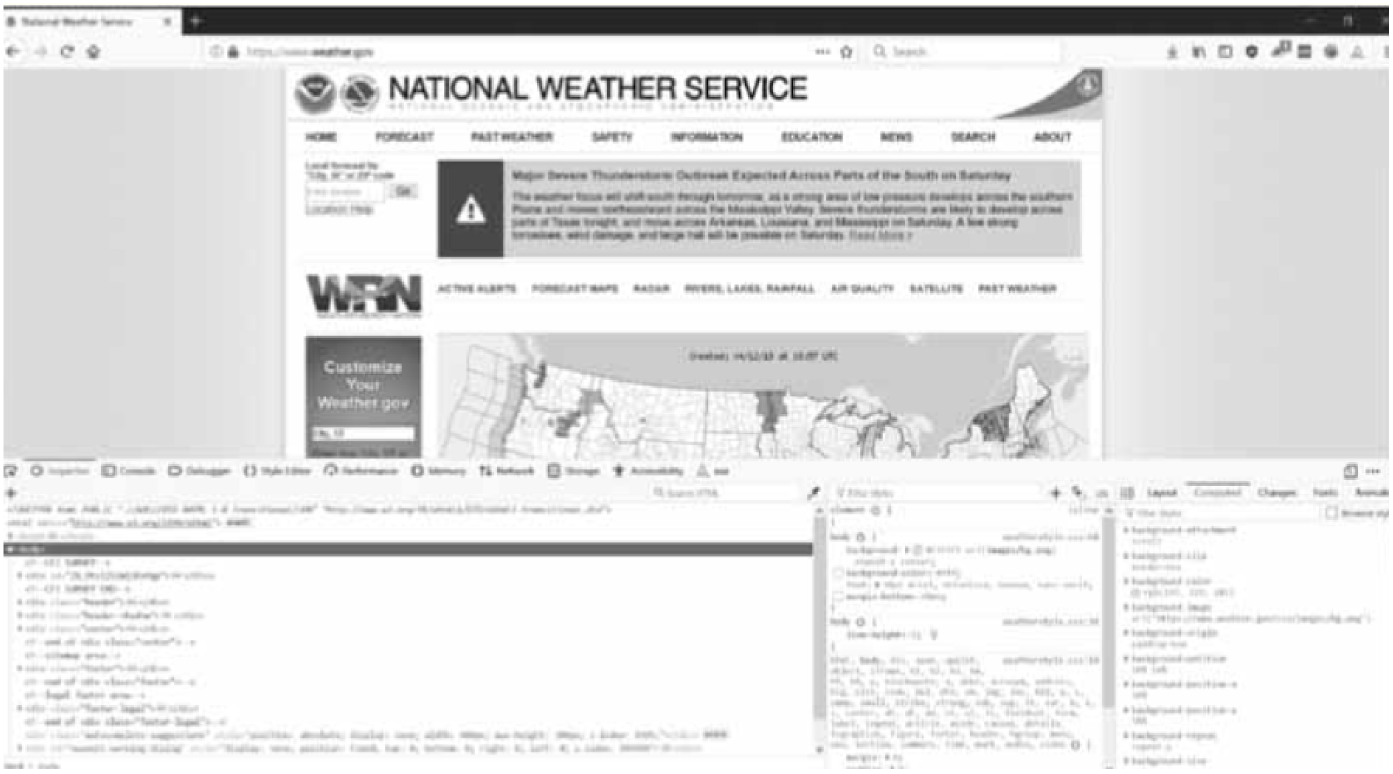


Using the Developer Tools to Find HTML Elements

Say you want to write a program to pull weather forecast data from https://weather.gov/ (https://weather.gov/). Before writing any code, do a little research. If you visit the site and search for the 94105 ZIP code, the site will take you to a page showing the forecast for that area.

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

Figure below shows the developer tools open to the HTML of the nearest forecast.



Parsing HTML with the bs4 Module

Beautiful Soup is a module for extracting information from an HTML page

The Beautiful Soup module’s name is bs4 (for Beautiful Soup, version 4).

To install it, you will need to run pip install --user beautifulsoup4 from the command line. (Check out Appendix A for instructions on installing third-party modules.)

While beautifulsoup4 is the name used for installation, to import Beautiful Soup you run import bs4.

the Beautiful Soup examples will parse (that is, analyze and identify the parts of) an HTML file on the hard drive

In [ ]:

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

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

Creating a BeautifulSoup Object from HTML

The bs4.BeautifulSoup() function needs to be called with a string containing the HTML it will parse. The bs4.BeautifulSoup() function returns a BeautifulSoup object.

In [ ]:

```
 import requests, bs4
>>> res = requests.get('https://nostarch.com')
>>> res.raise_for_status()
>>> noStarchSoup = bs4.BeautifulSoup(res.text, 'html.parser')
>>> type(noStarchSoup)
```

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

An HTML file can be loaded from hard drive by passing a File object to bs4.BeautifulSoup() along with a second argument that tells Beautiful Soup which parser to use to analyze the HTML.

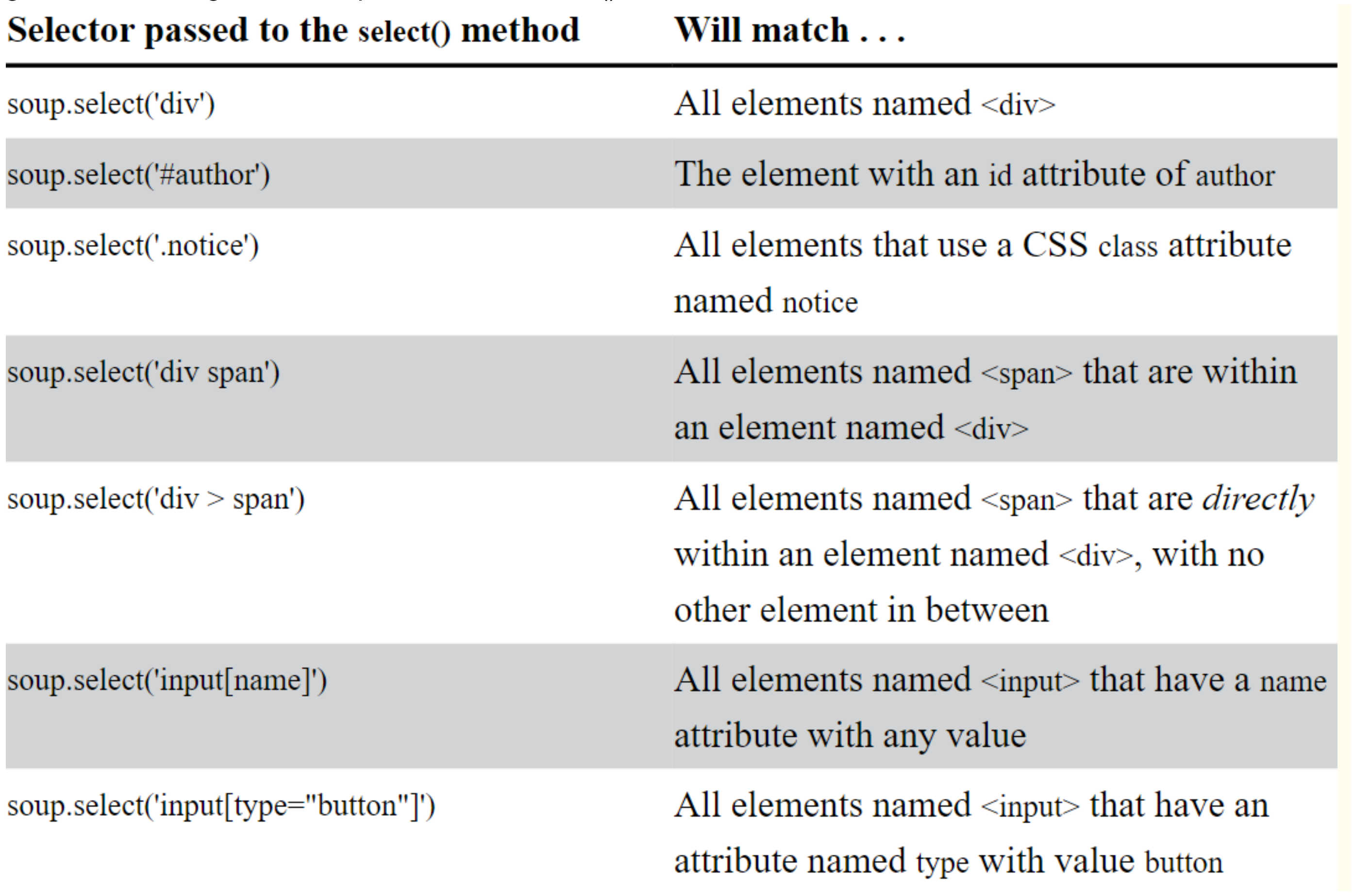
In [ ]:

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

Finding an Element with the select() Method

A web page element can be retrieved from a BeautifulSoup object by calling the select()method and passing a string of a CSS selector for the element you are looking for.

Selectors are like regular expressions: they specify a pattern to look for—in this case, in HTML pages instead of general text strings. Selector passed to the select() method



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 value button

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

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

In [ ]:

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

In [ ]:

```
print(len(elems))
print(type(elems[0]))
print(str(elems[0]) )
print(elems[0].getText())
print(elems[0].attrs)
```

In [ ]:

```
# to pull all the <p> elements from the BeautifulSoup object.
pElems = exampleSoup.select('p')
print(len(pElems))
print(  str(pElems[0]))
print(pElems[0].getText())
print(str(pElems[1]))
print(pElems[1].getText())
print(str(pElems[2]))
print(pElems[2].getText())
```

Getting Data from an Element’s Attributes

The get() method for Tag objects makes it simple to access attribute values from an element. The method is passed a string of an attribute name and returns that attribute’s value.

In [ ]:

```
import bs4
>>> soup = bs4.BeautifulSoup(open('example.html'), 'html.parser')
>>> spanElem = soup.select('span')[0]
>>> print(str(spanElem))
>>> print(spanElem.get('id'))
>>> print(spanElem.get('some_nonexistent_addr') == None)
>>> print(spanElem.attrs)
```

Project: Opening All Search Results

This is what your program does:

Gets search keywords from the command line arguments Retrieves the search results page Opens a browser tab for each result This means your code will need to do the following:

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

Project: Downloading All XKCD Comics

Here’s what your program does:

Loads the XKCD home page
Saves the comic image on that page
Follows the Previous Comic link

Repeats until it reaches the first comic

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.

Controlling the Browser with the selenium Module

The selenium module lets Python directly control the browser by programmatically clicking links and filling in login information, almost as though there were a human user interacting with the page.

Using selenium, you can interact with web pages in a much more advanced way than with requests and bs4; but because it launches a web browser, it is a bit slower and hard to run in the background

A major “tell” to websites that you’re using a script is the user-agent string, which identifies the web browser and is included in all HTTP requests. For example, the user-agent string for the requests module is something like 'python-requests/2.21.0'. You can visit a site such as https://www.whatsmyua.info/ (https://www.whatsmyua.info/) to see your user-agent string.

Using selenium, you’re much more likely to “pass for human” because not only is Selenium’s user-agent is the same as a regular browser (for instance, 'Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:65.0) Gecko/20100101 Firefox/65.0'), but it has the same traffic patterns: a selenium-controlled browser will download images, advertisements, cookies, and privacy-invading trackers just like a regular browser.

However, selenium can still be detected by websites, and major ticketing and ecommerce websites often block browsers controlled by selenium to prevent web scraping of their pages.

Starting a selenium-Controlled Browser

The following examples will show you how to control Firefox’s web browser.

Importing the modules for selenium is slightly tricky. Instead of import selenium, you need to run from selenium import webdriver.

After that, you can launch the Firefox browser with selenium.

In [ ]:

```
pip install  --user selenium
```

In [ ]:

```
pip install  webdriver-manager
```

In [ ]:

```
from selenium import webdriver
from webdriver_manager.chrome import ChromeDriverManager
browser = webdriver.Chrome(ChromeDriverManager().install())
type(browser)
browser.get('https://inventwithpython.com')
```

Finding Elements on the Page

WebDriver objects have quite a few methods for finding elements on a page. They are divided into the find_element_* and find_elements_* methods.

The find_element_* methods return a single WebElement object, representing the first element on the page that matches your query.

The find_elements_* methods return a list of WebElement_* objects for every matching element on the page.

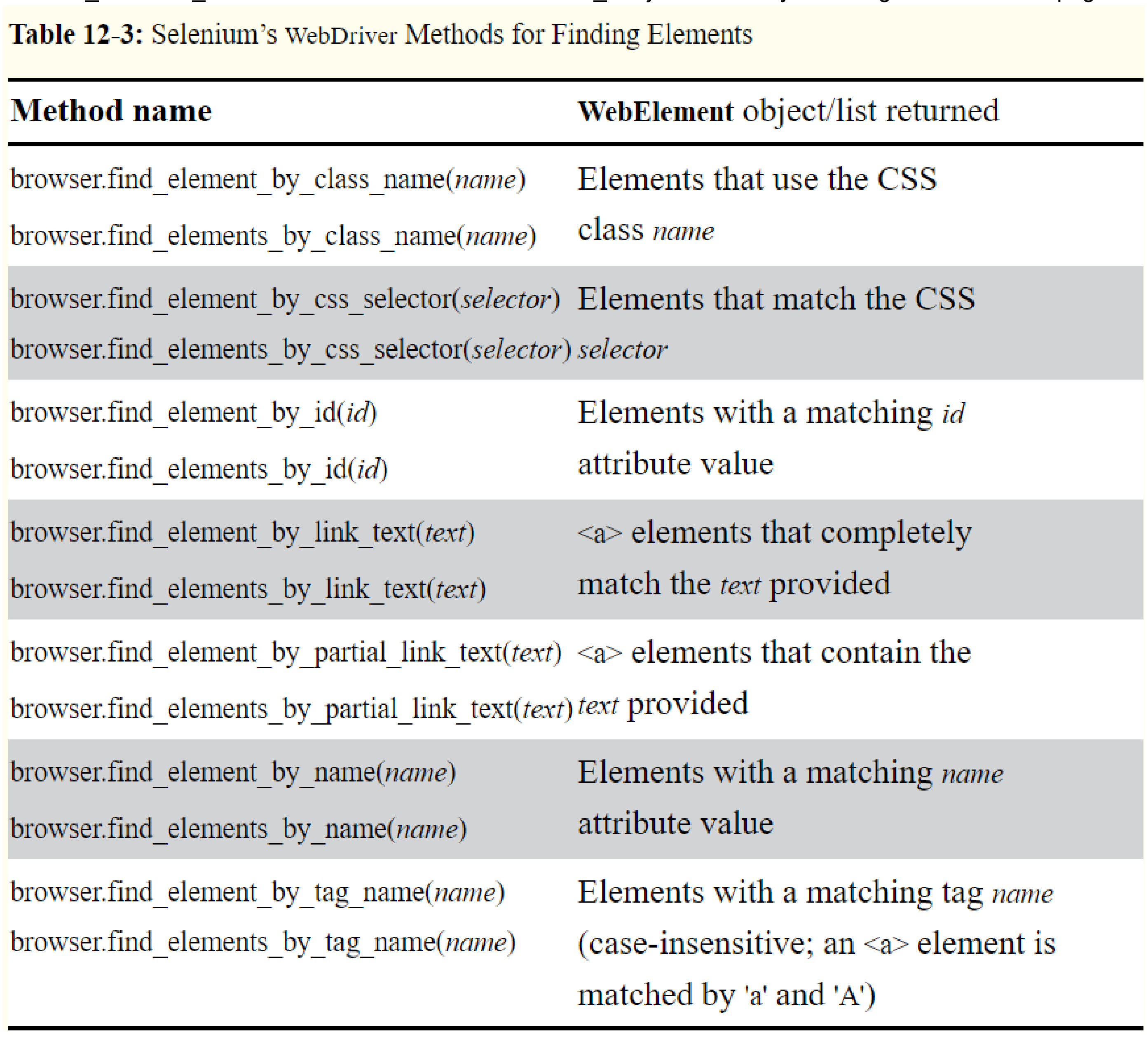


Table 12-3: Selenium's WebDriver Methods for Finding Elements

Method name	**WebElement** object/list returned

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

browser.find_element_by_css_selector(*selector*)
browser.find_elements_by_css_selector(*selector*)
	Elements that match the CSS *selector*

browser.find_element_by_id(*id*)
browser.find_elements_by_id(*id*)
	Elements with a matching *id* attribute value

browser.find_element_by_link_text(*text*)
browser.find_elements_by_link_text(*text*)
	<a> elements that completely match the *text* provided

browser.find_element_by_partial_link_text(*text*)
browser.find_elements_by_partial_link_text(*text*)
	<a> elements that contain the *text* provided

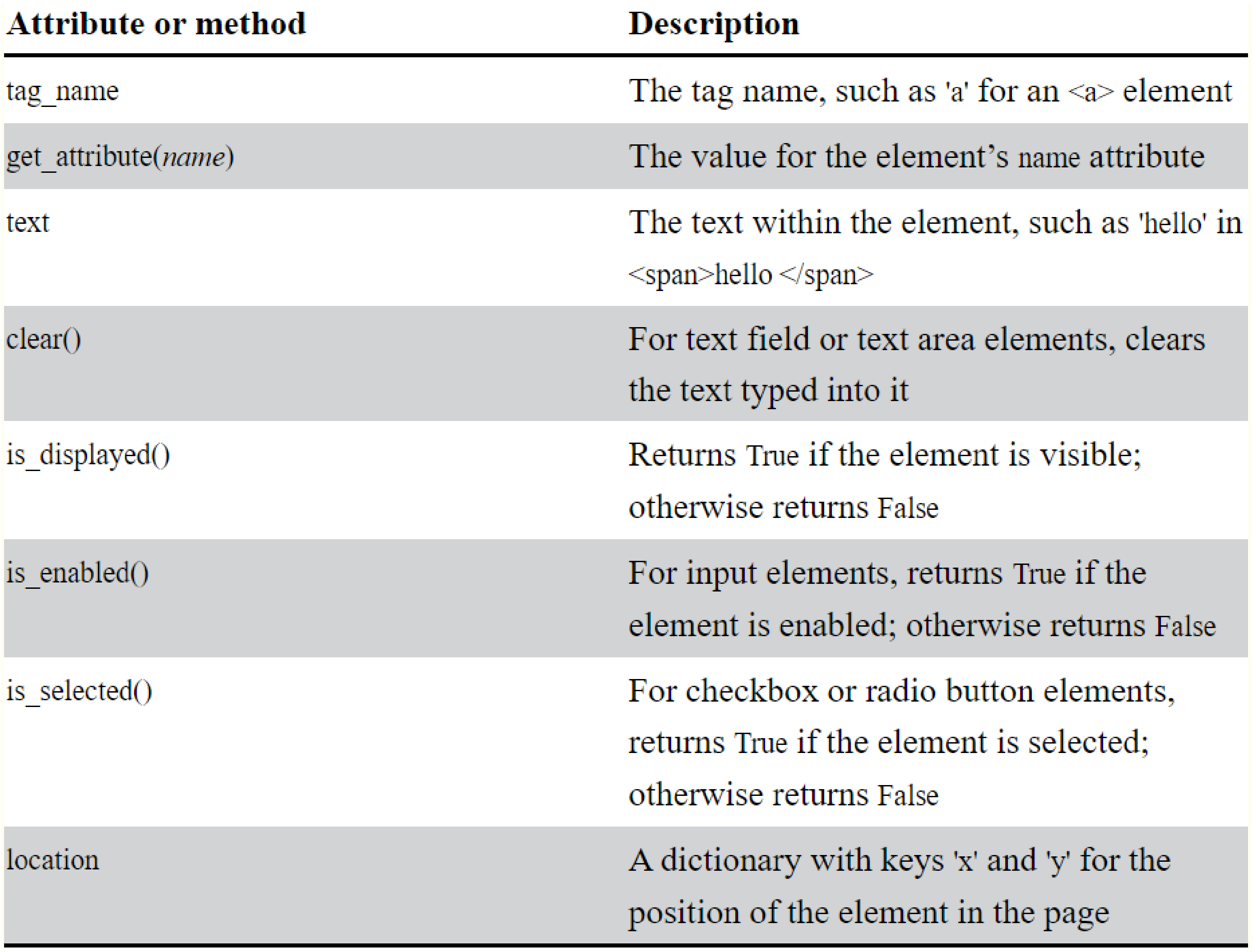
browser.find_element_by_name(*name*)
browser.find_elements_by_name(*name*)
	Elements with a matching *name* attribute value

browser.find_element_by_tag_name(*name*)
browser.find_elements_by_tag_name(*name*)
	Elements with a matching tag *name* (case-insensitive; an <a> element is matched by 'a' and 'A')

If no elements exist on the page that match what the method is looking for, the selenium module raises a NoSuchElement exception.

If you do not want this exception to crash the program, add try and except statements to the code.

Once you have the WebElement object, you can find out more about it by reading the attributes or calling the methods in Table below



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

In [ ]:

```
from selenium import webdriver
from webdriver_manager.chrome import ChromeDriverManager
browser = webdriver.Chrome(ChromeDriverManager().install())
type(browser)
browser.get('https://inventwithpython.com')
try:
    elem = browser.find_element_by_class_name('cover-thumb')
    print('Found <%s> element with that class name!' % (elem.tag_name))
except:
    print('Was not able to find an element with that name.')
```

Clicking the Page

WebElement objects returned from the find_element_* and find_elements_* methods have a click() method that simulates a mouse click on that element.

This method can be used to follow a link, make a selection on a radio button, click a Submit button, or trigger whatever else might happen when the element is clicked by the mouse.

In [ ]:

```
from selenium import webdriver
from webdriver_manager.chrome import ChromeDriverManager
browser = webdriver.Chrome(ChromeDriverManager().install())
type(browser)
browser.get('https://inventwithpython.com')
linkElem = browser.find_element_by_link_text('Read Online for Free')
print(type(linkElem))
linkElem.click() # follows the "Read Online for Free" 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. Example:

In [ ]:

```
from selenium import webdriver
from webdriver_manager.chrome import ChromeDriverManager
browser = webdriver.Chrome(ChromeDriverManager().install())
type(browser)
browser.get("https://login.metafilter.com")
userElem = browser.find_element_by_id('user_name')
userElem.send_keys('your_real_username_here')
passwordElem = browser.find_element_by_id('user_pass')
passwordElem.send_keys('your_real_password_here')
passwordElem.submit()
browser.refresh()
```

Sending Special Keys

The selenium module has a module for keyboard keys that are impossible to type into a string value, which function much like escape characters.

These values are stored in attributes in the selenium.webdriver.common.keys module.

Since that is such a long module name, it’s much easier to run from selenium.webdriver.common.keys import Keys at the top of your program;

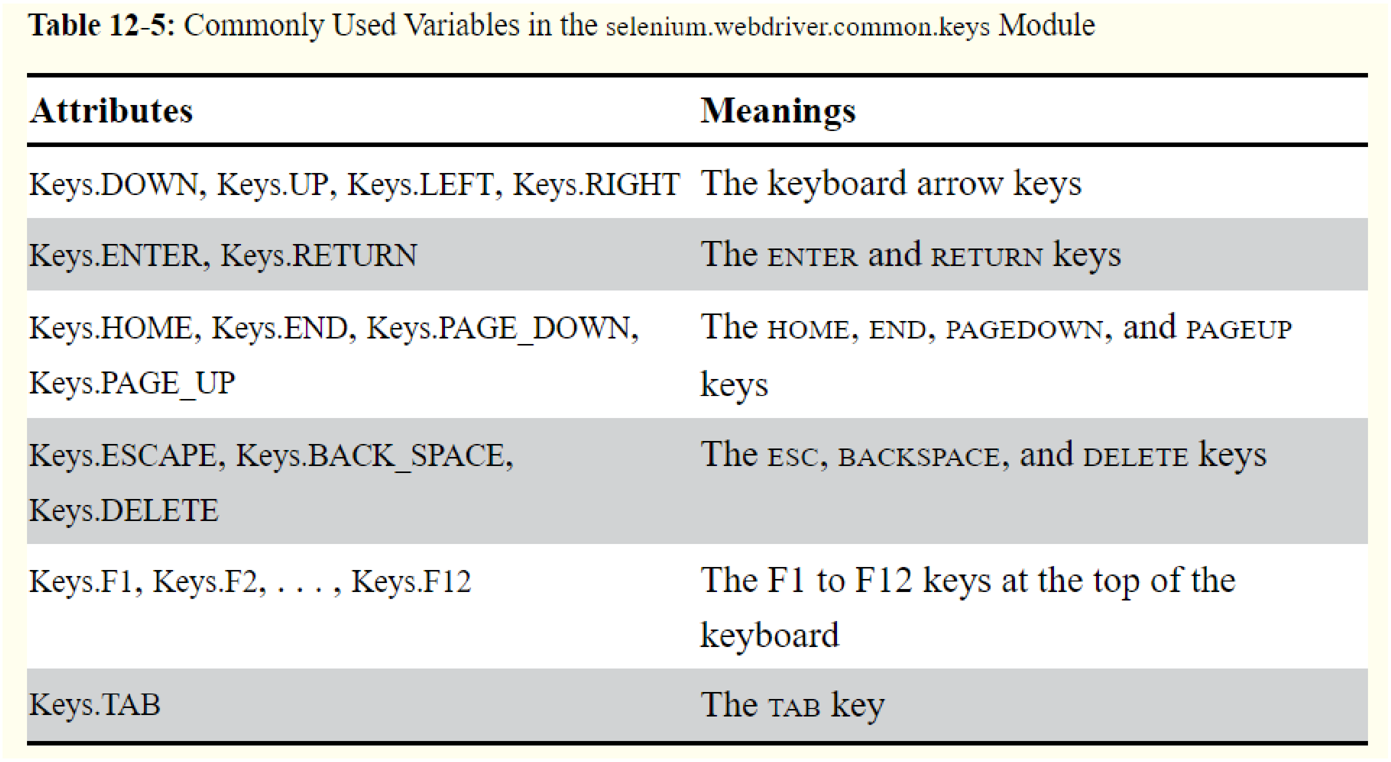


Table 12-5: Commonly Used Variables in the selenium.webdriver.common.keys Module

Attributes	Meanings
Keys.DOWN, Keys.UP, Keys.LEFT, Keys.RIGHT	The keyboard arrow keys
Keys.ENTER, Keys.RETURN	The ENTER and RETURN keys
Keys.HOME, Keys.END, Keys.PAGE_DOWN, Keys.PAGE_UP	The HOME, END, PAGEDOWN, and PAGEUP keys
Keys.ESCAPE, Keys.BACK_SPACE, Keys.DELETE	The ESC, BACKSPACE, and DELETE keys
Keys.F1, Keys.F2, . . . , Keys.F12	The F1 to F12 keys at the top of the keyboard
Keys.TAB	The TAB key

In [ ]:

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
from webdriver_manager.chrome import ChromeDriverManager
browser = webdriver.Chrome(ChromeDriverManager().install())
browser.get('https://nostarch.com')
htmlElem = browser.find_element_by_tag_name('html')
htmlElem.send_keys(Keys.END)     # scrolls to bottom
htmlElem.send_keys(Keys.HOME)    # scrolls to top
```

The html tag is the base tag in HTML files: the full content of the HTML file is enclosed within the <html> and <html> tags.

Calling browser.find_element_by_tag_name('html') is a good place to send keys to the general web page.

This would be useful if, for example, new content is loaded once you've scrolled to the bottom of the page.

Clicking Browser Buttons

The selenium module can simulate clicks on various browser buttons as well through the following methods:

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

WORKING WITH EXCEL SPREADSHEETS

Excel Documents

An Excel spreadsheet document is called a workbook. A single workbook is saved in a file with the .xlsx extension.

Each workbook can contain multiple sheets (also called worksheets). The sheet the user is currently viewing (or last viewed before closing Excel) is called the active sheet.

Each sheet has columns (addressed by letters starting at A) and rows (addressed by numbers starting at 1).

A box at a particular column and row is called a cell. Each cell can contain a number or text value. The grid of cells with data makes up a sheet.

Installing the openpyxl Module

Python does not come with OpenPyXL, so you'll have to install it.

In [ ]:

```
pip install --user -U openpyxl==2.6.2
```

In [ ]:

```
import openpyxl #to test module is installed
```

Reading Excel Documents

The examples in this chapter will use a spreadsheet named example.xlsx

Opening Excel Documents with OpenPyXL

Use the openpyxl.load_workbook() function. Enter the following into the interactive shell:

In [ ]:

```
import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> type(wb)
```

The openpyxl.load_workbook() function takes in the filename and returns a value of the workbook data type.

This Workbook object represents the Excel file, a bit like how a File object represents an opened text file.

Getting Sheets from the Workbook

To get a list of all the sheet names in the workbook can be done by accessing the sheetnames attribute.

In [ ]:

```
import openpyxl
wb = openpyxl.load_workbook('example.xlsx')
print(wb.sheetnames)# The workbook's sheets' names.
sheet = wb['Sheet3'] # Get a sheet from the workbook.
print(sheet)
print(type(sheet))
print(sheet.title) # Get the sheet's title as a string.
anotherSheet = wb.active # Get the active sheet.
print(anotherSheet)
```

Each sheet is represented by a Worksheet object, which you can obtain by using the square brackets with the sheet name string like a dictionary key.

Finally,use the active attribute of a Workbook object to get the workbook’s active sheet.

The active sheet is the sheet that’s on top when the workbook is opened in Excel.

Once you have the Worksheet object, you can get its name from the title attribute.

Getting Cells from the Sheets

With the Worksheet object,a Cellobject can be accessed by its name.

In [ ]:

```
import openpyxl
wb = openpyxl.load_workbook('example.xlsx')
sheet = wb['Sheet3'] # Get a sheet from the workbook.
print(sheet['A1']) # Get a cell from the sheet.
print(sheet['A1'].value) # Get the value from the cell.
c = sheet['B1'] # Get another cell from the sheet.
print(c.value)
# Get the row, column, and value from the cell.
print('Row %s, Column %s is %s' % (c.row, c.column, c.value))
print('Cell %s is %s' % (c.coordinate, c.value))
sheet['C1'].value
```

Specifying a column by letter can be tricky to program, especially because after column Z, the columns start by using two letters: AA, AB, AC, and so on.

As an alternative,a cell can also be got using the sheet’s cell() method and passing integers for its row and column keyword arguments.

In [ ]:

```
print(sheet.cell(row=1, column=2))
print(sheet.cell(row=1, column=2).value)
for i in range(1, 8, 2): # Go through every other row:
    print(i, sheet.cell(row=i, column=2).value)
```

Determining the size of the sheet can be done with the Worksheet object’s max_row and max_column attributes.

In [ ]:

```
import openpyxl
wb = openpyxl.load_workbook('example.xlsx')
sheet = wb['Sheet3']
print(sheet.max_row) # Get the highest row number.
print(sheet.max_column) # Get the highest column number.
```

In [ ]:

```
#To access the values of cells in a particular row or column,use a Worksheet object’s rows
#These attributes must be converted to lists with the list() function before using the squa
# and an index with them.
import openpyxl
wb = openpyxl.load_workbook('example.xlsx')
sheet = wb.active
print(list(sheet.columns)[1])
for cellObj in list(sheet.columns)[1]:
        print(cellObj.value)
```

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

Workbooks, Sheets, Cells

rundown of all the functions, methods, and data types involved in reading a cell out of a spreadsheet file:

- Import the openpyxl module.
- Call the openpyxl.load_workbook() function.
- Get a Workbook object.
- Use the active or sheetnames attributes.
- Get a Worksheet object.
- Use indexing or the cell() sheet method with row and column keyword arguments.
- Get a Cell object.
- Read the Cell object’s value attribute.

Project: Reading Data from a Spreadsheet

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

This is what your program does:

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

This means your code will need to do the following:

- Open and read the cells of an Excel document with the openpyxl module.
- Calculate all the tract and population data and store it in a data structure.
- Write the data structure to a text file with the .py extension using the pprint module.

Writing Excel Documents

Creating and Saving Excel Documents

Call the openpyxl.Workbook() function to create a new, blank Workbook object.

In [ ]:

```
import openpyxl
wb = openpyxl.Workbook() # Create a blank workbook.
print(wb.sheetnames) # It starts with one sheet.
sheet = wb.active
print(sheet.title)
sheet.title = 'Spam Bacon Eggs Sheet' # Change title.
print(wb.sheetnames)
wb.save('example_copy.xlsx')
# To save our changes, we pass a filename as a string to the save() method.
#Passing a different filename than the original, such as 'example_copy.xlsx',
#saves the changes to a copy of the spreadsheet.
```

Creating and Removing Sheets

Sheets can be added to and removed from a workbook with the create_sheet() method and del operator.

In [ ]:

```
import openpyxl
wb = openpyxl.Workbook()
print(wb.sheetnames)
print(wb.create_sheet()) # Add a new sheet.
print(wb.sheetnames)
#Create a new sheet at index 0.
print( wb.create_sheet(index=0, title='First Sheet'))
print(wb.sheetnames)
print(wb.create_sheet(index=2, title='Middle Sheet'))
print(wb.sheetnames)
wb.save("example3.xlsx")
#The create_sheet() method returns a new Worksheet object named SheetX, which by default is
#Optionally, the index and name of the new sheet can be specified with the index
#and title keyword arguments.
```

In [ ]:

```
del wb['Sheet']
wb.save("example3.xlsx")
```

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

Writing Values to Cells

Writing values to cells is much like writing values to keys in a dictionary.

In [ ]:

```
import openpyxl
wb = openpyxl.load_workbook("example3.xlsx")
sheet = wb['Sheet1']
sheet['A1'] = 'Hello, world!' # Edit the cell's value.
wb.save("example1.xlsx")
```

Project: Updating a Spreadsheet

program to update cells in a spreadsheet of produce sales. This program will look through the spreadsheet, find specific kinds of produce, and update their prices.

The Program will do the following:

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

This means your code will need to do the following:

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

The prices that you need to update are as follows:

Celery 1.19

Garlic 3.07

Lemon 1.27

In [2]:

```
import openpyxl

wb = openpyxl.load_workbook('produceSales.xlsx')
sheet = wb['Sheet']

# The produce types and their updated prices
PRICE_UPDATES = {'Garlic': 3.07,
                 'Celery': 1.19,
                 'Lemon': 1.27}
 # Loop through the rows and update the prices.
for rowNum in range(2, sheet.max_row):    # skip the first row
    produceName = sheet.cell(row=rowNum, column=1).value
    if produceName in PRICE_UPDATES:
          sheet.cell(row=rowNum, column=2).value = PRICE_UPDATES[produceName]
wb.save('updatedProduceSales.xlsx')
```

- loop through the rows starting at row 2, since row 1 is just the header.

- The cell in column 1 (that is, column A) will be stored in the variable produceName.
- If produceName exists as a key in the PRICE_UPDATES dictionary

Setting the Font Style of Cells

Styling certain cells, rows, or columns can help you emphasize important areas in your spreadsheet. To customize font styles in cells, important, import the Font() function from the openpyxl.styles module. Example that creates a new workbook and sets cell A1 to have a 24-point, italicized font.

In [3]:

```
import openpyxl
from openpyxl.styles import Font
wb = openpyxl.Workbook()
sheet = wb['Sheet']
italic24Font = Font(size=24, italic=True) # Create a font.
sheet['A1'].font = italic24Font # Apply the font to A1.
sheet['A1'] = 'Hello, world!'
wb.save('styles1.xlsx')
#In this example, Font(size=24, italic=True) returns a Font object, which is stored in ital
#The keyword arguments to Font(), size and italic, configure the Font object’s styling info
#And when sheet['A1'].font is assigned the italic24Font object,
#all that font styling information gets applied to cell A1.
```

Font Objects

To set font attributes,pass keyword arguments to Font().

Keyword Arguments for Font Objects

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

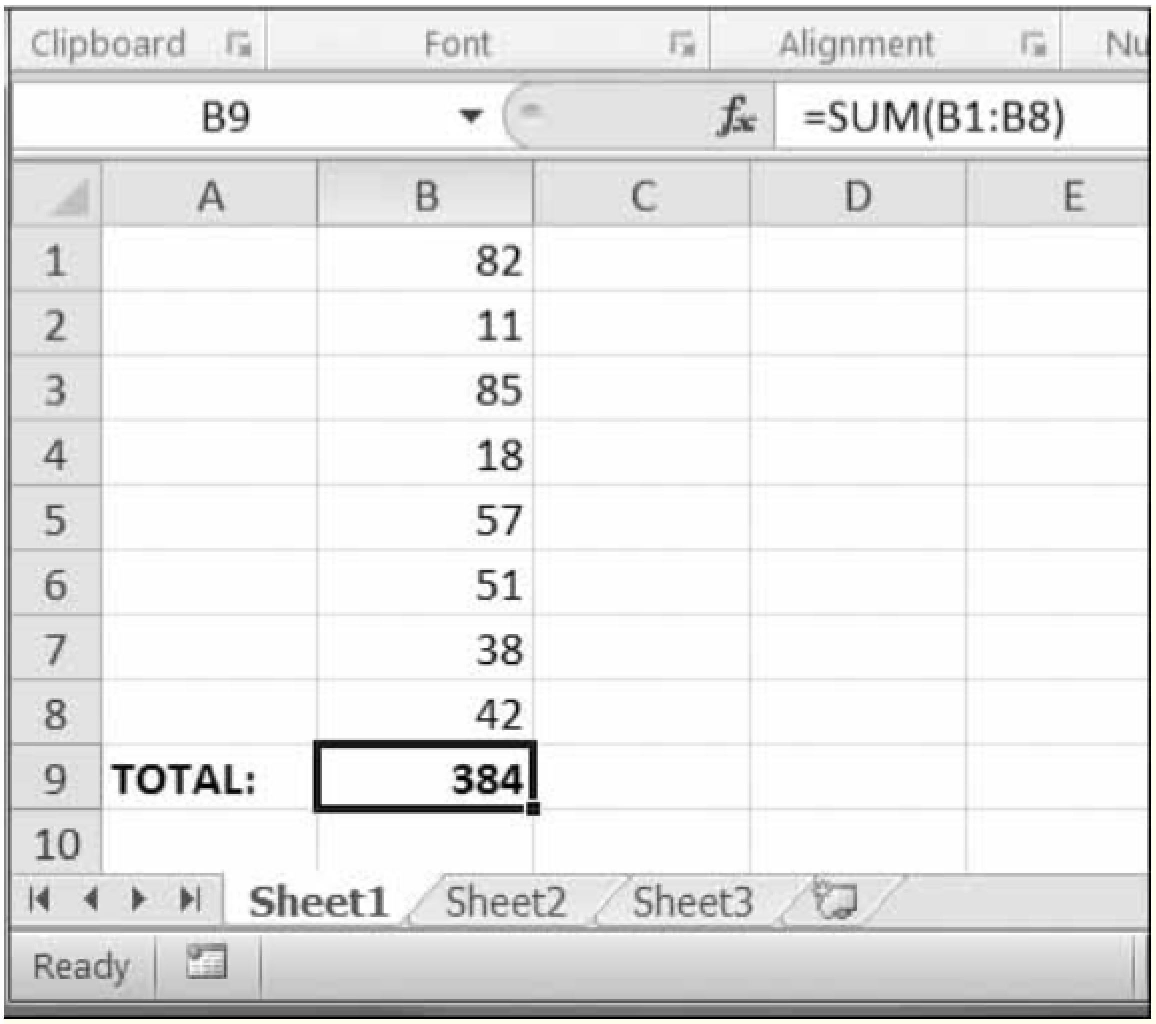
- call Font() to create a Font object and store that Font object in a variable.
- Then assign that variable to a Cell object’s font attribute. For example

In [1]:

```
import openpyxl
from openpyxl.styles import Font
wb = openpyxl.Workbook()
sheet = wb['Sheet']

fontObj1 = Font(name='Times New Roman', bold=True)
sheet['A1'].font = fontObj1
sheet['A1'] = 'Bold Times New Roman'

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

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

Formulas

```
Excel formulas, which begin with an equal sign, can configure cells to contain val
ues calculated from other cells.openpyxl module can be used to programmatically ad
d formulas to cells

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

    This will store =SUM(B1:B8) as the value in cell B9. This sets the B9 cell to
 a formula that calculates the sum of values in cells B1 to B8.
```



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

In [4]:

```
import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb.active
>>> sheet['A1'] = 200
>>> sheet['A2'] = 300
>>> sheet['A3'] = '=SUM(A1:A2)' # Set the formula.
>>> wb.save('writeFormula.xlsx')
# The cells in A1 and A2 are set to 200 and 300, respectively.
# The value in cell A3 is set to a formula that sums the values in A1 and A2. When the spre
# A3 will display its value as 500.
```

Adjusting Rows and Columns

To set a row or column's size based on its cells' contents or to set sizes in a la
rge number of spreadsheet files,it will be much quicker to write a Python program
 to do it.

Rows and columns can also be hidden entirely from view. Or they can be "frozen" in
place so that they are always visible on the screen and appear on every page when
 the spreadsheet is printed.

Setting Row Height and Column Width

```
Worksheet objects have row_dimensions and column_dimensions attributes that contro
l row heights and column widths.
```

In [7]:

```
import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb.active
>>> sheet['A1'] = 'Tall row'
>>> sheet['B2'] = 'Wide column'
# Set the height and width:
>>> sheet.row_dimensions[1].height = 0
>>> sheet.column_dimensions['B'].width = 20
>>> wb.save('dimensions.xlsx')
```

- A sheet's row_dimensions and column_dimensions are dictionary-like values,row_dimensions contains RowDimension objects and column_dimensions contains ColumnDimension objects.
- In row_dimensions, you can access one of the objects using the number of the row
- In column_dimensions, you can access one of the objects using the letter of the column
- The row height can be set to an integer or float value between 0 and 409. This value represents the height measured in points, where one point equals 1/72 of an inch. The default row height is 12.75.
- The column width can be set to an integer or float value between 0 and 255. This value represents the number of characters at the default font size (11 point) that can be displayed in the cell. The default column width is 8.43 characters.
- Columns with widths of 0 or rows with heights of 0 are hidden from the user.

Merging and Unmerging Cells

A rectangular area of cells can be merged into a single cell with the merge_cells() sheet method.

In [8]:

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

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

To unmerge cells, call the unmerge_cells() sheet method.

In [10]:

```
 import openpyxl
>>> wb = openpyxl.load_workbook('merged.xlsx')
>>> sheet = wb.active
>>> sheet.unmerge_cells('A1:D3') # Split these cells up.
>>> sheet.unmerge_cells('C5:D5')
>>> wb.save('merged1.xlsx')
```

Freezing Panes

```
For spreadsheets too large to be displayed all at once, it’s helpful to “freeze” a
few of the top rows or leftmost columns onscreen.

Frozen column or row headers, for example, are always visible to the user even as
 they scroll through the spreadsheet. These are known as freeze panes.

In OpenPyXL, each Worksheet object has a freeze_panes attribute that can be set to
a Cell object or a string of a cell’s coordinates.


all rows above and all columns to the left of this cell will be frozen, but the ro
w and column of the cell itself will not be frozen.


all rows above and all columns to the left of this cell will be frozen, but the ro
w and column of the cell itself will not be frozen.

To unfreeze all panes, set freeze_panes to None or 'A1'.
```

Example: >

In [2]:

```
import openpyxl
wb = openpyxl.load_workbook('produceSales.xlsx')
sheet = wb.active
sheet.freeze_panes = 'A2' # Freeze the rows above A2.
wb.save('freezeExample.xlsx')
```

Charts

OpenPyXL supports creating bar, line, scatter, and pie charts using the data in a sheet’s cells.

Do the following to make chart:

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

Reference objects are created by calling the openpyxl.chart.Reference() function and passing three arguments:

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

In [3]:

```
import openpyxl
wb = openpyxl.Workbook()
sheet = wb.active
for i in range(1, 11): # create some data in column A
    sheet['A' + str(i)] = i
refObj = openpyxl.chart.Reference(sheet, min_col=1, min_row=1, max_col=1,max_row=10)
seriesObj = openpyxl.chart.Series(refObj, title='First series')

chartObj = openpyxl.chart.BarChart()
chartObj.title = 'My Chart'
chartObj.append(seriesObj)

sheet.add_chart(chartObj, 'C5')
wb.save('sampleChart.xlsx')
```

creation line charts, scatter charts, and pie charts can be done by calling openpyxl.charts.LineChart(), openpyxl.chart.ScatterChart(), and openpyxl.chart.PieChart().

Working with CSV files and JSON data

- CSV stands for “comma-separated values,” and CSV files are simplified spreadsheets stored as plaintext files. Python’s csv module makes it easy to parse CSV files. \
- JSON is a format that stores information as JavaScript source code in plaintext files. (JSON is short for JavaScript Object Notation.)

The csv Module

```
Each line in a CSV file represents a row in the spreadsheet, and commas separate t
he cells in the row. For example, the spreadsheet example.xlsx would look like thi
s in a CSV file:

            4/5/2015 13:34,Apples,73
            4/5/2015 3:41,Cherries,85
            4/6/2015 12:46,Pears,14
            4/8/2015 8:59,Oranges,52
            4/10/2015 2:07,Apples,152
            4/10/2015 18:10,Bananas,23
            4/10/2015 2:40,Strawberries,98
 CSV files are simple, lacking many of the features of an Excel spreadsheet like :
- Don’t have types for their values—everything is a string
- Don’t have settings for font size or color
- Don’t have multiple worksheets
- Can’t specify cell widths and heights
- Can’t have merged cells
- Can’t have images or charts embedded in them

    The advantage of CSV files is simplicity. CSV files are widely supported by ma
ny types of programs, can be viewed in text editors, and are a straightforward way
to represent spreadsheet data.
The CSV format is exactly as advertised: it’s just a text file of comma-separated
 values.
```

reader Objects

```
    To read data from a CSV file with the csv module, create a reader object. A re
ader object lets to iterate over lines in the CSV file.
```

In [4]:

```
import csv
exampleFile = open('example.csv')
exampleReader = csv.reader(exampleFile)
exampleData = list(exampleReader)
exampleData
```

Out[4]:

```
[['4/5/2014 13:34', 'Apples', '73'],
 ['4/5/2014 3:41', 'Cherries', '85'],
 ['4/6/2014 12:46', 'Pears', '14'],
 ['4/8/2014 8:59', 'Oranges', '52'],
 ['4/10/2014 2:07', 'Apples', '152'],
 ['4/10/2014 18:10', 'Bananas', '23'],
 ['4/10/2014 2:40', 'Strawberries', '98']]
```

```
        To read a CSV file with the csv module, first open it using the open() fun
ction.
   But instead of calling the read() or readlines() method on the File object that
open() returns, pass it to the csv.reader() function. This will return a reader ob
ject to use. The most direct way to access the values in the reader object is to c
```

```
onvert it to a plain Python list by passing it to list().Using list() on this read
er object returns a list of lists, which can be stored in a variable like exampleD
ata. Entering exampleData in the shell displays the list of lists.

      To access the value at a particular row and column from CSV file can be done
with the expression exampleData[row][col], where row is the index of one of the li
sts in exampleData, and col is the index of the item you want from that list.
```

In [6]:

```
print(exampleData[0][0])
print(exampleData[0][1])
print(exampleData[0][2])
print(exampleData[1][1])
print(exampleData[6][1])
```

```
4/5/2014 13:34
Apples
73
Cherries
Strawberries
```

Reading Data from reader Objects in a for Loop

For large CSV files, the reader object is to be used in a for loop, to avoid loading the entire file into memory at once. Example:

In [7]:

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

```
Row #1 ['4/5/2014 13:34', 'Apples', '73']
Row #2 ['4/5/2014 3:41', 'Cherries', '85']
Row #3 ['4/6/2014 12:46', 'Pears', '14']
Row #4 ['4/8/2014 8:59', 'Oranges', '52']
Row #5 ['4/10/2014 2:07', 'Apples', '152']
Row #6 ['4/10/2014 18:10', 'Bananas', '23']
Row #7 ['4/10/2014 2:40', 'Strawberries', '98']
```

writer Objects

A writer object lets to write data to a CSV file.To create a writer object,use the csv.writer() function.
Example:

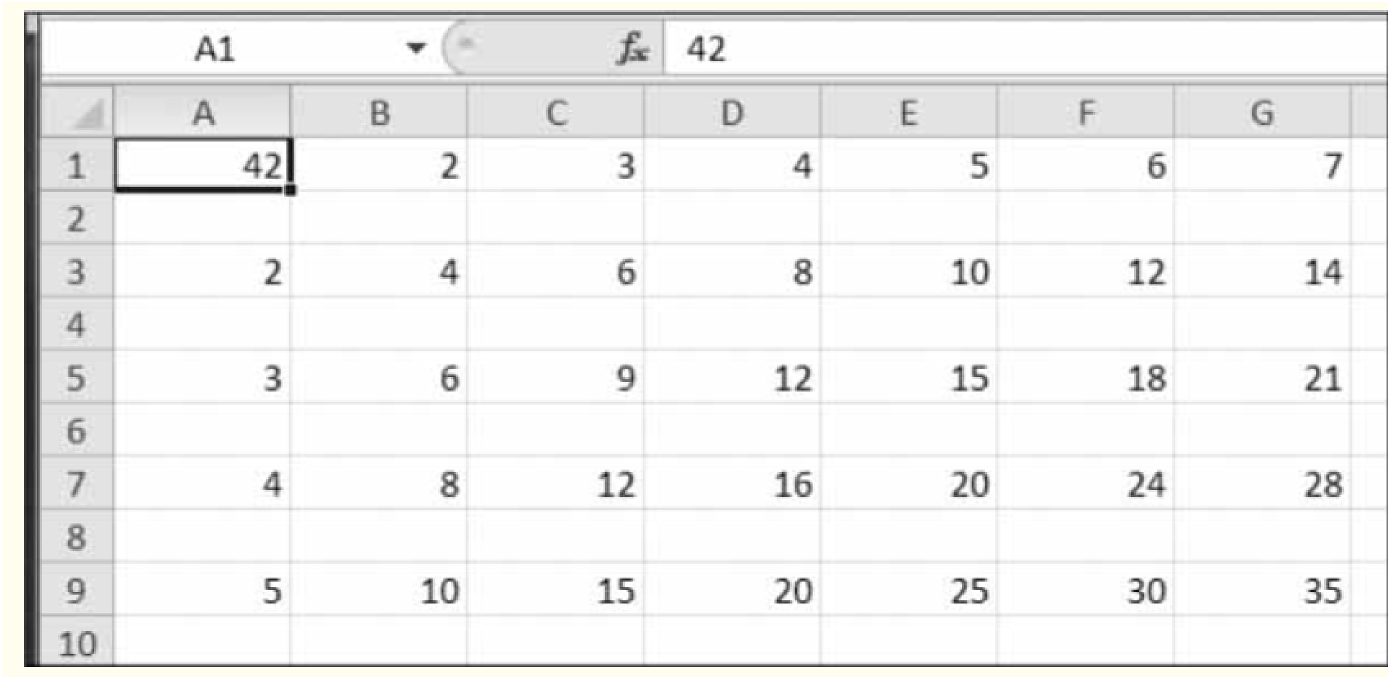
In [2]:

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

```
21
32
16
```

- First, call open() and pass it 'w' to open a file in write mode.
- This will create the object you can then pass to csv.writer() to create a writer object.
- The writerow() method for writer objects takes a list argument. Each value in the list is placed in its own cell in the output CSV file.
- The return value of writerow() is the number of characters written to the file for that row (including newline characters).

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



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

The delimiter and lineterminator Keyword Arguments

To separate cells with a tab character instead of a comma and the rows to be double-spaced do the following:

In [8]:

```
import csv
csvFile = open('example.tsv', 'w', newline='')
csvWriter = csv.writer(csvFile, delimiter='\t', lineterminator='\n\n')
csvWriter.writerow(['apples', 'oranges', 'grapes'])
csvWriter.writerow(['eggs', 'bacon', 'ham'])
csvWriter.writerow(['spam', 'spam', 'spam', 'spam', 'spam', 'spam'])
```

Out[8]:

```
31
```

- The delimiter is the character that appears between cells on a row. By default, the delimiter for a CSV file is a comma.
- The line terminator is the character that comes at the end of a row. By default, the line terminator is a newline.
  The characters can be changed to different values by using the delimiter and lineterminator keyword arguments with csv.writer().
  since the cells are separated by tabs, use the file extension .tsv, for tab-separated values.

DictReader and DictWriter CSV Objects

```
For CSV files that contain header rows, it’s often more convenient to work with th
e DictReader and DictWriter objects, rather than the reader and writer objects.
The DictReader and DictWriter CSV objects perform the same functions but use dicti
onaries instead, and they use the first row of the CSV file as the keys of these d
ictionaries.
```

In [15]:

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

```
4/5/2014 13:34 Apples 73
4/5/2014 3:41 Cherries 85
4/6/2014 12:46 Pears 14
4/8/2014 8:59 Oranges 52
4/10/2014 2:07 Apples 152
4/10/2014 18:10 Bananas 23
4/10/2014 2:40 Strawberries 98
```

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

In [9]:

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

```
4/5/2014 13:34 Apples 73
4/5/2014 3:41 Cherries 85
4/6/2014 12:46 Pears 14
4/8/2014 8:59 Oranges 52
4/10/2014 2:07 Apples 152
4/10/2014 18:10 Bananas 23
4/10/2014 2:40 Strawberries 98
```

In [18]:

```
# DictWriter objects use dictionaries to create CSV files.
import csv
outputFile = open('output.csv', 'w', newline='')
outputDictWriter = csv.DictWriter(outputFile, ['Name', 'Pet', 'Phone'])
outputDictWriter.writeheader()
print(outputDictWriter.writerow({'Name': 'Alice', 'Pet': 'cat', 'Phone': '555-1234'}))
print(outputDictWriter.writerow({'Name': 'Bob', 'Phone': '555-9999'}))
print(outputDictWriter.writerow({'Phone': '555-5555', 'Name': 'Carol', 'Pet':'dog'}))
outputFile.close()
```

```
20
15
20
```

Project: Removing the Header from CSV Files

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

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

The program will need to do the following:

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

In [49]:

```
import csv, os
os.chdir("D:\\python\\automate_online-materials")
os.makedirs('headerRemoved', exist_ok=True)
# Loop through every file in the current working directory.

for csvFilename in os.listdir('.'):
    if not csvFilename.endswith('.csv'):
               continue    # skip non-csv files
    print('Removing header from ' + csvFilename + '...')
    # Read the CSV file in (skipping first row).
    csvRows=[]
    csvFileObj=open(csvFilename)
    readerObj=csv.reader(csvFileObj)
    for row in readerObj:
        if readerObj.line_num == 1:
            continue    # skip first row
        csvRows.append(row)
    csvFileObj.close()
# Loop through every file in the current working directory.
    csvFileObj = open(os.path.join('headerRemoved', csvFilename), 'w',newline='')
    csvWriter = csv.writer(csvFileObj)
    for row in csvRows:
           csvWriter.writerow(row)
    csvFileObj.close()
```

```
Removing header from example.csv...
Removing header from exampleWithHeader.csv...
```

JSON and APIs

```
JavaScript Object Notation is a popular way to format data as a single human-reada
ble string.
JSON is useful to know, because many websites offer JSON content as a way for prog
rams to interact with the website. This is known as providing an application progr
amming interface (API).
Accessing an API is the same as accessing any other web page via a URL.
The difference is that the data returned by an API is formatted for machines; APIs
aren’t easy for people to read.
```

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

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

The json Module

- Python’s json module handles all the details of translating between a string with JSON data and Python values for the json.loads() and json.dumps() functions.

- JSON can't store every kind of Python value. It can contain values of only the following data types: strings, integers, floats, Booleans, lists, dictionaries, and NoneType.
- JSON cannot represent Python-specific objects, such as File objects, CSV reader or writer objects, Regex objects, or Selenium WebElement objects.

Reading JSON with the loads() Function

To translate a string containing JSON data into a Python value, pass it to the json.loads() function.

In [1]:

```
 stringOfJsonData = '{"name": "Zophie", "isCat": true, "miceCaught": 0,"felineIQ": null}'
import json
jsonDataAsPythonValue = json.loads(stringOfJsonData)
jsonDataAsPythonValue
```

Out[1]:

```
{'name': 'Zophie', 'isCat': True, 'miceCaught': 0, 'felineIQ': None}
```

JSON strings always use double quotes. It will return that data as a Python dictionary. Python dictionaries are not ordered, so the key-value pairs may appear in a different order when you print jsonDataAsPythonValue.

Writing JSON with the dumps() Function

The json.dumps() function (which means “dump string,” not “dumps”) will translate a Python value into a string of JSON-formatted data.

In [51]:

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

Out[51]:

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

Project: Fetching Current Weather Data

The program does the following:

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

So the code will need to do the following:

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

In [ ]:

```

```