

11

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 friends' Twitter feeds, or answer the question, "Did Kurtwood Smith have any major roles before he was in the original 1987 *Robo@op?*"¹

Since so much work on a computer involves going on the Internet, it'd be great if your programs could get online. *Web scraping* is the term for using a program to download and process content from the Web. For example, Google runs many web scraping programs to index web pages for its search engine. In this chapter, you will learn about several modules that make it easy to scrape web pages in Python.

1. The answer is no.

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

Requests Downloads files and web pages from the Internet.

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

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

Project: `mapIt.py` with the `webbrowser` Module

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('http://inventwithpython.com/')
```

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

This is what your program does:

- 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 window and save it as `mapIt.py`.

Step 1: Figure Out the URL

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

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

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

First you need to figure out what URL to use for a given street address. When you load `http://maps.google.com/` in the browser and search for an address, the URL in the address bar looks something like this: `https://www.google.com/maps/place/870+Valencia+St/@37.7590311,-122.4215096,17z/data=!3m1!4b1!4m2!3m1!1s0x808f7e3dadc07a37:0xc86b0b2bb93b73d8.`

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

Step 2: Handle the Command Line Arguments

Make your code look like this:

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

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

# TODO: Get address from clipboard.
```

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

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

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

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

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

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

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

Step 3: Handle the Clipboard Content and Launch the Browser

Make your code look like the following:

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

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

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

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

While some of the programs you write will perform huge tasks that save you hours, it can be just as satisfying to use a program that conveniently saves you a few seconds each time you perform a common task, such as getting a map of an address. Table 11-1 compares the steps needed to display a map with and without `mapIt.py`.

Table 11-1: Getting a Map with and Without *mapIt.py*

| Manually getting a map | Using <i>mapIt.py</i> |
|---|------------------------------|
| Highlight the address. | Highlight the address. |
| Copy the address. | Copy the address. |
| Open the web browser. | Run <i>mapIt.py</i> . |
| Go to http://maps.google.com/ . | |
| Click the address text field. | |
| Paste the address. | |
| Press ENTER. | |

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

Ideas for Similar Programs

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

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

Downloading Files from the Web with the `requests` Module

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

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

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

```
>>> import requests
```

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

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

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

```
>>> import requests
❶ >>> res = requests.get('http://www.gutenberg.org/cache/epub/1112/pg1112.txt')
>>> type(res)
<class 'requests.models.Response'>
❷ >>> res.status_code == requests.codes.ok
True
>>> len(res.text)
178981
>>> print(res.text[:250])
The Project Gutenberg EBook of Romeo and Juliet, by William Shakespeare
```

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

The URL goes to a text web page for the entire play of *Romeo and Juliet*, provided by Project Gutenberg ❶. You can tell that the request for this web page succeeded by checking the `status_code` attribute of the `Response` object.

If it is equal to the value of `requests.codes.ok`, then everything went fine ❷. (Incidentally, the status code for “OK” in the HTTP protocol is 200. You may already be familiar with the 404 status code for “Not Found.”)

If the request succeeded, the downloaded web page is stored as a string in the `Response` object’s `text` variable. This variable holds a large string of the entire page; the call to `len(res.text)` shows you that it is more than 178,000 characters long. Finally, calling `print(res.text[:250])` displays only the first 250 characters.

Checking for Errors

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

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

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

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

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

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

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

Saving Downloaded Files to the Hard Drive

From here, you can save the web page to a file on your hard drive with the standard `open()` function and `write()` method. There are some slight differences, though. First, you must open the file in *write binary* mode by passing the string '`wb`' as the second argument to `open()`. Even if the page is in plain-text (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.

UNICODE ENCODINGS

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

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

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

```
>>> import requests
>>> res = requests.get('http://www.gutenberg.org/cache/epub/1112/pg1112.txt')
>>> res.raise_for_status()
>>> playFile = open('RomeoAndJuliet.txt', 'wb')
>>> for chunk in res.iter_content(100000):
    playFile.write(chunk)

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

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

The file *RomeoAndJuliet.txt* will now exist in the current working directory. Note that while the filename on the website was *pg1112.txt*, the file on your hard drive has a different filename. The `requests` module simply handles downloading the contents of web pages. Once the page is downloaded, it is simply data in your program. Even if you were to lose your Internet connection after downloading the web page, all the page data would still be on your computer.

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

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

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

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

HTML

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

Resources for Learning HTML

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

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

A Quick Refresher

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

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

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

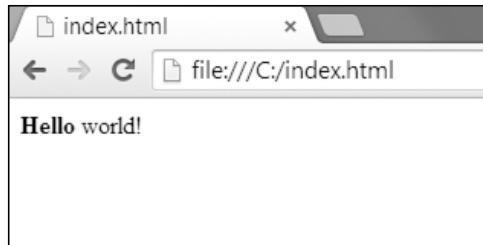


Figure 11-1: Hello world! rendered in the browser

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

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

Al's free `Python books`.

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

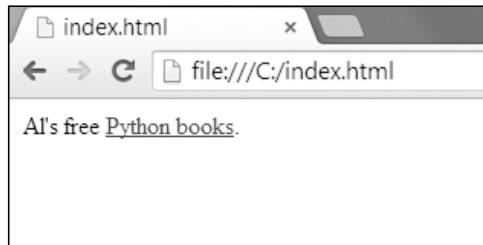


Figure 11-2: The link rendered in the browser

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

Viewing the Source HTML of a Web Page

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

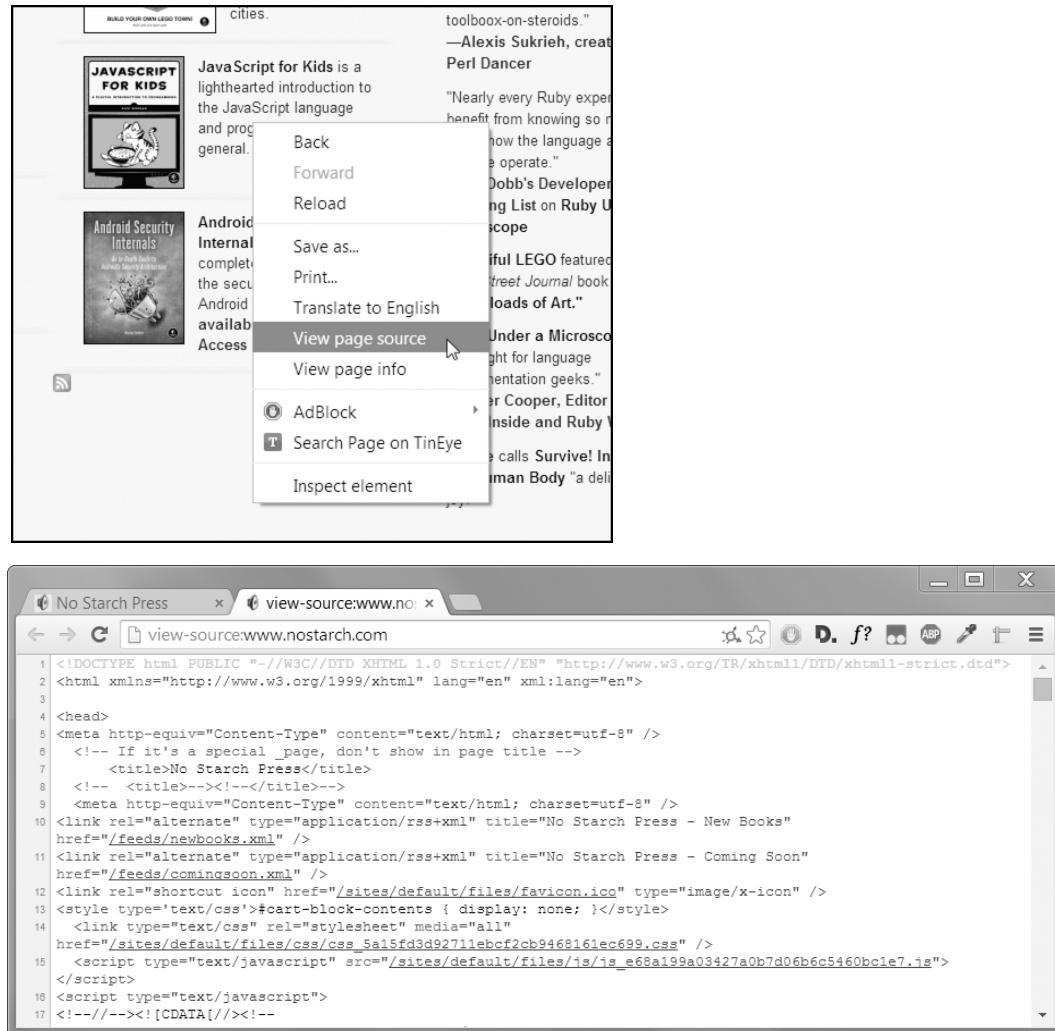


Figure 11-3: Viewing the source of a web page

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

Opening Your Browser's Developer Tools

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

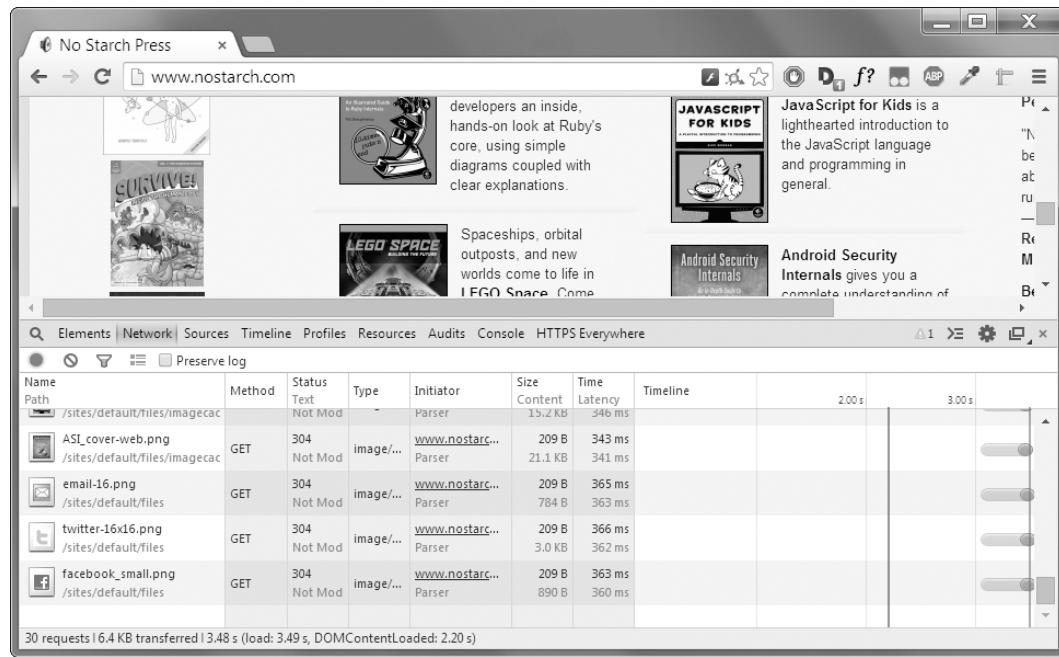


Figure 11-4: The Developer Tools window in the Chrome browser

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

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

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

DON'T USE REGULAR EXPRESSIONS TO PARSE HTML

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

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

Using the Developer Tools to Find HTML Elements

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

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

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

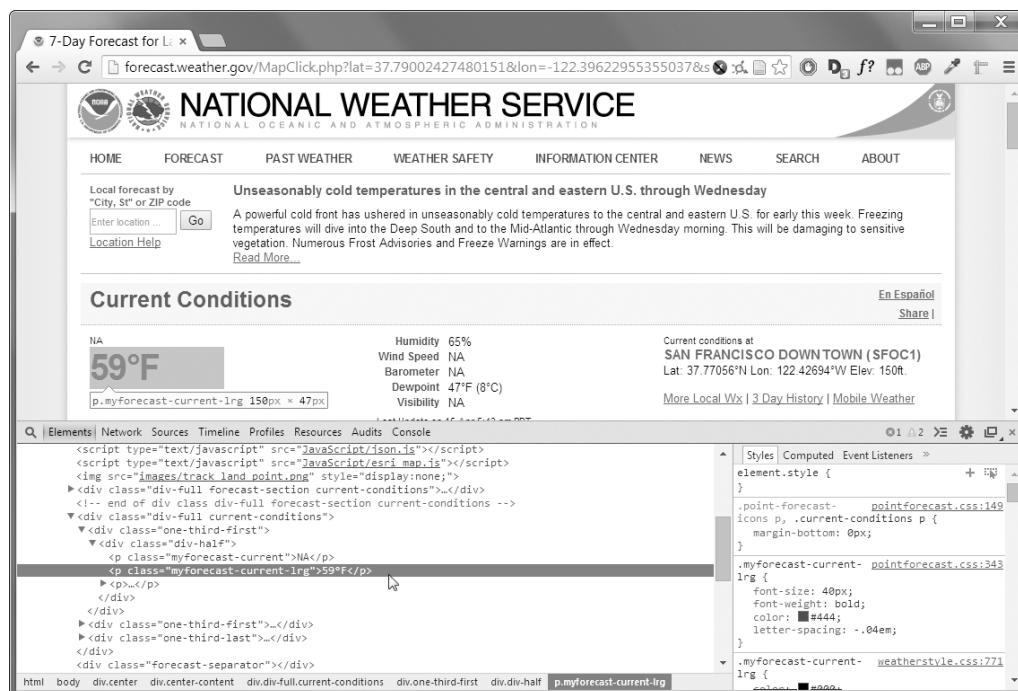


Figure 11-5: Inspecting the element that holds the temperature text with the developer tools

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

Parsing HTML with the BeautifulSoup Module

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

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

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

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

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

Creating a BeautifulSoup Object from HTML

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

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

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

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

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

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

Finding an Element with the `select()` Method

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

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

Table 11-2: Examples of CSS Selectors

| Selector passed to the <code>select()</code> method | Will match . . . |
|---|---|
| <code>soup.select('div')</code> | All elements named <code><div></code> |
| <code>soup.select('#author')</code> | The element with an <code>id</code> attribute of <code>author</code> |
| <code>soup.select('.notice')</code> | All elements that use a CSS class attribute named <code>notice</code> |
| <code>soup.select('div span')</code> | All elements named <code></code> that are within an element named <code><div></code> |
| <code>soup.select('div > span')</code> | All elements named <code></code> that are <i>directly</i> within an element named <code><div></code> , with no other element in between |
| <code>soup.select('input[name]')</code> | All elements named <code><input></code> that have a <code>name</code> attribute with <i>any</i> value |
| <code>soup.select('input[type="button"]')</code> | All elements named <code><input></code> that have an attribute named <code>type</code> with value <code>button</code> |

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

The `select()` method will return a list of `Tag` objects, which is how Beautiful Soup represents an HTML element. The list will contain one `Tag` object for every match in the `BeautifulSoup` object's HTML. `Tag` values can be passed to the `str()` function to show the HTML tags they represent.

Tag values also have an `attrs` attribute that shows all the HTML attributes of the tag as a dictionary. Using the `example.html` file from earlier, enter the following into the interactive shell:

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

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

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

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

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

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

Getting Data from an Element's Attributes

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

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

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

Project: “I’m Feeling Lucky” Google Search

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

This is what your program does:

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

This means your code will need to do the following:

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

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

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

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

Make your code look like the following:

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

import requests, sys, webbrowser, bs4

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

# TODO: Retrieve top search result links.

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

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

Step 2: Find All the Results

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

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

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

Make your code look like the following:

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

import requests, sys, webbrowser, bs4

--snip--

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

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

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

Step 3: Open Web Browsers for Each Result

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

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

import requests, sys, webbrowser, bs4

--snip--

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

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

The built-in Python function `min()` returns the smallest of the integer or float arguments it is passed. (There is also a built-in `max()` function that

returns the largest argument it is passed.) You can use `min()` to find out whether there are fewer than five links in the list and store the number of links to open in a variable named `numOpen`. Then you can run through a `for` loop by calling `range(numOpen)`.

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

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

Ideas for Similar Programs

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

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

Project: Downloading All XKCD Comics

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

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

Here's what your program does:

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

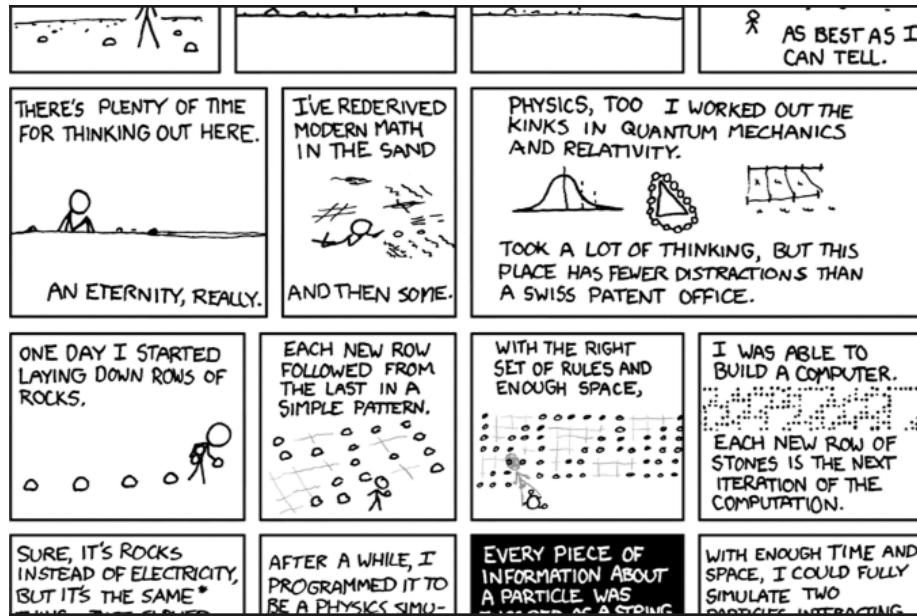


Figure 11-6: XKCD, “a webcomic of romance, sarcasm, math, and language”

This means your code will need to do the following:

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

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

Step 1: Design the Program

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

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

Make your code look like the following:

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

import requests, os, bs4

url = 'http://xkcd.com' # starting url
os.makedirs('xkcd', exist_ok=True) # store comics in ./xkcd
```

```
while not url.endswith('#'):
    # TODO: Download the page.

    # TODO: Find the URL of the comic image.

    # TODO: Download the image.

    # TODO: Save the image to ./xkcd.

    # TODO: Get the Prev button's url.

print('Done.')
```

You'll have a `url` variable that starts with the value '`http://xkcd.com`' and repeatedly update it (in a `for` loop) with the URL of the current page's `Prev` link. At every step in the loop, you'll download the comic at `url`. You'll know to end the loop when `url` ends with '#'.

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

Step 2: Download the Web Page

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

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

import requests, os, bs4

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

    soup = bs4.BeautifulSoup(res.text)

    # TODO: Find the URL of the comic image.

    # TODO: Download the image.

    # TODO: Save the image to ./xkcd.

    # TODO: Get the Prev button's url.

print('Done.')
```

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

Step 3: Find and Download the Comic Image

Make your code look like the following:

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

import requests, os, bs4

--snip--

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

# TODO: Save the image to ./xkcd.

# TODO: Get the Prev button's url.

print('Done.')
```

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

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

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

Step 4: Save the Image and Find the Previous Comic

Make your code look like the following:

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

import requests, os, bs4

--snip--

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

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

print('Done.')
```

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

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

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

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

The output of this program will look like this:

```
Downloading page http://xkcd.com...
Downloading image http://imgs.xkcd.com/comics/phone_alarm.png...
Downloading page http://xkcd.com/1358/...
```

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

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

Ideas for Similar Programs

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

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

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

Controlling the Browser with the `selenium` Module

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

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

Starting a Selenium-Controlled Browser

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

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

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

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



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

Finding Elements on the Page

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

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

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

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

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

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

Table 11-4: WebElement Attributes and Methods

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

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

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

```
try:  
    elem = browser.find_element_by_class_name('bookcover')  
    print('Found <%s> element with that class name!' % (elem.tag_name))  
except:  
    print('Was not able to find an element with that name.')
```

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

This program will output the following:

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

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

Clicking the Page

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

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

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

Filling Out and Submitting Forms

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

```
>>> from selenium import webdriver  
>>> browser = webdriver.Firefox()  
>>> browser.get('http://gmail.com')  
>>> emailElem = browser.find_element_by_id('Email')  
>>> emailElem.send_keys('not_my_real_email@gmail.com')  
>>> passwordElem = browser.find_element_by_id('Passwd')
```

```
>>> passwordElem.send_keys('12345')
>>> passwordElem.submit()
```

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

Sending Special Keys

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

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

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

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

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

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

Clicking Browser Buttons

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

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

More Information on Selenium

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

Summary

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

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

Practice Questions

1. Briefly describe the differences between the `webbrowser`, `requests`, `BeautifulSoup`, and `selenium` modules.
2. What type of object is returned by `requests.get()`? How can you access the downloaded content as a string value?
3. What `Requests` method checks that the download worked?
4. How can you get the HTTP status code of a `Requests` response?
5. How do you save a `Requests` response to a file?

6. What is the keyboard shortcut for opening a browser's developer tools?
7. How can you view (in the developer tools) the HTML of a specific element on a web page?
8. What is the CSS selector string that would find the element with an `id` attribute of `main`?
9. What is the CSS selector string that would find the elements with a CSS class of `highlight`?
10. What is the CSS selector string that would find all the `<div>` elements inside another `<div>` element?
11. What is the CSS selector string that would find the `<button>` element with a `value` attribute set to `favorite`?
12. Say you have a Beautiful Soup `Tag` object stored in the variable `spam` for the element `<div>Hello world!</div>`. How could you get a string '`Hello world!`' from the `Tag` object?
13. How would you store all the attributes of a Beautiful Soup `Tag` object in a variable named `linkElem`?
14. Running `import selenium` doesn't work. How do you properly import the `selenium` module?
15. What's the difference between the `find_element_*` and `find_elements_*` methods?
16. What methods do Selenium's `WebElement` objects have for simulating mouse clicks and keyboard keys?
17. You could call `send_keys(Keys.ENTER)` on the Submit button's `WebElement` object, but what is an easier way to submit a form with Selenium?
18. How can you simulate clicking a browser's Forward, Back, and Refresh buttons with Selenium?

Practice Projects

For practice, write programs to do the following tasks.

Command Line Emailer

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

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

Image Site Downloader

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

2048

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

Link Verification

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

12

WORKING WITH EXCEL SPREADSHEETS



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

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

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

Excel Documents

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

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

Installing the `openpyxl` Module

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

```
>>> import openpyxl
```

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

This book covers version 2.1.4 of OpenPyXL, but new versions are regularly released by the OpenPyXL team. Don't worry, though: New versions should stay backward compatible with the instructions in this book for quite some time. If you have a newer version and want to see what additional features may be available to you, you can check out the full documentation for OpenPyXL at <http://openpyxl.readthedocs.org/>.

Reading Excel Documents

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

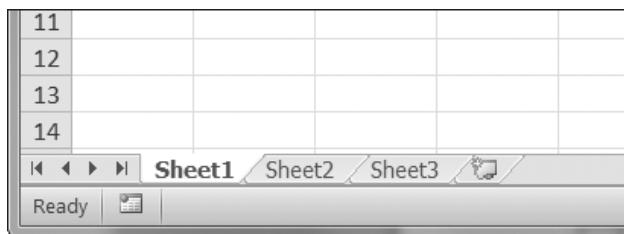


Figure 12-1: The tabs for a workbook's sheets are in the lower-left corner of Excel.

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

Table 12-1: The example.xlsx Spreadsheet

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

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

Opening Excel Documents with OpenPyXL

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

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

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

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

Getting Sheets from the Workbook

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

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

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

Getting Cells from the Sheets

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

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

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

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

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

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

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

1 Apples
3 Pears
5 Apples
7 Strawberries
```

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

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

You can determine the size of the sheet with the `Worksheet` object's `get_highest_row()` and `get_highest_column()` methods. Enter the following into the interactive shell:

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

Note that the `get_highest_column()` method returns an integer rather than the letter that appears in Excel.

Converting Between Column Letters and Numbers

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

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

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

Getting Rows and Columns from the Sheets

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

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.get_sheet_by_name('Sheet1')
>>> tuple(sheet['A1':'C3'])
((<Cell Sheet1.A1>, <Cell Sheet1.B1>, <Cell Sheet1.C1>), (<Cell Sheet1.A2>,
<Cell Sheet1.B2>, <Cell Sheet1.C2>), (<Cell Sheet1.A3>, <Cell Sheet1.B3>,
<Cell Sheet1.C3>))
❶ >>> for rowOfCellObjects in sheet['A1':'C3']:
❷     for cellObj in rowOfCellObjects:
         print(cellObj.coordinate, cellObj.value)
     print('--- END OF ROW ---')
```

```
A1 2015-04-05 13:34:02
B1 Apples
C1 73
--- END OF ROW ---
A2 2015-04-05 03:41:23
B2 Cherries
C2 85
--- END OF ROW ---
A3 2015-04-06 12:46:51
B3 Pears
C3 14
--- END OF ROW ---
```

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

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

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

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

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.get_active_sheet()
>>> sheet.columns[1]
(<Cell Sheet1.B1>, <Cell Sheet1.B2>, <Cell Sheet1.B3>, <Cell Sheet1.B4>,
<Cell Sheet1.B5>, <Cell Sheet1.B6>, <Cell Sheet1.B7>)
>>> for cell0bj in sheet.columns[1]:
    print(cell0bj.value)
```

```
Apples
Cherries
Pears
Oranges
Apples
Bananas
Strawberries
```

Using the `rows` attribute on a `Worksheet` object will give you a tuple of tuples. Each of these inner tuples represents a row, and contains the `Cell` objects in that row. The `columns` attribute also gives you a tuple of tuples, with each of the inner tuples containing the `Cell` objects in a particular

column. For *example.xlsx*, since there are 7 rows and 3 columns, `rows` gives us a tuple of 7 tuples (each containing 3 `Cell` objects), and `columns` gives us a tuple of 3 tuples (each containing 7 `Cell` objects).

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

Workbooks, Sheets, Cells

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

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

Project: Reading Data from a Spreadsheet

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

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

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

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

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

This is what your program does:

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

This means your code will need to do the following:

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

Step 1: Read the Spreadsheet Data

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

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

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

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

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

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

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

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

Step 2: Populate the Data Structure

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

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

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

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

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

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

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

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

--snip--
```

```

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

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

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

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

```

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

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

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

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

Step 3: Write the Results to a File

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

```

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

```

```
--snip--  
for row in range(2, sheet.get_highest_row() + 1):  
    --snip--  
  
    # Open a new text file and write the contents of countyData to it.  
    print('Writing results...')  
    resultFile = open('census2010.py', 'w')  
    resultFile.write('allData = ' + pprint.pformat(countyData))  
    resultFile.close()  
    print('Done.')

---


```

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

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

---


```

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

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

Ideas for Similar Programs

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

- Compare data across multiple rows in a spreadsheet.
- Open multiple Excel files and compare data between spreadsheets.

- Check whether a spreadsheet has blank rows or invalid data in any cells and alert the user if it does.
- Read data from a spreadsheet and use it as the input for your Python programs.

Writing Excel Documents

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

Creating and Saving Excel Documents

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

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

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

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

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

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

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

Creating and Removing Sheets

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

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

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

Continue the previous example by entering the following:

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

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

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

Writing Values to Cells

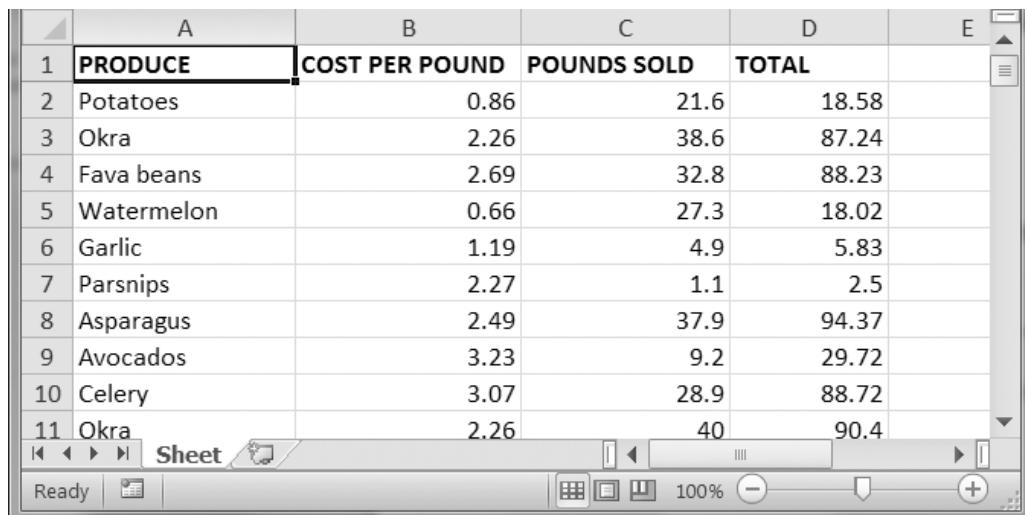
Writing values to cells is much like writing values to keys in a dictionary. Enter this into the interactive shell:

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

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

Project: Updating a Spreadsheet

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



The screenshot shows a Microsoft Excel spreadsheet titled "Sheet1". The data is organized into columns: A (PRODUCE), B (COST PER POUND), C (POUNDS SOLD), and D (TOTAL). Row 1 contains the column headers. Rows 2 through 11 list various produce items with their respective costs and total revenue. The TOTAL column uses a formula to calculate the product of COST PER POUND and POUNDS SOLD. The spreadsheet interface includes a ribbon bar at the top and standard Excel navigation tools at the bottom.

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

Figure 12-3: A spreadsheet of produce sales

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

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

Your program does the following:

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

This means your code will need to do the following:

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

Step 1: Set Up a Data Structure with the Update Information

The prices that you need to update are as follows:

| | |
|--------|------|
| Celery | 1.19 |
| Garlic | 3.07 |
| Lemon | 1.27 |

You could write code like this:

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

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

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

```
#! python3  
# updateProduce.py - Corrects costs in produce sales spreadsheet.  
  
import openpyxl  
  
wb = openpyxl.load_workbook('produceSales.xlsx')  
sheet = wb.get_sheet_by_name('Sheet')  
  
# The produce types and their updated prices  
PRICE_UPDATES = {'Garlic': 3.07,  
                 'Celery': 1.19,  
                 'Lemon': 1.27}  
  
# TODO: Loop through the rows and update the prices.
```

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

Step 2: Check All Rows and Update Incorrect Prices

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

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

--snip--

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

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

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

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

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

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

Ideas for Similar Programs

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

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

Setting the Font Style of Cells

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

To customize font styles in cells, import the `Font()` and `Style()` functions from the `openpyxl.styles` module.

```
from openpyxl.styles import Font, Style
```

This allows you to type `Font()` instead of `openpyxl.styles.Font()`. (See “Importing Modules” on page 57 to review this style of `import` statement.)

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

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

OpenPyXL represents the collection of style settings for a cell with a `Style` object, which is stored in the `Cell` object’s `style` attribute. A cell’s style can be set by assigning the `Style` object to the `style` attribute.

In this example, `Font(size=24, italic=True)` returns a `Font` object, which is stored in `italic24Font` ❶. The keyword arguments to `Font()`, `size` and `italic`, configure the `Font` object’s style attributes. This `Font` object is then passed into the `Style(font=italic24Font)` call, which returns the value you stored in `styleObj` ❷. And when `styleObj` is assigned to the cell’s `style` attribute ❸, all that font styling information gets applied to cell A1.

Font Objects

The `style` attributes in `Font` objects affect how the text in cells is displayed. To set font style attributes, you pass keyword arguments to `Font()`. Table 12-2 shows the possible keyword arguments for the `Font()` function.

Table 12-2: Keyword Arguments for Font style Attributes

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

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

```
>>> import openpyxl  
>>> from openpyxl.styles import Font, Style  
>>> wb = openpyxl.Workbook()  
>>> sheet = wb.get_sheet_by_name('Sheet')  
  
>>> fontObj1 = Font(name='Times New Roman', bold=True)  
>>> styleObj1 = Style(font=fontObj1)  
>>> sheet['A1'].style = styleObj1  
>>> sheet['A1'] = 'Bold Times New Roman'  
  
>>> fontObj2 = Font(size=24, italic=True)  
>>> styleObj2 = Style(font=fontObj2)  
>>> sheet['B3'].style = styleObj2  
>>> sheet['B3'] = '24 pt Italic'  
  
>>> wb.save('styles.xlsx')
```

Here, we store a `Font` object in `fontObj1` and use it to create a `Style` object, which we store in `styleObj1`, and then set the `A1` `Cell` object's `style` attribute to `styleObj1`. We repeat the process with another `Font` object and `Style` object to set the style of a second cell. After you run this code, the styles of the `A1` and `B3` cells in the spreadsheet will be set to custom font styles, as shown in Figure 12-4.

| | A | B | C | D |
|---|-----------------------------|---------------------|---|---|
| 1 | Bold Times New Roman | | | |
| 2 | | | | |
| 3 | | <i>24 pt Italic</i> | | |
| 4 | | | | |
| 5 | | | | |

Figure 12-4: A spreadsheet with custom font styles

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

Formulas

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

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

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

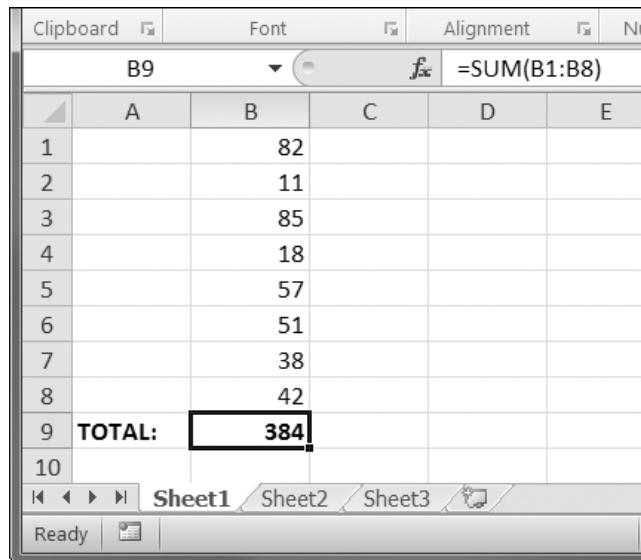


Figure 12-5: Cell B9 contains the formula `=SUM(B1:B8)`, which adds the cells B1 to B8.

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

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

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

You can also read the formula in a cell just as you would any value. However, if you want to see the result of the *calculation* for the formula instead of the literal formula, you must pass `True` for the `data_only` keyword argument to `load_workbook()`. This means a `Workbook` object can show either the formulas or the result of the formulas but not both. (But you can have multiple `Workbook` objects loaded for the same spreadsheet file.) Enter the following into the interactive shell to see the difference between loading a workbook with and without the `data_only` keyword argument:

```
>>> import openpyxl  
>>> wbFormulas = openpyxl.load_workbook('writeFormula.xlsx')  
>>> sheet = wbFormulas.get_active_sheet()  
>>> sheet['A3'].value  
'=SUM(A1:A2)'  
  
>>> wbDataOnly = openpyxl.load_workbook('writeFormula.xlsx', data_only=True)  
>>> sheet = wbDataOnly.get_active_sheet()  
>>> sheet['A3'].value  
500
```

Here, when `load_workbook()` is called with `data_only=True`, the A3 cell shows 500, the result of the `=SUM(A1:A2)` formula, rather than the text of the formula.

Excel formulas offer a level of programmability for spreadsheets but can quickly become unmanageable for complicated tasks. For example, even if you're deeply familiar with Excel formulas, it's a headache to try to decipher what `=IFERROR(TRIM(IF(LEN(VLOOKUP(F7, Sheet2!A1:B10000, 2, FALSE))>0,SUBSTITUTE(VLOOKUP(F7, Sheet2!A1:B10000, 2, FALSE), " ", ""), "")), "")` actually does. Python code is much more readable.

Adjusting Rows and Columns

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

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

Setting Row Height and Column Width

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

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

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

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

| | A | B | |
|---|----------|-------------|--|
| 1 | Tall row | | |
| 2 | | Wide column | |
| 3 | | | |

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

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

Merging and Unmerging Cells

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

```
>>> import openpyxl  
>>> wb = openpyxl.Workbook()
```

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

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

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

| | A | B | C | D | E |
|---|---|---|-------------------------------|-------------------|---|
| 1 | | | | | |
| 2 | | | | | |
| 3 | 3 | | Twelve cells merged together. | | |
| 4 | | | | | |
| 5 | 5 | | | Two merged cells. | |
| 6 | | | | | |
| 7 | | | | | |

Figure 12-7: Merged cells in a spreadsheet

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

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

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

Freeze Panes

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

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

Table 12-3: Frozen Pane Examples

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

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

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

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

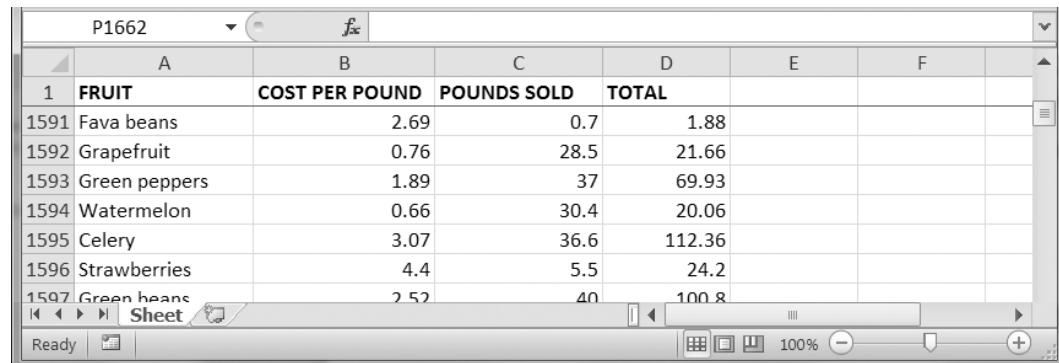


Figure 12-8: With `freeze_panes` set to '`A2`', row 1 is always visible even as the user scrolls down.

Charts

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

1. Create a `Reference` object from a rectangular selection of cells.
2. Create a `Series` object by passing in the `Reference` object.
3. Create a `Chart` object.

4. Append the Series object to the Chart object.
5. Optionally, set the drawing.top, drawing.left, drawing.width, and drawing.height variables of the Chart object.
6. Add the Chart object to the Worksheet object.

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

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

Figure 12-9 shows some sample coordinate arguments.

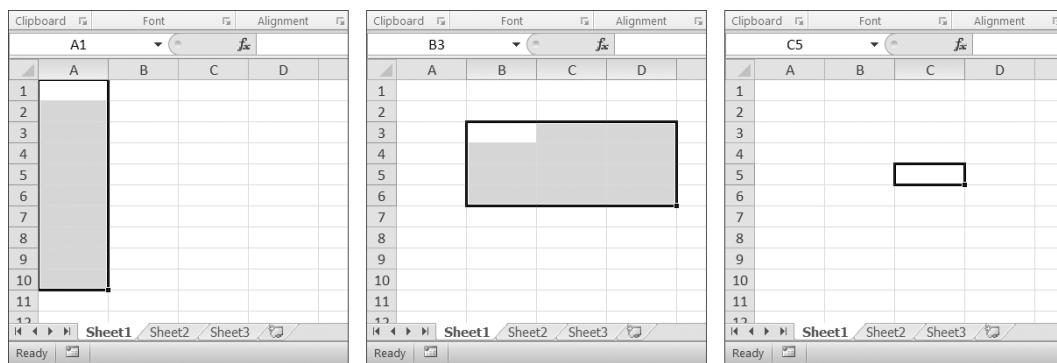


Figure 12-9: From left to right: `(1, 1), (10, 1); (3, 2), (6, 4); (5, 3), (5, 3)`

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

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

>>> ref0bj = openpyxl.charts.Reference(sheet, (1, 1), (10, 1))

>>> series0bj = openpyxl.charts.Series(ref0bj, title='First series')

>>> chart0bj = openpyxl.charts.BarChart()
>>> chart0bj.append(series0bj)
```

```

>>> chartObj.drawing.top = 50      # set the position
>>> chartObj.drawing.left = 100
>>> chartObj.drawing.width = 300   # set the size
>>> chartObj.drawing.height = 200

>>> sheet.add_chart(chartObj)
>>> wb.save('sampleChart.xlsx')

```

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

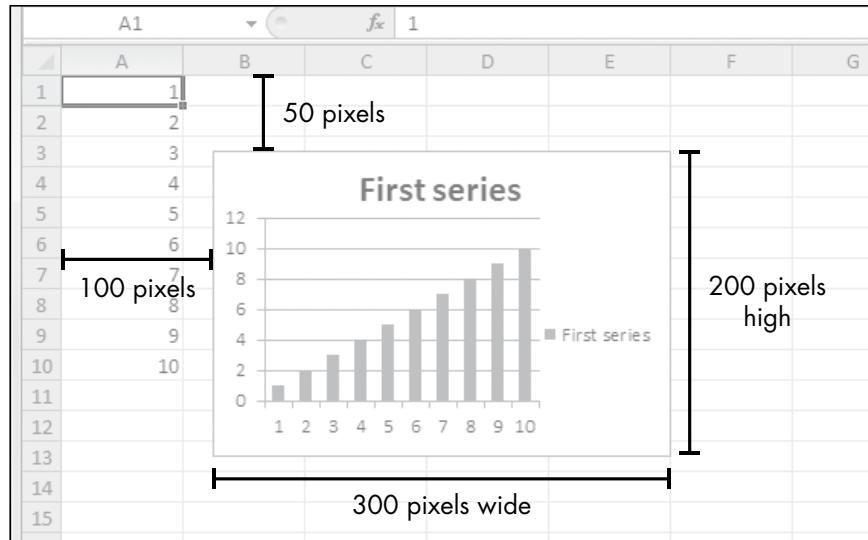


Figure 12-10: A spreadsheet with a chart added

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

Unfortunately, in the current version of OpenPyXL (2.1.4), the `load_workbook()` function does not load charts in Excel files. Even if the Excel file has charts, the loaded `Workbook` object will not include them. If you load a `Workbook` object and immediately save it to the same `.xlsx` filename, you will effectively remove the charts from it.

Summary

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

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

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

Practice Questions

For the following questions, imagine you have a `Workbook` object in the variable `wb`, a `Worksheet` object in `sheet`, a `Cell` object in `cell`, a `Comment` object in `comm`, and an `Image` object in `img`.

1. What does the `openpyxl.load_workbook()` function return?
2. What does the `get_sheet_names()` workbook method return?
3. How would you retrieve the `Worksheet` object for a sheet named 'Sheet1'?
4. How would you retrieve the `Worksheet` object for the workbook's active sheet?
5. How would you retrieve the value in the cell C5?
6. How would you set the value in the cell C5 to "Hello"?
7. How would you retrieve the cell's row and column as integers?
8. What do the `get_highest_column()` and `get_highest_row()` sheet methods return, and what is the data type of these return values?
9. If you needed to get the integer index for column 'M', what function would you need to call?
10. If you needed to get the string name for column 14, what function would you need to call?
11. How can you retrieve a tuple of all the `Cell` objects from A1 to F1?
12. How would you save the workbook to the filename `example.xlsx`?
13. How do you set a formula in a cell?
14. If you want to retrieve the result of a cell's formula instead of the cell's formula itself, what must you do first?
15. How would you set the height of row 5 to 100?
16. How would you hide column C?
17. Name a few features that OpenPyXL 2.1.4 does not load from a spreadsheet file.
18. What is a freeze pane?
19. What five functions and methods do you have to call to create a bar chart?

Practice Projects

For practice, write programs that perform the following tasks.

Multiplication Table Maker

Create a program `multiplicationTable.py` that takes a number N from the command line and creates an $N \times N$ multiplication table in an Excel spreadsheet. For example, when the program is run like this:

```
py multiplicationTable.py 6
```

... it should create a spreadsheet that looks like Figure 12-11.

Figure 12-11: A multiplication table generated in a spreadsheet

Row 1 and column A should be used for labels and should be in bold.

Blank Row Inserter

Create a program `blankRowInserter.py` that takes two integers and a filename string as command line arguments. Let's call the first integer N and the second integer M . Starting at row N , the program should insert M blank rows into the spreadsheet. For example, when the program is run like this:

```
python blankRowInserter.py 3 2 myProduce.xlsx
```

... the “before” and “after” spreadsheets should look like Figure 12-12.

| A1 | | | | | |
|----|-----------|-----------|------------|------------|------------|
| | A | B | C | D | E |
| 1 | Potatoes | Celery | Ginger | Yellow pea | Green bean |
| 2 | Okra | Okra | Corn | Garlic | Tomatoes |
| 3 | Fava bean | Spinach | Grapefruit | Grapes | Apricots |
| 4 | Watermel | Cucumber | Ginger | Watermel | Red onion |
| 5 | Garlic | Apricots | Eggplant | Cherries | Strawberri |
| 6 | Parsnips | Okra | Cucumber | Apples | Grapes |
| 7 | Asparagus | Fava bean | Green cab | Grapefruit | Ginger |
| 8 | Avocados | Watermel | Eggplant | Grapes | Strawberri |
| | | | | | Celery |

Figure 12-12: Before (left) and after (right) the two blank rows are inserted at row 3

You can write this program by reading in the contents of the spreadsheet. Then, when writing out the new spreadsheet, use a for loop to copy the first N lines. For the remaining lines, add M to the row number in the output spreadsheet.

Spreadsheet Cell Inverter

Write a program to invert the row and column of the cells in the spreadsheet. For example, the value at row 5, column 3 will be at row 3, column 5 (and vice versa). This should be done for all cells in the spreadsheet. For example, the “before” and “after” spreadsheets would look something like Figure 12-13.

| A | B | C | D | E | F | G | H | I | J |
|--------------|------|---|---|---|---|---|---|---|---|
| 1 ITEM | SOLD | | | | | | | | |
| 2 Eggplant | 334 | | | | | | | | |
| 3 Cucumber | 252 | | | | | | | | |
| 4 Green cabl | 238 | | | | | | | | |
| 5 Eggplant | 516 | | | | | | | | |
| 6 Garlic | 98 | | | | | | | | |
| 7 Parsnips | 16 | | | | | | | | |
| 8 Asparagus | 335 | | | | | | | | |
| 9 Avocados | 84 | | | | | | | | |
| 10 | | | | | | | | | |

| A | B | C | D | E | F | G | H | I | J |
|--------|----------|----------|------------|----------|--------|----------|-----------|----------|---|
| 1 ITEM | Eggplant | Cucumber | Green cabl | Eggplant | Garlic | Parsnips | Asparagus | Avocados | |
| 2 SOLD | 334 | 252 | 238 | 516 | 98 | 16 | 335 | 84 | |
| 3 | | | | | | | | | |
| 4 | | | | | | | | | |
| 5 | | | | | | | | | |
| 6 | | | | | | | | | |
| 7 | | | | | | | | | |
| 8 | | | | | | | | | |
| 9 | | | | | | | | | |
| 10 | | | | | | | | | |

Figure 12-13: The spreadsheet before (top) and after (bottom) inversion

You can write this program by using nested for loops to read in the spreadsheet's data into a list of lists data structure. This data structure could have `sheetData[x][y]` for the cell at column x and row y. Then, when writing out the new spreadsheet, use `sheetData[y][x]` for the cell at column x and row y.

Text Files to Spreadsheet

Write a program to read in the contents of several text files (you can make the text files yourself) and insert those contents into a spreadsheet, with one line of text per row. The lines of the first text file will be in the cells of column A, the lines of the second text file will be in the cells of column B, and so on.

Use the `readlines()` File object method to return a list of strings, one string per line in the file. For the first file, output the first line to column 1, row 1. The second line should be written to column 1, row 2, and so on. The next file that is read with `readlines()` will be written to column 2, the next file to column 3, and so on.

Spreadsheet to Text Files

Write a program that performs the tasks of the previous program in reverse order: The program should open a spreadsheet and write the cells of column A into one text file, the cells of column B into another text file, and so on.

13

WORKING WITH PDF AND WORD DOCUMENTS



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

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

PDF Documents

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

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

THE PROBLEMATIC PDF FORMAT

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

Extracting Text from PDFs

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

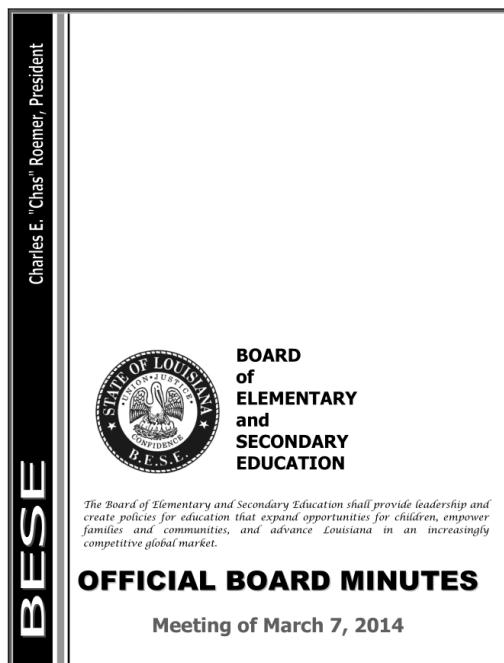


Figure 13-1: The PDF page that we will be extracting text from

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

```
>>> import PyPDF2
>>> pdfFileObj = open('meetingminutes.pdf', 'rb')
>>> pdfReader = PyPDF2.PdfFileReader(pdfFileObj)
❶ >>> pdfReader.numPages
19
❷ >>> pageObj = pdfReader.getPage(0)
❸ >>> pageObj.extractText()
'OOFFFFIICCIAALL BBOOAARRDD MMIINNUUTTEESS Meeting of March 7, 2015
\n      The Board of Elementary and Secondary Education shall provide leadership
and create policies for education that expand opportunities for children,
empower families and communities, and advance Louisiana in an increasingly
competitive global market. BOARD of ELEMENTARY and SECONDARY EDUCATION '
```

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

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

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

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

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

Decrypting PDFs

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

```
>>> import PyPDF2
>>> pdfReader = PyPDF2.PdfFileReader(open('encrypted.pdf', 'rb'))
❶ >>> pdfReader.isEncrypted
True
```

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

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

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

Creating PDFs

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

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

1. Open one or more existing PDFs (the source PDFs) into `PdfFileReader` objects.
2. Create a new `PdfFileWriter` object.
3. Copy pages from the `PdfFileReader` objects into the `PdfFileWriter` object.
4. Finally, use the `PdfFileWriter` object to write the output PDF.

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

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

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

Copying Pages

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

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

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

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

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

❾ >>> pdfOutputFile = open('combinedminutes.pdf', 'wb')
>>> pdfWriter.write(pdfOutputFile)
>>> pdfOutputFile.close()
>>> pdf1File.close()
>>> pdf2File.close()
```

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

Next, copy all the pages from the two source PDFs and add them to the `PdfFileWriter` object. Get the `Page` object by calling `getPage()` on a `PdfFileReader` object ❹. Then pass that `Page` object to your `PdfFileWriter`’s `addPage()` method ❺. These steps are done first for `pdf1Reader` and then

again for `pdf2Reader`. When you’re done copying pages, write a new PDF called `combinedminutes.pdf` by passing a `File` object to the `PdfFileWriter`’s `write()` method ❶.

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

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

Rotating Pages

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

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

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

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

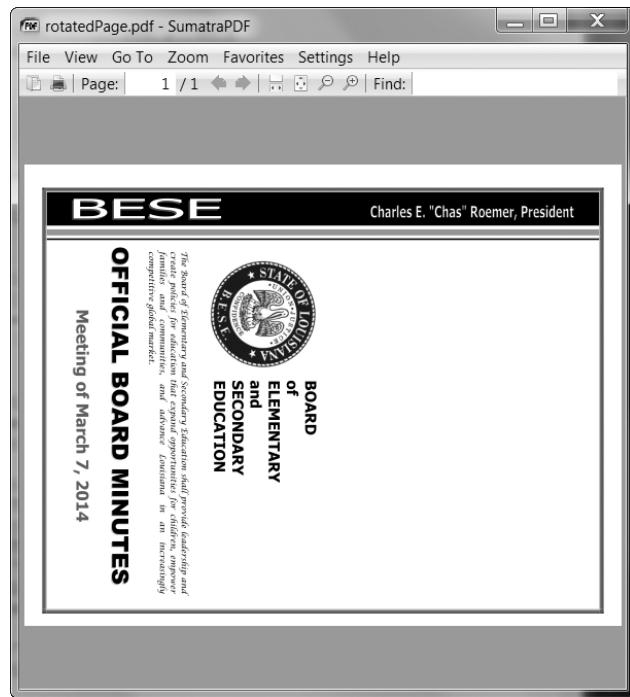


Figure 13-2: The rotatedPage.pdf file with the page rotated 90 degrees clockwise

Overlaying Pages

PyPDF2 can also overlay the contents of one page over another, which is useful for adding a logo, timestamp, or watermark to a page. With Python, it's easy to add watermarks to multiple files and only to pages your program specifies.

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

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

❼ >>> for pageNum in range(1, pdfReader.numPages):
    pageObj = pdfReader.getPage(pageNum)
    pdfWriter.addPage(pageObj)
```

```
>>> resultPdfFile = open('watermarkedCover.pdf', 'wb')
>>> pdfWriter.write(resultPdfFile)
>>> minutesFile.close()
>>> resultPdfFile.close()
```

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

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

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



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

Encrypting PDFs

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

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

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

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

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

Project: Combining Select Pages from Many PDFs

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

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

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

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

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

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

Step 1: Find All PDF Files

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

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

❶ import PyPDF2, os

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

❹ pdfWriter = PyPDF2.PdfFileWriter()

    # TODO: Loop through all the PDF files.

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

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

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

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

Step 2: Open Each PDF

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

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

import PyPDF2, os

    # Get all the PDF filenames.
    pdfFiles = []
    --snip--
```

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

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

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

Step 3: Add Each Page

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

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

import PyPDF2, os

--snip--

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

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

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

Step 4: Save the Results

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

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

```
import PyPDF2, os

--snip--

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

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

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

Ideas for Similar Programs

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

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

Word Documents

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

NOTE

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

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

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

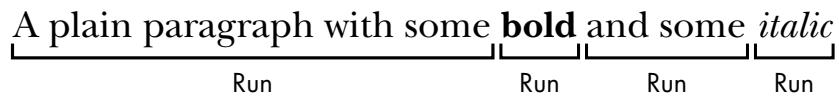


Figure 13-4: The Run objects identified in a Paragraph object

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

Reading Word Documents

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

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

At ❶, we open a *.docx* file in Python, call `docx.Document()`, and pass the filename `demo.docx`. This will return a Document object, which has a `paragraphs` attribute that is a list of Paragraph objects. When we call `len()` on `doc.paragraphs`, it returns 7, which tells us that there are seven Paragraph objects in this document ❷. Each of these Paragraph objects has a `text`

attribute that contains a string of the text in that paragraph (without the style information). Here, the first text attribute contains 'DocumentTitle' ❸, and the second contains 'A plain paragraph with some bold and some italic' ❹.

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

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

Getting the Full Text from a .docx File

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

```
#! python3

import docx

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

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

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

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

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

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

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

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

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

Styling Paragraph and Run Objects

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

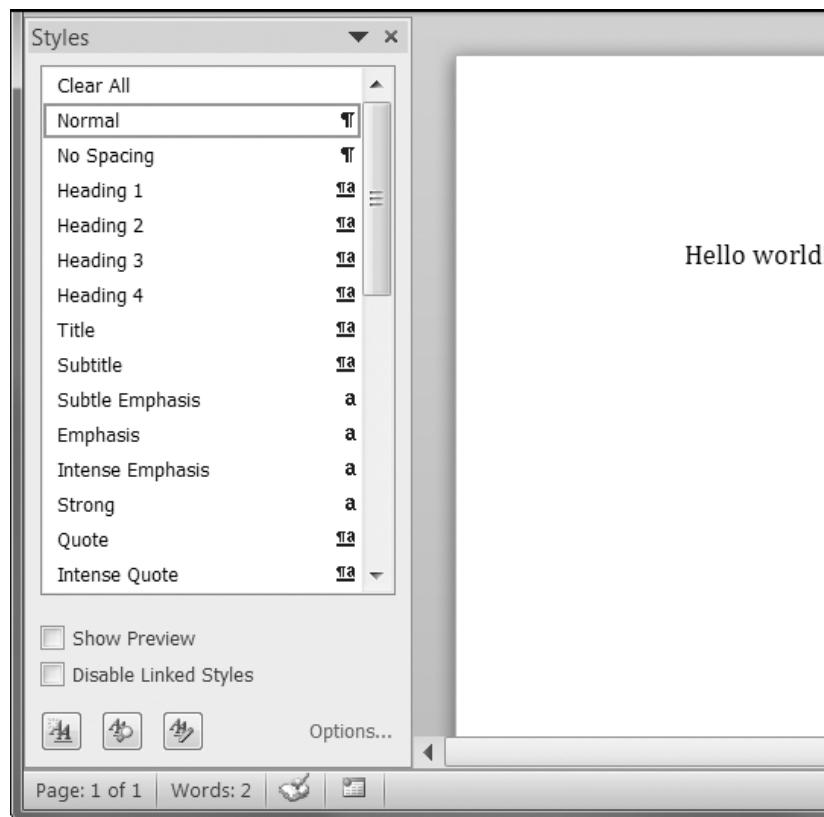


Figure 13-5: Display the Styles pane by pressing **CTRL-ALT-SHIFT-S** on Windows.

Word and other word processors use styles to keep the visual presentation of similar types of text consistent and easy to change. For example, perhaps you want to set body paragraphs in 11-point, Times New Roman, left-justified, ragged-right text. You can create a style with these settings and

assign it to all body paragraphs. Then, if you later want to change the presentation of all body paragraphs in the document, you can just change the style, and all those paragraphs will be automatically updated.

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

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

| | | | |
|-------------|----------------|-----------------|-----------------|
| 'Normal' | 'Heading5' | 'ListBullet' | 'ListParagraph' |
| 'BodyText' | 'Heading6' | 'ListBullet2' | 'MacroText' |
| 'BodyText2' | 'Heading7' | 'ListBullet3' | 'NoSpacing' |
| 'BodyText3' | 'Heading8' | 'ListContinue' | 'Quote' |
| 'Caption' | 'Heading9' | 'ListContinue2' | 'Subtitle' |
| 'Heading1' | 'IntenseQuote' | 'ListContinue3' | 'TOCHeading' |
| 'Heading2' | 'List' | 'ListNumber' | 'Title' |
| 'Heading3' | 'List2' | 'ListNumber2' | |
| 'Heading4' | 'List3' | 'ListNumber3' | |

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

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

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

Creating Word Documents with Nondefault Styles

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

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

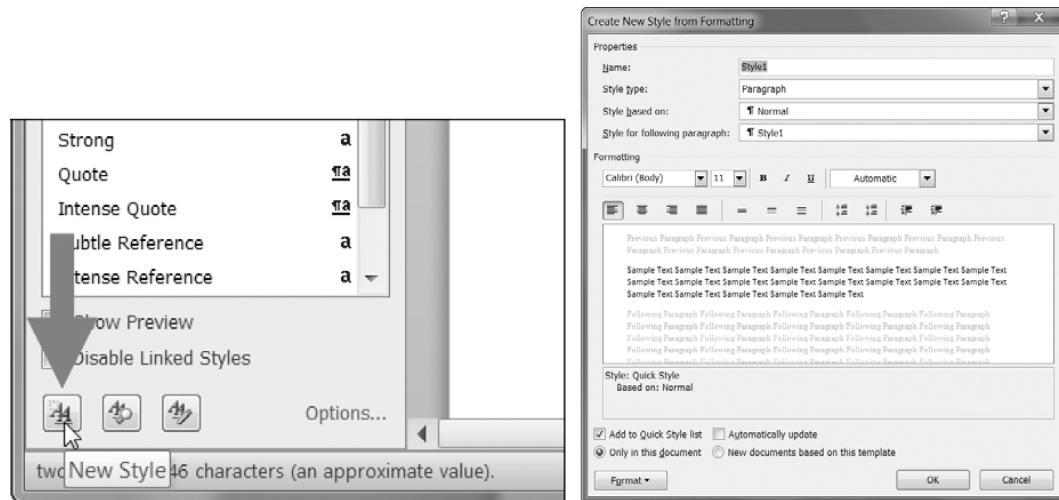


Figure 13-6: The New Style button (left) and the Create New Style from Formatting dialog (right)

Run Attributes

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

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

Table 13-1: Run Object text Attributes

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

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

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

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

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

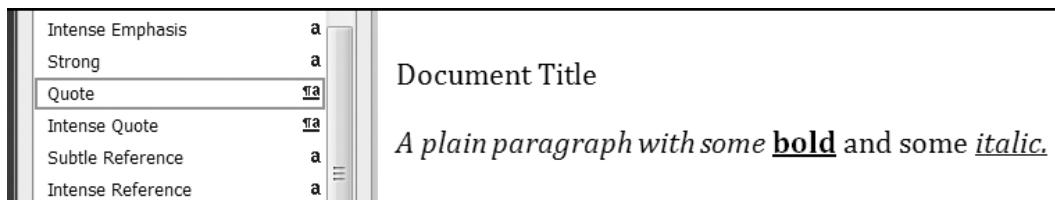


Figure 13-7: The *restyled.docx* file

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

Writing Word Documents

Enter the following into the interactive shell:

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

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

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

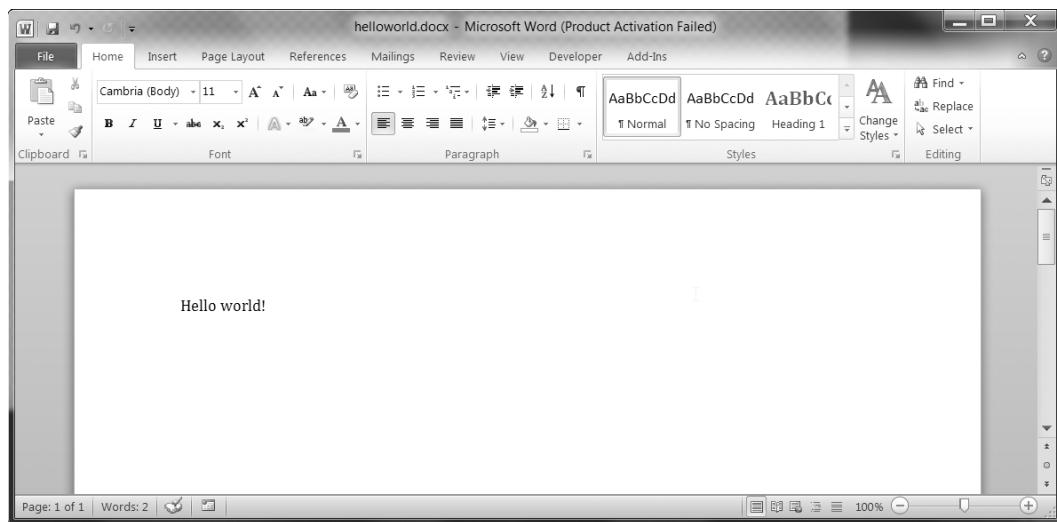


Figure 13-8: The Word document created using `add_paragraph('Hello world!')`

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

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

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

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

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

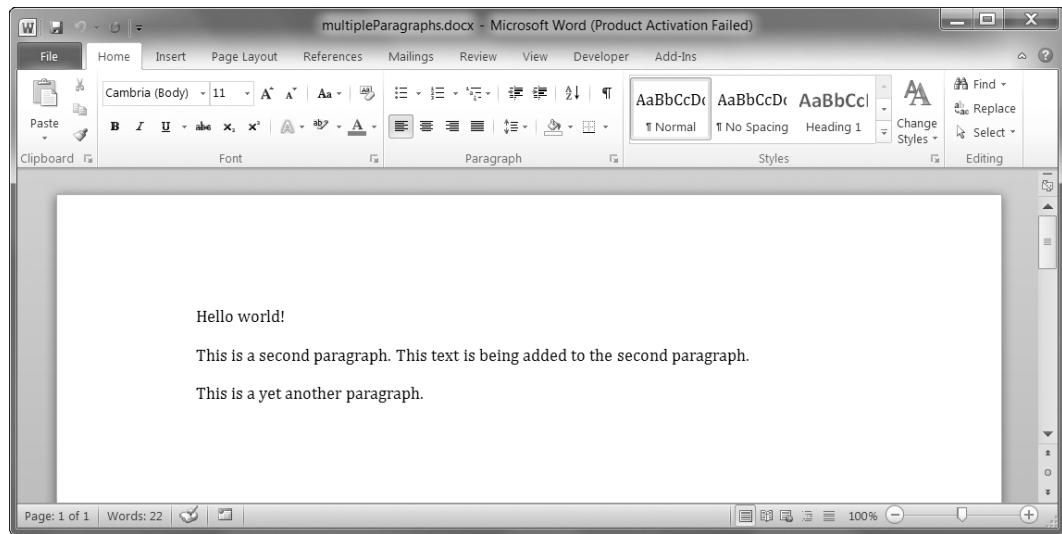


Figure 13-9: The document with multiple Paragraph and Run objects added

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

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

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

Adding Headings

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

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

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

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

Header 0

Header 1

Header 2

Header 3

Header 4

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

Adding Line and Page Breaks

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

```
>>> doc = docx.Document()
>>> doc.add_paragraph('This is on the first page!')
<docx.text.Paragraph object at 0x000000003785518>
❶ >>> doc.paragraphs[0].runs[0].add_break(docx.text.WD_BREAK.PAGE)
>>> doc.add_paragraph('This is on the second page!')
<docx.text.Paragraph object at 0x0000000037855F8>
>>> doc.save('twoPage.docx')
```

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

Adding Pictures

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

```
>>> doc.add_picture('zophie.png', width=docx.shared.Inches(1),
height=docx.shared.Cm(4))
<docx.shape.InlineShape object at 0x0000000036C7D30>
```

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

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

Summary

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

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

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

Practice Questions

1. A string value of the PDF filename is *not* passed to the `PyPDF2.PdfFileReader()` function. What do you pass to the function instead?
2. What modes do the `File` objects for `PdfFileReader()` and `PdfFileWriter()` need to be opened in?
3. How do you acquire a `Page` object for page 5 from a `PdfFileReader` object?
4. What `PdfFileReader` variable stores the number of pages in the PDF document?
5. If a `PdfFileReader` object’s PDF is encrypted with the password `swordfish`, what must you do before you can obtain `Page` objects from it?
6. What methods do you use to rotate a page?
7. What method returns a `Document` object for a file named `demo.docx`?

8. What is the difference between a Paragraph object and a Run object?
9. How do you obtain a list of Paragraph objects for a Document object that's stored in a variable named doc?
10. What type of object has bold, underline, italic, strike, and outline variables?
11. What is the difference between setting the bold variable to True, False, or None?
12. How do you create a Document object for a new Word document?
13. How do you add a paragraph with the text 'Hello there!' to a Document object stored in a variable named doc?
14. What integers represent the levels of headings available in Word documents?

Practice Projects

For practice, write programs that do the following.

PDF Paranoia

Using the `os.walk()` function from Chapter 9, write a script that will go through every PDF in a folder (and its subfolders) and encrypt the PDFs using a password provided on the command line. Save each encrypted PDF with an `_encrypted.pdf` suffix added to the original filename. Before deleting the original file, have the program attempt to read and decrypt the file to ensure that it was encrypted correctly.

Then, write a program that finds all encrypted PDFs in a folder (and its subfolders) and creates a decrypted copy of the PDF using a provided password. If the password is incorrect, the program should print a message to the user and continue to the next PDF.

Custom Invitations as Word Documents

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

Prof. Plum
Miss Scarlet
Col. Mustard
Al Sweigart
RoboCop

Write a program that would generate a Word document with custom invitations that look like Figure 13-11.

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

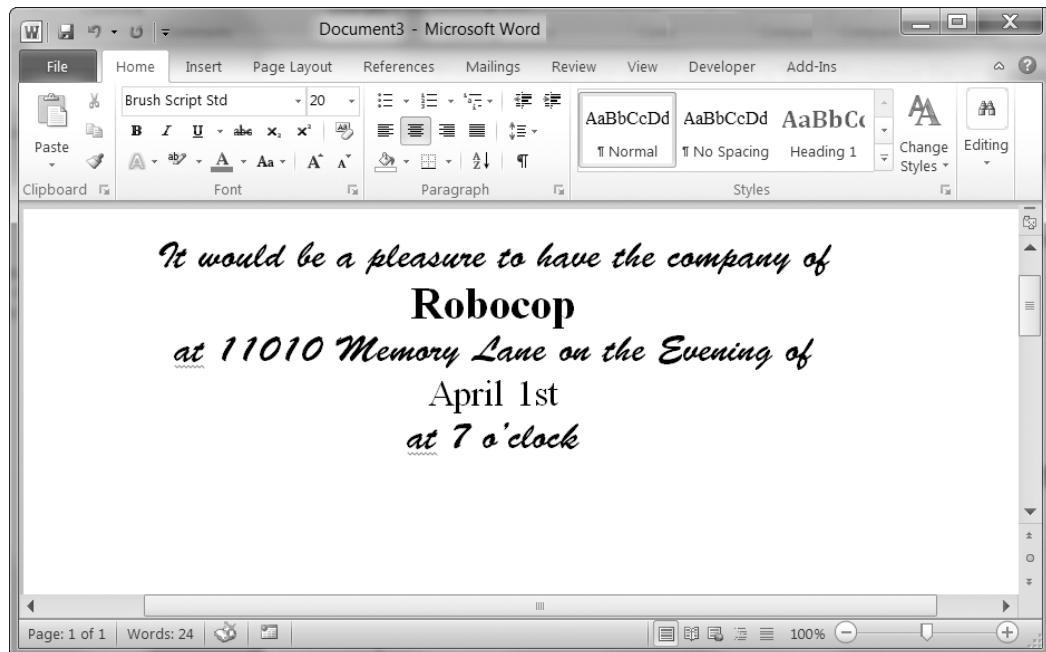


Figure 13-11: The Word document generated by your custom invite script

You can download a sample *guests.txt* file from <http://nostarch.com/automatestuff/>.

Brute-Force PDF Password Breaker

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

Using the file-reading skills you learned in Chapter 8, create a list of word strings by reading this file. Then loop over each word in this list, passing it to the `decrypt()` method. If this method returns the integer 0, the password was wrong and your program should continue to the next password. If `decrypt()` returns 1, then your program should break out of the loop and print the hacked password. You should try both the uppercase and lowercase form of each word. (On my laptop, going through all 88,000 uppercase and lowercase words from the dictionary file takes a couple of minutes. This is why you shouldn't use a simple English word for your passwords.)

14

WORKING WITH CSV FILES AND JSON DATA



In Chapter 13, you learned how to extract text from PDF and Word documents. These files were in a binary format, which required special Python modules to access their data.

CSV and JSON files, on the other hand, are just plain-text files. You can view them in a text editor, such as IDLE’s file editor. But Python also comes with the special `csv` and `json` modules, each providing functions to help you work with these file formats.

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

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

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

The `csv` Module

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

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

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

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

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

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

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

Reader Objects

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

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

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

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

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

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

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

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

Reading Data from Reader Objects in a for Loop

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

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

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

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

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

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

Writer Objects

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

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

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

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

The screenshot shows a Microsoft Excel spreadsheet with a single row of data. The data consists of the numbers 42, 2, 3, 4, 5, 6, 7, 8, 10, 12, 14, 15, 18, 21, 24, 28, 30, and 35. The cell containing '42' is selected. The formula bar at the top shows 'A1' and '42'. The cells are double-spaced vertically.

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

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

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

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

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

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

The delimiter and lineterminator Keyword Arguments

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

```
>>> import csv
>>> csvFile = open('example.tsv', 'w', newline='')
❶ >>> csvWriter = csv.writer(csvFile, delimiter='\t', lineterminator='\n\n')
>>> csvWriter.writerow(['apples', 'oranges', 'grapes'])
24
>>> csvWriter.writerow(['eggs', 'bacon', 'ham'])
17
>>> csvWriter.writerow(['spam', 'spam', 'spam', 'spam', 'spam', 'spam'])
32
>>> csvFile.close()
```

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

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

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

| | | |
|--------|---------|--------|
| apples | oranges | grapes |
| eggs | bacon | ham |
| spam | spam | spam |
| spam | spam | spam |

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

Project: Removing the Header from CSV Files

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

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

WARNING

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

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

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

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

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

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

Step 1: Loop Through Each CSV File

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

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

import csv, os

os.makedirs('headerRemoved', exist_ok=True)

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

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

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

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

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

Step 2: Read in the CSV File

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

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

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

--snip--
```

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

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

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

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

Step 3: Write Out the CSV File Without the First Row

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

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

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

    --snip--

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

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

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

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

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

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

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

Ideas for Similar Programs

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

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

JSON and APIs

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

Here's an example of data formatted as JSON:

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

JSON is useful to know, because many websites offer JSON content as a way for programs to interact with the website. This is known as providing an *application programming interface (API)*. Accessing an API is the same

as accessing any other web page via a URL. The difference is that the data returned by an API is formatted (with JSON, for example) for machines; APIs aren't easy for people to read.

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

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

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

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

The `json` Module

Python's `json` module handles all the details of translating between a string with JSON data and Python values for the `json.loads()` and `json.dumps()` functions. JSON can't store *every* kind of Python value. It can contain values of only the following data types: strings, integers, floats, Booleans, lists, dictionaries, and `NoneType`. JSON cannot represent Python-specific objects, such as `File` objects, CSV Reader or Writer objects, Regex objects, or Selenium `WebElement` objects.

Reading JSON with the `loads()` Function

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

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

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

Writing JSON with the `dumps()` Function

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

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

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

Project: Fetching Current Weather Data

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

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

Overall, the program does the following:

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

So the code will need to do the following:

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

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

Step 1: Get Location from the Command Line Argument

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

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

import json, requests, sys

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

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

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

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

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

Step 2: Download the JSON Data

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

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

--snip--

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

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

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

Step 3: Load JSON Data and Print Weather

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

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

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

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

--snip--
```

```
# Load JSON data into a Python variable.  
weatherData = json.loads(response.text)
```

```
# Print weather descriptions.  
❶ w = weatherData['list']  
print('Current weather in %s:' % (location))  
print(w[0]['weather'][0]['main'], '-', w[0]['weather'][0]['description'])  
print()  
print('Tomorrow:')  
print(w[1]['weather'][0]['main'], '-', w[1]['weather'][0]['description'])  
print()  
print('Day after tomorrow:')  
print(w[2]['weather'][0]['main'], '-', w[2]['weather'][0]['description'])
```

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

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

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

```
Tomorrow:  
Clouds - few clouds
```

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

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

Ideas for Similar Programs

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

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

Summary

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

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

In Chapter 15, you’ll break away from data formats and learn how to make your programs communicate with you by sending emails and text messages.

Practice Questions

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

Practice Project

For practice, write a program that does the following.

Excel-to-CSV Converter

Excel can save a spreadsheet to a CSV file with a few mouse clicks, but if you had to convert hundreds of Excel files to CSVs, it would take hours of clicking. Using the `openpyxl` module from Chapter 12, write a program that reads all the Excel files in the current working directory and outputs them as CSV files.

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

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

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

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

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

            # Write the rowData list to the CSV file.

csvFile.close()
```

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

15

KEEPING TIME, SCHEDULING TASKS, AND LAUNCHING PROGRAMS



Running programs while you're sitting at your computer is fine, but it's also useful to have programs run without your direct supervision. Your computer's clock can schedule programs to run code at some specified time and date or at regular intervals. For example, your program could scrape a website every hour to check for changes or do a CPU-intensive task at 4 AM while you sleep. Python's `time` and `datetime` modules provide these functions.

You can also write programs that launch other programs on a schedule by using the `subprocess` and `threading` modules. Often, the fastest way to program is to take advantage of applications that other people have already written.

The time Module

Your computer's system clock is set to a specific date, time, and time zone. The built-in `time` module allows your Python programs to read the system clock for the current time. The `time.time()` and `time.sleep()` functions are the most useful in the `time` module.

The `time.time()` Function

The *Unix epoch* is a time reference commonly used in programming: 12 AM on January 1, 1970, Coordinated Universal Time (UTC). The `time.time()` function returns the number of seconds since that moment as a float value. (Recall that a float is just a number with a decimal point.) This number is called an *epoch timestamp*. For example, enter the following into the interactive shell:

```
>>> import time  
>>> time.time()  
1425063955.068649
```

Here I'm calling `time.time()` on February 27, 2015, at 11:05 Pacific Standard Time, or 7:05 PM UTC. The return value is how many seconds have passed between the Unix epoch and the moment `time.time()` was called.

NOTE

The interactive shell examples will yield dates and times for when I wrote this chapter in February 2015. Unless you're a time traveler, your dates and times will be different.

Epoch timestamps can be used to *profile* code, that is, measure how long a piece of code takes to run. If you call `time.time()` at the beginning of the code block you want to measure and again at the end, you can subtract the first timestamp from the second to find the elapsed time between those two calls. For example, open a new file editor window and enter the following program:

```
import time  
❶ def calcProd():  
    # Calculate the product of the first 100,000 numbers.  
    product = 1  
    for i in range(1, 100000):  
        product = product * i  
    return product  
  
❷ startTime = time.time()  
prod = calcProd()  
❸ endTime = time.time()  
❹ print('The result is %s digits long.' % (len(str(prod))))  
❺ print('Took %s seconds to calculate.' % (endTime - startTime))
```

At ❶, we define a function `calcProd()` to loop through the integers from 1 to 99,999 and return their product. At ❷, we call `time.time()` and store it in `startTime`. Right after calling `calcProd()`, we call `time.time()` again and store it in `endTime` ❸. We end by printing the length of the product returned by `calcProd()` ❹ and how long it took to run `calcProd()` ❺.

Save this program as `calcProd.py` and run it. The output will look something like this:

```
The result is 456569 digits long.  
Took 2.844162940979004 seconds to calculate.
```

NOTE

Another way to profile your code is to use the `cProfile.run()` function, which provides a much more informative level of detail than the simple `time.time()` technique. The `cProfile.run()` function is explained at <https://docs.python.org/3/library/profile.html>.

The `time.sleep()` Function

If you need to pause your program for a while, call the `time.sleep()` function and pass it the number of seconds you want your program to stay paused. Enter the following into the interactive shell:

```
>>> import time  
>>> for i in range(3):  
❶    print('Tick')  
❷    time.sleep(1)  
❸    print('Tock')  
❹    time.sleep(1)
```

```
Tick  
Tock  
Tick  
Tock  
Tick  
Tock
```

```
❺ >>> time.sleep(5)
```

The for loop will print Tick ❶, pause for one second ❷, print Tock ❸, pause for one second ❹, print Tick, pause, and so on until Tick and Tock have each been printed three times.

The `time.sleep()` function will *block*—that is, it will not return and release your program to execute other code—until after the number of seconds you passed to `time.sleep()` has elapsed. For example, if you enter `time.sleep(5)` ❺, you'll see that the next prompt (`>>>`) doesn't appear until five seconds have passed.

Be aware that pressing CTRL-C will not interrupt `time.sleep()` calls in IDLE. IDLE waits until the entire pause is over before raising the `KeyboardInterrupt` exception. To work around this problem, instead of having a single `time.sleep(30)` call to pause for 30 seconds, use a for loop to make 30 calls to `time.sleep(1)`.

```
>>> for i in range(30):
    time.sleep(1)
```

If you press CTRL-C sometime during these 30 seconds, you should see the `KeyboardInterrupt` exception thrown right away.

Rounding Numbers

When working with times, you'll often encounter float values with many digits after the decimal. To make these values easier to work with, you can shorten them with Python's built-in `round()` function, which rounds a float to the precision you specify. Just pass in the number you want to round, plus an optional second argument representing how many digits after the decimal point you want to round it to. If you omit the second argument, `round()` rounds your number to the nearest whole integer. Enter the following into the interactive shell:

```
>>> import time
>>> now = time.time()
>>> now
1425064108.017826
>>> round(now, 2)
1425064108.02
>>> round(now, 4)
1425064108.0178
>>> round(now)
1425064108
```

After importing `time` and storing `time.time()` in `now`, we call `round(now, 2)` to round `now` to two digits after the decimal, `round(now, 4)` to round to four digits after the decimal, and `round(now)` to round to the nearest integer.

Project: Super Stopwatch

Say you want to track how much time you spend on boring tasks you haven't automated yet. You don't have a physical stopwatch, and it's surprisingly difficult to find a free stopwatch app for your laptop or smartphone that isn't covered in ads and doesn't send a copy of your browser history to marketers. (It says it can do this in the license agreement you agreed to. You did read the license agreement, didn't you?) You can write a simple stopwatch program yourself in Python.

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

- Track the amount of time elapsed between presses of the ENTER key, with each key press starting a new "lap" on the timer.
- Print the lap number, total time, and lap time.

This means your code will need to do the following:

- Find the current time by calling `time.time()` and store it as a timestamp at the start of the program, as well as at the start of each lap.
- Keep a lap counter and increment it every time the user presses ENTER.
- Calculate the elapsed time by subtracting timestamps.
- Handle the `KeyboardInterrupt` exception so the user can press CTRL-C to quit.

Open a new file editor window and save it as `stopwatch.py`.

Step 1: Set Up the Program to Track Times

The stopwatch program will need to use the current time, so you'll want to import the `time` module. Your program should also print some brief instructions to the user before calling `input()`, so the timer can begin after the user presses ENTER. Then the code will start tracking lap times.

Enter the following code into the file editor, writing a `TODO` comment as a placeholder for the rest of the code:

```
#! python3
# stopwatch.py - A simple stopwatch program.

import time

# Display the program's instructions.
print('Press ENTER to begin. Afterwards, press ENTER to "click" the stopwatch.
Press Ctrl-C to quit.')
input()                      # press Enter to begin
print('Started.')
startTime = time.time()      # get the first lap's start time
lastTime = startTime
lapNum = 1

# TODO: Start tracking the lap times.
```

Now that we've written the code to display the instructions, start the first lap, note the time, and set our lap count to 1.

Step 2: Track and Print Lap Times

Now let's write the code to start each new lap, calculate how long the previous lap took, and calculate the total time elapsed since starting the stopwatch. We'll display the lap time and total time and increase the lap count for each new lap. Add the following code to your program:

```
#! python3
# stopwatch.py - A simple stopwatch program.

import time

--snip--
```

```

# Start tracking the lap times.
❶ try:
❷     while True:
❸         input()
❹         lapTime = round(time.time() - lastTime, 2)
❺         totalTime = round(time.time() - startTime, 2)
❻         print('Lap %s: %s (%s)' % (lapNum, totalTime, lapTime), end='')
❽         lapNum += 1
❾         lastTime = time.time() # reset the last lap time
❿ except KeyboardInterrupt:
    # Handle the Ctrl-C exception to keep its error message from displaying.
    print('\nDone.')

```

If the user presses CTRL-C to stop the stopwatch, the `KeyboardInterrupt` exception will be raised, and the program will crash if its execution is not a `try` statement. To prevent crashing, we wrap this part of the program in a `try` statement ❶. We'll handle the exception in the `except` clause ❿, so when CTRL-C is pressed and the exception is raised, the program execution moves to the `except` clause to print `Done`, instead of the `KeyboardInterrupt` error message. Until this happens, the execution is inside an infinite loop ❷ that calls `input()` and waits until the user presses ENTER to end a lap. When a lap ends, we calculate how long the lap took by subtracting the start time of the lap, `lastTime`, from the current time, `time.time()` ❸. We calculate the total time elapsed by subtracting the overall start time of the stopwatch, `startTime`, from the current time ❹.

Since the results of these time calculations will have many digits after the decimal point (such as `4.766272783279419`), we use the `round()` function to round the float value to two digits at ❽ and ❾.

At ❻, we print the lap number, total time elapsed, and the lap time. Since the user pressing ENTER for the `input()` call will print a newline to the screen, pass `end=''` to the `print()` function to avoid double-spacing the output. After printing the lap information, we get ready for the next lap by adding 1 to the count `lapNum` and setting `lastTime` to the current time, which is the start time of the next lap.

Ideas for Similar Programs

Time tracking opens up several possibilities for your programs. Although you can download apps to do some of these things, the benefit of writing programs yourself is that they will be free and not bloated with ads and useless features. You could write similar programs to do the following:

- Create a simple timesheet app that records when you type a person's name and uses the current time to clock them in or out.
- Add a feature to your program to display the elapsed time since a process started, such as a download that uses the `requests` module. (See Chapter 11.)
- Intermittently check how long a program has been running and offer the user a chance to cancel tasks that are taking too long.

The `datetime` Module

The `time` module is useful for getting a Unix epoch timestamp to work with. But if you want to display a date in a more convenient format, or do arithmetic with dates (for example, figuring out what date was 205 days ago or what date is 123 days from now), you should use the `datetime` module.

The `datetime` module has its own `datetime` data type. `datetime` values represent a specific moment in time. Enter the following into the interactive shell:

```
>>> import datetime
❶ >>> datetime.datetime.now()
❷ datetime.datetime(2015, 2, 27, 11, 10, 49, 55, 53)
❸ >>> dt = datetime.datetime(2015, 10, 21, 16, 29, 0)
❹ >>> dt.year, dt.month, dt.day
(2015, 10, 21)
❺ >>> dt.hour, dt.minute, dt.second
(16, 29, 0)
```

Calling `datetime.datetime.now()` ❶ returns a `datetime` object ❷ for the current date and time, according to your computer’s clock. This object includes the year, month, day, hour, minute, second, and microsecond of the current moment. You can also retrieve a `datetime` object for a specific moment by using the `datetime.datetime()` function ❸, passing it integers representing the year, month, day, hour, and second of the moment you want. These integers will be stored in the `datetime` object’s `year`, `month`, `day` ❹, `hour`, `minute`, and `second` ❺ attributes.

A Unix epoch timestamp can be converted to a `datetime` object with the `datetime.datetime.fromtimestamp()` function. The date and time of the `datetime` object will be converted for the local time zone. Enter the following into the interactive shell:

```
>>> datetime.datetime.fromtimestamp(1000000)
datetime.datetime(1970, 1, 12, 5, 46, 40)
>>> datetime.datetime.fromtimestamp(time.time())
datetime.datetime(2015, 2, 27, 11, 13, 0, 604980)
```

Calling `datetime.datetime.fromtimestamp()` and passing it 1000000 returns a `datetime` object for the moment 1,000,000 seconds after the Unix epoch. Passing `time.time()`, the Unix epoch timestamp for the current moment, returns a `datetime` object for the current moment. So the expressions `datetime.datetime.now()` and `datetime.datetime.fromtimestamp(time.time())` do the same thing; they both give you a `datetime` object for the present moment.

NOTE

These examples were entered on a computer set to Pacific Standard Time. If you’re in another time zone, your results will look different.

`datetime` objects can be compared with each other using comparison operators to find out which one precedes the other. The later `datetime` object is the “greater” value. Enter the following into the interactive shell:

```
❶ >>> halloween2015 = datetime.datetime(2015, 10, 31, 0, 0, 0)
❷ >>> newyears2016 = datetime.datetime(2016, 1, 1, 0, 0, 0)
    >>> oct31_2015 = datetime.datetime(2015, 10, 31, 0, 0, 0)
❸ >>> halloween2015 == oct31_2015
True
❹ >>> halloween2015 > newyears2016
False
❺ >>> newyears2016 > halloween2015
True
    >>> newyears2016 != oct31_2015
True
```

Make a `datetime` object for the first moment (midnight) of October 31, 2015 and store it in `halloween2015` ❶. Make a `datetime` object for the first moment of January 1, 2016 and store it in `newyears2016` ❷. Then make another object for midnight on October 31, 2015 and store it in `oct31_2015`. Comparing `halloween2015` and `oct31_2015` shows that they’re equal ❸. Comparing `newyears2016` and `halloween2015` shows that `newyears2016` is greater (later) than `halloween2015` ❹ ❺.

The `timedelta` Data Type

The `datetime` module also provides a `timedelta` data type, which represents a *duration* of time rather than a *moment* in time. Enter the following into the interactive shell:

```
❶ >>> delta = datetime.timedelta(days=11, hours=10, minutes=9, seconds=8)
❷ >>> delta.days, delta.seconds, delta.microseconds
(11, 36548, 0)
    >>> delta.total_seconds()
986948.0
    >>> str(delta)
'11 days, 10:09:08'
```

To create a `timedelta` object, use the `datetime.timedelta()` function. The `datetime.timedelta()` function takes keyword arguments `weeks`, `days`, `hours`, `minutes`, `seconds`, `milliseconds`, and `microseconds`. There is no `month` or `year` keyword argument because “a month” or “a year” is a variable amount of time depending on the particular month or year. A `timedelta` object has the total duration represented in days, seconds, and microseconds. These numbers are stored in the `days`, `seconds`, and `microseconds` attributes, respectively. The `total_seconds()` method will return the duration in number of seconds alone. Passing a `timedelta` object to `str()` will return a nicely formatted, human-readable string representation of the object.

In this example, we pass keyword arguments to `datetime.delta()` to specify a duration of 11 days, 10 hours, 9 minutes, and 8 seconds, and store the returned `timedelta` object in `delta` ❶. This `timedelta` object's `days` attribute stores 11, and its `seconds` attribute stores 36548 (10 hours, 9 minutes, and 8 seconds, expressed in seconds) ❷. Calling `total_seconds()` tells us that 11 days, 10 hours, 9 minutes, and 8 seconds is 986,948 seconds. Finally, passing the `timedelta` object to `str()` returns a string clearly explaining the duration.

The arithmetic operators can be used to perform *date arithmetic* on `datetime` values. For example, to calculate the date 1,000 days from now, enter the following into the interactive shell:

```
>>> dt = datetime.datetime.now()
>>> dt
datetime.datetime(2015, 2, 27, 18, 38, 50, 636181)
>>> thousandDays = datetime.timedelta(days=1000)
>>> dt + thousandDays
datetime.datetime(2017, 11, 23, 18, 38, 50, 636181)
```

First, make a `datetime` object for the current moment and store it in `dt`. Then make a `timedelta` object for a duration of 1,000 days and store it in `thousandDays`. Add `dt` and `thousandDays` together to get a `datetime` object for the date 1,000 days from now. Python will do the date arithmetic to figure out that 1,000 days after February 27, 2015, will be November 23, 2017. This is useful because when you calculate 1,000 days from a given date, you have to remember how many days are in each month and factor in leap years and other tricky details. The `datetime` module handles all of this for you.

`timedelta` objects can be added or subtracted with `datetime` objects or other `timedelta` objects using the `+` and `-` operators. A `timedelta` object can be multiplied or divided by integer or float values with the `*` and `/` operators. Enter the following into the interactive shell:

```
❶ >>> oct21st = datetime.datetime(2015, 10, 21, 16, 29, 0)
❷ >>> aboutThirtyYears = datetime.timedelta(days=365 * 30)
>>> oct21st
datetime.datetime(2015, 10, 21, 16, 29)
>>> oct21st - aboutThirtyYears
datetime.datetime(1985, 10, 28, 16, 29)
>>> oct21st - (2 * aboutThirtyYears)
datetime.datetime(1955, 11, 5, 16, 29)
```

Here we make a `datetime` object for October 21, 2015 ❶ and a `timedelta` object for a duration of about 30 years (we're assuming 365 days for each of those years) ❷. Subtracting `aboutThirtyYears` from `oct21st` gives us a `datetime` object for the date 30 years before October 21, 2015. Subtracting `2 * aboutThirtyYears` from `oct21st` returns a `datetime` object for the date 60 years before October 21, 2015.

Pausing Until a Specific Date

The `time.sleep()` method lets you pause a program for a certain number of seconds. By using a `while` loop, you can pause your programs until a specific date. For example, the following code will continue to loop until Halloween 2016:

```
import datetime
import time
halloween2016 = datetime.datetime(2016, 10, 31, 0, 0, 0)
while datetime.datetime.now() < halloween2016:
    time.sleep(1)
```

The `time.sleep(1)` call will pause your Python program so that the computer doesn't waste CPU processing cycles simply checking the time over and over. Rather, the `while` loop will just check the condition once per second and continue with the rest of the program after Halloween 2016 (or whenever you program it to stop).

Converting `datetime` Objects into Strings

Epoch timestamps and `datetime` objects aren't very friendly to the human eye. Use the `strftime()` method to display a `datetime` object as a string. (The *f* in the name of the `strftime()` function stands for *format*.)

The `strftime()` method uses directives similar to Python's string formatting. Table 15-1 has a full list of `strftime()` directives.

Table 15-1: `strftime()` Directives

| <code>strftime</code> directive | Meaning |
|--|---|
| <code>%Y</code> | Year with century, as in '2014' |
| <code>%y</code> | Year without century, '00' to '99' (1970 to 2069) |
| <code>%m</code> | Month as a decimal number, '01' to '12' |
| <code>%B</code> | Full month name, as in 'November' |
| <code>%b</code> | Abbreviated month name, as in 'Nov' |
| <code>%d</code> | Day of the month, '01' to '31' |
| <code>%j</code> | Day of the year, '001' to '366' |
| <code>%w</code> | Day of the week, '0' (Sunday) to '6' (Saturday) |
| <code>%A</code> | Full weekday name, as in 'Monday' |
| <code>%a</code> | Abbreviated weekday name, as in 'Mon' |
| <code>%H</code> | Hour (24-hour clock), '00' to '23' |
| <code>%I</code> | Hour (12-hour clock), '01' to '12' |
| <code>%M</code> | Minute, '00' to '59' |
| <code>%S</code> | Second, '00' to '59' |
| <code>%p</code> | 'AM' or 'PM' |
| <code>%%</code> | Literal '%' character |

Pass `strptime()` a custom format string containing formatting directives (along with any desired slashes, colons, and so on), and `strptime()` will return the `datetime` object's information as a formatted string. Enter the following into the interactive shell:

```
>>> oct21st = datetime.datetime(2015, 10, 21, 16, 29, 0)
>>> oct21st.strftime('%Y/%m/%d %H:%M:%S')
'2015/10/21 16:29:00'
>>> oct21st.strftime('%I:%M %p')
'04:29 PM'
>>> oct21st.strftime("%B of '%y")
"October of '15"
```

Here we have a `datetime` object for October 21, 2015 at 4:29 PM, stored in `oct21st`. Passing `strptime()` the custom format string '`%Y/%m/%d %H:%M:%S`' returns a string containing 2015, 10, and 21 separated by slashes and 16, 29, and 00 separated by colons. Passing '`%I:%M %p`' returns '`04:29 PM`', and passing "`%B of '%y`" returns "`October of '15`". Note that `strptime()` doesn't begin with `datetime.datetime`.

Converting Strings into `datetime` Objects

If you have a string of date information, such as '`2015/10/21 16:29:00`' or '`October 21, 2015`', and need to convert it to a `datetime` object, use the `datetime.datetime.strptime()` function. The `strptime()` function is the inverse of the `strftime()` method. A custom format string using the same directives as `strftime()` must be passed so that `strptime()` knows how to parse and understand the string. (The `p` in the name of the `strptime()` function stands for *parse*.)

Enter the following into the interactive shell:

```
❶ >>> datetime.datetime.strptime('October 21, 2015', '%B %d, %Y')
datetime.datetime(2015, 10, 21, 0, 0)
>>> datetime.datetime.strptime('2015/10/21 16:29:00', '%Y/%m/%d %H:%M:%S')
datetime.datetime(2015, 10, 21, 16, 29)
>>> datetime.datetime.strptime("October of '15", "%B of '%y")
datetime.datetime(2015, 10, 1, 0, 0)
>>> datetime.datetime.strptime("November of '63", "%B of '%y")
datetime.datetime(2063, 11, 1, 0, 0)
```

To get a `datetime` object from the string '`October 21, 2015`', pass '`October 21, 2015`' as the first argument to `strptime()` and the custom format string that corresponds to '`October 21, 2015`' as the second argument ❶. The string with the date information must match the custom format string exactly, or Python will raise a `ValueError` exception.

Review of Python’s Time Functions

Dates and times in Python can involve quite a few different data types and functions. Here’s a review of the three different types of values used to represent time:

- A Unix epoch timestamp (used by the `time` module) is a float or integer value of the number of seconds since 12 AM on January 1, 1970, UTC.
- A `datetime` object (of the `datetime` module) has integers stored in the attributes `year`, `month`, `day`, `hour`, `minute`, and `second`.
- A `timedelta` object (of the `datetime` module) represents a time duration, rather than a specific moment.

Here’s a review of time functions and their parameters and return values:

- The `time.time()` function returns an epoch timestamp float value of the current moment.
- The `time.sleep(seconds)` function stops the program for the amount of seconds specified by the `seconds` argument.
- The `datetime.datetime(year, month, day, hour, minute, second)` function returns a `datetime` object of the moment specified by the arguments. If `hour`, `minute`, or `second` arguments are not provided, they default to 0.
- The `datetime.datetime.now()` function returns a `datetime` object of the current moment.
- The `datetime.datetime.fromtimestamp(epoch)` function returns a `datetime` object of the moment represented by the `epoch` timestamp argument.
- The `datetime.timedelta(weeks, days, hours, minutes, seconds, milliseconds, microseconds)` function returns a `timedelta` object representing a duration of time. The function’s keyword arguments are all optional and do not include `month` or `year`.
- The `total_seconds()` method for `timedelta` objects returns the number of seconds the `timedelta` object represents.
- The `strftime(format)` method returns a string of the time represented by the `datetime` object in a custom format that’s based on the `format` string. See Table 15-1 for the format details.
- The `datetime.datetime.strptime(time_string, format)` function returns a `datetime` object of the moment specified by `time_string`, parsed using the `format` string argument. See Table 15-1 for the format details.

Multithreading

To introduce the concept of multithreading, let's look at an example situation. Say you want to schedule some code to run after a delay or at a specific time. You could add code like the following at the start of your program:

```
import time, datetime

startTime = datetime.datetime(2029, 10, 31, 0, 0, 0)
while datetime.datetime.now() < startTime:
    time.sleep(1)

print('Program now starting on Halloween 2029')
--snip--
```

This code designates a start time of October 31, 2029, and keeps calling `time.sleep(1)` until the start time arrives. Your program cannot do anything while waiting for the loop of `time.sleep()` calls to finish; it just sits around until Halloween 2029. This is because Python programs by default have a single *thread* of execution.

To understand what a thread of execution is, remember the Chapter 2 discussion of flow control, when you imagined the execution of a program as placing your finger on a line of code in your program and moving to the next line or wherever it was sent by a flow control statement. A *single-threaded* program has only one finger. But a *multithreaded* program has multiple fingers. Each finger still moves to the next line of code as defined by the flow control statements, but the fingers can be at different places in the program, executing different lines of code at the same time. (All of the programs in this book so far have been single threaded.)

Rather than having all of your code wait until the `time.sleep()` function finishes, you can execute the delayed or scheduled code in a separate thread using Python's `threading` module. The separate thread will pause for the `time.sleep` calls. Meanwhile, your program can do other work in the original thread.

To make a separate thread, you first need to make a `Thread` object by calling the `threading.Thread()` function. Enter the following code in a new file and save it as `threadDemo.py`:

```
import threading, time
print('Start of program.')

❶ def takeANap():
    time.sleep(5)
    print('Wake up!')

❷ threadObj = threading.Thread(target=takeANap)
❸ threadObj.start()

print('End of program.')
```

At ❶, we define a function that we want to use in a new thread. To create a Thread object, we call `threading.Thread()` and pass it the keyword argument `target=takeANap` ❷. This means the function we want to call in the new thread is `takeANap()`. Notice that the keyword argument is `target=takeANap`, not `target=takeANap()`. This is because you want to pass the `takeANap()` function itself as the argument, not call `takeANap()` and pass its return value.

After we store the Thread object created by `threading.Thread()` in `threadObj`, we call `threadObj.start()` ❸ to create the new thread and start executing the target function in the new thread. When this program is run, the output will look like this:

```
Start of program.  
End of program.  
Wake up!
```

This can be a bit confusing. If `print('End of program.')` is the last line of the program, you might think that it should be the last thing printed. The reason `Wake up!` comes after it is that when `threadObj.start()` is called, the target function for `threadObj` is run in a new thread of execution. Think of it as a second finger appearing at the start of the `takeANap()` function. The main thread continues to `print('End of program.')`. Meanwhile, the new thread that has been executing the `time.sleep(5)` call, pauses for 5 seconds. After it wakes from its 5-second nap, it prints '`Wake up!`' and then returns from the `takeANap()` function. Chronologically, '`Wake up!`' is the last thing printed by the program.

Normally a program terminates when the last line of code in the file has run (or the `sys.exit()` function is called). But `threadDemo.py` has two threads. The first is the original thread that began at the start of the program and ends after `print('End of program.')`. The second thread is created when `threadObj.start()` is called, begins at the start of the `takeANap()` function, and ends after `takeANap()` returns.

A Python program will not terminate until all its threads have terminated. When you ran `threadDemo.py`, even though the original thread had terminated, the second thread was still executing the `time.sleep(5)` call.

Passing Arguments to the Thread's Target Function

If the target function you want to run in the new thread takes arguments, you can pass the target function's arguments to `threading.Thread()`. For example, say you wanted to run this `print()` call in its own thread:

```
>>> print('Cats', 'Dogs', 'Frogs', sep=' & ')  
Cats & Dogs & Frogs
```

This `print()` call has three regular arguments, '`Cats`', '`Dogs`', and '`Frogs`', and one keyword argument, `sep=' & '`. The regular arguments can be passed

as a list to the `args` keyword argument in `threading.Thread()`. The keyword argument can be specified as a dictionary to the `kwargs` keyword argument in `threading.Thread()`.

Enter the following into the interactive shell:

```
>>> import threading
>>> threadObj = threading.Thread(target=print, args=['Cats', 'Dogs', 'Frogs'],
    kwargs={'sep': ' & '})
>>> threadObj.start()
Cats & Dogs & Frogs
```

To make sure the arguments 'Cats', 'Dogs', and 'Frogs' get passed to `print()` in the new thread, we pass `args=['Cats', 'Dogs', 'Frogs']` to `threading.Thread()`. To make sure the keyword argument `sep=' & '` gets passed to `print()` in the new thread, we pass `kwargs={'sep': ' & '}` to `threading.Thread()`.

The `threadObj.start()` call will create a new thread to call the `print()` function, and it will pass 'Cats', 'Dogs', and 'Frogs' as arguments and ' & ' for the `sep` keyword argument.

This is an incorrect way to create the new thread that calls `print()`:

```
threadObj = threading.Thread(target=print('Cats', 'Dogs', 'Frogs', sep=' & '))
```

What this ends up doing is calling the `print()` function and passing its return value (`print()`'s return value is always `None`) as the `target` keyword argument. It *doesn't* pass the `print()` function itself. When passing arguments to a function in a new thread, use the `threading.Thread()` function's `args` and `kwargs` keyword arguments.

Concurrency Issues

You can easily create several new threads and have them all running at the same time. But multiple threads can also cause problems called *concurrency issues*. These issues happen when threads read and write variables at the same time, causing the threads to trip over each other. Concurrency issues can be hard to reproduce consistently, making them hard to debug.

Multithreaded programming is its own wide subject and beyond the scope of this book. What you have to keep in mind is this: To avoid concurrency issues, never let multiple threads read or write the same variables. When you create a new `Thread` object, make sure its target function uses only local variables in that function. This will avoid hard-to-debug concurrency issues in your programs.

NOTE

A beginner's tutorial on multithreaded programming is available at <http://nostarch.com/automatestuff/>.

Project: Multithreaded XKCD Downloader

In Chapter 11, you wrote a program that downloaded all of the XKCD comic strips from the XKCD website. This was a single-threaded program: It downloaded one comic at a time. Much of the program’s running time was spent establishing the network connection to begin the download and writing the downloaded images to the hard drive. If you have a broadband Internet connection, your single-threaded program wasn’t fully utilizing the available bandwidth.

A multithreaded program that has some threads downloading comics while others are establishing connections and writing the comic image files to disk uses your Internet connection more efficiently and downloads the collection of comics more quickly. Open a new file editor window and save it as *multidownloadXkcd.py*. You will modify this program to add multithreading. The completely modified source code is available to download from <http://nostarch.com/automatestuff/>.

Step 1: Modify the Program to Use a Function

This program will mostly be the same downloading code from Chapter 11, so I’ll skip the explanation for the Requests and BeautifulSoup code. The main changes you need to make are importing the threading module and making a `downloadXkcd()` function, which takes starting and ending comic numbers as parameters.

For example, calling `downloadXkcd(140, 280)` would loop over the downloading code to download the comics at <http://xkcd.com/140>, <http://xkcd.com/141>, <http://xkcd.com/142>, and so on, up to <http://xkcd.com/279>. Each thread that you create will call `downloadXkcd()` and pass a different range of comics to download.

Add the following code to your *multidownloadXkcd.py* program:

```
#! python3
# multidownloadXkcd.py - Downloads XKCD comics using multiple threads.

import requests, os, bs4, threading
❶ os.makedirs('xkcd', exist_ok=True)      # store comics in ./xkcd

❷ def downloadXkcd(startComic, endComic):
❸     for urlNumber in range(startComic, endComic):
❹         # Download the page.
❺         print('Downloading page http://xkcd.com/%s...' % (urlNumber))
❻         res = requests.get('http://xkcd.com/%s' % (urlNumber))
❾         res.raise_for_status()

❿         soup = bs4.BeautifulSoup(res.text)

❬         # Find the URL of the comic image.
❭         comicElem = soup.select('#comic img')
❮         if comicElem == []:
❯             print('Could not find comic image.')

❰         else:
❱             comicUrl = comicElem[0].get('src')
❲             # Download image.
❳             imgRes = requests.get(comicUrl)
❴             imgRes.raise_for_status()
❵             imgFile = open(os.path.join('xkcd', str(urlNumber) + '.png'), 'wb')
❶             for chunk in imgRes.iter_content(100000):
❷                 imgFile.write(chunk)
❸             imgFile.close()
```

```

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

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

# TODO: Create and start the Thread objects.
# TODO: Wait for all threads to end.

```

After importing the modules we need, we make a directory to store comics in ❶ and start defining `downloadXkcd()` ❷. We loop through all the numbers in the specified range ❸ and download each page ❹. We use Beautiful Soup to look through the HTML of each page ❺ and find the comic image ❻. If no comic image is found on a page, we print a message. Otherwise, we get the URL of the image ❾ and download the image ❿. Finally, we save the image to the directory we created.

Step 2: Create and Start Threads

Now that we've defined `downloadXkcd()`, we'll create the multiple threads that each call `downloadXkcd()` to download different ranges of comics from the XKCD website. Add the following code to `multidownloadXkcd.py` after the `downloadXkcd()` function definition:

```

#! python3
# multidownloadXkcd.py - Downloads XKCD comics using multiple threads.

--snip--

# Create and start the Thread objects.
downloadThreads = []                  # a list of all the Thread objects
for i in range(0, 1400, 100):          # loops 14 times, creates 14 threads
    downloadThread = threading.Thread(target=downloadXkcd, args=(i, i + 99))
    downloadThreads.append(downloadThread)
    downloadThread.start()

```

First we make an empty list `downloadThreads`; the list will help us keep track of the many `Thread` objects we'll create. Then we start our `for` loop. Each time through the loop, we create a `Thread` object with `threading.Thread()`, append the `Thread` object to the list, and call `start()` to start running `downloadXkcd()` in the new thread. Since the `for` loop sets the `i` variable from 0 to 1400 at steps of 100, `i` will be set to 0 on the first iteration, 100 on the second iteration, 200 on the third, and so on. Since we pass `args=(i, i + 99)` to `threading.Thread()`, the two arguments passed to `downloadXkcd()` will be 0 and 99 on the first iteration, 100 and 199 on the second iteration, 200 and 299 on the third, and so on.

As the `Thread` object's `start()` method is called and the new thread begins to run the code inside `downloadXkcd()`, the main thread will continue to the next iteration of the `for` loop and create the next thread.

Step 3: Wait for All Threads to End

The main thread moves on as normal while the other threads we create download comics. But say there's some code you don't want to run in the main thread until all the threads have completed. Calling a `Thread` object's `join()` method will block until that thread has finished. By using a `for` loop to iterate over all the `Thread` objects in the `downloadThreads` list, the main thread can call the `join()` method on each of the other threads. Add the following to the bottom of your program:

```
#! python3
# multidownloadXkcd.py - Downloads XKCD comics using multiple threads.

--snip--

# Wait for all threads to end.
for downloadThread in downloadThreads:
    downloadThread.join()
print('Done.')
```

The '`Done.`' string will not be printed until all of the `join()` calls have returned. If a `Thread` object has already completed when its `join()` method is called, then the method will simply return immediately. If you wanted to extend this program with code that runs only after all of the comics downloaded, you could replace the `print('Done.')` line with your new code.

Launching Other Programs from Python

Your Python program can start other programs on your computer with the `Popen()` function in the built-in `subprocess` module. (The *P* in the name of the `Popen()` function stands for *process*.) If you have multiple instances of an application open, each of those instances is a separate process of the same program. For example, if you open multiple windows of your web browser at the same time, each of those windows is a different process of the web browser program. See Figure 15-1 for an example of multiple calculator processes open at once.

Every process can have multiple threads. Unlike threads, a process cannot directly read and write another process's variables. If you think of a multithreaded program as having multiple fingers following source code, then having multiple processes of the same program open is like having a friend with a separate copy of the program's source code. You are both independently executing the same program.

If you want to start an external program from your Python script, pass the program's filename to `subprocess.Popen()`. (On Windows, right-click the application's **Start** menu item and select **Properties** to view the application's

filename. On OS X, CTRL-click the application and select **Show Package Contents** to find the path to the executable file.) The `Popen()` function will then immediately return. Keep in mind that the launched program is not run in the same thread as your Python program.

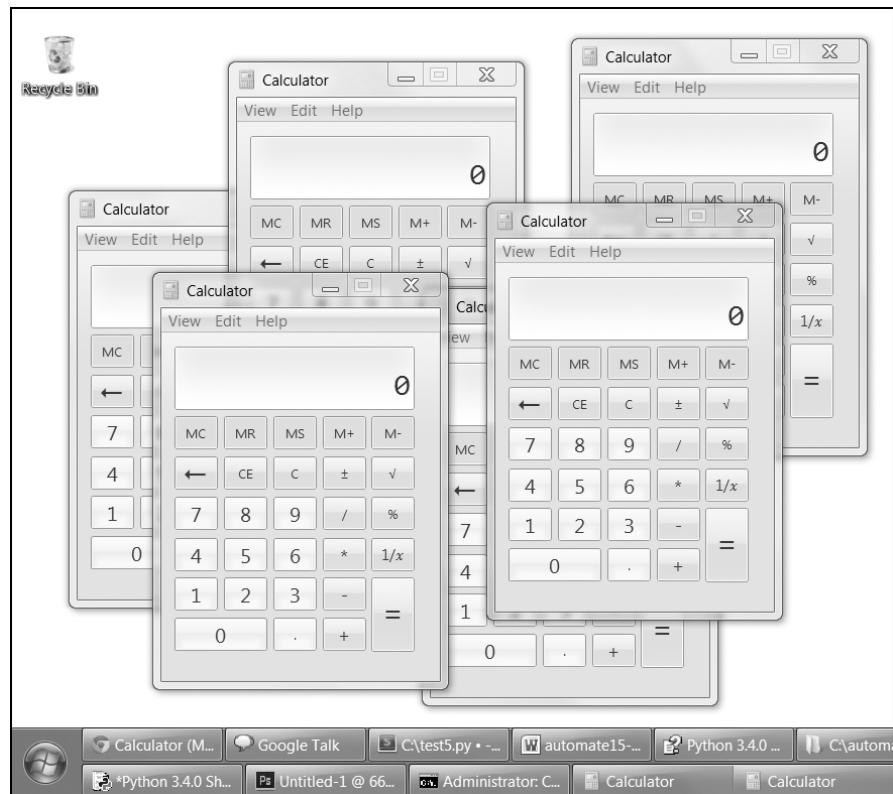


Figure 15-1: Six running processes of the same calculator program

On a Windows computer, enter the following into the interactive shell:

```
>>> import subprocess  
>>> subprocess.Popen('C:\\\\Windows\\\\System32\\\\calc.exe')  
<subprocess.Popen object at 0x000000003055A58>
```

On Ubuntu Linux, you would enter the following:

```
>>> import subprocess  
>>> subprocess.Popen('/usr/bin/gnome-calculator')  
<subprocess.Popen object at 0x7f2bcf93b20>
```

On OS X, the process is slightly different. See “Opening Files with Default Applications” on page 355.

The return value is a `Popen` object, which has two useful methods: `poll()` and `wait()`.

You can think of the `poll()` method as asking your friend if she’s finished running the code you gave her. The `poll()` method will return `None` if the process is still running at the time `poll()` is called. If the program has terminated, it will return the process’s integer *exit code*. An exit code is used

to indicate whether the process terminated without errors (an exit code of 0) or whether an error caused the process to terminate (a nonzero exit code—generally 1, but it may vary depending on the program).

The `wait()` method is like waiting for your friend to finish working on her code before you keep working on yours. The `wait()` method will block until the launched process has terminated. This is helpful if you want your program to pause until the user finishes with the other program. The return value of `wait()` is the process's integer exit code.

On Windows, enter the following into the interactive shell. Note that the `wait()` call will block until you quit the launched calculator program.

```
❶ >>> calcProc = subprocess.Popen('c:\\Windows\\System32\\calc.exe')
❷ >>> calcProc.poll() == None
True
❸ >>> calcProc.wait()
0
>>> calcProc.poll()
0
```

Here we open a calculator process ❶. While it's still running, we check if `poll()` returns `None` ❷. It should, as the process is still running. Then we close the calculator program and call `wait()` on the terminated process ❸. `wait()` and `poll()` now return 0, indicating that the process terminated without errors.

Passing Command Line Arguments to Popen()

You can pass command line arguments to processes you create with `Popen()`. To do so, you pass a list as the sole argument to `Popen()`. The first string in this list will be the executable filename of the program you want to launch; all the subsequent strings will be the command line arguments to pass to the program when it starts. In effect, this list will be the value of `sys.argv` for the launched program.

Most applications with a graphical user interface (GUI) don't use command line arguments as extensively as command line-based or terminal-based programs do. But most GUI applications will accept a single argument for a file that the applications will immediately open when they start. For example, if you're using Windows, create a simple text file called *C:\hello.txt* and then enter the following into the interactive shell:

```
>>> subprocess.Popen(['C:\\Windows\\notepad.exe', 'C:\\hello.txt'])
<subprocess.Popen object at 0x00000000032DCEB8>
```

This will not only launch the Notepad application but also have it immediately open the *C:\hello.txt* file.

Task Scheduler, launchd, and cron

If you are computer savvy, you may know about Task Scheduler on Windows, launchd on OS X, or the cron scheduler on Linux. These well-documented

and reliable tools all allow you to schedule applications to launch at specific times. If you'd like to learn more about them, you can find links to tutorials at <http://nostarch.com/automatestuff/>.

Using your operating system's built-in scheduler saves you from writing your own clock-checking code to schedule your programs. However, use the `time.sleep()` function if you just need your program to pause briefly. Or instead of using the operating system's scheduler, your code can loop until a certain date and time, calling `time.sleep(1)` each time through the loop.

Opening Websites with Python

The `webbrowser.open()` function can launch a web browser from your program to a specific website, rather than opening the browser application with `subprocess.Popen()`. See “Project: *mapIt.py* with the `webbrowser` Module” on page 234 for more details.

Running Other Python Scripts

You can launch a Python script from Python just like any other application. You just have to pass the `python.exe` executable to `Popen()` and the filename of the `.py` script you want to run as its argument. For example, the following would run the `hello.py` script from Chapter 1:

```
>>> subprocess.Popen(['C:\\\\python34\\\\python.exe', 'hello.py'])
<subprocess.Popen object at 0x000000000331CF28>
```

Pass `Popen()` a list containing a string of the Python executable's path and a string of the script's filename. If the script you're launching needs command line arguments, add them to the list after the script's filename. The location of the Python executable on Windows is `C:\\python34\\python.exe`. On OS X, it is `/Library/Frameworks/Python.framework/Versions/3.3/bin/python3`. On Linux, it is `/usr/bin/python3`.

Unlike importing the Python program as a module, when your Python program launches another Python program, the two are run in separate processes and will not be able to share each other's variables.

Opening Files with Default Applications

Double-clicking a `.txt` file on your computer will automatically launch the application associated with the `.txt` file extension. Your computer will have several of these file extension associations set up already. Python can also open files this way with `Popen()`.

Each operating system has a program that performs the equivalent of double-clicking a document file to open it. On Windows, this is the `start` program. On OS X, this is the `open` program. On Ubuntu Linux, this is the `see` program. Enter the following into the interactive shell, passing '`start`', '`open`', or '`see`' to `Popen()` depending on your system:

```
>>> fileObj = open('hello.txt', 'w')
>>> fileObj.write('Hello world!')
```

```
>>> fileObj.close()
>>> import subprocess
>>> subprocess.Popen(['start', 'hello.txt'], shell=True)
```

Here we write Hello world! to a new *hello.txt* file. Then we call `Popen()`, passing it a list containing the program name (in this example, 'start' for Windows) and the filename. We also pass the `shell=True` keyword argument, which is needed only on Windows. The operating system knows all of the file associations and can figure out that it should launch, say, *Notepad.exe* to handle the *hello.txt* file.

On OS X, the `open` program is used for opening both document files and programs. Enter the following into the interactive shell if you have a Mac:

```
>>> subprocess.Popen(['open', '/Applications/Calculator.app/'])
<subprocess.Popen object at 0x10202ff98>
```

The Calculator app should open.

THE UNIX PHILOSOPHY

Programs well designed to be launched by other programs become more powerful than their code alone. The *Unix philosophy* is a set of software design principles established by the programmers of the Unix operating system (on which the modern Linux and OS X are built). It says that it's better to write small, limited-purpose programs that can interoperate, rather than large, feature-rich applications. The smaller programs are easier to understand, and by being interoperable, they can be the building blocks of much more powerful applications.

Smartphone apps follow this approach as well. If your restaurant app needs to display directions to a café, the developers didn't reinvent the wheel by writing their own map code. The restaurant app simply launches a map app while passing it the café's address, just as your Python code would call a function and pass it arguments.

The Python programs you've been writing in this book mostly fit the Unix philosophy, especially in one important way: They use command line arguments rather than `input()` function calls. If all the information your program needs can be supplied up front, it is preferable to have this information passed as command line arguments rather than waiting for the user to type it in. This way, the command line arguments can be entered by a human user or supplied by another program. This interoperable approach will make your programs reusable as part of another program.

The sole exception is that you don't want passwords passed as command line arguments, since the command line may record them as part of its command history feature. Instead, your program should call the `input()` function when it needs you to enter a password.

You can read more about Unix philosophy at https://en.wikipedia.org/wiki/Unix_philosophy/.

Project: Simple Countdown Program

Just like it's hard to find a simple stopwatch application, it can be hard to find a simple countdown application. Let's write a countdown program that plays an alarm at the end of the countdown.

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

- Count down from 60.
- Play a sound file (*alarm.wav*) when the countdown reaches zero.

This means your code will need to do the following:

- Pause for one second in between displaying each number in the countdown by calling `time.sleep()`.
- Call `subprocess.Popen()` to open the sound file with the default application.

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

Step 1: Count Down

This program will require the `time` module for the `time.sleep()` function and the `subprocess` module for the `subprocess.Popen()` function. Enter the following code and save the file as *countdown.py*:

```
#! python3
# countdown.py - A simple countdown script.

import time, subprocess

❶ timeLeft = 60
while timeLeft > 0:
    ❷     print(timeLeft, end='')
    ❸     time.sleep(1)
    ❹     timeLeft = timeLeft - 1

# TODO: At the end of the countdown, play a sound file.
```

After importing `time` and `subprocess`, make a variable called `timeLeft` to hold the number of seconds left in the countdown ❶. It can start at 60—or you can change the value here to whatever you need or even have it get set from a command line argument.

In a `while` loop, you display the remaining count ❷, pause for one second ❸, and then decrement the `timeLeft` variable ❹ before the loop starts over again. The loop will keep looping as long as `timeLeft` is greater than 0. After that, the countdown will be over.

Step 2: Play the Sound File

While there are third-party modules to play sound files of various formats, the quick and easy way is to just launch whatever application the user already

uses to play sound files. The operating system will figure out from the `.wav` file extension which application it should launch to play the file. This `.wav` file could easily be some other sound file format, such as `.mp3` or `.ogg`.

You can use any sound file that is on your computer to play at the end of the countdown, or you can download `alarm.wav` from <http://nostarch.com/automatestuff/>.

Add the following to your code:

```
#! python3
# countdown.py - A simple countdown script.

import time, subprocess

--snip--

# At the end of the countdown, play a sound file.
subprocess.Popen(['start', 'alarm.wav'], shell=True)
```

After the while loop finishes, `alarm.wav` (or the sound file you choose) will play to notify the user that the countdown is over. On Windows, be sure to include 'start' in the list you pass to `Popen()` and pass the keyword argument `shell=True`. On OS X, pass 'open' instead of 'start' and remove `shell=True`.

Instead of playing a sound file, you could save a text file somewhere with a message like *Break time is over!* and use `Popen()` to open it at the end of the countdown. This will effectively create a pop-up window with a message. Or you could use the `webbrowser.open()` function to open a specific website at the end of the countdown. Unlike some free countdown application you'd find online, your own countdown program's alarm can be anything you want!

Ideas for Similar Programs

A countdown is a simple delay before continuing the program's execution. This can also be used for other applications and features, such as the following:

- Use `time.sleep()` to give the user a chance to press CTRL-C to cancel an action, such as deleting files. Your program can print a "Press CTRL-C to cancel" message and then handle any `KeyboardInterrupt` exceptions with `try` and `except` statements.
- For a long-term countdown, you can use `timedelta` objects to measure the number of days, hours, minutes, and seconds until some point (a birthday? an anniversary?) in the future.

Summary

The Unix epoch (January 1, 1970, at midnight, UTC) is a standard reference time for many programming languages, including Python. While the `time.time()` function module returns an epoch timestamp (that is, a float value of the number of seconds since the Unix epoch), the `datetime` module is better for performing date arithmetic and formatting or parsing strings with date information.

The `time.sleep()` function will block (that is, not return) for a certain number of seconds. It can be used to add pauses to your program. But if you want to schedule your programs to start at a certain time, the instructions at <http://nostarch.com/automatestuff/> can tell you how to use the scheduler already provided by your operating system.

The `threading` module is used to create multiple threads, which is useful when you need to download multiple files or do other tasks simultaneously. But make sure the thread reads and writes only local variables, or you might run into concurrency issues.

Finally, your Python programs can launch other applications with the `subprocess.Popen()` function. Command line arguments can be passed to the `Popen()` call to open specific documents with the application. Alternatively, you can use the `start`, `open`, or `see` program with `Popen()` to use your computer's file associations to automatically figure out which application to use to open a document. By using the other applications on your computer, your Python programs can leverage their capabilities for your automation needs.

Practice Questions

1. What is the Unix epoch?
2. What function returns the number of seconds since the Unix epoch?
3. How can you pause your program for exactly 5 seconds?
4. What does the `round()` function return?
5. What is the difference between a `datetime` object and a `timedelta` object?
6. Say you have a function named `spam()`. How can you call this function and run the code inside it in a separate thread?
7. What should you do to avoid concurrency issues with multiple threads?
8. How can you have your Python program run the `calc.exe` program located in the `C:\Windows\System32` folder?

Practice Projects

For practice, write programs that do the following.

Prettified Stopwatch

Expand the stopwatch project from this chapter so that it uses the `rjust()` and `ljust()` string methods to “prettyfify” the output. (These methods were covered in Chapter 6.) Instead of output such as this:

```
Lap #1: 3.56 (3.56)
Lap #2: 8.63 (5.07)
Lap #3: 17.68 (9.05)
Lap #4: 19.11 (1.43)
```

... the output will look like this:

```
Lap # 1: 3.56 ( 3.56)
Lap # 2: 8.63 ( 5.07)
Lap # 3: 17.68 ( 9.05)
Lap # 4: 19.11 ( 1.43)
```

Note that you will need string versions of the `lapNum`, `lapTime`, and `totalTime` integer and float variables in order to call the string methods on them.

Next, use the `pyperclip` module introduced in Chapter 6 to copy the text output to the clipboard so the user can quickly paste the output to a text file or email.

Scheduled Web Comic Downloader

Write a program that checks the websites of several web comics and automatically downloads the images if the comic was updated since the program’s last visit. Your operating system’s scheduler (Scheduled Tasks on Windows, launchd on OS X, and cron on Linux) can run your Python program once a day. The Python program itself can download the comic and then copy it to your desktop so that it is easy to find. This will free you from having to check the website yourself to see whether it has updated. (A list of web comics is available at <http://nostarch.com/automatestuff/>.)

16

SENDING EMAIL AND TEXT MESSAGES



Checking and replying to email is a huge time sink. Of course, you can't just write a program to handle all your email for you, since each message requires its own response.

But you can still automate plenty of email-related tasks once you know how to write programs that can send and receive email.

For example, maybe you have a spreadsheet full of customer records and want to send each customer a different form letter depending on their age and location details. Commercial software might not be able to do this for you; fortunately, you can write your own program to send these emails, saving yourself a lot of time copying and pasting form emails.

You can also write programs to send emails and SMS texts to notify you of things even while you're away from your computer. If you're automating a task that takes a couple of hours to do, you don't want to go back to your computer every few minutes to check on the program's status. Instead, the program can just text your phone when it's done—freeing you to focus on more important things while you're away from your computer.

SMTP

Much like HTTP is the protocol used by computers to send web pages across the Internet, *Simple Mail Transfer Protocol (SMTP)* is the protocol used for sending email. SMTP dictates how email messages should be formatted, encrypted, and relayed between mail servers, and all the other details that your computer handles after you click Send. You don't need to know these technical details, though, because Python's `smtplib` module simplifies them into a few functions.

SMTP just deals with sending emails to others. A different protocol, called IMAP, deals with retrieving emails sent to you and is described in “IMAP” on page 366.

Sending Email

You may be familiar with sending emails from Outlook or Thunderbird or through a website such as Gmail or Yahoo! Mail. Unfortunately, Python doesn't offer you a nice graphical user interface like those services. Instead, you call functions to perform each major step of SMTP, as shown in the following interactive shell example.

NOTE

Don't enter this example in IDLE; it won't work because `smtp.example.com`, `bob@example.com`, `MY_SECRET_PASSWORD`, and `alice@example.com` are just placeholders. This code is just an overview of the process of sending email with Python.

```
>>> import smtplib
>>> smtpObj = smtplib.SMTP('smtp.example.com', 587)
>>> smtpObj.ehlo()
(250, b'mx.example.com at your service, [216.172.148.131]\nSIZE 35882577\
n8BITMIME\nSTARTTLS\nENHANCEDSTATUSCODES\nCHUNKING')
>>> smtpObj.starttls()
(220, b'2.0.0 Ready to start TLS')
>>> smtpObj.login('bob@example.com', 'MY_SECRET_PASSWORD')
(235, b'2.7.0 Accepted')
>>> smtpObj.sendmail('bob@example.com', 'alice@example.com', 'Subject: So
long.\nDear Alice, so long and thanks for all the fish. Sincerely, Bob')
{}
>>> smtpObj.quit()
(221, b'2.0.0 closing connection ko10sm23097611pbd.52 - gsmtp')
```

In the following sections, we'll go through each step, replacing the placeholders with your information to connect and log in to an SMTP server, send an email, and disconnect from the server.

Connecting to an SMTP Server

If you've ever set up Thunderbird, Outlook, or another program to connect to your email account, you may be familiar with configuring the SMTP server and port. These settings will be different for each email provider, but a web search for *<your provider> smtp settings* should turn up the server and port to use.

The domain name for the SMTP server will usually be the name of your email provider's domain name, with *smtp.* in front of it. For example, Gmail's SMTP server is at *smtp.gmail.com*. Table 16-1 lists some common email providers and their SMTP servers. (The port is an integer value and will almost always be 587, which is used by the command encryption standard, TLS.)

Table 16-1: Email Providers and Their SMTP Servers

| Provider | SMTP server domain name |
|-------------------------|-------------------------------------|
| Gmail | <i>smtp.gmail.com</i> |
| Outlook.com/Hotmail.com | <i>smtp-mail.outlook.com</i> |
| Yahoo Mail | <i>smtp.mail.yahoo.com</i> |
| AT&T | <i>smpt.mail.att.net</i> (port 465) |
| Comcast | <i>smtp.comcast.net</i> |
| Verizon | <i>smtp.verizon.net</i> (port 465) |

Once you have the domain name and port information for your email provider, create an `SMTP` object by calling `smtplib.SMTP()`, passing the domain name as a string argument, and passing the port as an integer argument. The `SMTP` object represents a connection to an SMTP mail server and has methods for sending emails. For example, the following call creates an `SMTP` object for connecting to Gmail:

```
>>> smtpObj = smtplib.SMTP('smtp.gmail.com', 587)
>>> type(smtpObj)
<class 'smtplib.SMTP'>
```

Entering `type(smtpObj)` shows you that there's an `SMTP` object stored in `smtpObj`. You'll need this `SMTP` object in order to call the methods that log you in and send emails. If the `smtplib.SMTP()` call is not successful, your `SMTP` server might not support TLS on port 587. In this case, you will need to create an `SMTP` object using `smtplib.SMTP_SSL()` and port 465 instead.

```
>>> smtpObj = smtplib.SMTP_SSL('smtp.gmail.com', 465)
```

NOTE

If you are not connected to the Internet, Python will raise a `socket.gaierror`: [Errno 11004] `getaddrinfo` failed or similar exception.

For your programs, the differences between TLS and SSL aren't important. You only need to know which encryption standard your SMTP server uses so you know how to connect to it. In all of the interactive shell examples that follow, the `smtpObj` variable will contain an `SMTP` object returned by the `smtplib.SMTP()` or `smtplib.SMTP_SSL()` function.

Sending the SMTP “Hello” Message

Once you have the `SMTP` object, call its oddly named `ehlo()` method to “say hello” to the SMTP email server. This greeting is the first step in SMTP and is important for establishing a connection to the server. You don’t need to know the specifics of these protocols. Just be sure to call the `ehlo()` method first thing after getting the `SMTP` object or else the later method calls will result in errors. The following is an example of an `ehlo()` call and its return value:

```
>>> smtpObj.ehlo()
(250, b'mx.google.com at your service, [216.172.148.131]\nSIZE 35882577\
n8BITMIME\nSTARTTLS\nENHANCEDSTATUSCODES\nCHUNKING')
```

If the first item in the returned tuple is the integer 250 (the code for “success” in SMTP), then the greeting succeeded.

Starting TLS Encryption

If you are connecting to port 587 on the SMTP server (that is, you’re using TLS encryption), you’ll need to call the `starttls()` method next. This required step enables encryption for your connection. If you are connecting to port 465 (using SSL), then encryption is already set up, and you should skip this step.

Here’s an example of the `starttls()` method call:

```
>>> smtpObj.starttls()
(220, b'2.0.0 Ready to start TLS')
```

`starttls()` puts your SMTP connection in TLS mode. The 220 in the return value tells you that the server is ready.

Logging in to the SMTP Server

Once your encrypted connection to the SMTP server is set up, you can log in with your username (usually your email address) and email password by calling the `login()` method.

```
>>> smtpObj.login('my_email_address@gmail.com', 'MY_SECRET_PASSWORD')
(235, b'2.7.0 Accepted')
```

GMAIL'S APPLICATION-SPECIFIC PASSWORDS

Gmail has an additional security feature for Google accounts called *application-specific passwords*. If you receive an Application-specific password required error message when your program tries to log in, you will have to set up one of these passwords for your Python script. Check out the resources at <http://nostarch.com/automatestuff/> for detailed directions on how to set up an application-specific password for your Google account.

Pass a string of your email address as the first argument and a string of your password as the second argument. The 235 in the return value means authentication was successful. Python will raise an `smtplib.SMTPAuthenticationError` exception for incorrect passwords.

WARNING

Be careful about putting passwords in your source code. If anyone ever copies your program, they'll have access to your email account! It's a good idea to call `input()` and have the user type in the password. It may be inconvenient to have to enter a password each time you run your program, but this approach will prevent you from leaving your password in an unencrypted file on your computer where a hacker or laptop thief could easily get it.

Sending an Email

Once you are logged in to your email provider's SMTP server, you can call the `sendmail()` method to actually send the email. The `sendmail()` method call looks like this:

```
>>> smtpObj.sendmail('my_email_address@gmail.com', 'recipient@example.com',
'Subject: So long.\nDear Alice, so long and thanks for all the fish. Sincerely,
Bob')
{}
```

The `sendmail()` method requires three arguments.

- Your email address as a string (for the email's "from" address)
- The recipient's email address as a string or a list of strings for multiple recipients (for the "to" address)
- The email body as a string

The start of the email body string *must* begin with '`Subject: \n`' for the subject line of the email. The '`\n`' newline character separates the subject line from the main body of the email.

The return value from `sendmail()` is a dictionary. There will be one key-value pair in the dictionary for each recipient for whom email delivery failed. An empty dictionary means all recipients were successfully sent the email.

Disconnecting from the SMTP Server

Be sure to call the `quit()` method when you are done sending emails. This will disconnect your program from the SMTP server.

```
>>> smtpObj.quit()  
(221, b'2.0.0 closing connection ko10sm23097611pbd.52 - gsmtp')
```

The 221 in the return value means the session is ending.

To review all the steps for connecting and logging in to the server, sending email, and disconnection, see “[Sending Email](#)” on page 362.

IMAP

Just as SMTP is the protocol for sending email, the *Internet Message Access Protocol (IMAP)* specifies how to communicate with an email provider’s server to retrieve emails sent to your email address. Python comes with an `imaplib` module, but in fact the third-party `imapclient` module is easier to use. This chapter provides an introduction to using IMAPClient; the full documentation is at <http://imapclient.readthedocs.org/>.

The `imapclient` module downloads emails from an IMAP server in a rather complicated format. Most likely, you’ll want to convert them from this format into simple string values. The `pyzmail` module does the hard job of parsing these email messages for you. You can find the complete documentation for PyzMail at <http://www.magiksys.net/pyzmail/>.

Install `imapclient` and `pyzmail` from a Terminal window. Appendix A has steps on how to install third-party modules.

Retrieving and Deleting Emails with IMAP

Finding and retrieving an email in Python is a multistep process that requires both the `imapclient` and `pyzmail` third-party modules. Just to give you an overview, here’s a full example of logging in to an IMAP server, searching for emails, fetching them, and then extracting the text of the email messages from them.

```
>>> import imapclient  
>>> imapObj = imapclient.IMAPClient('imap.gmail.com', ssl=True)  
>>> imapObj.login('my_email_address@gmail.com', 'MY_SECRET_PASSWORD')  
'my_email_address@gmail.com Jane Doe authenticated (Success)'  
>>> imapObj.select_folder('INBOX', readonly=True)  
>>> UIDs = imapObj.search(['SINCE 05-Jul-2014'])  
>>> UIDs  
[40032, 40033, 40034, 40035, 40036, 40037, 40038, 40039, 40040, 40041]  
>>> rawMessages = imapObj.fetch([40041], ['BODY[]', 'FLAGS'])  
>>> import pyzmail  
>>> message = pyzmail.PyzMessage.factory(rawMessages[40041]['BODY[]'])  
>>> message.get_subject()  
'Hello!'
```

```

>>> message.get_addresses('from')
[('Edward Snowden', 'esnowden@nsa.gov')]
>>> message.get_addresses('to')
[(Jane Doe', 'jdoe@example.com')]
>>> message.get_addresses('cc')
[]
>>> message.get_addresses('bcc')
[]
>>> message.text_part != None
True
>>> message.text_part.get_payload().decode(message.text_part.charset)
'Follow the money.\r\n\r\n-Ed\r\n'
>>> message.html_part != None
True
>>> message.html_part.get_payload().decode(message.html_part.charset)
'<div dir="ltr"><div>So long, and thanks for all the fish!<br><br></div>-Al<br></div>\r\n'
>>> imapObj.logout()

```

You don't have to memorize these steps. After we go through each step in detail, you can come back to this overview to refresh your memory.

Connecting to an IMAP Server

Just like you needed an SMTP object to connect to an SMTP server and send email, you need an `IMAPClient` object to connect to an IMAP server and receive email. First you'll need the domain name of your email provider's IMAP server. This will be different from the SMTP server's domain name. Table 16-2 lists the IMAP servers for several popular email providers.

Table 16-2: Email Providers and Their IMAP Servers

| Provider | IMAP server domain name |
|-------------------------|------------------------------|
| Gmail | <i>imap.gmail.com</i> |
| Outlook.com/Hotmail.com | <i>imap-mail.outlook.com</i> |
| Yahoo Mail | <i>imap.mail.yahoo.com</i> |
| AT&T | <i>imap.mail.att.net</i> |
| Comcast | <i>imap.comcast.net</i> |
| Verizon | <i>incoming.verizon.net</i> |

Once you have the domain name of the IMAP server, call the `imapclient.IMAPClient()` function to create an `IMAPClient` object. Most email providers require SSL encryption, so pass the `ssl=True` keyword argument. Enter the following into the interactive shell (using your provider's domain name):

```

>>> import imapclient
>>> imapObj = imapclient.IMAPClient('imap.gmail.com', ssl=True)

```

In all of the interactive shell examples in the following sections, the `imapObj` variable will contain an `IMAPClient` object returned from the `imapclient.IMAPClient()` function. In this context, a *client* is the object that connects to the server.

Logging in to the IMAP Server

Once you have an `IMAPClient` object, call its `login()` method, passing in the username (this is usually your email address) and password as strings.

```
>>> imapObj.login('my_email_address@gmail.com', 'MY_SECRET_PASSWORD')
'my_email_address@gmail.com Jane Doe authenticated (Success)'
```

WARNING

Remember, never write a password directly into your code! Instead, design your program to accept the password returned from `input()`.

If the IMAP server rejects this username/password combination, Python will raise an `imaplib.error` exception. For Gmail accounts, you may need to use an application-specific password; for more details, see “Gmail’s Application-Specific Passwords” on page 365.

Searching for Email

Once you’re logged on, actually retrieving an email that you’re interested in is a two-step process. First, you must select a folder you want to search through. Then, you must call the `IMAPClient` object’s `search()` method, passing in a string of IMAP search keywords.

Selecting a Folder

Almost every account has an `INBOX` folder by default, but you can also get a list of folders by calling the `IMAPClient` object’s `list_folders()` method. This returns a list of tuples. Each tuple contains information about a single folder. Continue the interactive shell example by entering the following:

```
>>> import pprint
>>> pprint.pprint(imapObj.list_folders())
[('\\HasNoChildren',), '/', 'Drafts'),
 ('\\HasNoChildren',), '/', 'Filler'),
 ('\\HasNoChildren',), '/', 'INBOX'),
 ('\\HasNoChildren',), '/', 'Sent'),
 --snip--
 ('\\HasNoChildren', '\\Flagged'), '/', '[Gmail]/Starred'),
 ('\\HasNoChildren', '\\Trash'), '/', '[Gmail]/Trash')]
```

This is what your output might look like if you have a Gmail account. (Gmail calls its folders *labels*, but they work the same way as folders.) The three values in each of the tuples—for example, (('\\HasNoChildren',), '/', 'INBOX')—are as follows:

- A tuple of the folder's flags. (Exactly what these flags represent is beyond the scope of this book, and you can safely ignore this field.)
- The delimiter used in the name string to separate parent folders and subfolders.
- The full name of the folder.

To select a folder to search through, pass the folder's name as a string into the `IMAPClient` object's `select_folder()` method.

```
>>> imapObj.select_folder('INBOX', readonly=True)
```

You can ignore `select_folder()`'s return value. If the selected folder does not exist, Python will raise an `imaplib.error` exception.

The `readonly=True` keyword argument prevents you from accidentally making changes or deletions to any of the emails in this folder during the subsequent method calls. Unless you *want* to delete emails, it's a good idea to always set `readonly` to `True`.

Performing the Search

With a folder selected, you can now search for emails with the `IMAPClient` object's `search()` method. The argument to `search()` is a list of strings, each formatted to the IMAP's search keys. Table 16-3 describes the various search keys.

Table 16-3: IMAP Search Keys

| Search key | Meaning |
|--|---|
| 'ALL' | Returns all messages in the folder. You may run into <code>imaplib</code> size limits if you request all the messages in a large folder. See "Size Limits" on page 371. |
| 'BEFORE date', 'ON date', 'SINCE date' | These three search keys return, respectively, messages that were received by the IMAP server before, on, or after the given <i>date</i> . The date must be formatted like 05-Jul-2015. Also, while 'SINCE 05-Jul-2015' will match messages on and after July 5, 'BEFORE 05-Jul-2015' will match only messages before July 5 but not on July 5 itself. |
| 'SUBJECT string', 'BODY string', 'TEXT string' | Returns messages where <i>string</i> is found in the subject, body, or either, respectively. If <i>string</i> has spaces in it, then enclose it with double quotes: 'TEXT "search with spaces"'. |

(continued)

Table 16-3 (continued)

| Search key | Meaning |
|--|---|
| 'FROM <i>string</i> ', 'TO <i>string</i> ', 'CC <i>string</i> ', 'BCC <i>string</i> ' | Returns all messages where <i>string</i> is found in the "from" email address, "to" addresses, "cc" (carbon copy) addresses, or "bcc" (blind carbon copy) addresses, respectively. If there are multiple email addresses in <i>string</i> , then separate them with spaces and enclose them all with double quotes: 'CC "firstcc@example.com secondcc@example.com"'. |
| 'SEEN', 'UNSEEN' | Returns all messages with and without the \Seen flag, respectively. An email obtains the \Seen flag if it has been accessed with a <code>fetch()</code> method call (described later) or if it is clicked when you're checking your email in an email program or web browser. It's more common to say the email has been "read" rather than "seen," but they mean the same thing. |
| 'ANSWERED', 'UNANSWERED' | Returns all messages with and without the \Answered flag, respectively. A message obtains the \Answered flag when it is replied to. |
| 'DELETED', 'UNDELETED' | Returns all messages with and without the \Deleted flag, respectively. Email messages deleted with the <code>delete_messages()</code> method are given the \Deleted flag but are not permanently deleted until the <code>expunge()</code> method is called (see "Deleting Emails" on page 375). Note that some email providers, such as Gmail, automatically expunge emails. |
| 'DRAFT', 'UNDRAFT' | Returns all messages with and without the \Draft flag, respectively. Draft messages are usually kept in a separate Drafts folder rather than in the INBOX folder. |
| 'FLAGGED', 'UNFLAGGED' | Returns all messages with and without the \Flagged flag, respectively. This flag is usually used to mark email messages as "Important" or "Urgent." |
| 'LARGER <i>N</i> ', 'SMALLER <i>N</i> ' | Returns all messages larger or smaller than <i>N</i> bytes, respectively. |
| 'NOT <i>search-key</i> ' | Returns the messages that <i>search-key</i> would <i>not</i> have returned. |
| 'OR <i>search-key1</i> <i>search-key2</i> ' | Returns the messages that match <i>either</i> the first or second <i>search-key</i> . |

Note that some IMAP servers may have slightly different implementations for how they handle their flags and search keys. It may require some experimentation in the interactive shell to see exactly how they behave.

You can pass multiple IMAP search key strings in the list argument to the `search()` method. The messages returned are the ones that match *all* the search keys. If you want to match *any* of the search keys, use the OR search key. For the NOT and OR search keys, one and two complete search keys follow the NOT and OR, respectively.

Here are some example `search()` method calls along with their meanings:

`imapObj.search(['ALL'])` Returns every message in the currently selected folder.

`imapObj.search(['ON 05-Jul-2015'])` Returns every message sent on July 5, 2015.

```
imapObj.search(['SINCE 01-Jan-2015', 'BEFORE 01-Feb-2015', 'UNSEEN'])
```

Returns every message sent in January 2015 that is unread. (Note that this means *on and after* January 1 and up to *but not including* February 1.)

```
imapObj.search(['SINCE 01-Jan-2015', 'FROM alice@example.com'])
```

Returns every message from *alice@example.com* sent since the start of 2015.

```
imapObj.search(['SINCE 01-Jan-2015', 'NOT FROM alice@example.com'])
```

Returns every message sent from everyone except *alice@example.com* since the start of 2015.

```
imapObj.search(['OR FROM alice@example.com FROM bob@example.com'])
```

Returns every message ever sent from *alice@example.com* or *bob@example.com*.

```
imapObj.search(['FROM alice@example.com', 'FROM bob@example.com'])
```

Trick example! This search will never return any messages, because messages must match *all* search keywords. Since there can be only one “from” address, it is impossible for a message to be from both *alice@example.com* and *bob@example.com*.

The `search()` method doesn’t return the emails themselves but rather unique IDs (UIDs) for the emails, as integer values. You can then pass these UIDs to the `fetch()` method to obtain the email content.

Continue the interactive shell example by entering the following:

```
>>> UIDs = imapObj.search(['SINCE 05-Jul-2015'])
>>> UIDs
[40032, 40033, 40034, 40035, 40036, 40037, 40038, 40039, 40040, 40041]
```

Here, the list of message IDs (for messages received July 5 onward) returned by `search()` is stored in `UIDs`. The list of UIDs returned on your computer will be different from the ones shown here; they are unique to a particular email account. When you later pass UIDs to other function calls, use the UID values you received, not the ones printed in this book’s examples.

Size Limits

If your search matches a large number of email messages, Python might raise an exception that says `imaplib.error: got more than 10000 bytes`. When this happens, you will have to disconnect and reconnect to the IMAP server and try again.

This limit is in place to prevent your Python programs from eating up too much memory. Unfortunately, the default size limit is often too small. You can change this limit from 10,000 bytes to 10,000,000 bytes by running this code:

```
>>> import imaplib
>>> imaplib._MAXLINE = 10000000
```

This should prevent this error message from coming up again. You may want to make these two lines part of every IMAP program you write.

USING IMAPCLIENT'S GMAIL_SEARCH() METHOD

If you are logging in to the `imap.gmail.com` server to access a Gmail account, the `IMAPClient` object provides an extra search function that mimics the search bar at the top of the Gmail web page, as highlighted in Figure 16-1.

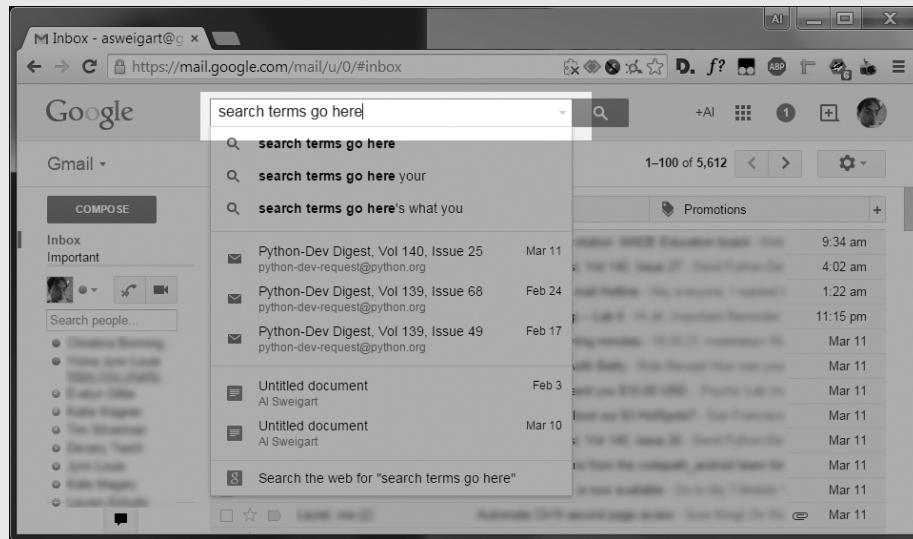


Figure 16-1: The search bar at the top of the Gmail web page

Instead of searching with IMAP search keys, you can use Gmail’s more sophisticated search engine. Gmail does a good job of matching closely related words (for example, a search for `driving` will also match `drive` and `drove`) and sorting the search results by most significant matches. You can also use Gmail’s advanced search operators (see <http://hostarch.com/automatestuff/> for more information). If you are logging in to a Gmail account, pass the search terms to the `gmail_search()` method instead of the `search()` method, like in the following interactive shell example:

```
>>> UIDs = imapObj.gmail_search('meaning of life')
>>> UIDs
[42]
```

Ah, yes—that’s that email with the meaning of life! I was looking for that.

Fetching an Email and Marking It As Read

Once you have a list of UIDs, you can call the `IMAPClient` object’s `fetch()` method to get the actual email content.

The list of UIDs will be `fetch()`’s first argument. The second argument should be the list `['BODY[]']`, which tells `fetch()` to download all the body content for the emails specified in your UID list.

Let's continue our interactive shell example.

```
>>> rawMessages = imapObj.fetch(UIDs, ['BODY[]'])
>>> import pprint
>>> pprint.pprint(rawMessages)
{40040: {'BODY[]': 'Delivered-To: my_email_address@gmail.com\r\n'
                  'Received: by 10.76.71.167 with SMTP id '
--snip--
                  '\r\n'
                  '-----_Part_6000970_707736290.1404819487066--\r\n',
 'SEQ': 5430}}
```

Import `pprint` and pass the return value from `fetch()`, stored in the variable `rawMessages`, to `pprint.pprint()` to “pretty print” it, and you’ll see that this return value is a nested dictionary of messages with UIDs as the keys. Each message is stored as a dictionary with two keys: `'BODY[]'` and `'SEQ'`. The `'BODY[]'` key maps to the actual body of the email. The `'SEQ'` key is for a *sequence number*, which has a similar role to the UID. You can safely ignore it.

As you can see, the message content in the `'BODY[]'` key is pretty unintelligible. It’s in a format called RFC 822, which is designed for IMAP servers to read. But you don’t need to understand the RFC 822 format; later in this chapter, the `pymail` module will make sense of it for you.

When you selected a folder to search through, you called `select_folder()` with the `readonly=True` keyword argument. Doing this will prevent you from accidentally deleting an email—but it also means that emails will not get marked as read if you fetch them with the `fetch()` method. If you *do* want emails to be marked as read when you fetch them, you will need to pass `readonly=False` to `select_folder()`. If the selected folder is already in read-only mode, you can reselect the current folder with another call to `select_folder()`, this time with the `readonly=False` keyword argument:

```
>>> imapObj.select_folder('INBOX', readonly=False)
```

Getting Email Addresses from a Raw Message

The raw messages returned from the `fetch()` method still aren’t very useful to people who just want to read their email. The `pymail` module parses these raw messages and returns them as `PyzMessage` objects, which make the subject, body, “To” field, “From” field, and other sections of the email easily accessible to your Python code.

Continue the interactive shell example with the following (using UIDs from your own email account, not the ones shown here):

```
>>> import pymail
>>> message = pymail.PyzMessage.factory(rawMessages[40041]['BODY[]'])
```

First, import `pymail`. Then, to create a `PyzMessage` object of an email, call the `pymail.PyzMessage.factory()` function and pass it the `'BODY[]'` section of the raw message. Store the result in `message`. Now `message` contains

a PyzMessage object, which has several methods that make it easy to get the email's subject line, as well as all sender and recipient addresses. The `get_subject()` method returns the subject as a simple string value. The `get_addresses()` method returns a list of addresses for the field you pass it. For example, the method calls might look like this:

```
>>> message.get_subject()
'Hello!'
>>> message.get_addresses('from')
[('Edward Snowden', 'esnowden@nsa.gov')]
>>> message.get_addresses('to')
[(Jane Doe', 'my_email_address@gmail.com')]
>>> message.get_addresses('cc')
[]
>>> message.get_addresses('bcc')
[]
```

Notice that the argument for `get_addresses()` is '`from`', '`to`', '`cc`', or '`bcc`'. The return value of `get_addresses()` is a list of tuples. Each tuple contains two strings: The first is the name associated with the email address, and the second is the email address itself. If there are no addresses in the requested field, `get_addresses()` returns a blank list. Here, the '`cc`' carbon copy and '`bcc`' blind carbon copy fields both contained no addresses and so returned empty lists.

Getting the Body from a Raw Message

Emails can be sent as plaintext, HTML, or both. Plaintext emails contain only text, while HTML emails can have colors, fonts, images, and other features that make the email message look like a small web page. If an email is only plaintext, its PyzMessage object will have its `html_part` attribute set to `None`. Likewise, if an email is only HTML, its PyzMessage object will have its `text_part` attribute set to `None`.

Otherwise, the `text_part` or `html_part` value will have a `get_payload()` method that returns the email's body as a value of the `bytes` data type. (The `bytes` data type is beyond the scope of this book.) But this *still* isn't a string value that we can use. Ugh! The last step is to call the `decode()` method on the `bytes` value returned by `get_payload()`. The `decode()` method takes one argument: the message's character encoding, stored in the `text_part.charset` or `html_part.charset` attribute. This, finally, will return the string of the email's body.

Continue the interactive shell example by entering the following:

```
❶ >>> message.text_part != None
True
>>> message.text_part.get_payload().decode(message.text_part.charset)
❷ 'So long, and thanks for all the fish!\r\n\r\nAl\r\n'
❸ >>> message.html_part != None
True
```

```
❶ >>> message.html_part.get_payload().decode(message.html_part.charset)
'<div dir="ltr"><div>So long, and thanks for all the fish!<br><br></div>-Al
<br></div>\r\n'
```

The email we're working with has both plaintext and HTML content, so the PyzMessage object stored in `message` has `text_part` and `html_part` attributes not equal to `None` ❶ ❸. Calling `get_payload()` on the message's `text_part` and then calling `decode()` on the bytes value returns a string of the text version of the email ❷. Using `get_payload()` and `decode()` with the message's `html_part` returns a string of the HTML version of the email ❹.

Deleting Emails

To delete emails, pass a list of message UIDs to the `IMAPClient` object's `delete_messages()` method. This marks the emails with the `\Deleted` flag. Calling the `expunge()` method will permanently delete all emails with the `\Deleted` flag in the currently selected folder. Consider the following interactive shell example:

```
❶ >>> imapObj.select_folder('INBOX', readonly=False)
❷ >>> UIDs = imapObj.search(['ON 09-Jul-2015'])
>>> UIDs
[40066]
>>> imapObj.delete_messages(UIDs)
❸ {40066: ('\\Seen', '\\Deleted')}
>>> imapObj.expunge()
('Success', [(5452, 'EXISTS')])
```

Here we select the inbox by calling `select_folder()` on the `IMAPClient` object and passing '`INBOX`' as the first argument; we also pass the keyword argument `readonly=False` so that we can delete emails ❶. We search the inbox for messages received on a specific date and store the returned message IDs in `UIDs` ❷. Calling `delete_message()` and passing it `UIDs` returns a dictionary; each key-value pair is a message ID and a tuple of the message's flags, which should now include `\Deleted` ❸. Calling `expunge()` then permanently deletes messages with the `\Deleted` flag and returns a success message if there were no problems expunging the emails. Note that some email providers, such as Gmail, automatically expunge emails deleted with `delete_messages()` instead of waiting for an `expunge` command from the IMAP client.

Disconnecting from the IMAP Server

When your program has finished retrieving or deleting emails, simply call the `IMAPClient`'s `logout()` method to disconnect from the IMAP server.

```
>>> imapObj.logout()
```

If your program runs for several minutes or more, the IMAP server may *time out*, or automatically disconnect. In this case, the next method call your program makes on the `IMAPClient` object will raise an exception like the following:

```
imaplib.abort: socket error: [WinError 10054] An existing connection was  
forcibly closed by the remote host
```

In this event, your program will have to call `imapclient.IMAPClient()` to connect again.

Whew! That's it. There were a lot of hoops to jump through, but you now have a way to get your Python programs to log in to an email account and fetch emails. You can always consult the overview in "Retrieving and Deleting Emails with IMAP" on page 366 whenever you need to remember all of the steps.

Project: Sending Member Dues Reminder Emails

Say you have been "volunteered" to track member dues for the Mandatory Volunteerism Club. This is a truly boring job, involving maintaining a spreadsheet of everyone who has paid each month and emailing reminders to those who haven't. Instead of going through the spreadsheet yourself and copying and pasting the same email to everyone who is behind on dues, let's—you guessed it—write a script that does this for you.

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

- Read data from an Excel spreadsheet.
- Find all members who have not paid dues for the latest month.
- Find their email addresses and send them personalized reminders.

This means your code will need to do the following:

- Open and read the cells of an Excel document with the `openpyxl` module. (See Chapter 12 for working with Excel files.)
- Create a dictionary of members who are behind on their dues.
- Log in to an SMTP server by calling `smtplib.SMTP()`, `ehlo()`, `starttls()`, and `login()`.
- For all members behind on their dues, send a personalized reminder email by calling the `sendmail()` method.

Open a new file editor window and save it as `sendDuesReminders.py`.

Step 1: Open the Excel File

Let's say the Excel spreadsheet you use to track membership dues payments looks like Figure 16-2 and is in a file named `duesRecords.xlsx`. You can download this file from <http://nostarch.com/automatestuff/>.

| | A | B | C | D | E | F | G | H |
|---|--------|-------------------|----------|----------|----------|----------|----------|----------|
| 1 | Member | Email | Jan 2014 | Feb 2014 | Mar 2014 | Apr 2014 | May 2014 | Jun 2014 |
| 2 | Alice | alice@example.com | paid | paid | paid | paid | paid | |
| 3 | Bob | bob@example.com | paid | paid | paid | paid | | |
| 4 | Carol | carol@example.com | paid | paid | paid | paid | paid | |
| 5 | David | david@example.com | paid | paid | paid | paid | paid | |
| 6 | Eve | eve@example.com | paid | paid | paid | | | |
| 7 | Fred | fred@example.com | paid | paid | paid | paid | paid | |
| 8 | | | | | | | | |
| 9 | | | | | | | | |

Figure 16-2: The spreadsheet for tracking member dues payments

This spreadsheet has every member's name and email address. Each month has a column tracking members' payment statuses. The cell for each member is marked with the text *paid* once they have paid their dues.

The program will have to open *duesRecords.xlsx* and figure out the column for the latest month by calling the `get_highest_column()` method. (You can consult Chapter 12 for more information on accessing cells in Excel spreadsheet files with the `openpyxl` module.) Enter the following code into the file editor window:

```

#! python3
# sendDuesReminders.py - Sends emails based on payment status in spreadsheet.

import openpyxl, smtplib, sys

# Open the spreadsheet and get the latest dues status.
❶ wb = openpyxl.load_workbook('duesRecords.xlsx')
❷ sheet = wb.get_sheet_by_name('Sheet1')

❸ lastCol = sheet.get_highest_column()
❹ latestMonth = sheet.cell(row=1, column=lastCol).value

# TODO: Check each member's payment status.

# TODO: Log in to email account.

# TODO: Send out reminder emails.

```

After importing the `openpyxl`, `smtplib`, and `sys` modules, we open our *duesRecords.xlsx* file and store the resulting `Workbook` object in `wb` ❶. Then we get Sheet 1 and store the resulting `Worksheet` object in `sheet` ❷. Now that we have a `Worksheet` object, we can access rows, columns, and cells. We store the highest column in `lastCol` ❸, and we then use row number 1 and `lastCol` to access the cell that should hold the most recent month. We get the value in this cell and store it in `latestMonth` ❹.

Step 2: Find All Unpaid Members

Once you've determined the column number of the latest month (stored in `lastCol`), you can loop through all rows after the first row (which has the column headers) to see which members have the text `paid` in the cell for that month's dues. If the member hasn't paid, you can grab the member's name and email address from columns 1 and 2, respectively. This information will go into the `unpaidMembers` dictionary, which will track all members who haven't paid in the most recent month. Add the following code to `sendDuesReminder.py`.

```
#! python3
# sendDuesReminders.py - Sends emails based on payment status in spreadsheet.

--snip--

# Check each member's payment status.
unpaidMembers = {}
❶ for r in range(2, sheet.get_highest_row() + 1):
❷     payment = sheet.cell(row=r, column=lastCol).value
        if payment != 'paid':
❸         name = sheet.cell(row=r, column=1).value
❹         email = sheet.cell(row=r, column=2).value
❺         unpaidMembers[name] = email
```

This code sets up an empty dictionary `unpaidMembers` and then loops through all the rows after the first ❶. For each row, the value in the most recent column is stored in `payment` ❷. If `payment` is not equal to `'paid'`, then the value of the first column is stored in `name` ❸, the value of the second column is stored in `email` ❹, and `name` and `email` are added to `unpaidMembers` ❺.

Step 3: Send Customized Email Reminders

Once you have a list of all unpaid members, it's time to send them email reminders. Add the following code to your program, except with your real email address and provider information:

```
#! python3
# sendDuesReminders.py - Sends emails based on payment status in spreadsheet.

--snip--

# Log in to email account.
smtpObj = smtplib.SMTP('smtp.gmail.com', 587)
smtpObj.ehlo()
smtpObj.starttls()
smtpObj.login('my_email_address@gmail.com', sys.argv[1])
```

Create an SMTP object by calling `smtplib.SMTP()` and passing it the domain name and port for your provider. Call `ehlo()` and `starttls()`, and then call `login()` and pass it your email address and `sys.argv[1]`, which will store your password string. You'll enter the password as a command line argument each time you run the program, to avoid saving your password in your source code.

Once your program has logged in to your email account, it should go through the `unpaidMembers` dictionary and send a personalized email to each member's email address. Add the following to `sendDuesReminders.py`:

```
#! python3
# sendDuesReminders.py - Sends emails based on payment status in spreadsheet.

--snip--

# Send out reminder emails.
for name, email in unpaidMembers.items():
    ❶    body = "Subject: %s dues unpaid.\nDear %s,\nRecords show that you have not
            paid dues for %s. Please make this payment as soon as possible. Thank you!" %
            (latestMonth, name, latestMonth)
    ❷    print('Sending email to %s...' % email)
    ❸    sendmailStatus = smtpObj.sendmail('my_email_address@gmail.com', email, body)

    ❹    if sendmailStatus != {}:
        print('There was a problem sending email to %s: %s' % (email,
            sendmailStatus))
smtpObj.quit()
```

This code loops through the names and emails in `unpaidMembers`. For each member who hasn't paid, we customize a message with the latest month and the member's name, and store the message in `body` ❶. We print output saying that we're sending an email to this member's email address ❷. Then we call `sendmail()`, passing it the from address and the customized message ❸. We store the return value in `sendmailStatus`.

Remember that the `sendmail()` method will return a nonempty dictionary value if the SMTP server reported an error sending that particular email. The last part of the for loop at ❹ checks if the returned dictionary is nonempty, and if it is, prints the recipient's email address and the returned dictionary.

After the program is done sending all the emails, the `quit()` method is called to disconnect from the SMTP server.

When you run the program, the output will look something like this:

```
Sending email to alice@example.com...
Sending email to bob@example.com...
Sending email to eve@example.com...
```

The recipients will receive an email that looks like Figure 16-3.

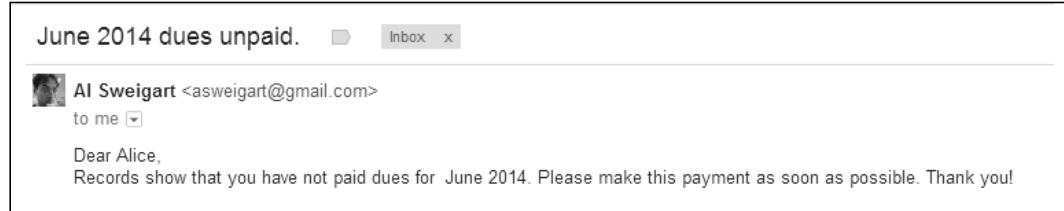


Figure 16-3: An automatically sent email from sendDuesReminders.py

Sending Text Messages with Twilio

Most people are more likely to be near their phones than their computers, so text messages can be a more immediate and reliable way of sending notifications than email. Also, the short length of text messages makes it more likely that a person will get around to reading them.

In this section, you'll learn how to sign up for the free Twilio service and use its Python module to send text messages. Twilio is an *SMS gateway service*, which means it's a service that allows you to send text messages from your programs. Although you will be limited in how many texts you can send per month and the texts will be prefixed with the words *Sent from a Twilio trial account*, this trial service is probably adequate for your personal programs. The free trial is indefinite; you won't have to upgrade to a paid plan later.

Twilio isn't the only SMS gateway service. If you prefer not to use Twilio, you can find alternative services by searching online for *free sms gateway*, *python sms api*, or even *twilio alternatives*.

Before signing up for a Twilio account, install the `twilio` module. Appendix A has more details about installing third-party modules.

NOTE

This section is specific to the United States. Twilio does offer SMS texting services for countries outside of the United States, but those specifics aren't covered in this book. The `twilio` module and its functions, however, will work the same outside the United States. See <http://twilio.com/> for more information.

Signing Up for a Twilio Account

Go to <http://twilio.com/> and fill out the sign-up form. Once you've signed up for a new account, you'll need to verify a mobile phone number that you want to send texts to. (This verification is necessary to prevent people from using the service to spam random phone numbers with text messages.)

After receiving the text with the verification number, enter it into the Twilio website to prove that you own the mobile phone you are verifying. You will now be able to send texts to this phone number using the `twilio` module.

Twilio provides your trial account with a phone number to use as the sender of text messages. You will need two more pieces of information: your account SID and the auth (authentication) token. You can find this information on the Dashboard page when you are logged in to your Twilio account. These values act as your Twilio username and password when logging in from a Python program.

Sending Text Messages

Once you've installed the `twilio` module, signed up for a Twilio account, verified your phone number, registered a Twilio phone number, and obtained your account SID and auth token, you will finally be ready to send yourself text messages from your Python scripts.

Compared to all the registration steps, the actual Python code is fairly simple. With your computer connected to the Internet, enter the following into the interactive shell, replacing the `accountSID`, `authToken`, `myTwilioNumber`, and `myCellPhone` variable values with your real information:

```
❶ >>> from twilio.rest import TwilioRestClient
>>> accountSID = 'ACxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
>>> authToken = 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
❷ >>> twilioCli = TwilioRestClient(accountSID, authToken)
>>> myTwilioNumber = '+14955551234'
>>> myCellPhone = '+14955558888'
❸ >>> message = twilioCli.messages.create(body='Mr. Watson - Come here - I want
to see you.', from_=myTwilioNumber, to=myCellPhone)
```

A few moments after typing the last line, you should receive a text message that reads *Sent from your Twilio trial account - Mr. Watson - Come here - I want to see you.*

Because of the way the `twilio` module is set up, you need to import it using `from twilio.rest import TwilioRestClient`, not just `import twilio` ❶. Store your account SID in `accountSID` and your auth token in `authToken` and then call `TwilioRestClient()` and pass it `accountSID` and `authToken`. The call to `TwilioRestClient()` returns a `TwilioRestClient` object ❷. This object has a `messages` attribute, which in turn has a `create()` method you can use to send text messages. This is the method that will instruct Twilio's servers to send your text message. After storing your Twilio number and cell phone number in `myTwilioNumber` and `myCellPhone`, respectively, call `create()` and pass it keyword arguments specifying the body of the text message, the sender's number (`myTwilioNumber`), and the recipient's number (`myCellPhone`) ❸.

The `Message` object returned from the `create()` method will have information about the text message that was sent. Continue the interactive shell example by entering the following:

```
>>> message.to
'+14955558888'
>>> message.from_
'+14955551234'
>>> message.body
'Mr. Watson - Come here - I want to see you.'
```

The `to`, `from_`, and `body` attributes should hold your cell phone number, Twilio number, and message, respectively. Note that the sending phone number is in the `from_` attribute—with an underscore at the end—not `from`. This is because `from` is a keyword in Python (you've seen it used in the `from`

`modulename import *` form of `import` statement, for example), so it cannot be used as an attribute name. Continue the interactive shell example with the following:

```
>>> message.status  
'queued'  
>>> message.date_created  
datetime.datetime(2015, 7, 8, 1, 36, 18)  
>>> message.date_sent == None  
True
```

The `status` attribute should give you a string. The `date_created` and `date_sent` attributes should give you a `datetime` object if the message has been created and sent. It may seem odd that the `status` attribute is set to `'queued'` and the `date_sent` attribute is set to `None` when you've already received the text message. This is because you captured the `Message` object in the `message` variable *before* the text was actually sent. You will need to refetch the `Message` object in order to see its most up-to-date `status` and `date_sent`. Every Twilio message has a unique string ID (SID) that can be used to fetch the latest update of the `Message` object. Continue the interactive shell example by entering the following:

```
>>> message.sid  
'SM09520de7639ba3af137c6fcf7c5f4b51'  
❶ >>> updatedMessage = twilioCli.messages.get(message.sid)  
>>> updatedMessage.status  
'delivered'  
>>> updatedMessage.date_sent  
datetime.datetime(2015, 7, 8, 1, 36, 18)
```

Entering `message.sid` shows you this message's long SID. By passing this SID to the Twilio client's `get()` method ❶, you can retrieve a new `Message` object with the most up-to-date information. In this new `Message` object, the `status` and `date_sent` attributes are correct.

The `status` attribute will be set to one of the following string values: `'queued'`, `'sending'`, `'sent'`, `'delivered'`, `'undelivered'`, or `'failed'`. These statuses are self-explanatory, but for more precise details, take a look at the resources at <http://nostarch.com/automatestuff/>.

RECEIVING TEXT MESSAGES WITH PYTHON

Unfortunately, receiving text messages with Twilio is a bit more complicated than sending them. Twilio requires that you have a website running its own web application. That's beyond the scope of this book, but you can find more details in the resources for this book (<http://nostarch.com/automatestuff/>).

Project: “Just Text Me” Module

The person you’ll most often text from your programs is probably you. Texting is a great way to send yourself notifications when you’re away from your computer. If you’ve automated a boring task with a program that takes a couple of hours to run, you could have it notify you with a text when it’s finished. Or you may have a regularly scheduled program running that sometimes needs to contact you, such as a weather-checking program that texts you a reminder to pack an umbrella.

As a simple example, here’s a small Python program with a `textmyself()` function that sends a message passed to it as a string argument. Open a new file editor window and enter the following code, replacing the account SID, auth token, and phone numbers with your own information. Save it as `textMyself.py`.

```
#! python3
# textMyself.py - Defines the textmyself() function that texts a message
# passed to it as a string.

# Preset values:
accountSID = 'ACxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
authToken  = 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
myNumber  = '+15559998888'
twilioNumber = '+15552225678'

from twilio.rest import TwilioRestClient

❶ def textmyself(message):
❷     twilioCli = TwilioRestClient(accountSID, authToken)
❸     twilioCli.messages.create(body=message, from_=twilioNumber, to=myNumber)
```

This program stores an account SID, auth token, sending number, and receiving number. It then defined `textmyself()` to take on argument ❶, make a `TwilioRestClient` object ❷, and call `create()` with the message you passed ❸.

If you want to make the `textmyself()` function available to your other programs, simply place the `textMyself.py` file in the same folder as the Python executable (`C:\Python34` on Windows, `/usr/local/lib/python3.4` on OS X, and `/usr/bin/python3` on Linux). Now you can use the function in your other programs. Whenever you want one of your programs to text you, just add the following:

```
import textmyself
textmyself.textmyself('The boring task is finished.')
```

You need to sign up for Twilio and write the texting code only once. After that, it’s just two lines of code to send a text from any of your other programs.

Summary

We communicate with each other on the Internet and over cell phone networks in dozens of different ways, but email and texting predominate. Your programs can communicate through these channels, which gives them powerful new notification features. You can even write programs running on different computers that communicate with one another directly via email, with one program sending emails with SMTP and the other retrieving them with IMAP.

Python's `smtplib` provides functions for using the SMTP to send emails through your email provider's SMTP server. Likewise, the third-party `imapclient` and `pymail` modules let you access IMAP servers and retrieve emails sent to you. Although IMAP is a bit more involved than SMTP, it's also quite powerful and allows you to search for particular emails, download them, and parse them to extract the subject and body as string values.

Texting is a bit different from email, since, unlike email, more than just an Internet connection is needed to send SMS texts. Fortunately, services such as Twilio provide modules to allow you to send text messages from your programs. Once you go through an initial setup process, you'll be able to send texts with just a couple lines of code.

With these modules in your skill set, you'll be able to program the specific conditions under which your programs should send notifications or reminders. Now your programs will have reach far beyond the computer they're running on!

Practice Questions

1. What is the protocol for sending email? For checking and receiving email?
2. What four `smtplib` functions/methods must you call to log in to an SMTP server?
3. What two `imapclient` functions/methods must you call to log in to an IMAP server?
4. What kind of argument do you pass to `imapObj.search()`?
5. What do you do if your code gets an error message that says got more than 10000 bytes?
6. The `imapclient` module handles connecting to an IMAP server and finding emails. What is one module that handles reading the emails that `imapclient` collects?
7. What three pieces of information do you need from Twilio before you can send text messages?

Practice Projects

For practice, write programs that do the following.

Random Chore Assignment Emailer

Write a program that takes a list of people's email addresses and a list of chores that need to be done and randomly assigns chores to people. Email each person their assigned chores. If you're feeling ambitious, keep a record of each person's previously assigned chores so that you can make sure the program avoids assigning anyone the same chore they did last time. For another possible feature, schedule the program to run once a week automatically.

Here's a hint: If you pass a list to the `random.choice()` function, it will return a randomly selected item from the list. Part of your code could look like this:

```
chores = ['dishes', 'bathroom', 'vacuum', 'walk dog']
randomChore = random.choice(chores)
chores.remove(randomChore)    # this chore is now taken, so remove it
```

Umbrella Reminder

Chapter 11 showed you how to use the `requests` module to scrape data from `http://weather.gov/`. Write a program that runs just before you wake up in the morning and checks whether it's raining that day. If so, have the program text you a reminder to pack an umbrella before leaving the house.

Auto Unsubscriber

Write a program that scans through your email account, finds all the unsubscribe links in all your emails, and automatically opens them in a browser. This program will have to log in to your email provider's IMAP server and download all of your emails. You can use BeautifulSoup (covered in Chapter 11) to check for any instance where the word `unsubscribe` occurs within an HTML link tag.

Once you have a list of these URLs, you can use `webbrowser.open()` to automatically open all of these links in a browser.

You'll still have to manually go through and complete any additional steps to unsubscribe yourself from these lists. In most cases, this involves clicking a link to confirm.

But this script saves you from having to go through all of your emails looking for unsubscribe links. You can then pass this script along to your friends so they can run it on their email accounts. (Just make sure your email password isn't hardcoded in the source code!)

Controlling Your Computer Through Email

Write a program that checks an email account every 15 minutes for any instructions you email it and executes those instructions automatically. For example, BitTorrent is a peer-to-peer downloading system. Using free BitTorrent software such as qBittorrent, you can download large media files on your home computer. If you email the program a (completely legal, not at all piratical) BitTorrent link, the program will eventually check its email, find this message, extract the link, and then launch qBittorrent to start downloading the file. This way, you can have your home computer begin downloads while you’re away, and the (completely legal, not at all piratical) download can be finished by the time you return home.

Chapter 15 covers how to launch programs on your computer using the `subprocess.Popen()` function. For example, the following call would launch the qBittorrent program, along with a torrent file:

```
qbProcess = subprocess.Popen(['C:\\\\Program Files (x86)\\\\qBittorrent\\\\
qbitTorrent.exe', 'shakespeare_complete_works.torrent'])
```

Of course, you’ll want the program to make sure the emails come from you. In particular, you might want to require that the emails contain a password, since it is fairly trivial for hackers to fake a “from” address in emails. The program should delete the emails it finds so that it doesn’t repeat instructions every time it checks the email account. As an extra feature, have the program email or text you a confirmation every time it executes a command. Since you won’t be sitting in front of the computer that is running the program, it’s a good idea to use the logging functions (see Chapter 10) to write a text file log that you can check if errors come up.

qBittorrent (as well as other BitTorrent applications) has a feature where it can quit automatically after the download completes. Chapter 15 explains how you can determine when a launched application has quit with the `wait()` method for `Popen` objects. The `wait()` method call will block until qBittorrent has stopped, and then your program can email or text you a notification that the download has completed.

There are a lot of possible features you could add to this project. If you get stuck, you can download an example implementation of this program from <http://nostarch.com/automatestuff/>.

17

MANIPULATING IMAGES



If you have a digital camera or even if you just upload photos from your phone to Facebook, you probably cross paths with digital image files all the time. You may know how to use basic graphics software, such as Microsoft Paint or Paintbrush, or even more advanced applications such as Adobe Photoshop. But if you need to edit a massive number of images, editing them by hand can be a lengthy, boring job.

Enter Python. Pillow is a third-party Python module for interacting with image files. The module has several functions that make it easy to crop, resize, and edit the content of an image. With the power to manipulate images the same way you would with software such as Microsoft Paint or Adobe Photoshop, Python can automatically edit hundreds or thousands of images with ease.

Computer Image Fundamentals

In order to manipulate an image, you need to understand the basics of how computers deal with colors and coordinates in images and how you can work with colors and coordinates in Pillow. But before you continue, install the `pillow` module. See Appendix A for help installing third-party modules.

Colors and *RGBA* Values

Computer programs often represent a color in an image as an *RGBA value*. An RGBA value is a group of numbers that specify the amount of red, green, blue, and *alpha* (or transparency) in a color. Each of these component values is an integer from 0 (none at all) to 255 (the maximum). These RGBA values are assigned to individual *pixels*; a pixel is the smallest dot of a single color the computer screen can show (as you can imagine, there are millions of pixels on a screen). A pixel's RGB setting tells it precisely what shade of color it should display. Images also have an alpha value to create RGBA values. If an image is displayed on the screen over a background image or desktop wallpaper, the alpha value determines how much of the background you can “see through” the image’s pixel.

In Pillow, RGBA values are represented by a tuple of four integer values. For example, the color red is represented by (255, 0, 0, 255). This color has the maximum amount of red, no green or blue, and the maximum alpha value, meaning it is fully opaque. Green is represented by (0, 255, 0, 255), and blue is (0, 0, 255, 255). White, the combination of all colors, is (255, 255, 255, 255), while black, which has no color at all, is (0, 0, 0, 255).

If a color has an alpha value of 0, it is invisible, and it doesn’t really matter what the RGB values are. After all, invisible red looks the same as invisible black.

Pillow uses the standard color names that HTML uses. Table 17-1 lists a selection of standard color names and their values.

Table 17-1: Standard Color Names and Their RGBA Values

| Name | RGBA value | Name | RGBA value |
|-------|----------------------|--------|--------------------|
| White | (255, 255, 255, 255) | Red | (255, 0, 0, 255) |
| Green | (0, 128, 0, 255) | Blue | (0, 0, 255, 255) |
| Gray | (128, 128, 128, 255) | Yellow | (255, 255, 0, 255) |
| Black | (0, 0, 0, 255) | Purple | (128, 0, 128, 255) |

Pillow offers the `ImageColor.getcolor()` function so you don’t have to memorize RGBA values for the colors you want to use. This function takes a color name string as its first argument, and the string ‘RGBA’ as its second argument, and it returns an RGBA tuple.

CMYK AND RGB COLORING

In grade school you learned that mixing red, yellow, and blue paints can form other colors; for example, you can mix blue and yellow to make green paint. This is known as the *subtractive color model*, and it applies to dyes, inks, and pigments. This is why color printers have CMYK ink cartridges: the *Cyan* (blue), *Magenta* (red), *Yellow*, and *black* ink can be mixed together to form any color.

However, the physics of light uses what's called an *additive color model*. When combining light (such as the light given off by your computer screen), red, green, and blue light can be combined to form any other color. This is why *RGB* values represent color in computer programs.

To see how this function works, enter the following into the interactive shell:

```
❶ >>> from PIL import ImageColor
❷ >>> ImageColor.getcolor('red', 'RGBA')
(255, 0, 0, 255)
❸ >>> ImageColor.getcolor('RED', 'RGBA')
(255, 0, 0, 255)
>>> ImageColor.getcolor('Black', 'RGBA')
(0, 0, 0, 255)
>>> ImageColor.getcolor('chocolate', 'RGBA')
(210, 105, 30, 255)
>>> ImageColor.getcolor('CornflowerBlue', 'RGBA')
(100, 149, 237, 255)
```

First, you need to import the `ImageColor` module from `PIL` ❶ (not from `Pillow`; you'll see why in a moment). The color name string you pass to `ImageColor.getcolor()` is case insensitive, so passing '`red`' ❷ and passing '`RED`' ❸ give you the same `RGBA` tuple. You can also pass more unusual color names, like '`chocolate`' and '`Cornflower Blue`'.

`Pillow` supports a huge number of color names, from '`aliceblue`' to '`whitesmoke`'. You can find the full list of more than 100 standard color names in the resources at <http://nostarch.com/automatestuff/>.

Coordinates and Box Tuples

Image pixels are addressed with x- and y-coordinates, which respectively specify a pixel's horizontal and vertical location in an image. The *origin* is the pixel at the top-left corner of the image and is specified with the notation `(0, 0)`. The first zero represents the x-coordinate, which starts at zero at the origin and increases going from left to right. The second zero represents the y-coordinate, which starts at zero at the origin and increases going

down the image. This bears repeating: y-coordinates increase going downward, which is the opposite of how you may remember y-coordinates being used in math class. Figure 17-1 demonstrates how this coordinate system works.

Many of Pillow's functions and methods take a *box tuple* argument. This means Pillow is expecting a tuple of four integer coordinates that represent a rectangular region in an image. The four integers are, in order, as follows:

- *Left*: The x-coordinate of the leftmost edge of the box.
- *Top*: The y-coordinate of the top edge of the box.
- *Right*: The x-coordinate of one pixel to the right of the rightmost edge of the box. This integer must be greater than the left integer.
- *Bottom*: The y-coordinate of one pixel lower than the bottom edge of the box. This integer must be greater than the top integer.

Note that the box includes the left and top coordinates and goes up to but does not include the right and bottom coordinates. For example, the box tuple (3, 1, 9, 6) represents all the pixels in the black box in Figure 17-2.

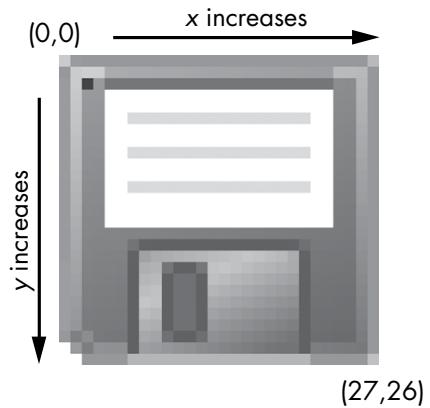


Figure 17-1: The x- and y-coordinates of a 27x26 image of some sort of ancient data storage device

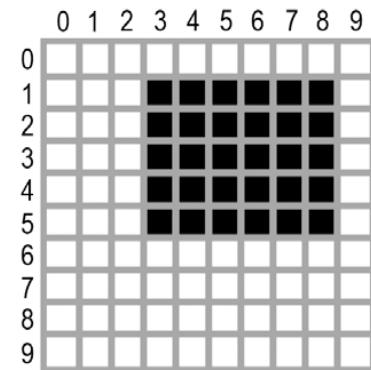


Figure 17-2: The area represented by the box tuple (3, 1, 9, 6)

Manipulating Images with Pillow

Now that you know how colors and coordinates work in Pillow, let's use Pillow to manipulate an image. Figure 17-3 is the image that will be used for all the interactive shell examples in this chapter. You can download it from <http://nostarch.com/automatestuff/>.

Once you have the image file *Zophie.png* in your current working directory, you'll be ready to load the image of Zophie into Python, like so:

```
>>> from PIL import Image  
>>> catIm = Image.open('zophie.png')
```



Figure 17-3: My cat Zophie. The camera adds 10 pounds (which is a lot for a cat).

To load the image, you import the `Image` module from Pillow and call `Image.open()`, passing it the image's filename. You can then store the loaded image in a variable like `CatIm`. The module name of Pillow is `PIL` to make it backward compatible with an older module called Python Imaging Library, which is why you must run `from PIL import Image` instead of `from Pillow import Image`. Because of the way Pillow's creators set up the `pillow` module, you must use the `from PIL import Image` form of `import` statement, rather than simply `import PIL`.

If the image file isn't in the current working directory, change the working directory to the folder that contains the image file by calling the `os.chdir()` function.

```
>>> import os  
>>> os.chdir('C:\\\\folder_with_image_file')
```

The `Image.open()` function returns a value of the `Image` object data type, which is how Pillow represents an image as a Python value. You can load an `Image` object from an image file (of any format) by passing the `Image.open()` function a string of the filename. Any changes you make to the `Image` object can be saved to an image file (also of any format) with the `save()` method. All the rotations, resizing, cropping, drawing, and other image manipulations will be done through method calls on this `Image` object.

To shorten the examples in this chapter, I'll assume you've imported Pillow's `Image` module and that you have the `Zophie` image stored in a variable named `catIm`. Be sure that the `zophie.png` file is in the current working directory so that the `Image.open()` function can find it. Otherwise, you will also have to specify the full absolute path in the string argument to `Image.open()`.

Working with the Image Data Type

An `Image` object has several useful attributes that give you basic information about the image file it was loaded from: its width and height, the filename, and the graphics format (such as JPEG, GIF, or PNG).

For example, enter the following into the interactive shell:

```
>>> from PIL import Image
>>> catIm = Image.open('zophie.png')
>>> catIm.size
❶ (816, 1088)
❷ >>> width, height = catIm.size
❸ >>> width
816
❹ >>> height
1088
>>> catIm.filename
'zophie.png'
>>> catIm.format
'PNG'
>>> catIm.format_description
'Portable network graphics'
❺ >>> catIm.save('zophie.jpg')
```

After making an `Image` object from *Zophie.png* and storing the `Image` object in `catIm`, we can see that the object's `size` attribute contains a tuple of the image's width and height in pixels ❶. We can assign the values in the tuple to `width` and `height` variables ❷ in order to access with `width` ❸ and `height` ❹ individually. The `filename` attribute describes the original file's name. The `format` and `format_description` attributes are strings that describe the image format of the original file (with `format_description` being a bit more verbose).

Finally, calling the `save()` method and passing it '*zophie.jpg*' saves a new image with the filename *zophie.jpg* to your hard drive ❺. Pillow sees that the file extension is *.jpg* and automatically saves the image using the JPEG image format. Now you should have two images, *zophie.png* and *zophie.jpg*, on your hard drive. While these files are based on the same image, they are not identical because of their different formats.

Pillow also provides the `Image.new()` function, which returns an `Image` object—much like `Image.open()`, except the image represented by `Image.new()`'s object will be blank. The arguments to `Image.new()` are as follows:

- The string '`RGBA`', which sets the color mode to `RGBA`. (There are other modes that this book doesn't go into.)
- The size, as a two-integer tuple of the new image's width and height.

- The background color that the image should start with, as a four-integer tuple of an RGBA value. You can use the return value of the `ImageColor.getcolor()` function for this argument. Alternatively, `Image.new()` also supports just passing the string of the standard color name.

For example, enter the following into the interactive shell:

```
>>> from PIL import Image
❶ >>> im = Image.new('RGBA', (100, 200), 'purple')
       >>> im.save('purpleImage.png')
❷ >>> im2 = Image.new('RGBA', (20, 20))
       >>> im2.save('transparentImage.png')
```

Here we create an `Image` object for an image that's 100 pixels wide and 200 pixels tall, with a purple background ❶. This image is then saved to the file `purpleImage.png`. We call `Image.new()` again to create another `Image` object, this time passing `(20, 20)` for the dimensions and nothing for the background color ❷. Invisible black, `(0, 0, 0, 0)`, is the default color used if no color argument is specified, so the second image has a transparent background; we save this 20×20 transparent square in `transparentImage.png`.

Cropping Images

Cropping an image means selecting a rectangular region inside an image and removing everything outside the rectangle. The `crop()` method on `Image` objects takes a box tuple and returns an `Image` object representing the cropped image. The cropping does not happen in place—that is, the original `Image` object is left untouched, and the `crop()` method returns a new `Image` object. Remember that a boxed tuple—in this case, the cropped section—includes the left column and top row of pixels but only goes up to and does *not* include the right column and bottom row of pixels.

Enter the following into the interactive shell:

```
>>> croppedIm = catIm.crop((335, 345, 565, 560))
>>> croppedIm.save('cropped.png')
```

This makes a new `Image` object for the cropped image, stores the object in `croppedIm`, and then calls `save()` on `croppedIm` to save the cropped image in `cropped.png`. The new file `cropped.png` will be created from the original image, like in Figure 17-4.

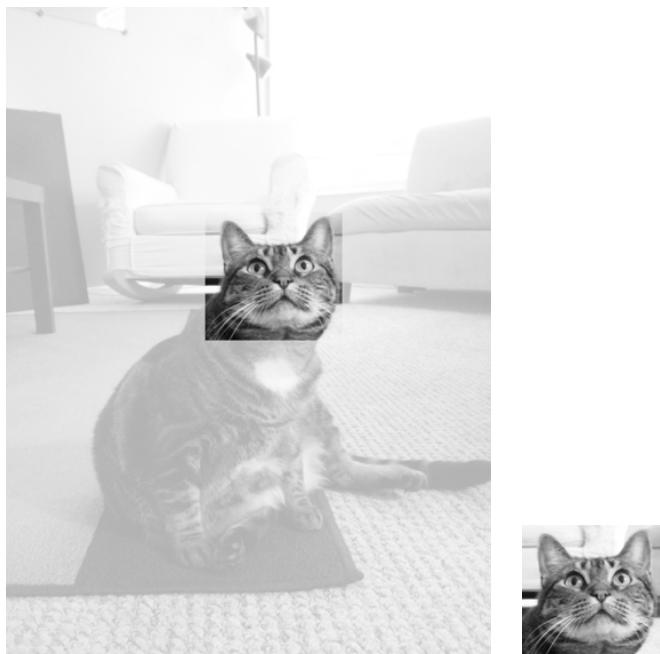


Figure 17-4: The new image will be just the cropped section of the original image.

Copying and Pasting Images onto Other Images

The `copy()` method will return a new `Image` object with the same image as the `Image` object it was called on. This is useful if you need to make changes to an image but also want to keep an untouched version of the original. For example, enter the following into the interactive shell:

```
>>> catIm = Image.open('zophie.png')
>>> catCopyIm = catIm.copy()
```

The `catIm` and `catCopyIm` variables contain two separate `Image` objects, which both have the same image on them. Now that you have an `Image` object stored in `catCopyIm`, you can modify `catCopyIm` as you like and save it to a new filename, leaving `zophie.png` untouched. For example, let's try modifying `catCopyIm` with the `paste()` method.

The `paste()` method is called on an `Image` object and pastes another image on top of it. Let's continue the shell example by pasting a smaller image onto `catCopyIm`.

```
>>> faceIm = catIm.crop((335, 345, 565, 560))
>>> faceIm.size
(230, 215)
>>> catCopyIm.paste(faceIm, (0, 0))
>>> catCopyIm.paste(faceIm, (400, 500))
>>> catCopyIm.save('pasted.png')
```

First we pass `crop()` a box tuple for the rectangular area in `zophie.png` that contains Zophie's face. This creates an `Image` object representing a 230×215 crop, which we store in `faceIm`. Now we can paste `faceIm` onto `catCopyIm`. The `paste()` method takes two arguments: a "source" `Image` object and a tuple of the x- and y-coordinates where you want to paste the top-left corner of the source `Image` object onto the main `Image` object. Here we call `paste()` twice on `catCopyIm`, passing $(0, 0)$ the first time and $(400, 500)$ the second time. This pastes `faceIm` onto `catCopyIm` twice: once with the top-left corner of `faceIm` at $(0, 0)$ on `catCopyIm`, and once with the top-left corner of `faceIm` at $(400, 500)$. Finally, we save the modified `catCopyIm` to `pasted.png`. The `pasted.png` image looks like Figure 17-5.

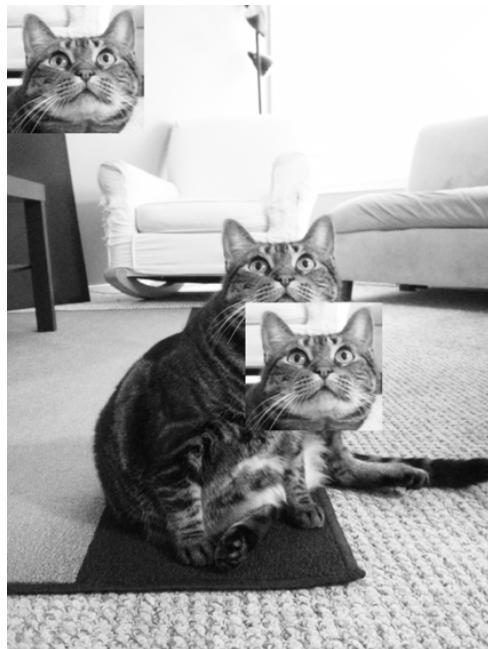


Figure 17-5: Zophie the cat, with her face pasted twice

NOTE

Despite their names, the `copy()` and `paste()` methods in Pillow do not use your computer's clipboard.

Note that the `paste()` method modifies its `Image` object *in place*; it does not return an `Image` object with the pasted image. If you want to call `paste()` but also keep an untouched version of the original image around, you'll need to first copy the image and then call `paste()` on that copy.

Say you want to tile Zophie's head across the entire image, as in Figure 17-6. You can achieve this effect with just a couple for loops. Continue the interactive shell example by entering the following:

```
>>> catImWidth, catImHeight = catIm.size  
>>> faceImWidth, faceImHeight = faceIm.size  
❶ >>> catCopyTwo = catIm.copy()
```

```

❷ >>> for left in range(0, catImWidth, faceImWidth):
❸         for top in range(0, catImHeight, faceImHeight):
                print(left, top)
                catCopyTwo.paste(faceIm, (left, top))

0 0
0 215
0 430
0 645
0 860
0 1075
230 0
230 215
--snip--
690 860
690 1075
>>> catCopyTwo.save('tiled.png')

```

Here we store the width of height of `catIm` in `catImWidth` and `catImHeight`. At ❶ we make a copy of `catIm` and store it in `catCopyTwo`. Now that we have a copy that we can paste onto, we start looping to paste `faceIm` onto `catCopyTwo`. The outer for loop's `left` variable starts at 0 and increases by `faceImWidth`(230) ❷. The inner for loop's `top` variable start at 0 and increases by `faceImHeight`(215) ❸. These nested for loops produce values for `left` and `top` to paste a grid of `faceIm` images over the `catCopyTwo` `Image` object, as in Figure 17-6. To see our nested loops working, we print `left` and `top`. After the pasting is complete, we save the modified `catCopyTwo` to `tiled.png`.



Figure 17-6: Nested for loops used with `paste()` to duplicate the cat's face (a dupli-cat, if you will).

PASTING TRANSPARENT PIXELS

Normally transparent pixels are pasted as white pixels. If the image you want to paste has transparent pixels, pass the `Image` object as the third argument so that a solid rectangle isn't pasted. This third argument is the "mask" `Image` object. A mask is an `Image` object where the alpha value is significant, but the red, green, and blue values are ignored. The mask tells the `paste()` function which pixels it should copy and which it should leave transparent. Advanced usage of masks is beyond this book, but if you want to paste an image that has transparent pixels, pass the `Image` object again as the third argument.

Resizing an Image

The `resize()` method is called on an `Image` object and returns a new `Image` object of the specified width and height. It accepts a two-integer tuple argument, representing the new width and height of the returned image. Enter the following into the interactive shell:

```
❶ >>> width, height = catIm.size
❷ >>> quartersizedIm = catIm.resize((int(width / 2), int(height / 2)))
       >>> quartersizedIm.save('quartersized.png')
❸ >>> svelteIm = catIm.resize((width, height + 300))
       >>> svelteIm.save('svelte.png')
```

Here we assign the two values in the `catIm.size` tuple to the variables `width` and `height` ❶. Using `width` and `height` instead of `catIm.size[0]` and `catIm.size[1]` makes the rest of the code more readable.

The first `resize()` call passes `int(width / 2)` for the new width and `int(height / 2)` for the new height ❷, so the `Image` object returned from `resize()` will be half the length and width of the original image, or one-quarter of the original image size overall. The `resize()` method accepts only integers in its tuple argument, which is why you needed to wrap both divisions by 2 in an `int()` call.

This resizing keeps the same proportions for the width and height. But the new width and height passed to `resize()` do not have to be proportional to the original image. The `svelteIm` variable contains an `Image` object that has the original width but a height that is 300 pixels taller ❸, giving Zophie a more slender look.

Note that the `resize()` method does not edit the `Image` object in place but instead returns a new `Image` object.

Rotating and Flipping Images

Images can be rotated with the `rotate()` method, which returns a new `Image` object of the rotated image and leaves the original `Image` object unchanged. The argument to `rotate()` is a single integer or float representing the number of degrees to rotate the image counterclockwise. Enter the following into the interactive shell:

```
>>> catIm.rotate(90).save('rotated90.png')
>>> catIm.rotate(180).save('rotated180.png')
>>> catIm.rotate(270).save('rotated270.png')
```

Note how you can *chain* method calls by calling `save()` directly on the `Image` object returned from `rotate()`. The first `rotate()` and `save()` call makes a new `Image` object representing the image rotated counterclockwise by 90 degrees and saves the rotated image to `rotated90.png`. The second and third calls do the same, but with 180 degrees and 270 degrees. The results look like Figure 17-7.



Figure 17-7: The original image (left) and the image rotated counterclockwise by 90, 180, and 270 degrees

Notice that the width and height of the image change when the image is rotated 90 or 270 degrees. If you rotate an image by some other amount, the original dimensions of the image are maintained. On Windows, a black background is used to fill in any gaps made by the rotation, like in Figure 17-8. On OS X, transparent pixels are used for the gaps instead.

The `rotate()` method has an optional `expand` keyword argument that can be set to `True` to enlarge the dimensions of the image to fit the entire rotated new image. For example, enter the following into the interactive shell:

```
>>> catIm.rotate(6).save('rotated6.png')
>>> catIm.rotate(6, expand=True).save('rotated6_expanded.png')
```

The first call rotates the image 6 degrees and saves it to `rotate6.png` (see the image on the left of Figure 17-8). The second call rotates the image 6 degrees with `expand` set to `True` and saves it to `rotate6_expanded.png` (see the image on the right of Figure 17-8).

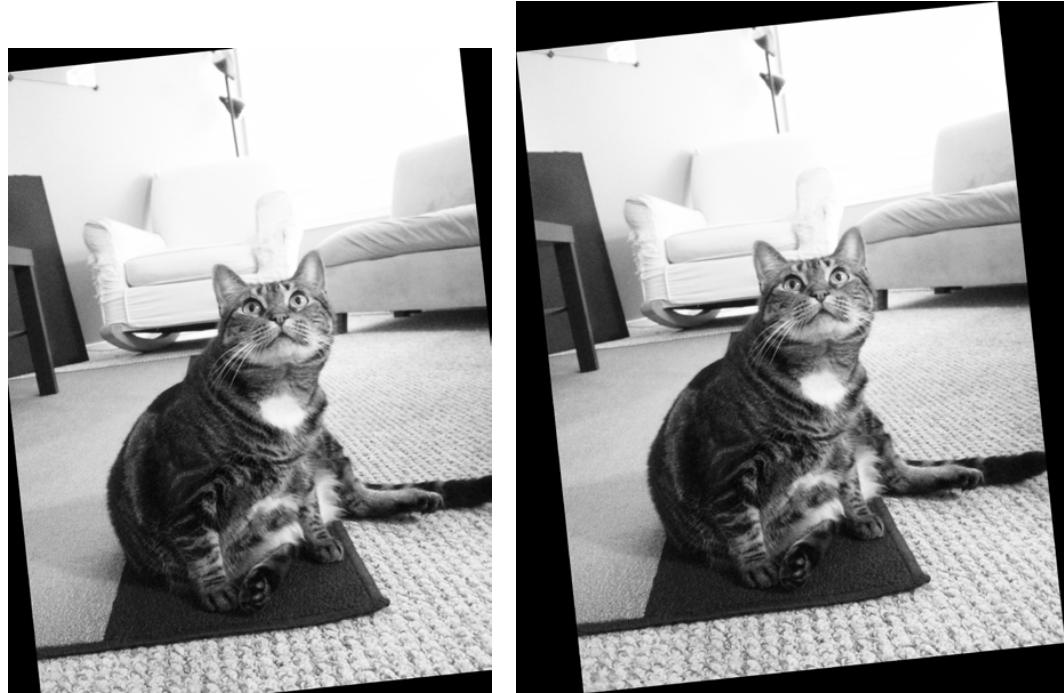


Figure 17-8: The image rotated 6 degrees normally (left) and with expand=True (right)

You can also get a “mirror flip” of an image with the `transpose()` method. You must pass either `Image.FLIP_LEFT_RIGHT` or `Image.FLIP_TOP_BOTTOM` to the `transpose()` method. Enter the following into the interactive shell:

```
>>> catIm.transpose(Image.FLIP_LEFT_RIGHT).save('horizontal_flip.png')
>>> catIm.transpose(Image.FLIP_TOP_BOTTOM).save('vertical_flip.png')
```

Like `rotate()`, `transpose()` creates a new `Image` object. Here we pass `Image.FLIP_LEFT_RIGHT` to flip the image horizontally and then save the result to `horizontal_flip.png`. To flip the image vertically, we pass `Image.FLIP_TOP_BOTTOM` and save to `vertical_flip.png`. The results look like Figure 17-9.

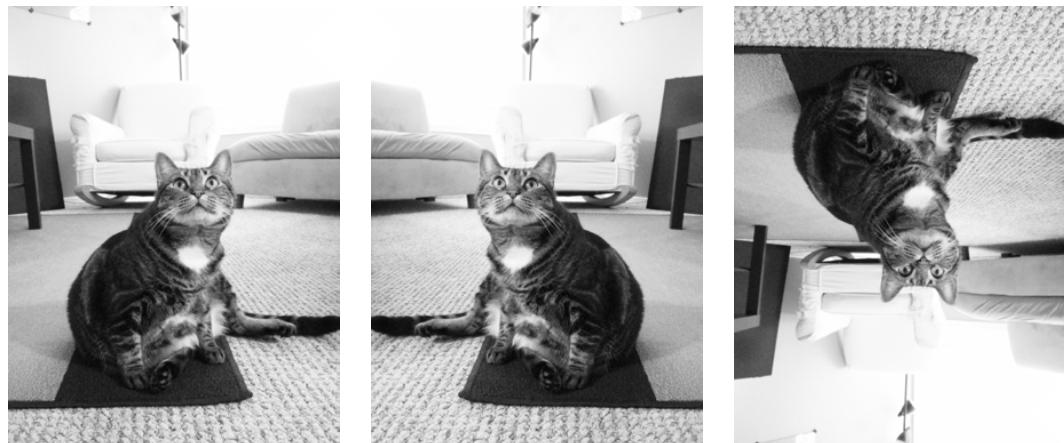


Figure 17-9: The original image (left), horizontal flip (center), and vertical flip (right)

Changing Individual Pixels

The color of an individual pixel can be retrieved or set with the `getpixel()` and `putpixel()` methods. These methods both take a tuple representing the x- and y-coordinates of the pixel. The `putpixel()` method also takes an additional tuple argument for the color of the pixel. This color argument is a four-integer RGBA tuple or a three-integer RGB tuple. Enter the following into the interactive shell:

```
❶ >>> im = Image.new('RGBA', (100, 100))
❷ >>> im.getpixel((0, 0))
(0, 0, 0, 0)
❸ >>> for x in range(100):
    for y in range(50):
        ❹         im.putpixel((x, y), (210, 210, 210))

    >>> from PIL import ImageColor
❺ >>> for x in range(100):
    for y in range(50, 100):
        ❻             im.putpixel((x, y), ImageColor.getcolor('darkgray', 'RGBA'))
    >>> im.getpixel((0, 0))
(210, 210, 210, 255)
    >>> im.getpixel((0, 50))
(169, 169, 169, 255)
    >>> im.save('putPixel.png')
```

At ❶ we make a new image that is a 100×100 transparent square. Calling `getpixel()` on some coordinates in this image returns `(0, 0, 0, 0)` because the image is transparent ❷. To color pixels in this image, we can use nested `for` loops to go through all the pixels in the top half of the image ❸ and color each pixel using `putpixel()` ❹. Here we pass `putpixel()` the RGB tuple `(210, 210, 210)`, a light gray.

Say we want to color the bottom half of the image dark gray but don't know the RGB tuple for dark gray. The `putpixel()` method doesn't accept a standard color name like `'darkgray'`, so you have to use `ImageColor.getcolor()` to get a color tuple from `'darkgray'`. Loop through the pixels in the bottom half of the image ❺ and pass `putpixel()` the return value of `ImageColor.getcolor()` ❻, and you should now have an image that is light gray in its top half and dark gray in the bottom half, as shown in Figure 17-10. You can call `getpixel()` on some coordinates to confirm that the color at any given pixel is what you expect. Finally, save the image to `putPixel.png`.

Of course, drawing one pixel at a time onto an image isn't very convenient. If you need to draw shapes, use the `ImageDraw` functions explained later in this chapter.



Figure 17-10: The `putPixel.png` image

Project: Adding a Logo

Say you have the boring job of resizing thousands of images and adding a small logo watermark to the corner of each. Doing this with a basic graphics program such as Paintbrush or Paint would take forever. A fancier graphics application such as Photoshop can do batch processing, but that software costs hundreds of dollars. Let's write a script to do it instead.

Say that Figure 17-11 is the logo you want to add to the bottom-right corner of each image: a black cat icon with a white border, with the rest of the image transparent.

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

- Load the logo image.
- Loop over all *.png* and *.jpg* files in the working directory.
- Check whether the image is wider or taller than 300 pixels.
- If so, reduce the width or height (whichever is larger) to 300 pixels and scale down the other dimension proportionally.
- Paste the logo image into the corner.
- Save the altered images to another folder.

This means the code will need to do the following:

- Open the *catlogo.png* file as an `Image` object.
- Loop over the strings returned from `os.listdir('.')`.
- Get the width and height of the image from the `size` attribute.
- Calculate the new width and height of the resized image.
- Call the `resize()` method to resize the image.
- Call the `paste()` method to paste the logo.
- Call the `save()` method to save the changes, using the original filename.

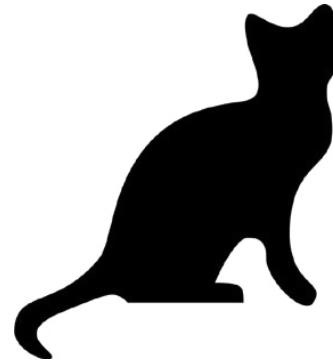


Figure 17-11: The logo to be added to the image.

Step 1: Open the Logo Image

For this project, open a new file editor window, enter the following code, and save it as *resizeAndAddLogo.py*:

```
#! python3
# resizeAndAddLogo.py - Resizes all images in current working directory to fit
# in a 300x300 square, and adds catlogo.png to the lower-right corner.
```

```
import os
from PIL import Image

❶ SQUARE_FIT_SIZE = 300
❷ LOGO_FILENAME = 'catlogo.png'

❸ logoIm = Image.open(LOGO_FILENAME)
❹ logoWidth, logoHeight = logoIm.size

# TODO: Loop over all files in the working directory.

# TODO: Check if image needs to be resized.

# TODO: Calculate the new width and height to resize to.

# TODO: Resize the image.

# TODO: Add the logo.

# TODO: Save changes.
```

By setting up the `SQUARE_FIT_SIZE` ❶ and `LOGO_FILENAME` ❷ constants at the start of the program, we've made it easy to change the program later. Say the logo that you're adding isn't the cat icon, or say you're reducing the output images' largest dimension to something other than 300 pixels. With these constants at the start of the program, you can just open the code, change those values once, and you're done. (Or you can make it so that the values for these constants are taken from the command line arguments.) Without these constants, you'd instead have to search the code for all instances of 300 and 'catlogo.png' and replace them with the values for your new project. In short, using constants makes your program more generalized.

The logo `Image` object is returned from `Image.open()` ❸. For readability, `logoWidth` and `logoHeight` are assigned to the values from `logoIm.size` ❹.

The rest of the program is a skeleton of TODO comments for now.

Step 2: Loop Over All Files and Open Images

Now you need to find every `.png` file and `.jpg` file in the current working directory. Note that you don't want to add the logo image to the logo image itself, so the program should skip any image with a filename that's the same as `LOGO_FILENAME`. Add the following to your code:

```
#! python3
# resizeAndAddLogo.py - Resizes all images in current working directory to fit
# in a 300x300 square, and adds catlogo.png to the lower-right corner.

import os
from PIL import Image

--snip--
```

```
os.makedirs('withLogo', exist_ok=True)
# Loop over all files in the working directory.
❶ for filename in os.listdir('.'):
❷     if not (filename.endswith('.png') or filename.endswith('.jpg')) \
        or filename == LOGO_FILENAME:
❸     continue    # skip non-image files and the logo file itself

❹     im = Image.open(filename)
width, height = im.size
```

--snip--

First, the `os.makedirs()` call creates a `withLogo` folder to store the finished images with logos, instead of overwriting the original image files. The `exist_ok=True` keyword argument will keep `os.makedirs()` from raising an exception if `withLogo` already exists. While looping through all the files in the working directory with `os.listdir('.')` ❶, the long `if` statement ❷ checks whether each filename doesn't end with `.png` or `.jpg`. If so—or if the file is the logo image itself—then the loop should skip it and use `continue` ❸ to go to the next file. If `filename` *does* end with `'.png'` or `'.jpg'` (and isn't the logo file), you can open it as an `Image` object ❹ and set `width` and `height`.

Step 3: Resize the Images

The program should resize the image only if the width or height is larger than `SQUARE_FIT_SIZE` (300 pixels, in this case), so put all of the resizing code inside an `if` statement that checks the `width` and `height` variables. Add the following code to your program:

```
#! python3
# resizeAndAddLogo.py - Resizes all images in current working directory to fit
# in a 300x300 square, and adds catlogo.png to the lower-right corner.

import os
from PIL import Image

--snip--

# Check if image needs to be resized.
if width > SQUARE_FIT_SIZE and height > SQUARE_FIT_SIZE:
    # Calculate the new width and height to resize to.
    if width > height:
❶        height = int((SQUARE_FIT_SIZE / width) * height)
        width = SQUARE_FIT_SIZE
    else:
❷        width = int((SQUARE_FIT_SIZE / height) * width)
        height = SQUARE_FIT_SIZE

    # Resize the image.
    print('Resizing %s...' % (filename))
❸    im = im.resize((width, height))

--snip--
```

If the image does need to be resized, you need to find out whether it is a wide or tall image. If `width` is greater than `height`, then the height should be reduced by the same proportion that the width would be reduced ❶. This proportion is the `SQUARE_FIT_SIZE` value divided by the current width. The new height value is this proportion multiplied by the current height value. Since the division operator returns a float value and `resize()` requires the dimensions to be integers, remember to convert the result to an integer with the `int()` function. Finally, the new `width` value will simply be set to `SQUARE_FIT_SIZE`.

If the height is greater than or equal to the `width` (both cases are handled in the `else` clause), then the same calculation is done, except with the `height` and `width` variables swapped ❷.

Once `width` and `height` contain the new image dimensions, pass them to the `resize()` method and store the returned `Image` object in `im` ❸.

Step 4: Add the Logo and Save the Changes

Whether or not the image was resized, the logo should still be pasted to the bottom-right corner. Where exactly the logo should be pasted depends on both the size of the image and the size of the logo. Figure 17-12 shows how to calculate the pasting position. The left coordinate for where to paste the logo will be the image width minus the logo width; the top coordinate for where to paste the logo will be the image height minus the logo height.

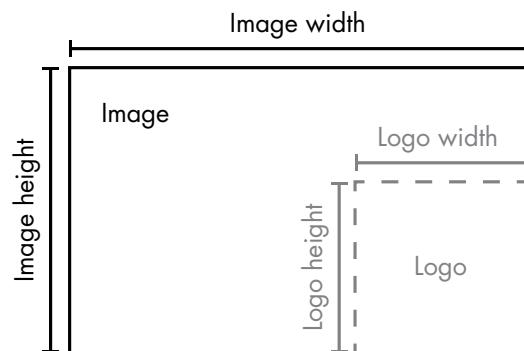


Figure 17-12: The left and top coordinates for placing the logo in the bottom-right corner should be the image width/height minus the logo width/height.

After your code pastes the logo into the image, it should save the modified `Image` object. Add the following to your program:

```
#! python3
# resizeAndAddLogo.py - Resizes all images in current working directory to fit
# in a 300x300 square, and adds catlogo.png to the lower-right corner.

import os
from PIL import Image
```

```
--snip--

# Check if image needs to be resized.
--snip--

# Add the logo.
❶ print('Adding logo to %s...' % (filename))
❷ im.paste(logoIm, (width - logoWidth, height - logoHeight), logoIm)

# Save changes.
❸ im.save(os.path.join('withLogo', filename))
```

The new code prints a message telling the user that the logo is being added ❶, pastes `logoIm` onto `im` at the calculated coordinates ❷, and saves the changes to a filename in the `withLogo` directory ❸. When you run this program with the `zophie.png` file as the only image in the working directory, the output will look like this:

```
Resizing zophie.png...
Adding logo to zophie.png...
```

The image `zophie.png` will be changed to a 225×300 -pixel image that looks like Figure 17-13. Remember that the `paste()` method will not paste the transparency pixels if you do not pass the `logoIm` for the third argument as well. This program can automatically resize and “logo-ify” hundreds of images in just a couple minutes.

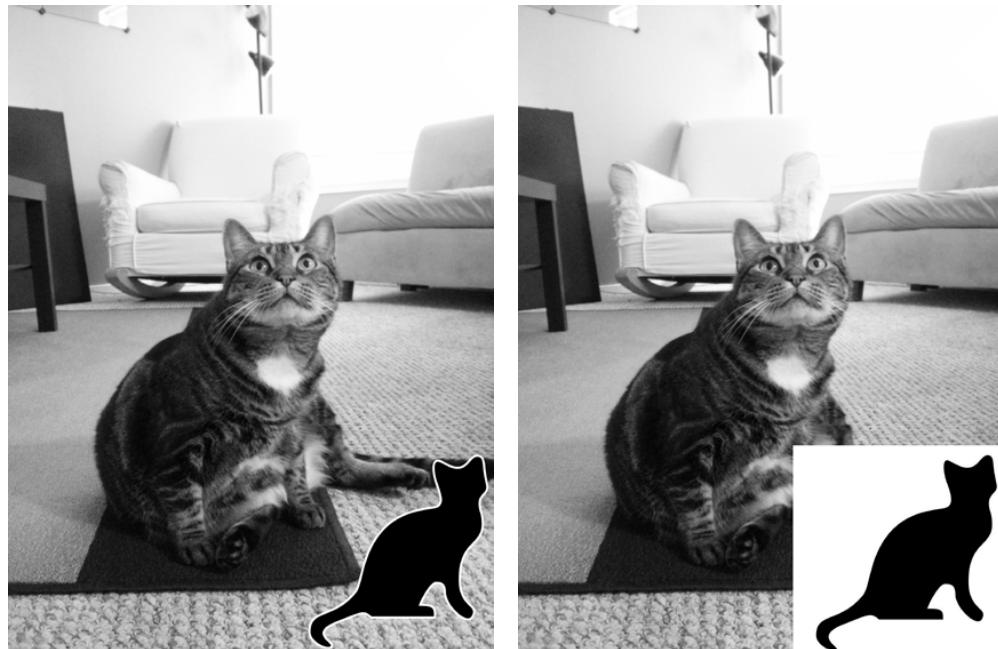


Figure 17-13: The image `zophie.png` resized and the logo added (left). If you forget the third argument, the transparent pixels in the logo will be copied as solid white pixels (right).

Ideas for Similar Programs

Being able to composite images or modify image sizes in a batch can be useful in many applications. You could write similar programs to do the following:

- Add text or a website URL to images.
- Add timestamps to images.
- Copy or move images into different folders based on their sizes.
- Add a mostly transparent watermark to an image to prevent others from copying it.

Drawing on Images

If you need to draw lines, rectangles, circles, or other simple shapes on an image, use Pillow's `ImageDraw` module. Enter the following into the interactive shell:

```
>>> from PIL import Image, ImageDraw
>>> im = Image.new('RGBA', (200, 200), 'white')
>>> draw = ImageDraw.Draw(im)
```

First, we import `Image` and `ImageDraw`. Then we create a new image, in this case, a 200×200 white image, and store the `Image` object in `im`. We pass the `Image` object to the `ImageDraw.Draw()` function to receive an `ImageDraw` object. This object has several methods for drawing shapes and text onto an `Image` object. Store the `ImageDraw` object in a variable like `draw` so you can use it easily in the following example.

Drawing Shapes

The following `ImageDraw` methods draw various kinds of shapes on the image. The `fill` and `outline` parameters for these methods are optional and will default to white if left unspecified.

Points

The `point(xy, fill)` method draws individual pixels. The `xy` argument represents a list of the points you want to draw. The list can be a list of x- and y-coordinate tuples, such as `[(x, y), (x, y), ...]`, or a list of x- and y-coordinates without tuples, such as `[x1, y1, x2, y2, ...]`. The `fill` argument is the color of the points and is either an RGBA tuple or a string of a color name, such as `'red'`. The `fill` argument is optional.

Lines

The `line(xy, fill, width)` method draws a line or series of lines. `xy` is either a list of tuples, such as `[(x, y), (x, y), ...]`, or a list of integers, such as `[x1, y1, x2, y2, ...]`. Each point is one of the connecting points on the

lines you’re drawing. The optional *fill* argument is the color of the lines, as an RGBA tuple or color name. The optional *width* argument is the width of the lines and defaults to 1 if left unspecified.

Rectangles

The `rectangle(xy, fill, outline)` method draws a rectangle. The *xy* argument is a box tuple of the form `(left, top, right, bottom)`. The *left* and *top* values specify the x- and y-coordinates of the upper-left corner of the rectangle, while *right* and *bottom* specify the lower-right corner. The optional *fill* argument is the color that will fill the inside of the rectangle. The optional *outline* argument is the color of the rectangle’s outline.

Ellipses

The `ellipse(xy, fill, outline)` method draws an ellipse. If the width and height of the ellipse are identical, this method will draw a circle. The *xy* argument is a box tuple `(left, top, right, bottom)` that represents a box that precisely contains the ellipse. The optional *fill* argument is the color of the inside of the ellipse, and the optional *outline* argument is the color of the ellipse’s outline.

Polygons

The `polygon(xy, fill, outline)` method draws an arbitrary polygon. The *xy* argument is a list of tuples, such as `[(x, y), (x, y), ...]`, or integers, such as `[x1, y1, x2, y2, ...]`, representing the connecting points of the polygon’s sides. The last pair of coordinates will be automatically connected to the first pair. The optional *fill* argument is the color of the inside of the polygon, and the optional *outline* argument is the color of the polygon’s outline.

Drawing Example

Enter the following into the interactive shell:

```
>>> from PIL import Image, ImageDraw
>>> im = Image.new('RGBA', (200, 200), 'white')
>>> draw = ImageDraw.Draw(im)
❶ >>> draw.line([(0, 0), (199, 0), (199, 199), (0, 199), (0, 0)], fill='black')
❷ >>> draw.rectangle((20, 30, 60, 60), fill='blue')
❸ >>> draw.ellipse((120, 30, 160, 60), fill='red')
❹ >>> draw.polygon(((57, 87), (79, 62), (94, 85), (120, 90), (103, 113)),
    fill='brown')
❺ >>> for i in range(100, 200, 10):
    draw.line([(i, 0), (200, i - 100)], fill='green')

>>> im.save('drawing.png')
```

After making an `Image` object for a 200×200 white image, passing it to `ImageDraw.Draw()` to get an `ImageDraw` object, and storing the `ImageDraw` object in `draw`, you can call drawing methods on `draw`. Here we make a thin, black

outline at the edges of the image ❶, a blue rectangle with its top-left corner at (20, 30) and bottom-right corner at (60, 60) ❷, a red ellipse defined by a box from (120, 30) to (160, 60) ❸, a brown polygon with five points ❹, and a pattern of green lines drawn with a for loop ❺. The resulting *drawing.png* file will look like Figure 17-14.

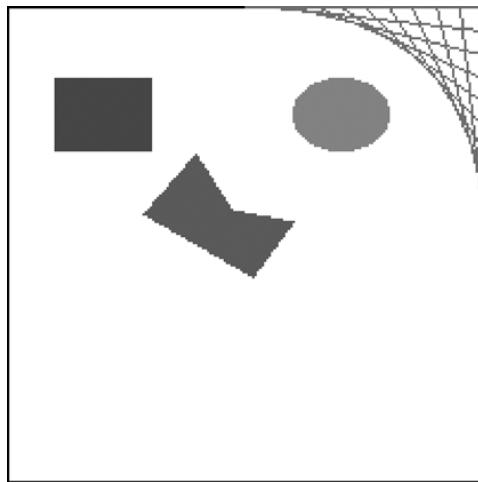


Figure 17-14: The resulting *drawing.png* image

There are several other shape-drawing methods for `ImageDraw` objects. The full documentation is available at <http://pillow.readthedocs.org/en/latest/reference/ImageDraw.html>.

Drawing Text

The `ImageDraw` object also has a `text()` method for drawing text onto an image. The `text()` method takes four arguments: `xy`, `text`, `fill`, and `font`.

- The `xy` argument is a two-integer tuple specifying the upper-left corner of the text box.
- The `text` argument is the string of text you want to write.
- The optional `fill` argument is the color of the text.
- The optional `font` argument is an `ImageFont` object, used to set the typeface and size of the text. This is described in more detail in the next section.

Since it's often hard to know in advance what size a block of text will be in a given font, the `ImageDraw` module also offers a `textsize()` method. Its first argument is the string of text you want to measure, and its second argument is an optional `ImageFont` object. The `textsize()` method will then return a two-integer tuple of the width and height that the text in the given

font would be if it were written onto the image. You can use this width and height to help you calculate exactly where you want to put the text on your image.

The first three arguments for `text()` are straightforward. Before we use `text()` to draw text onto an image, let's look at the optional fourth argument, the `ImageFont` object.

Both `text()` and `textsize()` take an optional `ImageFont` object as their final arguments. To create one of these objects, first run the following:

```
>>> from PIL import ImageFont
```

Now that you've imported Pillow's `ImageFont` module, you can call the `ImageFont.truetype()` function, which takes two arguments. The first argument is a string for the font's *TrueType file*—this is the actual font file that lives on your hard drive. A TrueType file has the `.ttf` file extension and can usually be found in the following folders:

- On Windows: `C:\Windows\Fonts`
- On OS X: `/Library/Fonts` and `/System/Library/Fonts`
- On Linux: `/usr/share/fonts/truetype`

You don't actually need to enter these paths as part of the TrueType file string because Python knows to automatically search for fonts in these directories. But Python will display an error if it is unable to find the font you specified.

The second argument to `ImageFont.truetype()` is an integer for the font size in *points* (rather than, say, pixels). Keep in mind that Pillow creates PNG images that are 72 pixels per inch by default, and a point is 1/72 of an inch.

Enter the following into the interactive shell, replacing `FONT_FOLDER` with the actual folder name your operating system uses:

```
>>> from PIL import Image, ImageDraw, ImageFont
>>> import os
❶ >>> im = Image.new('RGBA', (200, 200), 'white')
❷ >>> draw = ImageDraw.Draw(im)
❸ >>> draw.text((20, 150), 'Hello', fill='purple')
    >>> fontsFolder = 'FONT_FOLDER' # e.g. '/Library/Fonts'
❹ >>> arialFont = ImageFont.truetype(os.path.join(fontsFolder, 'arial.ttf'), 32)
❺ >>> draw.text((100, 150), 'Howdy', fill='gray', font=arialFont)
    >>> im.save('text.png')
```

After importing `Image`, `ImageDraw`, `ImageFont`, and `os`, we make an `Image` object for a new 200×200 white image ❶ and make an `ImageDraw` object from the `Image` object ❷. We use `text()` to draw *Hello* at (20, 150) in purple ❸. We didn't pass the optional fourth argument in this `text()` call, so the typeface and size of this text aren't customized.

To set a typeface and size, we first store the folder name (like `/Library/Fonts`) in `fontsFolder`. Then we call `ImageFont.truetype()`, passing it the `.ttf` file for the font we want, followed by an integer font size ❸. Store the `Font` object you get from `ImageFont.truetype()` in a variable like `arialFont`, and then pass the variable to `text()` in the final keyword argument. The `text()` call at ❹ draws *Howdy* at (100, 150) in gray in 32-point Arial.

The resulting `text.png` file will look like Figure 17-15.

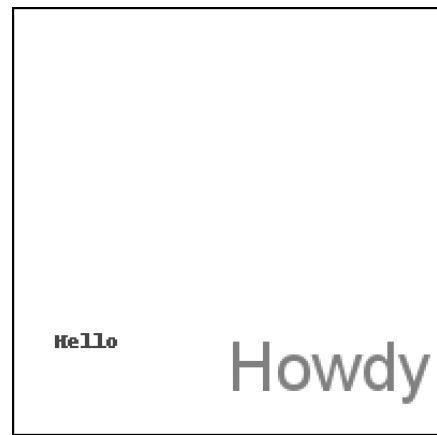


Figure 17-15: The resulting `text.png` image

Summary

Images consist of a collection of pixels, and each pixel has an RGBA value for its color and its addressable by x- and y-coordinates. Two common image formats are JPEG and PNG. The `pillow` module can handle both of these image formats and others.

When an image is loaded into an `Image` object, its width and height dimensions are stored as a two-integer tuple in the `size` attribute. Objects of the `Image` data type also have methods for common image manipulations: `crop()`, `copy()`, `paste()`, `resize()`, `rotate()`, and `transpose()`. To save the `Image` object to an image file, call the `save()` method.

If you want your program to draw shapes onto an image, use `ImageDraw` methods to draw points, lines, rectangles, ellipses, and polygons. The module also provides methods for drawing text in a typeface and font size of your choosing.

Although advanced (and expensive) applications such as Photoshop provide automatic batch processing features, you can use Python scripts to do many of the same modifications for free. In the previous chapters, you wrote Python programs to deal with plaintext files, spreadsheets, PDFs, and other formats. With the `pillow` module, you've extended your programming powers to processing images as well!

Practice Questions

1. What is an RGBA value?
2. How can you get the RGBA value of 'CornflowerBlue' from the `Pillow` module?
3. What is a box tuple?
4. What function returns an `Image` object for, say, an image file named `zophie.png`?

5. How can you find out the width and height of an `Image` object's image?
6. What method would you call to get `Image` object for a 100×100 image, excluding the lower left quarter of it?
7. After making changes to an `Image` object, how could you save it as an image file?
8. What module contains Pillow's shape-drawing code?
9. `Image` objects do not have drawing methods. What kind of object does? How do you get this kind of object?

Practice Projects

For practice, write programs that do the following.

Extending and Fixing the Chapter Project Programs

The `resizeAndAddLogo.py` program in this chapter works with PNG and JPEG files, but Pillow supports many more formats than just these two. Extend `resizeAndAddLogo.py` to process GIF and BMP images as well.

Another small issue is that the program modifies PNG and JPEG files only if their file extensions are set in lowercase. For example, it will process `zophie.png` but not `zophie.PNG`. Change the code so that the file extension check is case insensitive.

Finally, the logo added to the bottom-right corner is meant to be just a small mark, but if the image is about the same size as the logo itself, the result will look like Figure 17-16. Modify `resizeAndAddLogo.py` so that the image must be at least twice the width and height of the logo image before the logo is pasted. Otherwise, it should skip adding the logo.

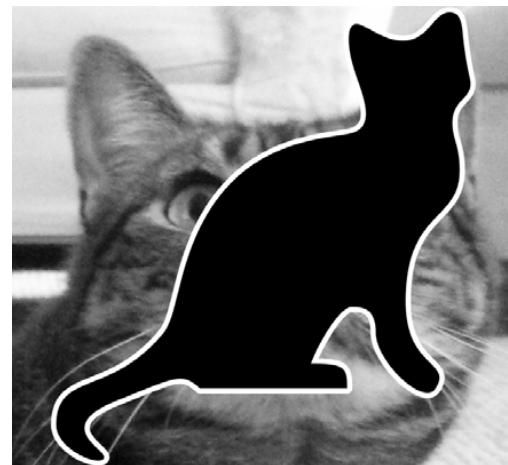


Figure 17-16: When the image isn't much larger than the logo, the results look ugly.

Identifying Photo Folders on the Hard Drive

I have a bad habit of transferring files from my digital camera to temporary folders somewhere on the hard drive and then forgetting about these folders. It would be nice to write a program that could scan the entire hard drive and find these leftover “photo folders.”

Write a program that goes through every folder on your hard drive and finds potential photo folders. Of course, first you'll have to define what you consider a “photo folder” to be; let's say that it's any folder where more than half of the files are photos. And how do you define what files are photos?

First, a photo file must have the file extension `.png` or `.jpg`. Also, photos are large images; a photo file's width and height must both be larger than 500 pixels. This is a safe bet, since most digital camera photos are several thousand pixels in width and height.

As a hint, here's a rough skeleton of what this program might look like:

```
#! python3
# Import modules and write comments to describe this program.

for foldername, subfolders, filenames in os.walk('C:\\\\'):
    numPhotoFiles = 0
    numNonPhotoFiles = 0
    for filename in filenames:
        # Check if file extension isn't .png or .jpg.
        if TODO:
            numNonPhotoFiles += 1
            continue    # skip to next filename

        # Open image file using Pillow.

        # Check if width & height are larger than 500.
        if TODO:
            # Image is large enough to be considered a photo.
            numPhotoFiles += 1
        else:
            # Image is too small to be a photo.
            numNonPhotoFiles += 1

    # If more than half of files were photos,
    # print the absolute path of the folder.
    if TODO:
        print(TODO)
```

When the program runs, it should print the absolute path of any photo folders to the screen.

Custom Seating Cards

Chapter 13 included a practice project to create custom invitations from a list of guests in a plaintext file. As an additional project, use the `Pillow` module to create images for custom seating cards for your guests. For each of the guests listed in the `guests.txt` file from the resources at <http://nostarch.com/automatestuff/>, generate an image file with the guest name and some flowery decoration. A public domain flower image is available in the resources at <http://nostarch.com/automatestuff/>.

To ensure that each seating card is the same size, add a black rectangle on the edges of the invitation image so that when the image is printed out, there will be a guideline for cutting. The PNG files that Pillow produces are set to 72 pixels per inch, so a 4×5-inch card would require a 288×360-pixel image.

18

CONTROLLING THE KEYBOARD AND MOUSE WITH GUI AUTOMATION



Knowing various Python modules for editing spreadsheets, downloading files, and launching programs is useful, but sometimes there just aren't any modules for the applications you need to work with. The ultimate tools for automating tasks on your computer are programs you write that directly control the keyboard and mouse. These programs can control other applications by sending them virtual keystrokes and mouse clicks, just as if you were sitting at your computer and interacting with the applications yourself. This technique is known as *graphical user interface automation*, or *GUI automation* for short. With GUI automation, your programs can do anything that a human user sitting at the computer can do, except spill coffee on the keyboard.

Think of GUI automation as programming a robotic arm. You can program the robotic arm to type at your keyboard and move your mouse for you. This technique is particularly useful for tasks that involve a lot of mindless clicking or filling out of forms.

The `pyautogui` module has functions for simulating mouse movements, button clicks, and scrolling the mouse wheel. This chapter covers only a subset of PyAutoGUI's features; you can find the full documentation at <http://pyautogui.readthedocs.org/>.

Installing the `pyautogui` Module

The `pyautogui` module can send virtual keypresses and mouse clicks to Windows, OS X, and Linux. Depending on which operating system you're using, you may have to install some other modules (called *dependencies*) before you can install PyAutoGUI.

- On Windows, there are no other modules to install.
- On OS X, run `sudo pip3 install pyobjc-framework-Quartz`, `sudo pip3 install pyobjc-core`, and then `sudo pip3 install pyobjc`.
- On Linux, run `sudo pip3 install python3-xlib`, `sudo apt-get install scrot`, `sudo apt-get install python3-tk`, and `sudo apt-get install python3-dev`. (Scrot is a screenshot program that PyAutoGUI uses.)

After these dependencies are installed, run `pip install pyautogui` (or `pip3` on OS X and Linux) to install PyAutoGUI.

Appendix A has complete information on installing third-party modules. To test whether PyAutoGUI has been installed correctly, run `import pyautogui` from the interactive shell and check for any error messages.

Staying on Track

Before you jump in to a GUI automation, you should know how to escape problems that may arise. Python can move your mouse and type keystrokes at an incredible speed. In fact, it might be too fast for other programs to keep up with. Also, if something goes wrong but your program keeps moving the mouse around, it will be hard to tell what exactly the program is doing or how to recover from the problem. Like the enchanted brooms from Disney's *The Sorcerer's Apprentice*, which kept filling—and then overfilling—Mickey's tub with water, your program could get out of control even though it's following your instructions perfectly. Stopping the program can be difficult if the mouse is moving around on its own, preventing you from clicking the IDLE window to close it. Fortunately, there are several ways to prevent or recover from GUI automation problems.

Shutting Down Everything by Logging Out

Perhaps the simplest way to stop an out-of-control GUI automation program is to log out, which will shut down all running programs. On Windows and Linux, the logout hotkey is CTRL-ALT-DEL. On OS X, it is ⌘-SHIFT-OPTION-Q. By logging out, you'll lose any unsaved work, but at least you won't have to wait for a full reboot of the computer.

Pauses and Fail-Safes

You can tell your script to wait after every function call, giving you a short window to take control of the mouse and keyboard if something goes wrong. To do this, set the `pyautogui.PAUSE` variable to the number of seconds you want it to pause. For example, after setting `pyautogui.PAUSE = 1.5`, every PyAutoGUI function call will wait one and a half seconds after performing its action. Non-PyAutoGUI instructions will not have this pause.

PyAutoGUI also has a fail-safe feature. Moving the mouse cursor to the upper-left corner of the screen will cause PyAutoGUI to raise the `pyautogui.FailSafeException` exception. Your program can either handle this exception with `try` and `except` statements or let the exception crash your program. Either way, the fail-safe feature will stop the program if you quickly move the mouse as far up and left as you can. You can disable this feature by setting `pyautogui.FAILSAFE = False`. Enter the following into the interactive shell:

```
>>> import pyautogui  
>>> pyautogui.PAUSE = 1  
>>> pyautogui.FAILSAFE = True
```

Here we import `pyautogui` and set `pyautogui.PAUSE` to 1 for a one-second pause after each function call. We set `pyautogui.FAILSAFE` to `True` to enable the fail-safe feature.

Controlling Mouse Movement

In this section, you'll learn how to move the mouse and track its position on the screen using PyAutoGUI, but first you need to understand how PyAutoGUI works with coordinates.

The mouse functions of PyAutoGUI use x- and y-coordinates. Figure 18-1 shows the coordinate system for the computer screen; it's similar to the coordinate system used for images, discussed in Chapter 17. The *origin*, where x and y are both zero, is at the upper-left corner of the screen. The x-coordinates increase going to the right, and the y-coordinates increase going down. All coordinates are positive integers; there are no negative coordinates.

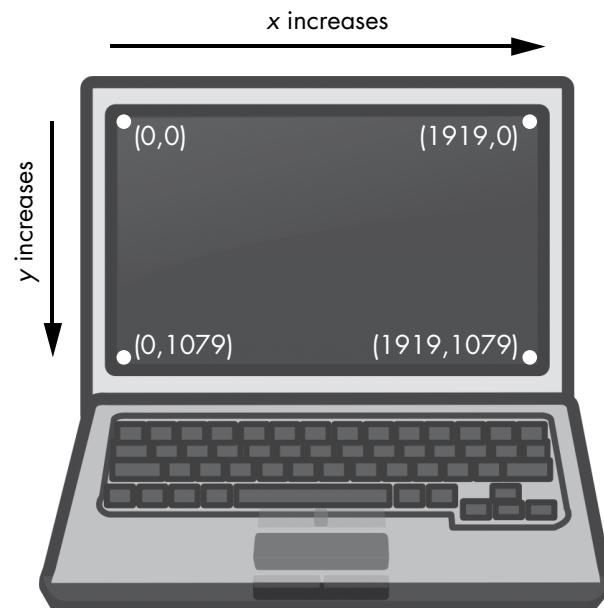


Figure 18-1: The coordinates of a computer screen with 1920x1080 resolution

Your *resolution* is how many pixels wide and tall your screen is. If your screen's resolution is set to 1920×1080, then the coordinate for the upper-left corner will be (0, 0), and the coordinate for the bottom-right corner will be (1919, 1079).

The `pyautogui.size()` function returns a two-integer tuple of the screen's width and height in pixels. Enter the following into the interactive shell:

```
>>> import pyautogui  
>>> pyautogui.size()  
(1920, 1080)  
>>> width, height = pyautogui.size()
```

`pyautogui.size()` returns (1920, 1080) on a computer with a 1920×1080 resolution; depending on your screen's resolution, your return value may be different. You can store the width and height from `pyautogui.size()` in variables like `width` and `height` for better readability in your programs.

Moving the Mouse

Now that you understand screen coordinates, let's move the mouse. The `pyautogui.moveTo()` function will instantly move the mouse cursor to a specified position on the screen. Integer values for the x- and y-coordinates make up the function's first and second arguments, respectively. An optional duration integer or float keyword argument specifies the number of seconds it should take to move the mouse to the destination. If you leave it out, the default is 0 for instantaneous movement. (All of the duration keyword arguments in PyAutoGUI functions are optional.) Enter the following into the interactive shell:

```
>>> import pyautogui  
>>> for i in range(10):  
    pyautogui.moveTo(100, 100, duration=0.25)  
    pyautogui.moveTo(200, 100, duration=0.25)  
    pyautogui.moveTo(200, 200, duration=0.25)  
    pyautogui.moveTo(100, 200, duration=0.25)
```

This example moves the mouse cursor clockwise in a square pattern among the four coordinates provided a total of ten times. Each movement takes a quarter of a second, as specified by the `duration=0.25` keyword argument. If you hadn't passed a third argument to any of the `pyautogui.moveTo()` calls, the mouse cursor would have instantly teleported from point to point.

The `pyautogui.moveRel()` function moves the mouse cursor *relative* to its current position. The following example moves the mouse in the same square pattern, except it begins the square from wherever the mouse happens to be on the screen when the code starts running:

```
>>> import pyautogui  
>>> for i in range(10):  
    pyautogui.moveRel(100, 0, duration=0.25)  
    pyautogui.moveRel(0, 100, duration=0.25)
```

```
pyautogui.moveRel(-100, 0, duration=0.25)
pyautogui.moveRel(0, -100, duration=0.25)
```

`pyautogui.moveRel()` also takes three arguments: how many pixels to move horizontally to the right, how many pixels to move vertically downward, and (optionally) how long it should take to complete the movement. A negative integer for the first or second argument will cause the mouse to move left or upward, respectively.

Getting the Mouse Position

You can determine the mouse's current position by calling the `pyautogui.position()` function, which will return a tuple of the mouse cursor's *x* and *y* positions at the time of the function call. Enter the following into the interactive shell, moving the mouse around after each call:

```
>>> pyautogui.position()
(311, 622)
>>> pyautogui.position()
(377, 481)
>>> pyautogui.position()
(1536, 637)
```

Of course, your return values will vary depending on where your mouse cursor is.

Project: “Where Is the Mouse Right Now?”

Being able to determine the mouse position is an important part of setting up your GUI automation scripts. But it's almost impossible to figure out the exact coordinates of a pixel just by looking at the screen. It would be handy to have a program that constantly displays the *x*- and *y*-coordinates of the mouse cursor as you move it around.

At a high level, here's what your program should do:

- Display the current *x*- and *y*-coordinates of the mouse cursor.
- Update these coordinates as the mouse moves around the screen.

This means your code will need to do the following:

- Call the `position()` function to fetch the current coordinates.
- Erase the previously printed coordinates by printing \b backspace characters to the screen.
- Handle the `KeyboardInterrupt` exception so the user can press CTRL-C to quit.

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

Step 1: Import the Module

Start your program with the following:

```
#! python3
# mouseNow.py - Displays the mouse cursor's current position.
import pyautogui
print('Press Ctrl-C to quit.')
#TODO: Get and print the mouse coordinates.
```

The beginning of the program imports the `pyautogui` module and prints a reminder to the user that they have to press CTRL-C to quit.

Step 2: Set Up the Quit Code and Infinite Loop

You can use an infinite `while` loop to constantly print the current mouse coordinates from `mouse.position()`. As for the code that quits the program, you'll need to catch the `KeyboardInterrupt` exception, which is raised whenever the user presses CTRL-C. If you don't handle this exception, it will display an ugly traceback and error message to the user. Add the following to your program:

```
#! python3
# mouseNow.py - Displays the mouse cursor's current position.
import pyautogui
print('Press Ctrl-C to quit.')
try:
    while True:
        # TODO: Get and print the mouse coordinates.
❶ except KeyboardInterrupt:
❷     print('\nDone.'
```

To handle the exception, enclose the infinite `while` loop in a `try` statement. When the user presses CTRL-C, the program execution will move to the `except` clause ❶ and `Done.` will be printed in a new line ❷.

Step 3: Get and Print the Mouse Coordinates

The code inside the `while` loop should get the current mouse coordinates, format them to look nice, and print them. Add the following code to the inside of the `while` loop:

```
#! python3
# mouseNow.py - Displays the mouse cursor's current position.
import pyautogui
print('Press Ctrl-C to quit.')
--snip--
    # Get and print the mouse coordinates.
    x, y = pyautogui.position()
    positionStr = 'X: ' + str(x).rjust(4) + ' Y: ' + str(y).rjust(4)
--snip--
```

Using the multiple assignment trick, the `x` and `y` variables are given the values of the two integers returned in the tuple from `pyautogui.position()`. By passing `x` and `y` to the `str()` function, you can get string forms of the integer coordinates. The `rjust()` string method will right-justify them so that they take up the same amount of space, whether the coordinate has one, two, three, or four digits. Concatenating the right-justified string coordinates with '`X:` ' and '`Y:` ' labels gives us a neatly formatted string, which will be stored in `positionStr`.

At the end of your program, add the following code:

```
#! python3
# mouseNow.py - Displays the mouse cursor's current position.
--snip--
    print(positionStr, end='')
❶     print('\b' * len(positionStr), end='', flush=True)
```

This actually prints `positionStr` to the screen. The `end=''` keyword argument to `print()` prevents the default newline character from being added to the end of the printed line. It's possible to erase text you've already printed to the screen—but only for the most recent line of text. Once you print a newline character, you can't erase anything printed before it.

To erase text, print the `\b` backspace escape character. This special character erases a character at the end of the current line on the screen. The line at ❶ uses string replication to produce a string with as many `\b` characters as the length of the string stored in `positionStr`, which has the effect of erasing the `positionStr` string that was last printed.

For a technical reason beyond the scope of this book, always pass `flush=True` to `print()` calls that print `\b` backspace characters. Otherwise, the screen might not update the text as desired.

Since the `while` loop repeats so quickly, the user won't actually notice that you're deleting and reprinting the whole number on the screen. For example, if the x-coordinate is 563 and the mouse moves one pixel to the right, it will look like only the 3 in 563 is changed to a 4.

When you run the program, there will be only two lines printed. They should look like something like this:

```
Press Ctrl-C to quit.
X: 290 Y: 424
```

The first line displays the instruction to press CTRL-C to quit. The second line with the mouse coordinates will change as you move the mouse around the screen. Using this program, you'll be able to figure out the mouse coordinates for your GUI automation scripts.

Controlling Mouse Interaction

Now that you know how to move the mouse and figure out where it is on the screen, you're ready to start clicking, dragging, and scrolling.

Clicking the Mouse

To send a virtual mouse click to your computer, call the `pyautogui.click()` method. By default, this click uses the left mouse button and takes place wherever the mouse cursor is currently located. You can pass x- and y-coordinates of the click as optional first and second arguments if you want it to take place somewhere other than the mouse's current position.

If you want to specify which mouse button to use, include the `button` keyword argument, with a value of `'left'`, `'middle'`, or `'right'`. For example, `pyautogui.click(100, 150, button='left')` will click the left mouse button at the coordinates `(100, 150)`, while `pyautogui.click(200, 250, button='right')` will perform a right-click at `(200, 250)`.

Enter the following into the interactive shell:

```
>>> import pyautogui  
>>> pyautogui.click(10, 5)
```

You should see the mouse pointer move to near the top-left corner of your screen and click once. A full “click” is defined as pushing a mouse button down and then releasing it back up without moving the cursor. You can also perform a click by calling `pyautogui.mouseDown()`, which only pushes the mouse button down, and `pyautogui.mouseUp()`, which only releases the button. These functions have the same arguments as `click()`, and in fact, the `click()` function is just a convenient wrapper around these two function calls.

As a further convenience, the `pyautogui.doubleClick()` function will perform two clicks with the left mouse button, while the `pyautogui.rightClick()` and `pyautogui.middleClick()` functions will perform a click with the right and middle mouse buttons, respectively.

Dragging the Mouse

Dragging means moving the mouse while holding down one of the mouse buttons. For example, you can move files between folders by dragging the folder icons, or you can move appointments around in a calendar app.

PyAutoGUI provides the `pyautogui.dragTo()` and `pyautogui.dragRel()` functions to drag the mouse cursor to a new location or a location relative to its current one. The arguments for `dragTo()` and `dragRel()` are the same as `moveTo()` and `moveRel()`: the x-coordinate/horizontal movement, the y-coordinate/vertical movement, and an optional duration of time. (OS X does not drag correctly when the mouse moves too quickly, so passing a `duration` keyword argument is recommended.)

To try these functions, open a graphics-drawing application such as Paint on Windows, Paintbrush on OS X, or GNU Paint on Linux. (If you don't have a drawing application, you can use the online one at <http://sumopaint.com/>.) I will use PyAutoGUI to draw in these applications.

With the mouse cursor over the drawing application’s canvas and the Pencil or Brush tool selected, enter the following into a new file editor window and save it as *spiralDraw.py*:

```
import pyautogui, time
❶ time.sleep(5)
❷ pyautogui.click()      # click to put drawing program in focus
distance = 200
while distance > 0:
    ❸   pyautogui.dragRel(distance, 0, duration=0.2)    # move right
    ❹   distance = distance - 5
    ❺   pyautogui.dragRel(0, distance, duration=0.2)    # move down
    ❻   pyautogui.dragRel(-distance, 0, duration=0.2)    # move left
        distance = distance - 5
    ❼   pyautogui.dragRel(0, -distance, duration=0.2)    # move up
```

When you run this program, there will be a five-second delay ❶ for you to move the mouse cursor over the drawing program’s window with the Pencil or Brush tool selected. Then *spiralDraw.py* will take control of the mouse and click to put the drawing program in focus ❷. A window is in *focus* when it has an active blinking cursor, and the actions you take—like typing or, in this case, dragging the mouse—will affect that window. Once the drawing program is in focus, *spiralDraw.py* draws a square spiral pattern like the one in Figure 18-2.

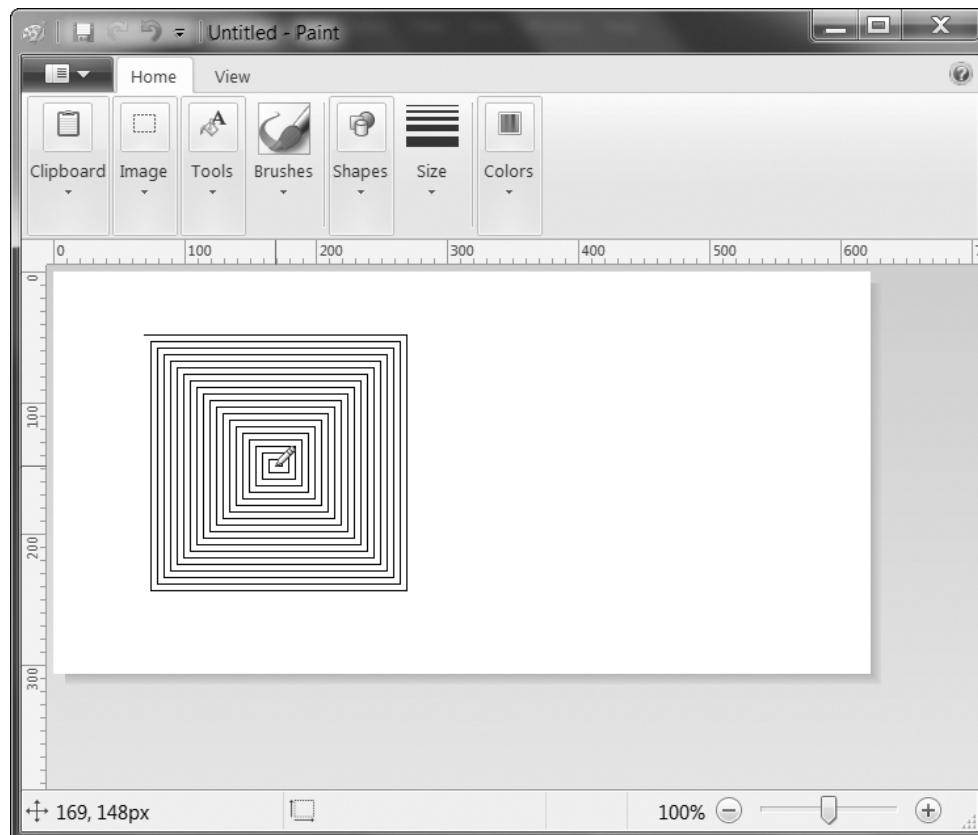


Figure 18-2: The results from the `pyautogui.dragRel()` example

The `distance` variable starts at 200, so on the first iteration of the `while` loop, the first `dragRel()` call drags the cursor 200 pixels to the right, taking 0.2 seconds ❸. `distance` is then decreased to 195 ❹, and the second `dragRel()` call drags the cursor 195 pixels down ❺. The third `dragRel()` call drags the cursor -195 horizontally (195 to the left) ❻, `distance` is decreased to 190, and the last `dragRel()` call drags the cursor 190 pixels up. On each iteration, the mouse is dragged right, down, left, and up, and `distance` is slightly smaller than it was in the previous iteration. By looping over this code, you can move the mouse cursor to draw a square spiral.

You could draw this spiral by hand (or rather, by mouse), but you'd have to work slowly to be so precise. PyAutoGUI can do it in a few seconds!

NOTE

You could have your code draw the image using the `pillow` module's drawing functions—see Chapter 17 for more information. But using GUI automation allows you to make use of the advanced drawing tools that graphics programs can provide, such as gradients, different brushes, or the fill bucket.

Scrolling the Mouse

The final PyAutoGUI mouse function is `scroll()`, which you pass an integer argument for how many units you want to scroll the mouse up or down. The size of a unit varies for each operating system and application, so you'll have to experiment to see exactly how far it scrolls in your particular situation. The scrolling takes place at the mouse cursor's current position. Passing a positive integer scrolls up, and passing a negative integer scrolls down. Run the following in IDLE's interactive shell while the mouse cursor is over the IDLE window:

```
>>> pyautogui.scroll(200)
```

You'll see IDLE briefly scroll upward—and then go back down. The downward scrolling happens because IDLE automatically scrolls down to the bottom after executing an instruction. Enter this code instead:

```
>>> import pyperclip
>>> numbers = ''
>>> for i in range(200):
...     numbers = numbers + str(i) + '\n'
>>> pyperclip.copy(numbers)
```

This imports `pyperclip` and sets up an empty string, `numbers`. The code then loops through 200 numbers and adds each number to `numbers`, along with a newline. After `pyperclip.copy(numbers)`, the clipboard will be loaded with 200 lines of numbers. Open a new file editor window and paste the text into it. This will give you a large text window to try scrolling in. Enter the following code into the interactive shell:

```
>>> import time, pyautogui
>>> time.sleep(5); pyautogui.scroll(100)
```

On the second line, you enter two commands separated by a semicolon, which tells Python to run the commands as if they were on separate lines. The only difference is that the interactive shell won't prompt you for input between the two instructions. This is important for this example because we want the call to `pyautogui.scroll()` to happen automatically after the wait. (Note that while putting two commands on one line can be useful in the interactive shell, you should still have each instruction on a separate line in your programs.)

After pressing ENTER to run the code, you will have five seconds to click the file editor window to put it in focus. Once the pause is over, the `pyautogui.scroll()` call will cause the file editor window to scroll up after the five-second delay.

Working with the Screen

Your GUI automation programs don't have to click and type blindly. PyAutoGUI has screenshot features that can create an image file based on the current contents of the screen. These functions can also return a Pillow `Image` object of the current screen's appearance. If you've been skipping around in this book, you'll want to read Chapter 17 and install the `pillow` module before continuing with this section.

On Linux computers, the `scrot` program needs to be installed to use the screenshot functions in PyAutoGUI. In a Terminal window, run `sudo apt-get install scrot` to install this program. If you're on Windows or OS X, skip this step and continue with the section.

Getting a Screenshot

To take screenshots in Python, call the `pyautogui.screenshot()` function. Enter the following into the interactive shell:

```
>>> import pyautogui  
>>> im = pyautogui.screenshot()
```

The `im` variable will contain the `Image` object of the screenshot. You can now call methods on the `Image` object in the `im` variable, just like any other `Image` object. Enter the following into the interactive shell:

```
>>> im.getpixel((0, 0))  
(176, 176, 175)  
>>> im.getpixel((50, 200))  
(130, 135, 144)
```

Pass `getpixel()` a tuple of coordinates, like `(0, 0)` or `(50, 200)`, and it'll tell you the color of the pixel at those coordinates in your image. The return value from `getpixel()` is an RGB tuple of three integers for the amount of red, green, and blue in the pixel. (There is no fourth value for alpha, because screenshot images are fully opaque.) This is how your programs can "see" what is currently on the screen.

Analyzing the Screenshot

Say that one of the steps in your GUI automation program is to click a gray button. Before calling the `click()` method, you could take a screenshot and look at the pixel where the script is about to click. If it's not the same gray as the gray button, then your program knows something is wrong. Maybe the window moved unexpectedly, or maybe a pop-up dialog has blocked the button. At this point, instead of continuing—and possibly wreaking havoc by clicking the wrong thing—your program can “see” that it isn’t clicking on the right thing and stop itself.

PyAutoGUI’s `pixelMatchesColor()` function will return `True` if the pixel at the given x- and y-coordinates on the screen matches the given color. The first and second arguments are integers for the x- and y-coordinates, and the third argument is a tuple of three integers for the RGB color the screen pixel must match. Enter the following into the interactive shell:

```
>>> import pyautogui
>>> im = pyautogui.screenshot()
❶ >>> im.getpixel((50, 200))
(130, 135, 144)
❷ >>> pyautogui.pixelMatchesColor(50, 200, (130, 135, 144))
True
❸ >>> pyautogui.pixelMatchesColor(50, 200, (255, 135, 144))
False
```

After taking a screenshot and using `getpixel()` to get an RGB tuple for the color of a pixel at specific coordinates ❶, pass the same coordinates and RGB tuple to `pixelMatchesColor()` ❷, which should return `True`. Then change a value in the RGB tuple and call `pixelMatchesColor()` again for the same coordinates ❸. This should return `false`. This method can be useful to call whenever your GUI automation programs are about to call `click()`. Note that the color at the given coordinates must *exactly* match. If it is even slightly different—for example, `(255, 255, 254)` instead of `(255, 255, 255)`—then `pixelMatchesColor()` will return `False`.

Project: Extending the mouseNow Program

You could extend the `mouseNow.py` project from earlier in this chapter so that it not only gives the x- and y-coordinates of the mouse cursor’s current position but also gives the RGB color of the pixel under the cursor. Modify the code inside the `while` loop of `mouseNow.py` to look like this:

```
#! python3
# mouseNow.py - Displays the mouse cursor's current position.
--snip--
    positionStr = 'X: ' + str(x).rjust(4) + ' Y: ' + str(y).rjust(4)
    pixelColor = pyautogui.screenshot().getpixel((x, y))
    positionStr += ' RGB: (' + str(pixelColor[0]).rjust(3)
    positionStr += ', ' + str(pixelColor[1]).rjust(3)
    positionStr += ', ' + str(pixelColor[2]).rjust(3) + ')'
```

```
    print(positionStr, end=' ')
--snip--
```

Now, when you run *mouseNow.py*, the output will include the RGB color value of the pixel under the mouse cursor.

```
Press Ctrl-C to quit.
X: 406 Y: 17 RGB: (161, 50, 50)
```

This information, along with the `pixelMatchesColor()` function, should make it easy to add pixel color checks to your GUI automation scripts.

Image Recognition

But what if you do not know beforehand where PyAutoGUI should click? You can use image recognition instead. Give PyAutoGUI an image of what you want to click and let it figure out the coordinates.

For example, if you have previously taken a screenshot to capture the image of a Submit button in *submit.png*, the `locateOnScreen()` function will return the coordinates where that image is found. To see how `locateOnScreen()` works, try taking a screenshot of a small area on your screen; then save the image and enter the following into the interactive shell, replacing '*submit.png*' with the filename of your screenshot:

```
>>> import pyautogui
>>> pyautogui.locateOnScreen('submit.png')
(643, 745, 70, 29)
```

The four-integer tuple that `locateOnScreen()` returns has the x-coordinate of the left edge, the y-coordinate of the top edge, the width, and the height for the first place on the screen the image was found. If you're trying this on your computer with your own screenshot, your return value will be different from the one shown here.

If the image cannot be found on the screen, `locateOnScreen()` will return `None`. Note that the image on the screen must match the provided image perfectly in order to be recognized. If the image is even a pixel off, `locateOnScreen()` will return `None`.

If the image can be found in several places on the screen, `locateAllOnScreen()` will return a `Generator` object, which can be passed to `list()` to return a list of four-integer tuples. There will be one four-integer tuple for each location where the image is found on the screen. Continue the interactive shell example by entering the following (and replacing '*submit.png*' with your own image filename):

```
>>> list(pyautogui.locateAllOnScreen('submit.png'))
[(643, 745, 70, 29), (1007, 801, 70, 29)]
```

Each of the four-integer tuples represents an area on the screen. If your image is only found in one area, then using `list()` and `locateAllOnScreen()` just returns a list containing one tuple.

Once you have the four-integer tuple for the area on the screen where your image was found, you can click the center of this area by passing the tuple to the `center()` function to return x- and y-coordinates of the area's center. Enter the following into the interactive shell, replacing the arguments with your own filename, four-integer tuple, and coordinate pair:

```
>>> pyautogui.locateOnScreen('submit.png')
(643, 745, 70, 29)
>>> pyautogui.center((643, 745, 70, 29))
(678, 759)
>>> pyautogui.click((678, 759))
```

Once you have center coordinates from `center()`, passing the coordinates to `click()` should click the center of the area on the screen that matches the image you passed to `locateOnScreen()`.

Controlling the Keyboard

PyAutoGUI also has functions for sending virtual keypresses to your computer, which enables you to fill out forms or enter text into applications.

Sending a String from the Keyboard

The `pyautogui.typewrite()` function sends virtual keypresses to the computer. What these keypresses do depends on what window and text field have focus. You may want to first send a mouse click to the text field you want in order to ensure that it has focus.

As a simple example, let's use Python to automatically type the words *Hello world!* into a file editor window. First, open a new file editor window and position it in the upper-left corner of your screen so that PyAutoGUI will click in the right place to bring it into focus. Next, enter the following into the interactive shell:

```
>>> pyautogui.click(100, 100); pyautogui.typewrite('Hello world!')
```

Notice how placing two commands on the same line, separated by a semicolon, keeps the interactive shell from prompting you for input between running the two instructions. This prevents you from accidentally bringing a new window into focus between the `click()` and `typewrite()` calls, which would mess up the example.

Python will first send a virtual mouse click to the coordinates (100, 100), which should click the file editor window and put it in focus. The `typewrite()` call will send the text *Hello world!* to the window, making it look like Figure 18-3. You now have code that can type for you!

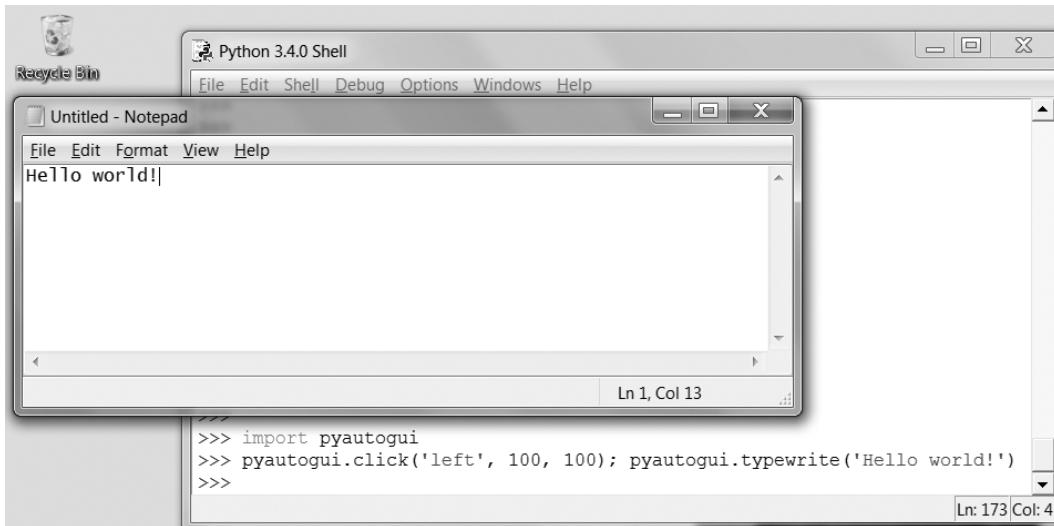


Figure 18-3: Using PyAutoGUI to click the file editor window and type Hello world! into it

By default, the `typewrite()` function will type the full string instantly. However, you can pass an optional second argument to add a short pause between each character. This second argument is an integer or float value of the number of seconds to pause. For example, `pyautogui.typewrite('Hello world!', 0.25)` will wait a quarter-second after typing *H*, another quarter-second after *e*, and so on. This gradual typewriter effect may be useful for slower applications that can't process keystrokes fast enough to keep up with PyAutoGUI.

For characters such as *A* or *!*, PyAutoGUI will automatically simulate holding down the SHIFT key as well.

Key Names

Not all keys are easy to represent with single text characters. For example, how do you represent SHIFT or the left arrow key as a single character? In PyAutoGUI, these keyboard keys are represented by short string values instead: 'esc' for the ESC key or 'enter' for the ENTER key.

Instead of a single string argument, a list of these keyboard key strings can be passed to `typewrite()`. For example, the following call presses the A key, then the B key, then the left arrow key twice, and finally the X and Y keys:

```
>>> pyautogui.typewrite(['a', 'b', 'left', 'left', 'X', 'Y'])
```

Because pressing the left arrow key moves the keyboard cursor, this will output XYab. Table 18-1 lists the PyAutoGUI keyboard key strings that you can pass to `typewrite()` to simulate pressing any combination of keys.

You can also examine the `pyautogui.KEYBOARD_KEYS` list to see all possible keyboard key strings that PyAutoGUI will accept. The 'shift' string refers to the left SHIFT key and is equivalent to 'shiftleft'. The same applies for 'ctrl', 'alt', and 'win' strings; they all refer to the left-side key.

Table 18-1: PyKeyboard Attributes

| Keyboard key string | Meaning |
|--|--|
| 'a', 'b', 'c', 'A', 'B', 'C', '1', '2', '3', '! ', '@ ', '# ', and so on | The keys for single characters |
| 'enter' (or 'return' or '\n') | The ENTER key |
| 'esc' | The ESC key |
| 'shiftleft', 'shiftright' | The left and right SHIFT keys |
| 'altright', 'altright' | The left and right ALT keys |
| 'ctrlleft', 'ctrlright' | The left and right CTRL keys |
| 'tab' (or '\t') | The TAB key |
| 'backspace', 'delete' | The BACKSPACE and DELETE keys |
| 'pageup', 'pagedown' | The PAGE UP and PAGE DOWN keys |
| 'home', 'end' | The HOME and END keys |
| 'up', 'down', 'left', 'right' | The up, down, left, and right arrow keys |
| 'f1', 'f2', 'f3', and so on | The F1 to F12 keys |
| 'volumemute', 'volumedown', 'volumeup' | The mute, volume down, and volume up keys (some keyboards do not have these keys, but your operating system will still be able to understand these simulated keypresses) |
| 'pause' | The PAUSE key |
| 'capslock', 'numlock', 'scrolllock' | The CAPS LOCK, NUM LOCK, and SCROLL LOCK keys |
| 'insert' | The INS or INSERT key |
| 'printscreen' | The PRTSC or PRINT SCREEN key |
| 'winleft', 'winright' | The left and right WIN keys (on Windows) |
| 'command' | The Command (⌘) key (on OS X) |
| 'option' | The OPTION key (on OS X) |

Pressing and Releasing the Keyboard

Much like the `mouseDown()` and `mouseUp()` functions, `pyautogui.keyDown()` and `pyautogui.keyUp()` will send virtual keypresses and releases to the computer. They are passed a keyboard key string (see Table 18-1) for their argument. For convenience, PyAutoGUI provides the `pyautogui.press()` function, which calls both of these functions to simulate a complete keypress.

Run the following code, which will type a dollar sign character (obtained by holding the SHIFT key and pressing 4):

```
>>> pyautogui.keyDown('shift'); pyautogui.press('4'); pyautogui.keyUp('shift')
```

This line presses down SHIFT, presses (and releases) 4, and then releases SHIFT. If you need to type a string into a text field, the `typewrite()` function is more suitable. But for applications that take single-key commands, the `press()` function is the simpler approach.

Hotkey Combinations

A *hotkey* or *shortcut* is a combination of keypresses to invoke some application function. The common hotkey for copying a selection is CTRL-C (on Windows and Linux) or ⌘-C (on OS X). The user presses and holds the CTRL key, then presses the C key, and then releases the C and CTRL keys. To do this with PyAutoGUI's `keyDown()` and `keyUp()` functions, you would have to enter the following:

```
pyautogui.keyDown('ctrl')
pyautogui.keyDown('c')
pyautogui.keyUp('c')
pyautogui.keyUp('ctrl')
```

This is rather complicated. Instead, use the `pyautogui.hotkey()` function, which takes multiple keyboard key string arguments, presses them in order, and releases them in the reverse order. For the CTRL-C example, the code would simply be as follows:

```
pyautogui.hotkey('ctrl', 'c')
```

This function is especially useful for larger hotkey combinations. In Word, the CTRL-ALT-SHIFT-S hotkey combination displays the Style pane. Instead of making eight different function calls (four `keyDown()` calls and four `keyUp()` calls), you can just call `hotkey('ctrl', 'alt', 'shift', 's')`.

With a new IDLE file editor window in the upper-left corner of your screen, enter the following into the interactive shell (in OS X, replace 'alt' with 'ctrl'):

```
>>> import pyautogui, time
>>> def commentAfterDelay():
    ❶    pyautogui.click(100, 100)
    ❷    pyautogui.typewrite('In IDLE, Alt-3 comments out a line.')
    ❸    time.sleep(2)
    ❹    pyautogui.hotkey('alt', '3')

>>> commentAfterDelay()
```

This defines a function `commentAfterDelay()` that, when called, will click the file editor window to bring it into focus ❶, type *In IDLE, Alt-3 comments out a line* ❷, pause for 2 seconds, and then simulate pressing the ALT-3 hotkey (or CTRL-3 on OS X) ❸. This keyboard shortcut adds two # characters to the current line, commenting it out. (This is a useful trick to know when writing your own code in IDLE.)

Review of the PyAutoGUI Functions

Since this chapter covered many different functions, here is a quick summary reference:

- `moveTo(x, y)` Moves the mouse cursor to the given `x` and `y` coordinates.
- `moveRel(xOffset, yOffset)` Moves the mouse cursor relative to its current position.
- `dragTo(x, y)` Moves the mouse cursor while the left button is held down.
- `dragRel(xOffset, yOffset)` Moves the mouse cursor relative to its current position while the left button is held down.
- `click(x, y, button)` Simulates a click (left button by default).
- `rightClick()` Simulates a right-button click.
- `middleClick()` Simulates a middle-button click.
- `doubleClick()` Simulates a double left-button click.
- `mouseDown(x, y, button)` Simulates pressing down the given button at the position `x, y`.
- `mouseUp(x, y, button)` Simulates releasing the given button at the position `x, y`.
- `scroll(units)` Simulates the scroll wheel. A positive argument scrolls up; a negative argument scrolls down.
- `typewrite(message)` Types the characters in the given message string.
- `typewrite([key1, key2, key3])` Types the given keyboard key strings.
- `press(key)` Presses the given keyboard key string.
- `keyDown(key)` Simulates pressing down the given keyboard key.
- `keyUp(key)` Simulates releasing the given keyboard key.
- `hotkey([key1, key2, key3])` Simulates pressing the given keyboard key strings down in order and then releasing them in reverse order.
- `screenshot()` Returns a screenshot as an `Image` object. (See Chapter 17 for information on `Image` objects.)

Project: Automatic Form Filler

Of all the boring tasks, filling out forms is the most dreaded of chores. It's only fitting that now, in the final chapter project, you will slay it. Say you have a huge amount of data in a spreadsheet, and you have to tediously retype it into some other application's form interface—with no intern to do it for you. Although some applications will have an Import feature that will allow you to upload a spreadsheet with the information, sometimes it seems that there is no other way than mindlessly clicking and typing for hours on end. You've come this far in this book; you know that *of course* there's another way.

The form for this project is a Google Docs form that you can find at <http://nostarch.com/automatestuff>. It looks like Figure 18-4.

The screenshot shows a Google Form titled "Generic Form". The form contains the following fields:

- A text input field labeled "Name *".
- A text input field labeled "Greatest Fear(s)".
- A dropdown menu labeled "What is the source of your wizard powers?" with "Wand" selected.
- A question "Robocop was the greatest action movie of the 1980s." followed by a scale from 1 to 5 and radio buttons for "Strongly Disagree" through "Strongly Agree".
- A text area labeled "Additional Comments".
- A "Submit" button.

At the bottom, a note says "Never submit passwords through Google Forms."

Figure 18-4: The form used for this project

At a high level, here's what your program should do:

- Click the first text field of the form.
- Move through the form, typing information into each field.
- Click the Submit button.
- Repeat the process with the next set of data.

This means your code will need to do the following:

- Call `pyautogui.click()` to click the form and Submit button.
- Call `pyautogui.typewrite()` to enter text into the fields.
- Handle the `KeyboardInterrupt` exception so the user can press CTRL-C to quit.

Open a new file editor window and save it as `formFiller.py`.

Step 1: Figure Out the Steps

Before writing code, you need to figure out the exact keystrokes and mouse clicks that will fill out the form once. The *mouseNow.py* script on page 417 can help you figure out specific mouse coordinates. You need to know only the coordinates of the first text field. After clicking the first field, you can just press TAB to move focus to the next field. This will save you from having to figure out the x- and y-coordinates to click for every field.

Here are the steps for entering data into the form:

1. Click the Name field. (Use *mouseNow.py* to determine the coordinates after maximizing the browser window. On OS X, you may need to click twice: once to put the browser in focus and again to click the Name field.)
2. Type a name and then press TAB.
3. Type a greatest fear and then press TAB.
4. Press the down arrow key the correct number of times to select the wizard power source: once for *wand*, twice for *amulet*, three times for *crystal ball*, and four times for *money*. Then press TAB. (Note that on OS X, you will have to press the down arrow key one more time for each option. For some browsers, you may need to press the ENTER key as well.)
5. Press the right arrow key to select the answer to the RoboCop question. Press it once for *2*, twice for *3*, three times for *4*, or four times for *5*; or just press the spacebar to select *1* (which is highlighted by default). Then press TAB.
6. Type an additional comment and then press TAB.
7. Press the ENTER key to “click” the Submit button.
8. After submitting the form, the browser will take you to a page where you will need to click a link to return to the form page.

Note that if you run this program again later, you may have to update the mouse click coordinates, since the browser window might have changed position. To work around this, always make sure the browser window is maximized before finding the coordinates of the first form field. Also, different browsers on different operating systems might work slightly differently from the steps given here, so check that these keystroke combinations work for your computer before running your program.

Step 2: Set Up Coordinates

Load the example form you downloaded (Figure 18-4) in a browser and maximize your browser window. Open a new Terminal or command line window to run the *mouseNow.py* script, and then mouse over the Name field to figure out its the x- and y-coordinates. These numbers will be assigned to the `nameField` variable in your program. Also, find out the x- and y-coordinates and RGB tuple value of the blue Submit button. These values will be assigned to the `submitButton` and `submitButtonColor` variables, respectively.

Next, fill in some dummy data for the form and click **Submit**. You need to see what the next page looks like so that you can use *mouseNow.py* to find the coordinates of the *Submit another response* link on this new page.

Make your source code look like the following, being sure to replace all the values in italics with the coordinates you determined from your own tests:

```
#! python3
# formFiller.py - Automatically fills in the form.

import pyautogui, time

# Set these to the correct coordinates for your computer.
nameField = (648, 319)
submitButton = (651, 817)
submitButtonColor = (75, 141, 249)
submitAnotherLink = (760, 224)

# TODO: Give the user a chance to kill the script.

# TODO: Wait until the form page has loaded.

# TODO: Fill out the Name Field.

# TODO: Fill out the Greatest Fear(s) field.

# TODO: Fill out the Source of Wizard Powers field.

# TODO: Fill out the RoboCop field.

# TODO: Fill out the Additional Comments field.

# TODO: Click Submit.

# TODO: Wait until form page has loaded.

# TODO: Click the Submit another response link.
```

Now you need the data you actually want to enter into this form. In the real world, this data might come from a spreadsheet, a plaintext file, or a website, and it would require additional code to load into the program. But for this project, you'll just hardcode all this data in a variable. Add the following to your program:

```
#! python3
# formFiller.py - Automatically fills in the form.

--snip--

formData = [{"name": "Alice", "fear": "eavesdroppers", "source": "wand",
    'robocop': 4, 'comments': 'Tell Bob I said hi.'},
    {"name": "Bob", "fear": "bees", "source": "amulet", "robocop": 4,
    'comments': 'n/a'},
```

```
{'name': 'Carol', 'fear': 'puppets', 'source': 'crystal ball',
'robocop': 1, 'comments': 'Please take the puppets out of the
break room.'},
{'name': 'Alex Murphy', 'fear': 'ED-209', 'source': 'money',
'robocop': 5, 'comments': 'Protect the innocent. Serve the public
trust. Uphold the law.'},
]
```

--snip--

The `formData` list contains four dictionaries for four different names. Each dictionary has names of text fields as keys and responses as values. The last bit of setup is to set PyAutoGUI's `PAUSE` variable to wait half a second after each function call. Add the following to your program after the `formData` assignment statement:

```
pyautogui.PAUSE = 0.5
```

Step 3: Start Typing Data

A for loop will iterate over each of the dictionaries in the `formData` list, passing the values in the dictionary to the PyAutoGUI functions that will virtually type in the text fields.

Add the following code to your program:

```
#! python3
# formFiller.py - Automatically fills in the form.

--snip--

for person in formData:
    # Give the user a chance to kill the script.
    print('>>> 5 SECOND PAUSE TO LET USER PRESS CTRL-C <<<')
❶    time.sleep(5)

    # Wait until the form page has loaded.
❷    while not pyautogui.pixelMatchesColor(submitButton[0], submitButton[1],
                                             submitButtonColor):
        time.sleep(0.5)
```

--snip--

As a small safety feature, the script has a five-second pause ❶ that gives the user a chance to hit CTRL-C (or move the mouse cursor to the upper-left corner of the screen to raise the `FailSafeException` exception) to shut the program down in case it's doing something unexpected. Then the program waits until the Submit button's color is visible ❷, letting the program know that the form page has loaded. Remember that you figured out the

coordinate and color information in step 2 and stored it in the `submitButton` and `submitButtonColor` variables. To use `pixelMatchesColor()`, you pass the coordinates `submitButton[0]` and `submitButton[1]`, and the color `submitButtonColor`.

After the code that waits until the Submit button's color is visible, add the following:

```
#! python3
# formFiller.py - Automatically fills in the form.

--snip--
```

```
❶  print('Entering %s info...' % (person['name']))
❷  pyautogui.click(nameField[0], nameField[1])

    # Fill out the Name field.
❸  pyautogui.typewrite(person['name'] + '\t')

    # Fill out the Greatest Fear(s) field.
❹  pyautogui.typewrite(person['fear'] + '\t')
```

```
--snip--
```

We add an occasional `print()` call to display the program's status in its Terminal window to let the user know what's going on ❶.

Since the program knows that the form is loaded, it's time to call `click()` to click the Name field ❷ and `typewrite()` to enter the string in `person['name']` ❸. The '`\t`' character is added to the end of the string passed to `typewrite()` to simulate pressing TAB, which moves the keyboard focus to the next field, Greatest Fear(s). Another call to `typewrite()` will type the string in `person['fear']` into this field and then tab to the next field in the form ❹.

Step 4: Handle Select Lists and Radio Buttons

The drop-down menu for the “wizard powers” question and the radio buttons for the RoboCop field are trickier to handle than the text fields. To click these options with the mouse, you would have to figure out the x- and y-coordinates of each possible option. It's easier to use the keyboard arrow keys to make a selection instead.

Add the following to your program:

```
#! python3
# formFiller.py - Automatically fills in the form.
```

```
--snip--

    # Fill out the Source of Wizard Powers field.
❶  if person['source'] == 'wand':
❷      pyautogui.typewrite(['down', '\t'])
```

```

        elif person['source'] == 'amulet':
            pyautogui.typewrite(['down', 'down', '\t'])
        elif person['source'] == 'crystal ball':
            pyautogui.typewrite(['down', 'down', 'down', '\t'])
        elif person['source'] == 'money':
            pyautogui.typewrite(['down', 'down', 'down', 'down', '\t'])

    # Fill out the RoboCop field.
❸    if person['robocop'] == 1:
❹        pyautogui.typewrite([' ', '\t'])
        elif person['robocop'] == 2:
            pyautogui.typewrite(['right', '\t'])
        elif person['robocop'] == 3:
            pyautogui.typewrite(['right', 'right', '\t'])
        elif person['robocop'] == 4:
            pyautogui.typewrite(['right', 'right', 'right', '\t'])
        elif person['robocop'] == 5:
            pyautogui.typewrite(['right', 'right', 'right', 'right', '\t'])

--snip--

```

Once the drop-down menu has focus (remember that you wrote code to simulate pressing TAB after filling out the Greatest Fear(s) field), pressing the down arrow key will move to the next item in the selection list. Depending on the value in person['source'], your program should send a number of down arrow keypresses before tabbing to the next field. If the value at the 'source' key in this user's dictionary is 'wand' ❶, we simulate pressing the down arrow key once (to select *Wand*) and pressing TAB ❷. If the value at the 'source' key is 'amulet', we simulate pressing the down arrow key twice and pressing TAB, and so on for the other possible answers.

The radio buttons for the RoboCop question can be selected with the right arrow keys—or, if you want to select the first choice ❸, by just pressing the spacebar ❹.

Step 5: Submit the Form and Wait

You can fill out the Additional Comments field with the typewrite() function by passing person['comments'] as an argument. You can type an additional '\t' to move the keyboard focus to the next field or the Submit button. Once the Submit button is in focus, calling pyautogui.press('enter') will simulate pressing the ENTER key and submit the form. After submitting the form, your program will wait five seconds for the next page to load.

Once the new page has loaded, it will have a *Submit another response* link that will direct the browser to a new, empty form page. You stored the coordinates of this link as a tuple in submitAnotherLink in step 2, so pass these coordinates to pyautogui.click() to click this link.

With the new form ready to go, the script's outer for loop can continue to the next iteration and enter the next person's information into the form.

Complete your program by adding the following code:

```
#!/usr/bin/python3
# formFiller.py - Automatically fills in the form.

--snip--

# Fill out the Additional Comments field.
pyautogui.typewrite(person['comments'] + '\t')

# Click Submit.
pyautogui.press('enter')

# Wait until form page has loaded.
print('Clicked Submit.')
time.sleep(5)

# Click the Submit another response link.
pyautogui.click(submitAnotherLink[0], submitAnotherLink[1])
```

Once the main for loop has finished, the program will have plugged in the information for each person. In this example, there are only four people to enter. But if you had 4,000 people, then writing a program to do this would save you a lot of time and typing!

Summary

GUI automation with the pyautogui module allows you to interact with applications on your computer by controlling the mouse and keyboard. While this approach is flexible enough to do anything that a human user can do, the downside is that these programs are fairly blind to what they are clicking or typing. When writing GUI automation programs, try to ensure that they will crash quickly if they're given bad instructions. Crashing is annoying, but it's much better than the program continuing in error.

You can move the mouse cursor around the screen and simulate mouse clicks, keystrokes, and keyboard shortcuts with PyAutoGUI. The pyautogui module can also check the colors on the screen, which can provide your GUI automation program with enough of an idea of the screen contents to know whether it has gotten offtrack. You can even give PyAutoGUI a screenshot and let it figure out the coordinates of the area you want to click.

You can combine all of these PyAutoGUI features to automate any mindlessly repetitive task on your computer. In fact, it can be downright hypnotic to watch the mouse cursor move on its own and see text appear on the screen automatically. Why not spend the time you saved by sitting back and watching your program do all your work for you? There's a certain satisfaction that comes from seeing how your cleverness has saved you from the boring stuff.

Practice Questions

1. How can you trigger PyAutoGUI's fail safe to stop a program?
2. What function returns the current `resolution()`?
3. What function returns the coordinates for the mouse cursor's current position?
4. What is the difference between `pyautogui.moveTo()` and `pyautogui.moveRel()`?
5. What functions can be used to drag the mouse?
6. What function call will type out the characters of "Hello world!"?
7. How can you do keypresses for special keys such as the keyboard's left arrow key?
8. How can you save the current contents of the screen to an image file named `screenshot.png`?
9. What code would set a two second pause after every PyAutoGUI function call?

Practice Projects

For practice, write programs that do the following.

Looking Busy

Many instant messaging programs determine whether you are idle, or away from your computer, by detecting a lack of mouse movement over some period of time—say, ten minutes. Maybe you'd like to sneak away from your desk for a while but don't want others to see your instant messenger status go into idle mode. Write a script to nudge your mouse cursor slightly every ten seconds. The nudge should be small enough so that it won't get in the way if you do happen to need to use your computer while the script is running.

Instant Messenger Bot

Google Talk, Skype, Yahoo Messenger, AIM, and other instant messaging applications often use proprietary protocols that make it difficult for others to write Python modules that can interact with these programs. But even these proprietary protocols can't stop you from writing a GUI automation tool.

The Google Talk application has a search bar that lets you enter a username on your friend list and open a messaging window when you press ENTER. The keyboard focus automatically moves to the new window. Other instant messenger applications have similar ways to open new message windows. Write a program that will automatically send out a notification message to a select group of people on your friend list. Your program may have to deal with exceptional cases, such as friends being offline, the chat window appearing at different coordinates on the screen, or confirmation

boxes that interrupt your messaging. Your program will have to take screenshots to guide its GUI interaction and adopt ways of detecting when its virtual keystrokes aren't being sent.

NOTE

You may want to set up some fake test accounts so that you don't accidentally spam your real friends while writing this program.

Game-Playing Bot Tutorial

There is a great tutorial titled “How to Build a Python Bot That Can Play Web Games” that you can find at <http://nostarch.com/automatestuff/>. This tutorial explains how to create a GUI automation program in Python that plays a Flash game called Sushi Go Round. The game involves clicking the correct ingredient buttons to fill customers’ sushi orders. The faster you fill orders without mistakes, the more points you get. This is a perfectly suited task for a GUI automation program—and a way to cheat to a high score! The tutorial covers many of the same topics that this chapter covers but also includes descriptions of PyAutoGUI’s basic image recognition features.

A

INSTALLING THIRD-PARTY MODULES



Beyond the standard library of modules packaged with Python, other developers have written their own modules to extend Python's capabilities even further. The primary way to install third-party modules is to use Python's pip tool. This tool securely downloads and installs Python modules onto your computer from <https://pypi.python.org/>, the website of the Python Software Foundation. PyPI, or the Python Package Index, is a sort of free app store for Python modules.

The pip Tool

The executable file for the pip tool is called *pip* on Windows and *pip3* on OS X and Linux. On Windows, you can find pip at *C:\Python34\Scripts\pip.exe*. On OS X, it is in */Library/Frameworks/Python.framework/Versions/3.4/bin/pip3*. On Linux, it is in */usr/bin/pip3*.

While pip comes automatically installed with Python 3.4 on Windows and OS X, you must install it separately on Linux. To install pip3 on Ubuntu or Debian Linux, open a new Terminal window and enter `sudo apt-get install python3-pip`. To install pip3 on Fedora Linux, enter `sudo yum install python3-pip` into a Terminal window. You will need to enter the administrator password for your computer in order to install this software.

Installing Third-Party Modules

The pip tool is meant to be run from the command line: You pass it the command `install` followed by the name of the module you want to install. For example, on Windows you would enter `pip install ModuleName`, where *ModuleName* is the name of the module. On OS X and Linux, you'll have to run pip3 with the `sudo` prefix to grant administrative privileges to install the module. You would need to type `sudo pip3 install ModuleName`.

If you already have the module installed but would like to upgrade it to the latest version available on PyPI, run `pip install -U ModuleName` (or `pip3 install -U ModuleName` on OS X and Linux).

After installing the module, you can test that it installed successfully by running `import ModuleName` in the interactive shell. If no error messages are displayed, you can assume the module was installed successfully.

You can install all of the modules covered in this book by running the commands listed next. (Remember to replace `pip` with `pip3` if you're on OS X or Linux.)

- `pip install send2trash`
- `pip install requests`
- `pip install beautifulsoup4`
- `pip install selenium`
- `pip install openpyxl`
- `pip install PyPDF2`
- `pip install python-docx` (install `python-docx`, not `docx`)
- `pip install imapclient`
- `pip install pyzmail`
- `pip install twilio`
- `pip install pillow`
- `pip install pyobjc-core` (on OS X only)
- `pip install pyobjc` (on OS X only)
- `pip install python3-xlib` (on Linux only)
- `pip install pyautogui`

NOTE

For OS X users: The `pyobjc` module can take 20 minutes or longer to install, so don't be alarmed if it takes a while. You should also install the `pyobjc-core` module first, which will reduce the overall installation time.

B

RUNNING PROGRAMS



If you have a program open in IDLE's file editor, running it is a simple matter of pressing F5 or selecting the Run ▶ Run Module menu item. This is an easy way to run programs while writing them, but opening IDLE to run your finished programs can be a burden. There are more convenient ways to execute Python scripts.

Shebang Line

The first line of all your Python programs should be a *shebang* line, which tells your computer that you want Python to execute this program. The shebang line begins with `#!`, but the rest depends on your operating system.

- On Windows, the shebang line is `#! python3`.
- On OS X, the shebang line is `#! /usr/bin/env python3`.
- On Linux, the shebang line is `#! /usr/bin/python3`.

You will be able to run Python scripts from IDLE without the shebang line, but the line is needed to run them from the command line.

Running Python Programs on Windows

On Windows, the Python 3.4 interpreter is located at `C:\Python34\python.exe`. Alternatively, the convenient `py.exe` program will read the shebang line at the top of the `.py` file's source code and run the appropriate version of Python for that script. The `py.exe` program will make sure to run the Python program with the correct version of Python if multiple versions are installed on your computer.

To make it convenient to run your Python program, create a `.bat` batch file for running the Python program with `py.exe`. To make a batch file, make a new text file containing a single line like the following:

```
@py.exe C:\path\to\your\pythonScript.py %*
```

Replace this path with the absolute path to your own program, and save this file with a `.bat` file extension (for example, `pythonScript.bat`). This batch file will keep you from having to type the full absolute path for the Python program every time you want to run it. I recommend you place all your batch and `.py` files in a single folder, such as `C:\MyPythonScripts` or `C:\Users\YourName\PythonScripts`.

The `C:\MyPythonScripts` folder should be added to the system path on Windows so that you can run the batch files in it from the Run dialog. To do this, modify the PATH environment variable. Click the **Start** button and type **Edit environment variables for your account**. This option should autocomplete after you've begun to type it. The Environment Variables window that appears will look like Figure B-1.

From System variables, select the Path variable and click **Edit**. In the Value text field, append a semicolon, type `C:\MyPythonScripts`, and then click **OK**. Now you can run any Python script in the `C:\MyPythonScripts` folder by simply pressing WIN-R and entering the script's name. Running `pythonScript`, for instance, will run `pythonScript.bat`, which in turn will save you from having to run the whole command `py.exe C:\MyPythonScripts\pythonScript.py` from the Run dialog.

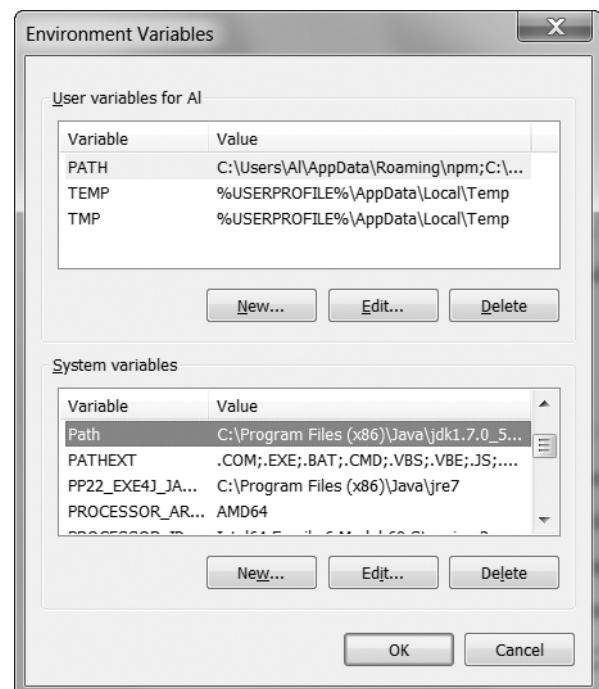


Figure B-1: The Environment Variables window on Windows

Running Python Programs on OS X and Linux

On OS X, selecting Applications ▶ Utilities ▶ Terminal will bring up a *Terminal* window. A Terminal window is a way to enter commands on your computer using only text, rather than clicking through a graphic interface. To bring up the Terminal window on Ubuntu Linux, press the WIN (or SUPER) key to bring up Dash and type in **Terminal**.

The Terminal window will begin in the home folder of your user account. If my username is *asweigart*, the home folder will be */Users/asweigart* on OS X and */home/asweigart* on Linux. The tilde (~) character is a shortcut for your home folder, so you can enter `cd ~` to change to your home folder. You can also use the `cd` command to change the current working directory to any other directory. On both OS X and Linux, the `pwd` command will print the current working directory.

To run your Python programs, save your *.py* file to your home folder. Then, change the *.py* file's permissions to make it executable by running `chmod +x pythonScript.py`. File permissions are beyond the scope of this book, but you will need to run this command on your Python file if you want to run the program from the Terminal window. Once you do so, you will be able to run your script whenever you want by opening a Terminal window and entering `./pythonScript.py`. The shebang line at the top of the script will tell the operating system where to locate the Python interpreter.

Running Python Programs with Assertions Disabled

You can disable the `assert` statements in your Python programs for a slight performance improvement. When running Python from the terminal, include the `-O` switch after `python` or `python3` and before the name of the *.py* file. This will run an optimized version of your program that skips the assertion checks.

C

ANSWERS TO THE PRACTICE QUESTIONS



This appendix contains the answers to the practice problems at the end of each chapter. I highly recommend that you take the time to work through these problems. Programming is more than memorizing syntax and a list of function names. As when learning a foreign language, the more practice you put into it, the more you will get out of it. There are many websites with practice programming problems as well. You can find a list of these at <http://nostarch.com/automatestuff/>.

Chapter 1

1. The operators are `+`, `-`, `*`, and `/`. The values are `'hello'`, `-88.8`, and `5`.
2. The string is `'spam'`; the variable is `spam`. Strings always start and end with quotes.
3. The three data types introduced in this chapter are integers, floating-point numbers, and strings.
4. An expression is a combination of values and operators. All expressions evaluate (that is, reduce) to a single value.
5. An expression evaluates to a single value. A statement does not.
6. The `bacon` variable is set to `20`. The `bacon + 1` expression does not reassign the value in `bacon` (that would need an assignment statement: `bacon = bacon + 1`).
7. Both expressions evaluate to the string `'spamspamspam'`.
8. Variable names cannot begin with a number.
9. The `int()`, `float()`, and `str()` functions will evaluate to the integer, floating-point number, and string versions of the value passed to them.
10. The expression causes an error because `99` is an integer, and only strings can be concatenated to other strings with the `+` operator. The correct way is `I have eaten ' + str(99) + ' burritos.'`.

Chapter 2

1. True and False, using capital `T` and `F`, with the rest of the word in lowercase
 - 2. and, or, and not
 - 3. True and True is True.
True and False is False.
False and True is False.
False and False is False.
True or True is True.
True or False is True.
False or True is True.
False or False is False.
not True is False.
not False is True.
4. False
False
True
False
False
True

5. `==`, `!=`, `<`, `>`, `<=`, and `>=`.
 6. `==` is the equal to operator that compares two values and evaluates to a Boolean, while `=` is the assignment operator that stores a value in a variable.
 7. A condition is an expression used in a flow control statement that evaluates to a Boolean value.
 8. The three blocks are everything inside the `if` statement and the lines `print('bacon')` and `print('ham')`.
-

```
print('eggs')
if spam > 5:
    print('bacon')
else:
    print('ham')
print('spam')
```

9. The code:
-

```
if spam == 1:
    print('Hello')
elif spam == 2:
    print('Howdy')
else:
    print('Greetings!')
```

10. Press `CTRL-C` to stop a program stuck in an infinite loop.
11. The `break` statement will move the execution outside and just after a loop. The `continue` statement will move the execution to the start of the loop.
12. They all do the same thing. The `range(10)` call ranges from 0 up to (but not including) 10, `range(0, 10)` explicitly tells the loop to start at 0, and `range(0, 10, 1)` explicitly tells the loop to increase the variable by 1 on each iteration.

13. The code:
-

```
for i in range(1, 11):
    print(i)
```

and:

```
i = 1
while i <= 10:
    print(i)
    i = i + 1
```

14. This function can be called with `spam.bacon()`.

Chapter 3

1. Functions reduce the need for duplicate code. This makes programs shorter, easier to read, and easier to update.
2. The code in a function executes when the function is called, not when the function is defined.
3. The `def` statement defines (that is, creates) a function.
4. A function consists of the `def` statement and the code in its `def` clause. A function call is what moves the program execution into the function, and the function call evaluates to the function's return value.
5. There is one global scope, and a local scope is created whenever a function is called.
6. When a function returns, the local scope is destroyed, and all the variables in it are forgotten.
7. A return value is the value that a function call evaluates to. Like any value, a return value can be used as part of an expression.
8. If there is no return statement for a function, its return value is `None`.
9. A `global` statement will force a variable in a function to refer to the global variable.
10. The data type of `None` is `NoneType`.
11. That `import` statement imports a module named `areallyourpetsnamederic`. (This isn't a real Python module, by the way.)
12. This function can be called with `spam.bacon()`.
13. Place the line of code that might cause an error in a `try` clause.
14. The code that could potentially cause an error goes in the `try` clause. The code that executes if an error happens goes in the `except` clause.

Chapter 4

1. The empty list value, which is a list value that contains no items. This is similar to how `''` is the empty string value.
2. `spam[2] = 'hello'` (Notice that the third value in a list is at index 2 because the first index is 0.)
3. `'d'` (Note that `'3' * 2` is the string `'33'`, which is passed to `int()` before being divided by `11`. This eventually evaluates to `3`. Expressions can be used wherever values are used.)
4. `'d'` (Negative indexes count from the end.)
5. `['a', 'b']`
6. `1`
7. `[3.14, 'cat', 11, 'cat', True, 99]`
8. `[3.14, 11, 'cat', True]`

9. The operator for list concatenation is `+`, while the operator for replication is `*`. (This is the same as for strings.)
10. While `append()` will add values only to the end of a list, `insert()` can add them anywhere in the list.
11. The `del` statement and the `remove()` list method are two ways to remove values from a list.
12. Both lists and strings can be passed to `len()`, have indexes and slices, be used in `for` loops, be concatenated or replicated, and be used with the `in` and `not in` operators.
13. Lists are mutable; they can have values added, removed, or changed. Tuples are immutable; they cannot be changed at all. Also, tuples are written using parentheses, `(` and `)`, while lists use the square brackets, `[` and `]`.
14. `(42,)` (The trailing comma is mandatory.)
15. The `tuple()` and `list()` functions, respectively
16. They contain references to list values.
17. The `copy.copy()` function will do a shallow copy of a list, while the `copy.deepcopy()` function will do a deep copy of a list. That is, only `copy.deepcopy()` will duplicate any lists inside the list.

Chapter 5

1. Two curly brackets: `{}`
2. `{'foo': 42}`
3. The items stored in a dictionary are unordered, while the items in a list are ordered.
4. You get a `KeyError` error.
5. There is no difference. The `in` operator checks whether a value exists as a key in the dictionary.
6. `'cat' in spam` checks whether there is a `'cat'` key in the dictionary, while `'cat' in spam.values()` checks whether there is a value `'cat'` for one of the keys in `spam`.
7. `spam.setdefault('color', 'black')`
8. `pprint.pprint()`

Chapter 6

1. Escape characters represent characters in string values that would otherwise be difficult or impossible to type into code.
2. `\n` is a newline; `\t` is a tab.
3. The `\\"` escape character will represent a backslash character.

4. The single quote in `Howl's` is fine because you've used double quotes to mark the beginning and end of the string.
5. Multiline strings allow you to use newlines in strings without the `\n` escape character.
6. The expressions evaluate to the following:
 - `'e'`
 - `'Hello'`
 - `'Hello'`
 - `'lo world!'`
7. The expressions evaluate to the following:
 - `'HELLO'`
 - `True`
 - `'hello'`
8. The expressions evaluate to the following:
 - `['Remember', 'remember', 'the', 'fifth', 'of', 'November.']`
 - `'There-can-be-only-one.'`
9. The `rjust()`, `ljust()`, and `center()` string methods, respectively
10. The `lstrip()` and `rstrip()` methods remove whitespace from the left and right ends of a string, respectively.

Chapter 7

1. The `re.compile()` function returns Regex objects.
2. Raw strings are used so that backslashes do not have to be escaped.
3. The `search()` method returns Match objects.
4. The `group()` method returns strings of the matched text.
5. Group 0 is the entire match, group 1 covers the first set of parentheses, and group 2 covers the second set of parentheses.
6. Periods and parentheses can be escaped with a backslash: `\.`, `\(`, and `\)`.
7. If the regex has no groups, a list of strings is returned. If the regex has groups, a list of tuples of strings is returned.
8. The `|` character signifies matching “either, or” between two groups.
9. The `?` character can either mean “match zero or one of the preceding group” or be used to signify nongreedy matching.
10. The `+` matches one or more. The `*` matches zero or more.
11. The `{3}` matches exactly three instances of the preceding group. The `{3,5}` matches between three and five instances.
12. The `\d`, `\w`, and `\s` shorthand character classes match a single digit, word, or space character, respectively.
13. The `\D`, `\W`, and `\S` shorthand character classes match a single character that is not a digit, word, or space character, respectively.

14. Passing `re.I` or `re.IGNORECASE` as the second argument to `re.compile()` will make the matching case insensitive.
15. The `.` character normally matches any character except the newline character. If `re.DOTALL` is passed as the second argument to `re.compile()`, then the dot will also match newline characters.
16. The `.*` performs a greedy match, and the `.*?` performs a nongreedy match.
17. Either `[0-9a-z]` or `[a-zA-Z]`
18. 'X drummers, X pipers, five rings, X hens'
19. The `re.VERBOSE` argument allows you to add whitespace and comments to the string passed to `re.compile()`.
20. `re.compile(r'^\d{1,3}(,\d{3})*$')` will create this regex, but other regex strings can produce a similar regular expression.
21. `re.compile(r'[A-Z][a-z]*\sNakamoto')`
22. `re.compile(r'(Alice|Bob|Carol)\s(eats|pets|throws)\s(apples|cats|baseballs)\.', re.IGNORECASE)`

Chapter 8

1. Relative paths are relative to the current working directory.
2. Absolute paths start with the root folder, such as `/` or `C:\`.
3. The `os.getcwd()` function returns the current working directory. The `os.chdir()` function *changes* the current working directory.
4. The `.` folder is the current folder, and `..` is the parent folder.
5. `C:\bacon\eggs` is the dir name, while `spam.txt` is the base name.
6. The string `'r'` for read mode, `'w'` for write mode, and `'a'` for append mode
7. An existing file opened in write mode is erased and completely overwritten.
8. The `read()` method returns the file's entire contents as a single string value. The `readlines()` method returns a list of strings, where each string is a line from the file's contents.
9. A shelf value resembles a dictionary value; it has keys and values, along with `keys()` and `values()` methods that work similarly to the dictionary methods of the same names.

Chapter 9

1. The `shutil.copy()` function will copy a single file, while `shutil.copytree()` will copy an entire folder, along with all its contents.
2. The `shutil.move()` function is used for renaming files, as well as moving them.

3. The `send2trash` functions will move a file or folder to the recycle bin, while `shutil` functions will permanently delete files and folders.
4. The `zipfile.ZipFile()` function is equivalent to the `open()` function; the first argument is the filename, and the second argument is the mode to open the ZIP file in (read, write, or append).

Chapter 10

1. `assert(spam >= 10, 'The spam variable is less than 10.')`
2. `assert(eggs.lower() != bacon.lower(), 'The eggs and bacon variables are the same!') or assert(eggs.upper() != bacon.upper(), 'The eggs and bacon variables are the same!')`
3. `assert(False, 'This assertion always triggers.')`
4. To be able to call `logging.debug()`, you must have these two lines at the start of your program:

```
import logging
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s -
%(levelname)s - %(message)s')
```

5. To be able to send logging messages to a file named `programLog.txt` with `logging.debug()`, you must have these two lines at the start of your program:

```
import logging
>>> logging.basicConfig(filename='programLog.txt', level=logging.DEBUG,
format=' %(asctime)s - %(levelname)s - %(message)s')
```

6. DEBUG, INFO, WARNING, ERROR, and CRITICAL
7. `logging.disable(logging.CRITICAL)`
8. You can disable logging messages without removing the logging function calls. You can selectively disable lower-level logging messages. You can create logging messages. Logging messages provides a timestamp.
9. The Step button will move the debugger into a function call. The Over button will quickly execute the function call without stepping into it. The Out button will quickly execute the rest of the code until it steps out of the function it currently is in.
10. After you click Go, the debugger will stop when it has reached the end of the program or a line with a breakpoint.
11. A breakpoint is a setting on a line of code that causes the debugger to pause when the program execution reaches the line.
12. To set a breakpoint in IDLE, right-click the line and select **Set Breakpoint** from the context menu.

Chapter 11

1. The `webbrowser` module has an `open()` method that will launch a web browser to a specific URL, and that's it. The `requests` module can download files and pages from the Web. The `BeautifulSoup` module parses HTML. Finally, the `selenium` module can launch and control a browser.
2. The `requests.get()` function returns a `Response` object, which has a `text` attribute that contains the downloaded content as a string.
3. The `raise_for_status()` method raises an exception if the download had problems and does nothing if the download succeeded.
4. The `status_code` attribute of the `Response` object contains the HTTP status code.
5. After opening the new file on your computer in '`wb`' "write binary" mode, use a `for` loop that iterates over the `Response` object's `iter_content()` method to write out chunks to the file. Here's an example:

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

6. F12 brings up the developer tools in Chrome. Pressing **CTRL-SHIFT-C** (on Windows and Linux) or **⌘-OPTION-C** (on OS X) brings up the developer tools in Firefox.
7. Right-click the element in the page, and select **Inspect Element** from the menu.
8. `'#main'`
9. `'.highlight'`
10. `'div div'`
11. `'button[value="favorite"]'`
12. `spam.getText()`
13. `linkElem.attrs`
14. The `selenium` module is imported with `from selenium import webdriver`.
15. The `find_element_*` methods return the first matching element as a `WebElement` object. The `find_elements_*` methods return a list of all matching elements as `WebElement` objects.
16. The `click()` and `send_keys()` methods simulate mouse clicks and keyboard keys, respectively.
17. Calling the `submit()` method on any element within a form submits the form.
18. The `forward()`, `back()`, and `refresh()` `WebDriver` object methods simulate these browser buttons.

Chapter 12

1. The `openpyxl.load_workbook()` function returns a `Workbook` object.
2. The `get_sheet_names()` method returns a `Worksheet` object.
3. Call `wb.get_sheet_by_name('Sheet1')`.
4. Call `wb.get_active_sheet()`.
5. `sheet['C5'].value` or `sheet.cell(row=5, column=3).value`
6. `sheet['C5'] = 'Hello'` or `sheet.cell(row=5, column=3).value = 'Hello'`
7. `cell.row` and `cell.column`
8. They return the highest column and row with values in the sheet, respectively, as integer values.
9. `openpyxl.cell.column_index_from_string('M')`
10. `openpyxl.cell.get_column_letter(14)`
11. `sheet['A1':'F1']`
12. `wb.save('example.xlsx')`
13. A formula is set the same way as any value. Set the cell's `value` attribute to a string of the formula text. Remember that formulas begin with the `=` sign.
14. When calling `load_workbook()`, pass `True` for the `data_only` keyword argument.
15. `sheet.row_dimensions[5].height = 100`
16. `sheet.column_dimensions['C'].hidden = True`
17. OpenPyXL 2.0.5 does not load freeze panes, print titles, images, or charts.
18. Freeze panes are rows and columns that will always appear on the screen. They are useful for headers.
19. `openpyxl.charts.Reference()`, `openpyxl.charts.Series()`, `openpyxl.charts.BarChart()`, `chartObj.append(seriesObj)`, and `add_chart()`

Chapter 13

1. A `File` object returned from `open()`
2. Read-binary ('rb') for `PdfFileReader()` and write-binary ('wb') for `PdfFileWriter()`
3. Calling `getPage(4)` will return a `Page` object for page 5, since page 0 is the first page.
4. The `numPages` variable stores an integer of the number of pages in the `PdfFileReader` object.
5. Call `decrypt('swordfish')`.
6. The `rotateClockwise()` and `rotateCounterClockwise()` methods. The degrees to rotate is passed as an integer argument.

7. `docx.Document('demo.docx')`
8. A document contains multiple paragraphs. A paragraph begins on a new line and contains multiple runs. Runs are contiguous groups of characters within a paragraph.
9. Use `doc.paragraphs`.
10. A Run object has these variables (*not* a Paragraph).
11. True always makes the Run object bolded and False makes it always not bolded, no matter what the style's bold setting is. None will make the Run object just use the style's bold setting.
12. Call the `docx.Document()` function.
13. `doc.add_paragraph('Hello there!')`
14. The integers 0, 1, 2, 3, and 4

Chapter 14

1. In Excel, spreadsheets can have values of data types other than strings; cells can have different fonts, sizes, or color settings; cells can have varying widths and heights; adjacent cells can be merged; and you can embed images and charts.
2. You pass a File object, obtained from a call to `open()`.
3. File objects need to be opened in read-binary ('rb') for Reader objects and write-binary ('wb') for Writer objects.
4. The `writerow()` method
5. The `delimiter` argument changes the string used to separate cells in a row. The `lineterminator` argument changes the string used to separate rows.
6. `json.loads()`
7. `json.dumps()`

Chapter 15

1. A reference moment that many date and time programs use. The moment is January 1st, 1970, UTC.
2. `time.time()`
3. `time.sleep(5)`
4. It returns the closest integer to the argument passed. For example, `round(2.4)` returns 2.
5. A `datetime` object represents a specific moment in time. A `timedelta` object represents a duration of time.
6. `threadObj = threading.Thread(target=spam)`
7. `threadObj.start()`

8. Make sure that code running in one thread does not read or write the same variables as code running in another thread.
9. `subprocess.Popen('c:\\Windows\\System32\\calc.exe')`

Chapter 16

1. SMTP and IMAP, respectively
2. `smtplib.SMTP()`, `smtplib.ehlo()`, `smtplib.starttls()`, and `smtplib.login()`
3. `imapclient.IMAPClient()` and `imaplib.login()`
4. A list of strings of IMAP keywords, such as '`BEFORE <date>`', '`FROM <string>`', or '`SEEN`'
5. Assign the variable `imaplib._MAXLINE` a large integer value, such as `10000000`.
6. The `pyzmail` module reads downloaded emails.
7. You will need the Twilio account SID number, the authentication token number, and your Twilio phone number.

Chapter 17

1. An RGBA value is a tuple of 4 integers, each ranging from 0 to 255. The four integers correspond to the amount of red, green, blue, and alpha (transparency) in the color.
2. A function call to `ImageColor.getcolor('CornflowerBlue', 'RGBA')` will return `(100, 149, 237, 255)`, the RGBA value for that color.
3. A box tuple is a tuple value of four integers: the left edge x-coordinate, the top edge y-coordinate, the width, and the height, respectively.
4. `Image.open('zophie.png')`
5. `imageObj.size` is a tuple of two integers, the width and the height.
6. `imageObj.crop((0, 50, 50, 50))`. Notice that you are passing a box tuple to `crop()`, not four separate integer arguments.
7. Call the `imageObj.save('new_filename.png')` method of the `Image` object.
8. The `ImageDraw` module contains code to draw on images.
9. `ImageDraw` objects have shape-drawing methods such as `point()`, `line()`, or `rectangle()`. They are returned by passing the `Image` object to the `ImageDraw.Draw()` function.

Chapter 18

1. Move the mouse to the top-left corner of the screen, that is, the `(0, 0)` coordinates.
2. `pyautogui.size()` returns a tuple with two integers for the width and height of the screen.

3. `pyautogui.position()` returns a tuple with two integers for the x- and y-coordinates of the mouse cursor.
4. The `moveTo()` function moves the mouse to absolute coordinates on the screen, while the `moveRel()` function moves the mouse relative to the mouse's current position.
5. `pyautogui.dragTo()` and `pyautogui.dragRel()`
6. `pyautogui.typewrite('Hello world!')`
7. Either pass a list of keyboard key strings to `pyautogui.typewrite()` (such as `'left'`) or pass a single keyboard key string to `pyautogui.press()`.
8. `pyautogui.screenshot('screenshot.png')`
9. `pyautogui.PAUSE = 2`

INDEX

Symbols

= (assignment) operator, 18, 34
\ (backslash), 124, 151, 162, 174–175
 line continuation character, 93
^ (caret symbol), 162
 matching beginning of string,
 159–160
 negative character classes, 159
: (colon), 38, 45, 54, 82, 127
{ } (curly brackets), 105, 162
 greedy vs. nongreedy matching,
 156–157
 matching specific repetitions
 with, 156
\$ (dollar sign), 159–160, 162
. (dot character), 160–162
 using in paths, 175–176
 wildcard matches, 160–162
" (double quotes), 124
** (exponent) operator, 15
== (equal to) operator, 33, 34
/ (forward slash), 174–175
 division operator, 15, 88
> (greater than) operator, 33
>= (greater than or equal to)
 operator, 33
(hash character), 126
// (integer division/floored quotient)
 operator, 15
< (less than) operator, 33
<= (less than or equal to) operator, 33
% (modulus/remainder) operator,
 15, 88
* (multiplication) operator, 15, 83, 88
!= (not equal to) operator, 33
() (parentheses), 96–97, 152–153
| (pipe character), 153–154, 164–165
+ (plus sign), 155–156, 162
 addition operator, 15, 17, 83, 88
? (question mark), 154–155, 162
' (single quote), 124

[] (square brackets), 80, 162
* (star), 162
 using with wildcard character, 161
 zero or more matches with, 155
- (subtraction) operator, 15, 88
''' (triple quotes), 125, 164
_ (underscore), 20

A

%A directive, 344
%a directive, 344
absolute paths, 175–179
abspath() function, 177
addition (+) operator, 15, 17, 83, 88

B

\b backspace escape character, 419
%B directive, 344
%b directive, 344
back() method, 261
backslash (\), 124, 151, 162, 174–175
BarChart() function, 290
basename() function, 178
BCC search key, 370
Beautiful Soup, 245. *See also* bs4 module
BeautifulSoup objects, 245–246
BEFORE search key, 369
binary files, 180–181, 184–185
binary operators, 35–37
bitwise or operator, 164–165
blank strings, 17
blocking execution, 337
blocks of code, 37–38
BODY search key, 369
bold attribute, 311
Boolean data type
 binary operators, 35–36
 flow control and, 32–33
 in operator, 87
 not in operator, 87
 “truthy” and “falsey” values, 53
 using binary and comparison
 operators together, 36–37
box tuples, 390
breakpoints, debugging using, 229–231
break statements
 overview, 49–50
 using in for loop, 55
browser, opening using `webbrowser`
 module, 234–236
bs4 module
 creating object from HTML,
 245–246
 finding element with `select()`
 method, 246–247
 getting attribute, 248
 overview, 245
built-in functions, 57
bulleted list, creating in Wiki markup,
 139–141
copying and pasting clipboard,
 139–140
joining modified lines, 141
overview, 139
separating lines of text, 140

C

calling functions, 23
call stack, defined, 217
camelcase, 21
caret symbol (^), 162
 matching beginning of string,
 159–160
 negative character classes, 159
Cascading Style Sheets (CSS)
 matching with selenium
 module, 258
 selectors, 246–247
case sensitivity, 21, 163
CC search key, 370
Cell objects, 268–269
cells, in Excel spreadsheets, 266
 accessing Cell object by its name,
 268–269
 merging and unmerging, 286–287
 writing values to, 278–279
center() method, 133–134, 426
chaining method calls, 398
character classes, 158–159, 162
character styles, 310
charts, Excel, 288–290
chdir() function, 175
Chrome, developer tools in, 242–243
clear() method, 258
click() function, 420, 430, 431
clicking mouse, 420
click() method, 259
clipboard, using string from, 236
CMYK color model, 389
colon (:), 38, 45, 54, 82, 127
color values
 CMYK vs. RGB color models, 389
 RGBA values, 388–389
column_index_from_string() function, 270
columns, in Excel spreadsheets
 setting height and width of,
 285–286
 slicing Worksheet objects to get Cell
 objects in, 270–272
Comcast mail, 363, 367
comma-delimited items, 80
command line arguments, 235
commentAfterDelay() function, 429
comments
 multiline, 126
 overview, 23

comparison operators
 overview, 33–35
 using binary operators with, 36–37

`compile()` function, 151, 152, 164–165

compressed files
 backing up folder into, 209–212
 creating ZIP files, 205–206
 extracting ZIP files, 205
 overview, 203–204
 reading ZIP files, 204

computer screen
 coordinates of, 415
 resolution of, 416

concatenation
 of lists, 83
 string, 17–18

concurrency issues, 349

conditions, defined, 37

`continue` statements
 overview, 50–53
 using in `for` loop, 55

Coordinated Universal Time (UTC), 336

coordinates
 of computer screen, 415
 of an image, 389–390

`copy()` function, 100–101, 135, 198, 394

`copytree()` function, 198–199

countdown project, 357–358
 counting down, 357
 overview, 357
 playing sound file, 357–358

`cProfile.run()` function, 337

crashes, program, 14

`create_sheet()` method, 278

CRITICAL level, 224

cron, 354

cropping images, 393–394

CSS (Cascading Style Sheets)
 matching with `selenium` module, 258
 selectors, 246–247

CSV files
 defined, 319
 delimiter for, 324
 format overview, 320
 line terminator for, 324
 Reader objects, 321
 reading data in loop, 322
 removing header from, 324–327
 looping through CSV files, 325
 overview, 324–325
 reading in CSV file, 325–326
 writing out CSV file, 326–327

`Writer` objects, 322–323

curly brackets (`{}`), 105, 162
greedy vs. nongreedy matching,
 156–157
matching specific repetitions
 with, 156

current working directory, 175

D

\D character class, 158

\d character class, 158

\d directive, 344

data structures
 algebraic chess notation, 112–113
 tic-tac-toe board, 113–117

data types
 Booleans, 32
 defined, 16
 dictionaries, 105–106
 floating-point numbers, 17
 integers, 17
 `list()` function, 97
 lists, 80
 mutable vs. immutable, 94–96
 `None` value, 65
 strings, 17
 `tuple()` function, 97
 tuples, 96–97

`datetime` module
 arithmetic using, 343
 converting objects to strings,
 344–345
 converting strings to objects, 345
 `fromtimestamp()` function, 341
 `now()` function, 341
 overview, 341–342, 346
 pausing program until time, 344
 `timedelta` data type, 342–343
 `total_seconds()` method, 342

`datetime` objects, 341–342
 converting to strings, 344–345
 converting from strings to, 345

`debug()` function, 222

debugging
 assertions, 219–221
 defined, 4
 getting traceback as string, 217–218
 in IDLE
 overview, 225–227
 stepping through program,
 227–229
 using breakpoints, 229–231

debugging (*continued*)
 logging
 disabling, 224–225
 to file, 225
 levels of, 223–224
 logging module, 221–223
 print() function and, 223
 raising exceptions, 216–217
 DEBUG level, 223
 decimal numbers. *See* floating-point numbers
 decode() method, 374–375
 decryption, of PDF files, 297–298
 deduplicating code, 62
 deepcopy() function, 100–101
 def statements, 62
 with parameters, 63
 DELETED search key, 370
 delete_messages() method, 375
 deleting files/folders
 permanently, 200–201
 using send2trash module, 201–202
 del statements, 84
 dictionaries
 copy() function, 100–101
 deepcopy() function, 100–101
 get() method, 109
 in operator, 109
 items() method, 107–108
 keys() method, 107–108
 lists vs., 106–107
 nesting, 117–119
 not in operator, 109
 overview, 105–106
 setdefault() method, 110–111
 values() method, 107–108
 directories
 absolute vs. relative paths, 175–176
 backslash vs. forward slash, 174–175
 copying, 198–199
 creating, 176
 current working directory, 175
 defined, 173–174
 deleting permanently, 200–201
 deleting using send2trash module, 201–202
 moving, 199–200
 os.path module
 absolute paths in, 177–179
 file sizes, 179–180
 folder contents, 179–180
 overview, 177
 path validity, 180
 relative paths in, 177–179
 renaming, 199–200
 walking, 202–203
 dirname() function, 178
 disable() function, 224
 division (/) operator, 15, 88
 Document objects, 307–308
 dollar sign (\$), 159–160, 162
 dot character (.), 160–162
 using in paths, 175–176
 wildcard matches, 160–162
 dot-star character (*), 161
 doubleClick() function, 420, 430
 double quotes ("), 124
 double_strike attribute, 311
 downloading
 files from web, 239–240
 web pages, 237–238
 XKCD comics, 251–256, 350–352
 DRAFT search key, 370
 dragging mouse, 420–422
 dragRel() function, 420, 422, 430
 dragTo() function, 420, 430
 drawing on images
 ellipses, 407
 example program, 407–408
 ImageDraw module, 406
 lines, 406–407
 points, 406
 polygons, 407
 rectangles, 407
 text, 408–410
 dumps() function, 329
 duration keyword arguments, 416

E

ehlo() method, 364, 379
 elements, HTML, 240
 elif statements, 40–45
 ellipse() method, 407
 else statements, 39–40
 email addresses, extracting, 165–169
 creating regex, 166–167
 finding matches on clipboard, 167–168
 joining matches into a string, 168
 overview, 165–166
 emails
 deleting, 375
 disconnecting from server, 375–376
 fetching

folders, 368–369
getting message content, 372–373
logging into server, 368
overview, 366–367
raw messages, 373–375
`gmail_search()` method, 372
IMAP, 366
marking message as read, 372–373
searching, 368–371
sending
 connecting to SMTP server, 363–364
 disconnecting from server, 366
 logging into server, 364–365
 overview, 362
 reminder, 376–380
 sending “hello” message, 364
 sending message, 365
 TLS encryption, 364
SMTP, 362
`emboss` attribute, 311
encryption, of PDF files, 302–303
`endswith()` method, 131
epoch timestamps, 336, 341, 346
equal to (==) operator, 33, 34
ERROR level, 224
errors
 crashes and, 14
 help for, 8–9
escape characters, 124–125
evaluation, defined, 14
Excel spreadsheets
 application support, 265–266
 charts in, 288–290
 column width, 285–286
 converting between column letters
 and numbers, 270
 creating documents, 277
 creating worksheets, 278
 deleting worksheets, 278
 font styles, 282–284
 formulas in, 284–285
 freezing panes, 287–288
 getting cell values, 268–269
 getting rows and columns, 270–272
 getting worksheet names, 268
 merging and unmerging cells, 286–287
 opening documents, 267
 `openpyxl` module, 266
 overview, 266–267
reading files
 overview, 272–273
 populating data structure, 274–275
 reading data, 273–274
 writing results to file, 275–276
and reminder emails project, 376–380
row height, 285–286
saving workbooks, 277
updating, 279–281
 overview, 279–280
 setup, 280
workbooks vs., 266
writing values to cells, 278–279
Exception objects, 217
exceptions
 assertions and, 219–221
 getting traceback as string, 217–218
 handling, 72–74
 raising, 216–217
execution, program
 defined, 31
 overview, 38
 pausing until specific time, 344
 terminating program with
 `sys.exit()`, 58
`exists()` function, 180
exit codes, 353–354
`expand` keyword, 398
`exponent (**)` operator, 15
expressions
 conditions and, 37
 in interactive shell, 14–16
`expunge()` method, 375
extensions, file, 173
`extractall()` method, 205
extracting ZIP files, 205
`extract()` method, 205

F

`FailSafeException` exception, 434
“falsey” values, 53
`fetch()` method, 371, 372–373
file editor, 21
file management
 absolute vs. relative paths, 175–176
 backslash vs. forward slash, 174–175
 compressed files
 backing up to, 209–212
 creating ZIP files, 205–206

file management (*continued*)
 compressed files (*continued*)
 extracting ZIP files, 205
 overview, 203–204
 reading ZIP files, 204
 creating directories, 176
 current working directory, 175
 multiclipboard project, 191–193
 opening files, 181–182
 os.path module
 absolute paths in, 177–179
 file sizes, 179–180
 folder contents, 179–180
 overview, 177
 path validity, 180
 relative paths in, 177–179
 overview, 173–174
 paths, 173–174
 plaintext vs. binary files, 180–181
 reading files, 182–183
 renaming files, date styles, 206–209
 saving variables with pformat()
 function, 185–186
 send2trash module, 201–202
 shelve module, 184–185
 shutil module
 copying files/folders, 198–199
 deleting files/folders, 200–201
 moving files/folders, 199–200
 renaming files/folders, 199–200
 walking directory trees, 202–203
 writing files, 183–184
filenames, defined, 173
File objects, 182
findall() method, 157–158
find_element_by_* methods, 257–258
find_elements_by_* methods, 257–258
Firefox, developer tools in, 243
FLAGGED search key, 370
flipping images, 398–399
float() function, 25–28
floating-point numbers
 integer equivalence, 27
 overview, 17
 rounding, 338
flow control
 binary operators, 35–36
 blocks of code, 37–38
 Boolean values and, 32–33
 break statements, 49–50
 comparison operators, 33–35
 conditions, 37
continue statements, 50–53
elif statements, 40–45
else statements, 39–40
if statements, 38–39
overview, 31–32
using binary and comparison
 operators together, 36–37
while loops, 45–49
folders
 absolute vs. relative paths, 175–176
 backing up to ZIP file, 209–212
 creating new ZIP file, 211
 figuring out ZIP filename,
 210–211
 walking directory tree, 211–212
 backslash vs. forward slash, 174–175
 copying, 198–199
 creating, 176
 current working directory, 175
 defined, 173–174
 deleting permanently, 200–201
 deleting using send2trash module,
 201–202
 moving, 199–200
os.path module
 absolute paths in, 177–179
 file sizes, 179–180
 folder contents, 179–180
 overview, 177
 path validity, 180
 relative paths in, 177–179
renaming, 199–200
walking directory trees, 202–203
Font objects, 282–283
font styles, in Excel spreadsheets,
 282–284
for loops
 overview, 53–56
 using dictionary items in, 108
 using lists with, 86
format attribute, 392
format_description attribute, 392
formData list, 434
form filler project, 430–437
 overview, 430–431
 radio buttons, 435–436
 select lists, 435–436
 setting up coordinates, 432–434
 steps in process, 431
 submitting form, 436–437
 typing data, 434–435

formulas, in Excel spreadsheets, 284–285
`forward()` method, 261
forward slash (/), 174–175
FROM search key, 370
`fromtimestamp()` function, 341, 346
functions. *See also names of individual functions*
 arguments, 23, 63
 as “black box”, 72
 built-in, 57
 `def` statements, 63
 exception handling, 72–74
 keyword arguments, 65–66
 None value and, 65
 overview, 61–62
 parameters, 63
 return values, 63–65

G

`get_active_sheet()` method, 268
`get_addresses()` method, 374
`get_attribute()` method, 258
`getcolor()` function, 388–389, 393
`get_column_letter()` function, 270
`getcwd()` function, 175
`get()` function
 overview, 109
 requests module, 237
`get_highest_column()` method, 269, 377
`get_highest_row()` method, 269
`get_payload()` method, 374–375
`getpixel()` function, 400, 423, 424
`get_sheet_by_name()` method, 268
`get_sheet_names()` method, 268
`getsize()` function, 179
`get_subject()` method, 374
`getText()` function, 308–309
GIF format, 392
global scope, 70–71
Gmail, 363, 365, 367
`gmail_search()` method, 372
Google Maps, 234–236
graphical user interface automation.
 See GUI (graphical user interface) automation
greater than (>) operator, 33
greater than or equal to (>=)
 operator, 33
greedy matching
 dot-star for, 161
 in regular expressions, 156–157

`group()` method, 151, 152–153
groups, regular expression
 matching
 greedy, 156–157
 nongreedy, 157
 one or more, 155–156
 optional, 154–155
 specific repetitions, 156
 zero or more, 155
 using parentheses, 152–153
 using pipe character in, 153–154
Guess the Number program, 74–76
GUI (graphical user interface)
 automation. *See also*
 form filler project
 controlling keyboard, 426–429
 hotkey combinations, 429
 key names, 427–428
 pressing and releasing, 428–429
 sending string from keyboard,
 426–427
 controlling mouse, 415–417,
 419–423
 clicking mouse, 420
 dragging mouse, 420–422
 scrolling mouse, 422–423
 determining mouse position,
 417–419
 image recognition, 425–426
 installing pyautogui module, 414
 logging out of program, 414
 overview, 413–414
 screenshots, 423–424
 stopping program, 414–415

H

`%H` directive, 344
hash character (#), 126
headings, Word document, 314–315
help
 asking online, 9–10
 for error messages, 8–9
hotkey combinations, 429
`hotkey()` function, 429, 430
Hotmail.com, 363, 367
HTML (Hypertext Markup Language)
 browser developer tools and,
 242–243
 finding elements, 244
 learning resources, 240
 overview, 240–241
 viewing page source, 241–242

I

- %I directive, 344
- id attribute, 241
- IDLE (interactive development environment)
 - creating programs, 21–22
 - debugging in
 - overview, 225–227
 - stepping through program, 227–229
 - using breakpoints, 229–231
 - expressions in, 14–16
 - overview, 8
 - running scripts outside of, 136
 - starting, 7–8
- if statements
 - overview, 38–39
 - using in while loop, 46–47
- imageDraw module, 406
- imageDraw objects, 406–408
- ImageFont objects, 408–410
- Image objects, 391–399
- images
 - adding logo to, 401–405
 - attributes for, 392–393
 - box tuples, 390
 - color values in, 388–389
 - coordinates in, 389–390
 - copying and pasting in, 394–396
 - cropping, 393–394
 - drawing on
 - example program, 407–408
 - ellipses, 407
 - ImageDraw module, 406
 - lines, 406–407
 - points, 406
 - polygons, 407
 - rectangles, 407
 - text, 408–410
 - flipping, 398–399
 - opening with Pillow, 390–391
 - pixel manipulation, 400
 - recognition of, 425–426
 - resizing, 397
 - RGBA values, 388–389
 - rotating, 398–399
 - transparent pixels, 397
- IMAP (Internet Message Access Protocol)
 - defined, 366
 - deleting messages, 375
- disconnecting from server, 375–376
- fetching messages, 372–375
- folders, 368–369
- logging into server, 368
- searching messages, 368–371
- imapclient module, 366
- IMAPClient objects, 367–368
- immutable data types, 94–96
- importing modules
 - overview, 57–58
- pyautogui module, 417
- imprint attribute, 311
- im variable, 423
- indentation, 93
- indexes
 - for dictionaries. *See* keys, dictionary for lists
 - changing values using, 83
 - getting value using, 80–81
 - negative, 82
 - removing values from list
 - using, 84
 - for strings, 126–127
- IndexError, 106
- index() method, 89
- infinite loops, 49, 51, 418
- INFO level, 223
- in operator
 - using with dictionaries, 109
 - using with lists, 87
 - using with strings, 127
- input() function
 - overview, 23–24, 89–90
 - using for sensitive information, 365
- installing
 - openpyxl module, 266
 - pyautogui module, 414
 - Python, 6–7
 - selenium module, 256
 - third-party modules, 441–442
- int, 17. *See also* integers
- integer division/floored quotient (//) operator, 15
- integers
 - floating-point equivalence, 27
 - overview, 17
- interactive development environment.
 - See* IDLE (interactive development environment)
- interactive shell. *See* IDLE
- Internet Explorer, developer tools in, 242–243

Internet Message Access Protocol. *See*
IMAP (Internet Message
Access Protocol)

interpreter, Python, 7
`int()` function, 25–28
`isabs()` function, 177
`isalnum()` method, 129–131
`isalpha()` method, 129–130
`isdecimal()` method, 129–131
`isdir()` function, 180
`is_displayed()` method, 258
`is_enabled()` method, 258
`.isfile()` function, 180
`islower()` method, 128–129
`is_selected()` method, 258
`isspace()` method, 130
`istitle()` method, 130
`isupper()` method, 128–129
italic attribute, 311
`items()` method, 107–108
`iter_content()` method, 239–240

J

`%j` directive, 344
`join()` method, 131–132, 174–175,
177, 352
JPEG format, 392
JSON files
 APIs for, 327–328
 defined, 319–320
 format overview, 327–328
 reading, 328–329
 and weather data project, 329–332
 writing, 329
justifying text, 133–134

K

keyboard
 controlling, with PyAutoGUI
 hotkey combinations, 429
 pressing and releasing keys,
 428–429
 sending string from keyboard,
 426–427
 key names, 427–428
 `KeyboardInterrupt` exception, 340,
 417, 418
`keyDown()` function, 428, 429, 430
keys, dictionary, 105

`keys()` method, 107–108
`keyUp()` function, 428, 429, 430
keyword arguments, 65–66

L

LARGER search key, 370
launchd, 354–355
launching programs
 and countdown project, 357–358
 opening files with default
 applications, 355–356
 opening websites, 355
 overview, 352–354
 passing command line arguments
 to processes, 354
`poll()` method, 353
running Python scripts, 355
scheduling, 354–355
`sleep()` function, 355
`wait()` method, 354
`len()` function, 307–308
 finding number of values in list, 83
 overview, 24–25

less than (`<`) operator, 33
less than or equal to (`<=`) operator, 33
LibreOffice, 265, 306
line breaks, Word document, 315
`LineChart()` function, 290
line continuation character (`\`), 93
`line()` method, 406–407
linked styles, 310

Linux
 backslash vs. forward slash, 174–175
 cron, 354
 installing Python, 7
 installing third-party modules, 442
 launching processes from
 Python, 353
 logging out of automation
 program, 414
 opening files with default
 applications, 355
 pip tool on, 441–442
 Python support, 4
 running Python programs on, 445
 starting IDLE, 8
 Unix philosophy, 356
`listdir()` function, 179
`list_folders()` method, 368–369
`list()` function, 321, 426

lists
 append() method, 89–90
 augmented assignment operators,
 88–89
 changing values using index, 83
 concatenation of, 83
 copy() function, 100–101
 deepcopy() function, 100–101
 dictionaries vs., 106–107
 finding number of values using
 len(), 83
 getting sublists with slices, 82–83
 getting value using index, 80–81
 index() method, 89
 in operator, 87
 insert() method, 89–90
 list() function, 97
 Magic 8 Ball example program
 using, 92–93
 multiple assignment trick, 87–88
 mutable vs. immutable data types,
 94–96
 negative indexes, 82
 nesting, 117–119
 not in operator, 87
 overview, 80
 remove() method, 90–91
 removing values from, 84
 replication of, 83
 sort() method, 91–92
 storing variables as, 84–85
 using with for loops, 86
ljust() method, 133–134
load_workbook() function, 267
loads() function, 328–329, 331
local scope, 67–70
locateAllOnScreen() function, 426
locateOnScreen() function, 425
location attribute, 258
logging
 disabling, 224–225
 to file, 225
 levels of, 223–224
 print() function and, 223
logging module, 221–223
logging out, of automation
 program, 414
login() method, 364, 368, 379
logo, adding to an image, 401–406
 looping over files, 402–403
 opening logo image, 401–402
 overview, 404
 resizing image, 403–404

logout() method, 375–376
LogRecord objects, 221
loops
 break statements, 49–50
 continue statements, 50–53
 for loop, 53–56
 range() function for, 56–57
 reading data from CSV file, 322
 using lists with, 86
 while loop, 45–49
lower() method, 128–129
lstrip() method, 134–135

M

%M directive, 344
%m directive, 344
Mac OS X. *See* OS X
Magic 8 Ball example program, 92–93
makedirs() function, 176, 403
maps, open when location is copied,
 234–236
 figuring out URL, 234–235
 handling clipboard content, 236
 handling command line argument,
 235–236
 launching browser, 236
 overview, 234
Match objects, 151
math
 operators for, 15
 programming and, 4
mergePage() method, 302
Message objects, 381–382
methods
 chaining calls, 398
 defined, 89
 dictionary
 get() method, 109
 items() method, 107–108
 keys() method, 107–108
 setdefault() method, 110–111
 values() method, 107–108
list
 append() method, 89–90
 index() method, 89
 insert() method, 89–90
 remove() method, 90–91
 sort() method, 91–92
string
 center() method, 133–134
 copy() method, 135
 endswith() method, 131

`isalnum()` method, 129–131
`isalpha()` method, 129–130
`isdecimal()` method, 129–131
`islower()` method, 128–129
`isspace()` method, 130
`istitle()` method, 130
`isupper()` method, 128–129
`join()` method, 131–132
`ljust()` method, 133–134
`lower()` method, 128–129
`lstrip()` method, 134–135
`paste()` method, 135
`rjust()` method, 133–134
`rstrip()` method, 134–135
`split()` method, 131–133
`startswith()` method, 131
`strip()` method, 134–135
`upper()` method, 128–129

Microsoft Windows. *See* Windows OS

`middleClick()` function, 420, 430

modules

- importing, 57–58
- third-party, installing, 442

modulus/remainder (%) operator, 15, 88

Monty Python, 4

mouse

- controlling, 415–417, 419–423
- clicking mouse, 420
- dragging mouse, 420–422
- scrolling mouse, 422–423
- determining position of, 417–419
- locating, 417–419
 - getting coordinates, 418–419
 - handling `KeyboardInterrupt` exception, 418
- importing `pyautogui` module, 418
- infinite loop, 418
- overview, 417

and pixels, identifying colors of, 424–425

`mouseDown()` function, 420, 430

`mouse.position()` function, 418

`mouseUp()` function, 430

`move()` function, 199–200

`moveRel()` function, 416, 417, 420, 430

`moveTo()` function, 416, 420, 430

moving files/folders, 199–200

multiclipboard project, 191–193

- listing keywords, 193
- loading keyword content, 193

overview, 191

saving clipboard content, 192

setting up shelf file, 192

multiline comments, 126

multiline strings, 125–126

multiple assignment trick, 87–88

multiplication (*) operator, 15, 83, 88

multithreading

- concurrency issues, 349
- downloading multiple images, , 350–352
- creating and starting threads, 351–352
- using `downloadXkcd()` function, 350–351
- waiting for threads to end, 352

`join()` method, 352

overview, 347–348

passing arguments to threads, 348–349

`start()` method, 348, 349

`Thread()` function, 347–348

mutable data types, 94–96

N

`NameError`, 84

`namelist()` method, 204

negative character classes, 159

negative indexes, 82

nested lists and dictionaries, 117–119

newline keyword argument, 322

`None` value, 65

nongreedy matching

- dot, star, and question mark for, 161
- in regular expressions, 157

not equal to (!=) operator, 33

`not in` operator

- using with dictionaries, 109
- using with lists, 87
- using with strings, 127

`not` operator, 36

`NOT` search key, 370

`now()` function, 341, 346

O

`ON` search key, 369

`open()` function, 181–182, 234, 355–356, 391

opening files, 181–182

OpenOffice, 265, 306

open program, 355

- `openpyxl` module, installing, 266
- operators
- augmented assignment, 88–89
 - binary, 35–36
 - comparison, 33–35
 - defined, 14
 - math, 15
 - using binary and comparison operators together, 36–37
- order of operations, 15
- or operator, 36
- OR search key, 370
- OS X
- backslash vs. forward slash, 174–175
 - installing Python, 7
 - installing third-party modules, 442
 - launchd, 354
 - launching processes from Python, 356
 - logging out of automation program, 414
 - opening files with default applications, 355–356
 - pip tool on, 441–442
 - Python support, 4
 - running Python programs on, 445
 - starting IDLE, 8
 - Unix philosophy, 356
- outline attribute, 311
- Outlook.com, 363, 367
- P**
- `%p` directive, 344
- page breaks, Word document, 315
- Page objects, 297
- Paragraph objects, 307
- paragraphs, Word document, 309–310
- parameters, function, 63
- parentheses (), 96–97, 152–153
- parsing, defined, 245
- passing arguments, 23
- passing references, 100
- passwords
- application-specific, 365
 - managing project, 136–138
 - command-line arguments, 137
 - copying password, 137–138
 - data structures, 136–137
 - overview, 136
- pastebin.com, 10
- `paste()` method, 135, 394, 395
- paths
- absolute vs. relative, 175–176
 - backslash vs. forward slash, 174–175
 - current working directory, 175
 - overview, 173–174
 - `os.path` module
 - absolute paths in, 177–179
 - file sizes, 179–180
 - folder contents, 179–180
 - overview, 177
 - path validity, 180
 - relative paths in, 177–179
- `PAUSE` variable, 415, 434
- `PdfFileReader` objects, 297–298
- PDF files
- combining pages from multiple files, 303–306
 - adding pages, 303
 - finding PDF files, 304
 - opening PDFs, 304–305
 - overview, 303
 - saving results, 305–306
 - creating, 298–299
 - decrypting, 297–298
 - encrypting, 302–303
 - extracting text from, 296–297
 - format overview, 295–296
 - pages in
 - copying, 299–300
 - overlaying, 301–302
 - rotating, 300–301
- `PdfFileWriter` objects, 298–299
- `printf()` function
- overview, 111–112
 - saving variables in text files using, 185–186
- phone numbers, extracting, 165–169
- creating regex, 166
 - finding matches on clipboard, 167–168
 - joining matches into a string, 168
 - overview, 165–166
- Pillow
- copying and pasting in images, 394–396
 - cropping images, 393–394
 - drawing on images
 - ellipses, 407
 - example program, 407–408
 - `ImageDraw` module, 406
 - lines, 406–407

points, 406
polygons, 407
rectangles, 407
text, 408–410
flipping images, 398–399
image attributes, 392–393
module, 388
opening images, 390–391
pixel manipulation, 400
resizing images, 397
rotating images, 398–399
transparent pixels, 397
pipe character (|), 153–154, 164–165
pip tool, 441–442
`pixelMatchesColor()` function, 424, 435
pixels, 388, 400
plaintext files, 180–181
plus sign (+), 155–156, 162
PNG format, 392
`point()` method, 406
`poll()` method, 353
`polygon()` method, 407
`Popen()` function, 352–353
 opening files with default
 applications, 355–356
 passing command line
 arguments to, 354
`position()` function, 417, 419
`pprint()` function, 111–112
precedence of math operators, 15
`press()` function, 429, 430, 436
`print()` function, 435
 logging and, 223
 overview, 23
 passing multiple arguments to, 66
 using variables with, 24
processes
 and countdown project, 357–358
 defined, 352
 opening files with default
 applications, 355–356
 opening websites, 355
 passing command line
 arguments to, 354
`poll()` method, 353
`Popen()` function, 352–353
 `wait()` method, 354
profiling code, 336–337
programming
 blocks of code, 37–38
 comments, 23
 creativity needed for, 5
deduplicating code, 62
defined, 3
exception handling, 72–74
execution, program, 38
functions as “black boxes”, 72
global scope, 70–71
indentation, 93
local scope, 67–70
math and, 4
Python, 4
terminating program with
 `sys.exit()`, 58
projects
 Adding Bullets to Wiki Markup,
 139–141
 Adding a Logo, 401–406
 Automatic Form Filler, 430–437
 Backing Up a Folder into a ZIP File,
 209–212
 Combining Select Pages from Many
 PDFs, 303–306
 Downloading All XKCD Comics,
 251–256
 Extending the mouseNow Program,
 424–425
 Fetching Current Weather Data,
 329–332
 Generating Random Quiz Files,
 186–191
 “I’m Feeling Lucky” Google Search,
 248–251
 “Just Text Me” Module, 383
 `mapIt.py` with the `webbrowser`
 Module, 234–236
 Multiclipboard, 191–193
 Multithreaded XKCD Downloader,
 350–352
 Password Locker, 136–138
 Phone Number and Email Address
 Extractor, 165–169
 Reading Data from a Spreadsheet,
 272–276
 Removing the Header from CSV
 Files, 324–327
 Renaming Files with American-
 Style Dates to European-
 Style Dates, 206–209
 Sending Member Dues Reminder
 Emails, 376–379
 Simple Countdown Program,
 357–358
 Super Stopwatch, 338–340

- projects (*continued*)
 - Updating a Spreadsheet, 279–281
 - “Where Is the Mouse Right Now?”, 417–419
 - `putpixel()` method, 400
 - `pyautogui.click()` function, 431
 - `pyautogui.click()` method, 420
 - `pyautogui.doubleClick()` function, 420
 - `pyautogui.dragTo()` function, 420
 - `pyautogui.FailSafeException` exception, 415
 - `pyautogui.hotkey()` function, 429
 - `pyautogui.keyDown()` function, 428
 - `pyautogui.keyUp()` function, 428
 - `pyautogui.middleClick()` function, 420
 - `pyautogui` module
 - form filler project, 430–437
 - controlling keyboard, 426–429
 - hotkey combinations, 429
 - key names, 427–428
 - pressing and releasing keys, 428–429
 - sending string from keyboard, 426–427
 - controlling mouse, 415–417, 419–423
 - clicking mouse, 420
 - dragging mouse, 420–422
 - scrolling mouse, 422–423
 - documentation for, 414
 - fail-safe feature, 415
 - functions, 430
 - image recognition, 425–426
 - importing, 417
 - installing, 414
 - pausing function calls, 415
 - screenshots, 423–424
 - `pyautogui.mouseDown()` function, 420
 - `pyautogui.moveRel()` function, 416, 417
 - `pyautogui.moveTo()` function, 416
 - `pyautogui.PAUSE` variable, 415
 - `pyautogui.position()` function, 419
 - `pyautogui.press()` function, 436
 - `pyautogui.rightClick()` function, 420
 - `pyautogui.screenshot()` function, 423
 - `pyautogui.size()` function, 416
 - `pyautogui.typewrite()` function, 426, 427, 431
 - `py.exe` program, 444–445
 - `pyobjc` module, 442
- PyPDF2 module**
 - combining pages from multiple PDFs, 303–306
 - creating PDFs, 298–299
 - decrypting PDFs, 297–298
 - encrypting PDFs, 302–303
 - extracting text from PDFs, 296–297
 - format overview, 295–296
 - pages in PDFs
 - copying, 299–300
 - overlaying, 301–302
 - rotating, 300–301
- pyperclip module**, 135
- Python**
 - data types, 16–17
 - downloading, 6
 - example program, 21–22
 - help, 8–9
 - installing, 6–7
 - interactive shell, 8
 - interpreter, defined, 7
 - math and, 4
 - overview, 4
 - programming overview, 3
 - starting IDLE, 7–8
- python-docx module**, 306
- pymail module**, 366, 373–375
- PyzMessage objects**, 373–375
- Q**
- question mark (?), 154–155, 162
- quit() method**, 261, 366, 379
- quiz generator, 186–190
 - creating quiz file, 188
 - creating answer options, 189
 - overview, 186
 - shuffling question order, 188
 - storing quiz data in dictionary, 187
 - writing content to files, 189–191
- R**
- radio buttons, 435–436
- raise_for_status() method**, 238
- raise keyword, 216–217
- range() function**, 56–57
- raw strings, 125, 151
- Reader objects, 321–322
- reading files, 182–183, 204
- readlines() method**, 182

`read()` method, 182
`rectangle()` method, 407
Reddit, 9
Reference objects, 289
references
 overview, 97–99
 passing, 100
`refresh()` method, 261
Regex objects
 creating, 150
 matching, 151
regular expressions
 beginning of string matches, 159–160
 case sensitivity, 163
 character classes, 158–159
 creating Regex objects, 150–151
 defined, 147–148
 end of string matches, 159–160
 extracting phone numbers and emails addresses, 165–169
 `findall()` method, 157–158
 finding text without, 148–150
 greedy matching, 156–157
 grouping
 matching specific repetitions, 156
 one or more matches, 155–156
 optional matching, 154–155
 using parentheses, 152–153
 using pipe character in, 153–154
 zero or more matches, 155
 HTML and, 243
 matching with, 151–152
 multiple arguments for `compile()` function, 164–165
 nongreedy matching, 157
 patterns for, 150
 spreading over multiple lines, 164
 substituting strings using, 163–164
 symbol reference, 162
 wildcard character, 160–162
relative paths, 175–179
`relpath()` function, 177, 178
remainder/modulus (%) operator, 15, 88
`remove()` method, 90–91
`remove_sheet()` method, 278
renaming files/folders, 199–200
 date styles, 206–209
 creating regex for dates, 206
 identifying dates in filenames, 207–208
 overview, 206
 renaming files, 209
replication
 of lists, 83
 string, 18
requests module
 downloading files, 239–240
 downloading pages, 237–238
resolution of computer screen, 416
Response objects, 237–238
return values, function, 63–65
reverse keyword, 91
RGBA values, 388–389
RGB color model, 389
`rightClick()` function, 420, 430
`rjust()` method, 133–134, 419
`rmdir()` function, 200
`rmtree()` function, 200
`rotateClockwise()` method, 300
`rotateCounterClockwise()` method, 300
rotating images, 398–399
rounding numbers, 338
rows, in Excel spreadsheets
 setting height and width of, 285–286
 slicing `Worksheet` objects to get Cell objects in, 270–272
`rstrip()` method, 134–135
rtl attribute, 311
Run objects, 310, 311–312
running programs
 on Linux, 445
 on OS X, 445
 overview, 443
 on Windows, 444–445
 shebang line, 443–444

S

`\S` character class, 158
`\s` character class, 158
`%$` directive, 344
Safari, developer tools in, 243
`save()` method, 391
scope
 global, 70–71
 local, 67–70
`screenshot()` function, 423, 430
screenshots
 analyzing, 424
 getting, 423
scripts
 running from Python program, 355
 running outside of IDLE, 136

`scroll()` function, 422, 423, 430
scrolling mouse, 422–423
searching
 email, 368–371
 the Web, 248–251
 finding results, 249–250
 getting command line
 arguments, 249
 opening web browser for
 results, 250–251
 overview, 248
 requesting search page, 249
`search()` method, 151
SEEN search key, 370
see program, 355
`select_folder()` method, 369
`select lists`, 435–436
`select()` method, `bs4` module, 246–247
selectors, CSS, 246–247, 258
`selenium` module
 clicking buttons, 261
 finding elements, 257–259
 following links, 259
 installing, 256
 sending special keystrokes, 260–261
 submitting forms, 259–260
 using Firefox with, 256–257
`send2trash` module, 201–202
sending reminder emails, 376–379
 finding unpaid members, 378
 opening Excel file, 376–377
 overview, 376
 sending emails, 378–379
`send_keys()` method, 259–260
`sendmail()` method, 365, 379
sequence numbers, 373
sequences, 86
`setdefault()` method, 110–111
shadow attribute, 311
shebang line, 443–444
`shelve` module, 184–185
Short Message Service (SMS)
 sending messages, 381–382
 Twilio service, 380
`shutil` module
 deleting files/folders, 200–201
 moving files/folders, 199–200
 renaming files/folders, 199–200
SID (string ID), 382
Simple Mail Transfer Protocol. *See*
 SMTP (Simple Mail
 Transfer Protocol)
SINCE search key, 369
single quote ('), 124
single-threaded programs, 347
`size()` function, 416
`sleep()` function, 337–338, 344, 346, 355
slices
 getting sublists with, 82–83
 for strings, 126–127
small_caps attribute, 311
SMALLER search key, 370
SMS (Short Message Service)
 sending messages, 381–382
 Twilio service, 380
SMTP (Simple Mail Transfer Protocol)
 connecting to server, 363–364
 defined, 362
 disconnecting from server, 366
 logging into server, 364–365
 sending “hello” message, 364
 sending message, 365
 TLS encryption, 364
SMTP objects, 363–364
`sort()` method, 91–92
sound files, playing, 357–358
source code, defined, 3
`split()` method, 131–133, 178, 320
spreadsheets. *See* Excel spreadsheets
square brackets [], 80
Stack Overflow, 9
standard library, 57
star (*), 161, 162
 using with wildcard character, 161
 zero or more matches with, 155
`start()` method, 348, 349, 351
start program, 355
`startswith()` method, 131
`starttls()` method, 364, 379
step argument, 56
stopwatch project, 338–340
 overview, 338–339
 set up, 339
 tracking lap times, 339–340
`strftime()` function, 344–345, 346
`str()` function, 25–28, 97, 419
strike attribute, 311
string ID (SID), 382
strings
 center() method, 133–134
 concatenation, 17–18
 converting `datetime` objects to,
 344–345
 converting to `datetime` objects, 345

copying and pasting, 135
double quotes for, 124
`endswith()` method, 131
escape characters, 124–125
extracting PDF text as, 296–297
getting traceback as, 217–218
indexes for, 126–127
`in` operator, 127
`isalnum()` method, 129–131
`isalpha()` method, 129–130
`isdecimal()` method, 129–131
`islower()` method, 128–129
`isspace()` method, 130
`istitle()` method, 130
`isupper()` method, 128–129
`join()` method, 131–132
literals, 124
`ljust()` method, 133–134
`lower()` method, 128–129
`lstrip()` method, 134–135
multiline, 125–126
mutable vs. immutable data types, 94–96
`not in` operator, 127
overview, 17
raw, 125
replication of, 18
`rjust()` method, 133–134
`rstrip()` method, 134–135
slicing, 126–127
`split()` method, 131–133
`startswith()` method, 131
`strip()` method, 134–135
substituting using regular expressions, 163–164
`upper()` method, 128–129
`strip()` method, 134–135
`strptime()` function, 345, 346
strs, 17, *See also* strings
Style objects, 282–283
SUBJECT search key, 369
sublists, getting with slices, 82–83
`sub()` method, 163–164
`submitButtonColor` variable, 432, 435
`submitButton` variable, 432
`submit()` method, 260
subprocess module, 335, 352–354
subtraction (-) operator, 15, 88
subtractive color model, 389
Sudoku puzzles, 4
`sys.exit()` function, 58

T

`tag_name` attribute, 258
Tag objects, 246–247
tags, HTML, 240
Task Scheduler, 354
termination, program, 22, 58
text attribute, 308, 311
text messaging
 automatic notifications, 383
 sending messages, 381–382
 Twilio service, 380
`text()` method, 408–410
TEXT search key, 369
`textsize()` method, 409
third-party modules, installing, 441–442
`Thread()` function, 347–348, 351
threading module, 335, 347
Thread objects, 347–348
threads
 concurrency issues, 349
 `join()` method, 352
 multithreading, 347–348
 image downloader, 350–352
 passing arguments to, 348–349
 processes vs., 352
tic-tac-toe board, 113–117
`timedelta` data type, 342–343, 346
`timedelta` objects, 342–343
time module
 overview, 346
 `sleep()` function, 337–338, 344
 stopwatch project, 338–340
 `time()` function, 336–337
TLS encryption, 364
top-level domains, 167
T0 search key, 370
`total_seconds()` method, 342, 346
traceback, getting from error, 217–218
transparency, 388, 397
`transpose()` method, 399
triple quotes (''), 125, 164
`truetype()` function, 409
truth tables, 35–36
“truthy” values, 53
tuple data type
 overview, 96–97
 `tuple()` function, 97
twilio module, 380
`TwilioRestClient` objects, 381

- Twilio service
 automatic text messages, 383
 overview, 380
 sending text messages, 381–382
- `TypeError`, 81, 94
- `typewrite()` function, 426, 427, 430, 431, 435, 436
- U**
- Ubuntu, 7
 cron, 354–355
 launching processes from Python, 353
 opening files with default applications, 355
 Unix philosophy, 356
- UNANSWERED search key, 370
- UNDELETED search key, 370
- underline attribute, 311
- underscore (_), 20
- UNDRAFT search key, 370
- UNFLAGGED search key, 370
- Unicode encodings, 239
- Unix epoch, 336, 341, 346
- Unix philosophy, 356
- `unlink()` function, 200
- UNSEEN search key, 370
- `upper()` method, 128–129
- UTC (Coordinated Universal Time), 336
- V**
- `ValueError`, 88, 345
- values, defined, 14, 150
- `values()` method, 107–108
- variables. *See also* lists
 assignment statements, 18–19
 defined, 18
 global, 70–71
 initializing, 19
 local, 67–70
 naming, 20–21
 None value and, 65
 overwriting, 19–20
 references, 97–99
 saving with `shelve` module, 184–185
 storing as list, 84–85
- Verizon mail, 363, 367
- volumes, defined, 174
- W**
- \W character class, 158
- \w character class, 158
- %w directive, 344
- `walk()` function, 202–203, 354
- WARNING level, 223
- weather data, fetching, 329–332
 downloading JSON data, 330–331
 getting location, 330
 loading JSON data, 331–332
 overview, 329
- `webbrowser` module
 `open()` function, 355
 opening browser using, 234–236
- `WebDriver` objects, 257
- `WebElement` objects, 257–258
- web scraping
 `bs4` module
 creating object from HTML, 245–246
 finding element with `select()` method, 246–247
 getting attribute, 248
 overview, 245
 downloading
 files, 239–240
 images, 251–256
 pages, 237–238
 and Google maps project, 234–236
 and Google search project, 248–251
 HTML
 browser developer tools and, 242–243
 finding elements, 244
 learning resources, 240
 overview, 240–241
 viewing page source, 241–242
 overview, 233–234
 `requests` module, 237–238
- `selenium` module
 clicking buttons, 261
 finding elements, 257–259
 following links, 259
 installing, 256
 sending special keystrokes, 260–261
 submitting forms, 259–260
 using Firefox with, 256–257
- websites, opening from script, 355

`while` loops
 getting and printing mouse coordinates using, 418
 infinite, 418
 overview, 45–49
whitespace, removing, 134–135
wildcard character (.), 160–162
Windows OS
 backslash vs. forward slash, 174–175
 installing Python, 6–7
 installing third-party modules, 442
 launching processes from Python, 353
 logging out of automation program, 414
 opening files with default applications, 355–356
 pip tool on, 441–442
 Python support, 4
 running Python programs on, 444–445
 starting IDLE, 7
 Task Scheduler, 354
Word documents
 adding headings, 314–315
 creating documents with nondefault styles, 310–311
 format overview, 306–307
 getting text from, 308–309
 line/page breaks, 315
 pictures in, 315–316
 python-docx module, 306
 reading, 307–308
 Run object attributes, 311–312
 styling paragraphs, 309–310
 writing to file, 312–314
Workbook objects, 267
workbooks, Excel, 266
 creating worksheets, 278
 deleting worksheets, 278
 opening, 267
 saving, 277
Worksheet objects, 268
`write()` method, 183–184
Writer objects, 322–323
`writerow()` method, 323

X

XKCD comics
 downloading project, 251–256
 designing program, 252–253
 downloading web page, 253–254
 overview, 251–252
 saving image, 255–256
multithreaded downloading project, 350–352
 creating and starting threads, 351–352
 using `downloadXkcd()` function, 350–351
 waiting for threads to end, 352

Y

%Y directive, 344
%y directive, 344
Yahoo! Mail, 363, 367

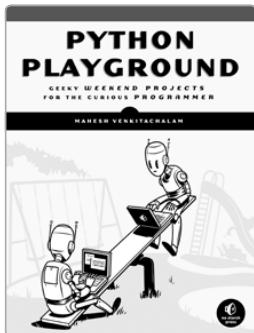
Z

zipfile module
 creating ZIP files, 205–206
 extracting ZIP files, 205
 and folders, 209–212
 overview, 203–204
 reading ZIP files, 204
ZipFile objects, 204–205
ZipInfo objects, 204

RESOURCES

Visit <http://nostarch.com/automatestuff/> for resources, errata, and more information.

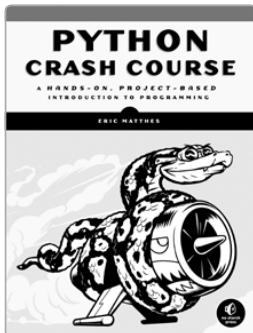
More no-nonsense books from  NO STARCH PRESS



PYTHON PLAYGROUND

**Geeky Weekend Projects
for the Curious Programmer**

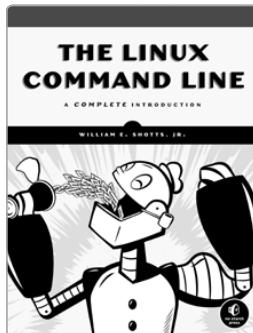
by MAHESH VENKITACHALAM
MAY 2015, 304 PP., \$29.95
ISBN 978-1-59327-604-1



PYTHON CRASH COURSE

**A Hands-On, Project-Based
Introduction to Programming**

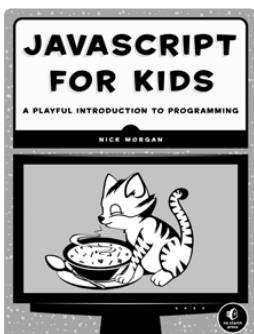
by ERIC MATTHES
JULY 2015, 624 PP., \$34.95
ISBN 978-1-59327-603-4



THE LINUX COMMAND LINE

A Complete Introduction

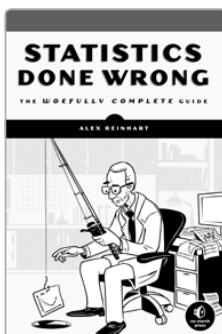
by WILLIAM E. SHOTTS, JR.
JANUARY 2012, 480 PP., \$39.95
ISBN 978-1-59327-389-7



JAVASCRIPT FOR KIDS

A Playful Introduction to Programming

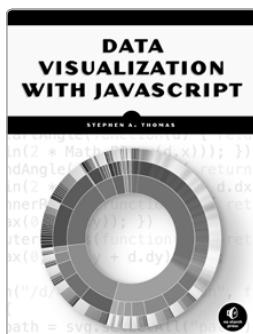
by NICK MORGAN
DECEMBER 2014, 336 PP., \$34.95
ISBN 978-1-59327-408-5
full color



STATISTICS DONE WRONG

The Woefully Complete Guide

by ALEX REINHART
MARCH 2015, 176 PP., \$24.95
ISBN 978-1-59327-620-1



DATA VISUALIZATION WITH JAVASCRIPT

by STEPHEN A. THOMAS
MARCH 2015, 384 PP., \$39.95
ISBN 978-1-59327-605-8
full color

PHONE:
800.420.7240 OR
415.863.9900

EMAIL:
SALES@NOSTARCH.COM
WEB:
WWW.NOSTARCH.COM

**LEARN PYTHON.
GET STUFF DONE.**



If you've ever spent hours renaming files or updating hundreds of spreadsheet cells, you know how tedious tasks like these can be. But what if you could have your computer do them for you?

In *Automate the Boring Stuff with Python*, you'll learn how to use Python to write programs that do in minutes what would take you hours to do by hand—no prior programming experience required. Once you've mastered the basics of programming, you'll create Python programs that effortlessly perform useful and impressive feats of automation to:

- Search for text in a file or across multiple files
- Create, update, move, and rename files and folders
- Search the Web and download online content
- Update and format data in Excel spreadsheets of any size
- Split, merge, watermark, and encrypt PDFs

- Send reminder emails and text notifications
- Fill out online forms

Step-by-step instructions walk you through each program, and practice projects at the end of each chapter challenge you to improve those programs and use your newfound skills to automate similar tasks.

Don't spend your time doing work a well-trained monkey could do. Even if you've never written a line of code, you can make your computer do the grunt work. Learn how in *Automate the Boring Stuff with Python*.

ABOUT THE AUTHOR

Al Sweigart is a software developer and teaches programming to kids and adults. He has written several Python books for beginners, including *Hacking Secret Ciphers with Python*, *Invent Your Own Computer Games with Python*, and *Making Games with Python & Pygame*.

COVERS PYTHON 3



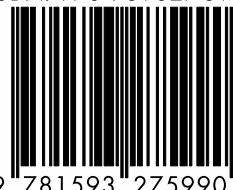
THE FINEST IN GEEK ENTERTAINMENT™
www.nostarch.com

"I LIE FLAT."

This book uses a durable binding that won't snap shut.



ISBN: 978-1-59327-599-0



\$29.95 (\$34.95 CDN)



SHELF IN:
PROGRAMMING LANGUAGES/
PYTHON