

## Web Scrapping

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. Python offers several modules that make it easy to scrape web pages.

- webbrowser Comes with Python and opens a browser to a specific page.
- Requests Downloads files and web pages from the Internet.
- BeautifulSoup 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.

### Webbrowser module

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

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

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).

```
import webbrowser, sys, pyperclip
```

```
if len(sys.argv) > 1:
```

```
    # Get address from command line.
```

```
    address = ' '.join(sys.argv[1:])
```

```
else:
```

```
    # Get address from clipboard.
```

```
    #givenadd= '870 Valencia St'
```

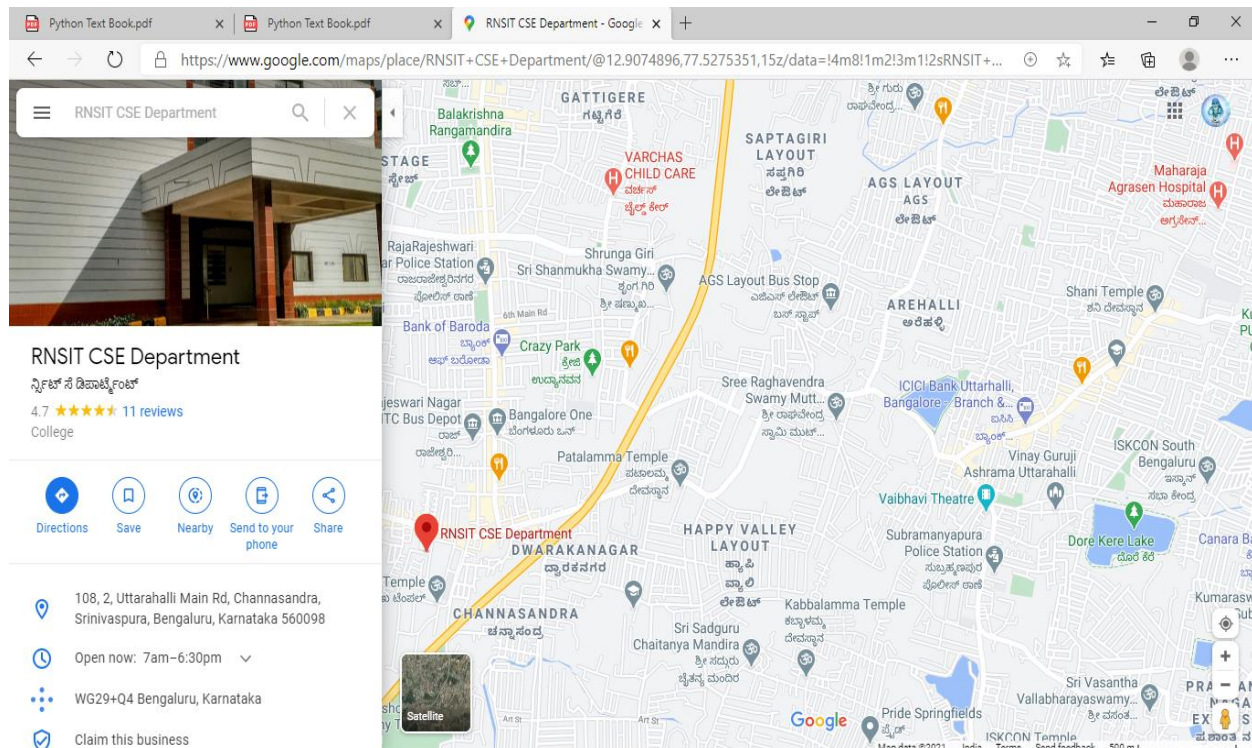
```
    givenadd='RNSIT CSE Department'
```

```
    pyperclip.copy(givenadd)
```

```
    address = pyperclip.paste()
```

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

O/P



**Table 1 compares the steps needed to display 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 <a href="http://maps.google.com/">http://maps.google.com/</a> .	
Click the address text field.	
Paste the address.	
Press ENTER.	

## 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`.

- `pip install requests`
- `import requests`

```

Administrator: Command Prompt
operable program or batch file.

C:\Users\home\AppData\Local\Programs\Python\Python38-32>cd Scripts

C:\Users\home\AppData\Local\Programs\Python\Python38-32\Scripts>dir
Volume in drive C has no label.
Volume Serial Number is FE52-D858

Directory of C:\Users\home\AppData\Local\Programs\Python\Python38-32\Scripts

02/05/2020  03:38 PM  <DIR>          .
02/05/2020  03:38 PM  <DIR>          ..
02/05/2020  03:38 PM                93,077 easy_install-3.8.exe
02/05/2020  03:38 PM                93,077 easy_install.exe
02/05/2020  03:38 PM                93,059 pip.exe
02/05/2020  03:38 PM                93,059 pip3.8.exe
02/05/2020  03:38 PM                93,059 pip3.exe
02/05/2020  03:38 PM                5 File(s)      465,331 bytes
                                2 Dir(s)      187,723,472,896 bytes free

C:\Users\home\AppData\Local\Programs\Python\Python38-32\Scripts>pip install requests
Collecting requests
  Downloading https://files.pythonhosted.org/packages/29/c1/24814557f1d22c56d50280771a17307e6bf87b70727d975fd6b2ce6b014a/requests-2.25.1-py2.py3-none-any.whl (61kB)
    |#####| 61kB 975kB/s
Collecting urllib3<1.27,>=1.21.1 (from requests)
  Downloading https://files.pythonhosted.org/packages/f5/71/45d36a8df68f3ebb098d6861b2c017f3d094538c0fb98fa61d4dc43e69b9/urllib3-1.26.2-py2.py3-none-any.whl (136kB)
    |#####| 143kB 1.0MB/s
Collecting chardet<5,>=3.0.2 (from requests)
  Downloading https://files.pythonhosted.org/packages/19/c7/fa589626997dd07bd87d9269342ccb74b1720384a4d739a1872bd84f6e68/chardet-4.0.0-py2.py3-none-any.whl (178kB)
    |#####| 184kB 2.2MB/s
Collecting idna<3,>=2.5 (from requests)
  Downloading https://files.pythonhosted.org/packages/a2/38/928ddce2273eaa564f6f50de919327bf3a00f091b5baba8dfa9460f3a8a8/idna-2.10-py2.py3-none-any.whl (58kB)
    |#####| 61kB 3.8MB/s
Collecting certifi<2017.4.17 (from requests)
  Downloading https://files.pythonhosted.org/packages/5e/a0/5f06e1e1d463903cf0c0eebeb751791119ed7a4b3737fdc9a77f1cdfb51f/certifi-2020.12.5-py2.py3-none-any.whl (147kB)
    |#####| 153kB 2.2MB/s
Installing collected packages: urllib3, chardet, idna, certifi, requests
  WARNING: The script chardet.exe is installed in 'c:\users\home\appdata\local\programs\python\python38-32\Scripts' which is not on PATH.
  Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn-script-location.
Successfully installed certifi-2020.12.5 chardet-4.0.0 idna-2.10 requests-2.25.1 urllib3-1.26.2
WARNING: You are using pip version 19.2.3, however version 20.3.3 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.

C:\Users\home\AppData\Local\Programs\Python\Python38-32\Scripts>

```

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.

```
import requests
```

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

```
print(type(res))
```

```
print(res.status_code == requests.codes.ok)
```

```
print(len(res.text))
```

```
print(res.text[:250])
```

**O/P**

```
= RESTART: C:\Users\home\AppData\Local\Programs\Python\Python38-32\requestmoduleuse.py
```

```
<class 'requests.models.Response'>
```

```
True
```

```
178978
```

```
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
```

## Checking for Errors

- 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.

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

```
>>> res.raise_for_status()
```

**o/p**

```
>>> res.raise_for_status()
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    res.raise_for_status()
  File "C:\Users\home\AppData\Local\Programs\Python\Python38-32\lib\site-packages\requests\models.py", line 943, in raise_for_status
    raise HTTPError(http_error_msg, response=self)
requests.exceptions.HTTPError: 404 Client Error: Not Found for url: http://inventwithpython.com/page_that_does_not_exist
```

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))
```

**o/p**

```
===== RESTART: C:\Users\home\AppData\Local\Programs\Python\Python38-32\request2.py =====
There was a problem: 404 Client Error: Not Found for url: http://inventwithpython.com/page_that_does_not_exist
```

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

You can save the web page to a file on your hard drive with the standard `open()` function and `write()` method. First, you must open the file in write binary mode by passing the string `'wb'` as the second argument to `open()`. Even if the page is in plaintext, you need to write binary data instead of text data in order to maintain the Unicode encoding of the text.

The **Unicode** Standard provides a unique number for every character, no matter what platform, device, application or language. It has been adopted by all modern software providers and now allows data to be transported through many different platforms, devices and applications without corruption.

```
>>> import requests

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

>>> res.raise_for_status()

>>> playFile = open('RomeoAndJuliet.txt', 'wb')

>>> for chunk in res.iter_content(100000):

    playFile.write(chunk)

100000

78981

>>> playFile.close()
```

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

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.

## 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.

ensure that the `requests` module doesn't eat up too much memory even if you download massive files.

## Hypertext Markup Language (HTML)

HTML is the format that web pages are written in. An HTML file is a plaintext file with the .html file extension. The text in these files is surrounded by tags, which are words enclosed in angle brackets. The tags tell the browser how to format the web page. A starting tag and closing tag can enclose some text to form an element. The text (or inner HTML) is the content between the starting and closing tags.

**<strong>Hello</strong> world!**

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

The closing `</strong>` tag 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.

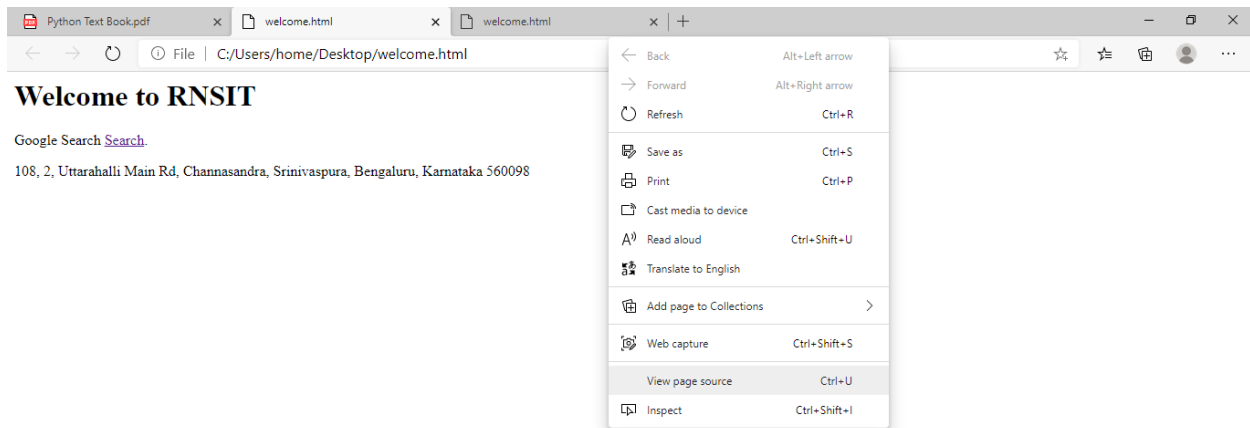
**Search `<a href="http://google.com">` Google Search`</a>`.**

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.

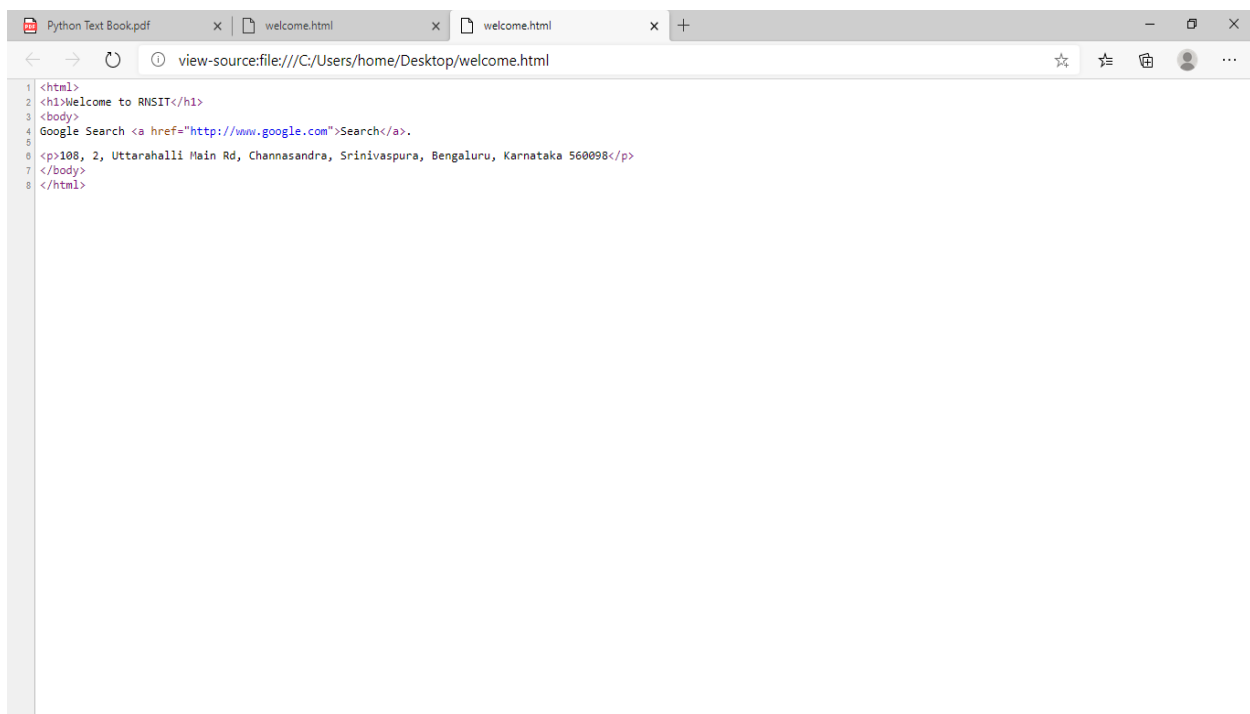
Some elements have an id attribute that is used to uniquely identify the element in the page. You will often instruct your programs to seek out an element by its id attribute, so figuring out an element's id attribute using the browser's developer tools is a common task in writing web scraping programs.



## Viewing the Source HTML of a Web Page



This is the text your browser actually receives. The browser knows how to display, or render, the web page from this HTML

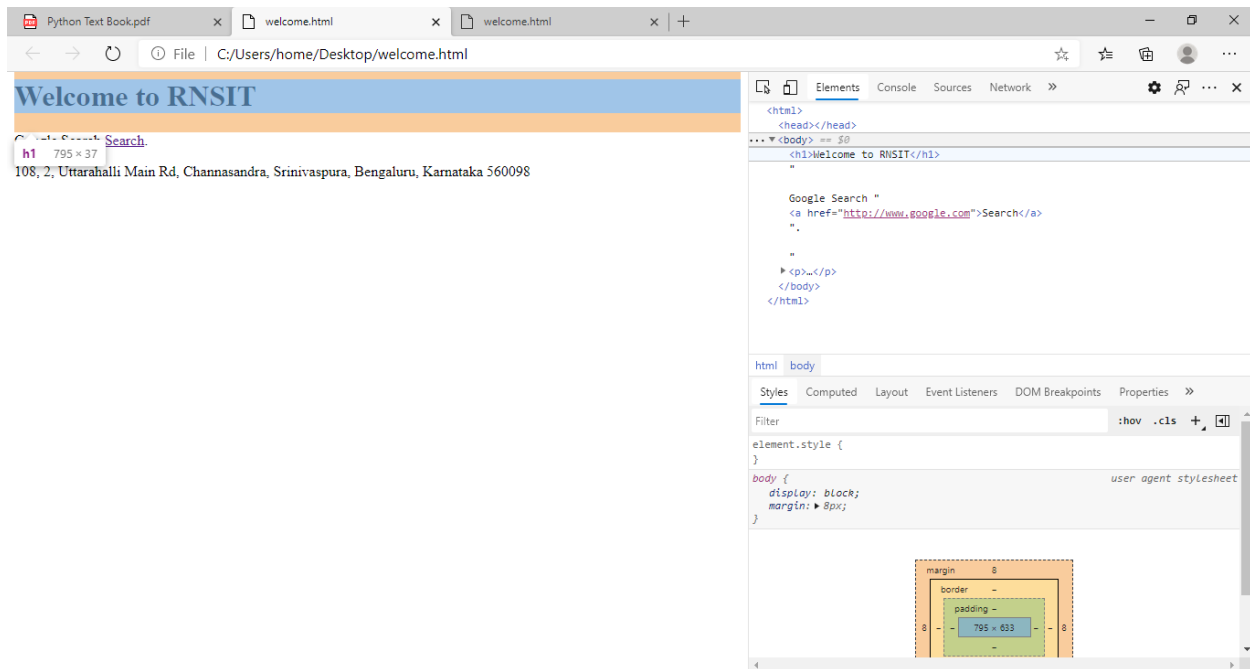


**To perform web scraping you just need enough knowledge to pick out data from an existing site.**



## 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. 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.



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

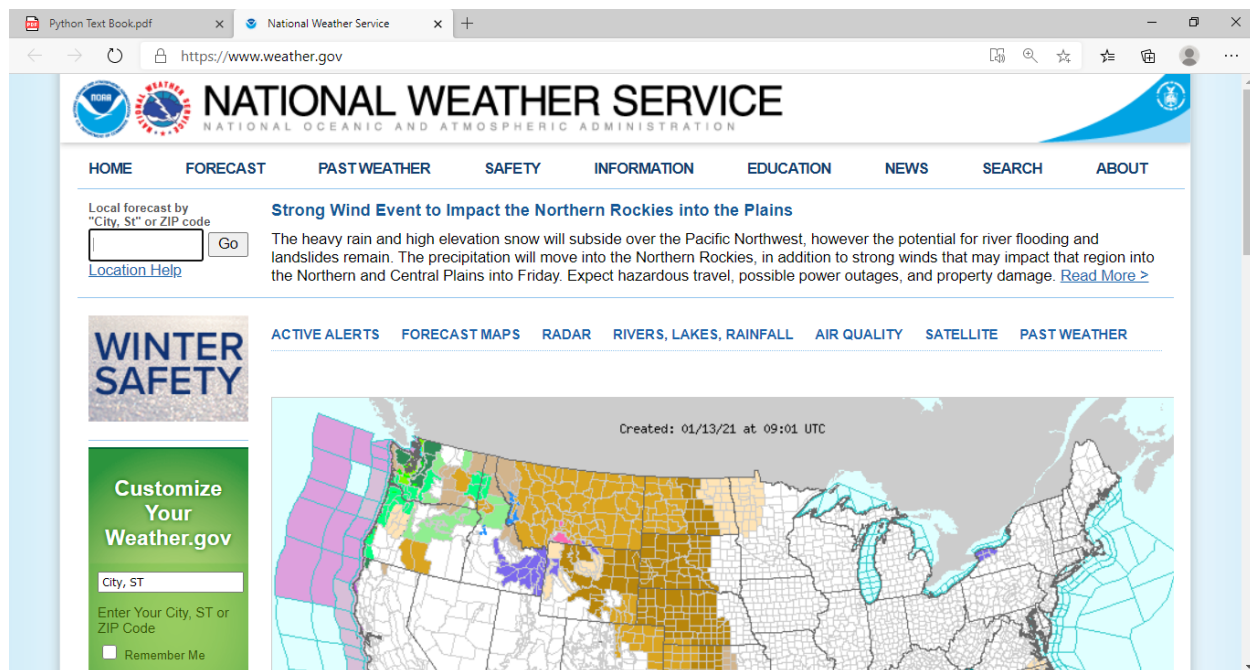
## Using the Developer Tools to Find HTML Elements

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.

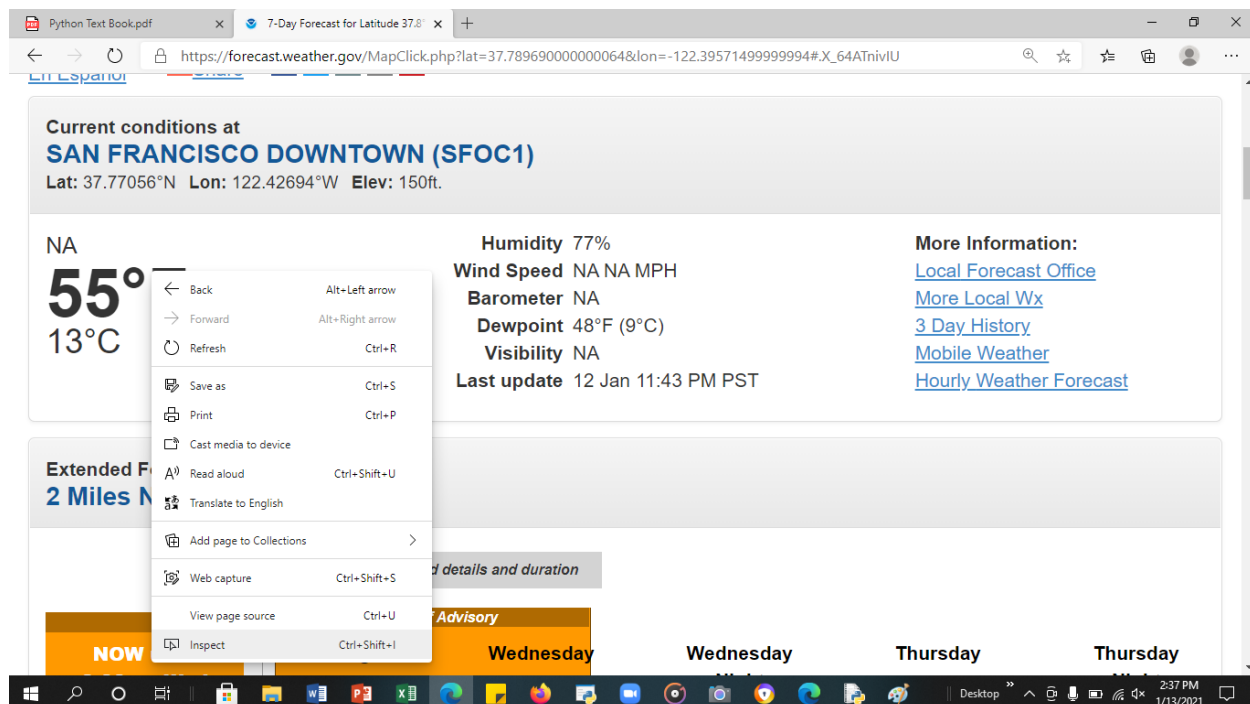
**Write a program to pull weather forecast data from <http://weather.gov/>**

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.

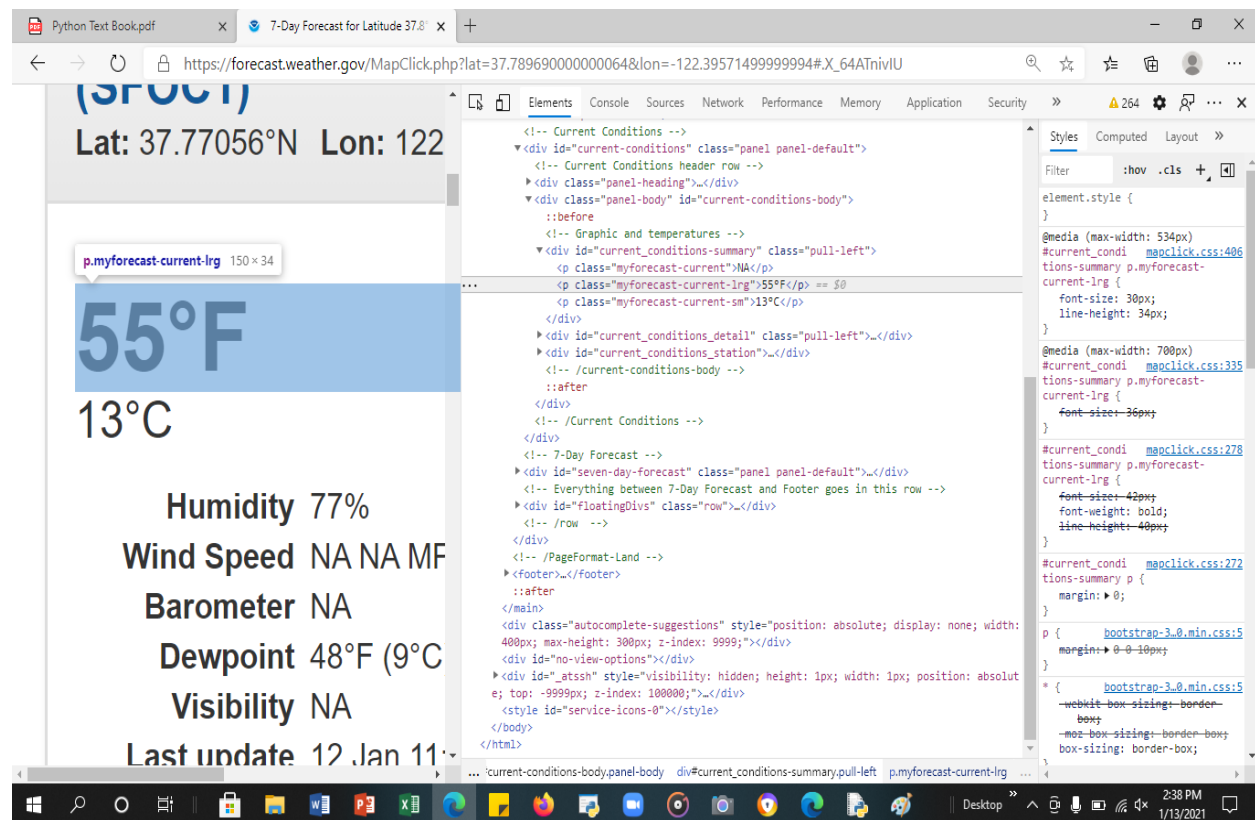




What if you're interested in scraping the temperature information for that ZIP code? Right-click where it is on the page and select Inspect Element



This will bring up the Developer Tools window, which shows you the HTML that produces this particular part of the web page



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">59°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. **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 .
- 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.
- While `beautifulsoup4` is the name used for installation, to import Beautiful Soup you run `import bs4`

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

The example.html file

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

```
<html><head><title>The Website Title</title></head>
```

```
<body>
```

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



## Creating a 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.

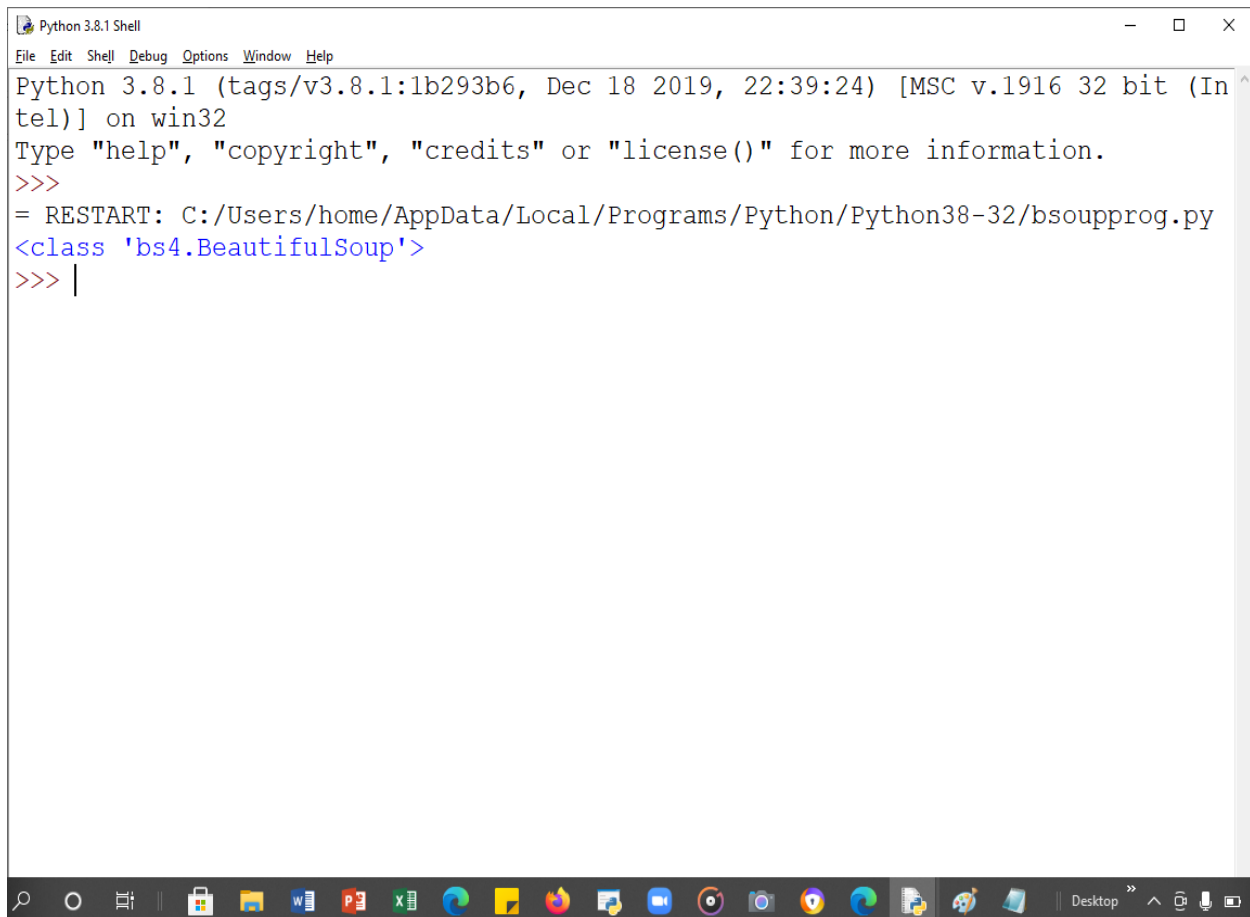
```
import requests, bs4
res = requests.get('http://nostarch.com')
print(res)
res.raise_for_status()
noStarchSoup = bs4.BeautifulSoup(res.text, features="html.parser")
print(type(noStarchSoup))
```

```
===== RESTART: C:/Users/home/AppData/Local/Programs/Python/Python38-32/bsoupprog.py =====
<Response [200]>
<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()`. make sure the `example.html` file is in the working directory. Once you have a BeautifulSoup object, you can use its methods to locate specific parts of an HTML document.

```
import requests, bs4
exampleFile = open('example.html')
exampleSoup = bs4.BeautifulSoup(exampleFile,features="html.parser")
print(type(exampleSoup))
```



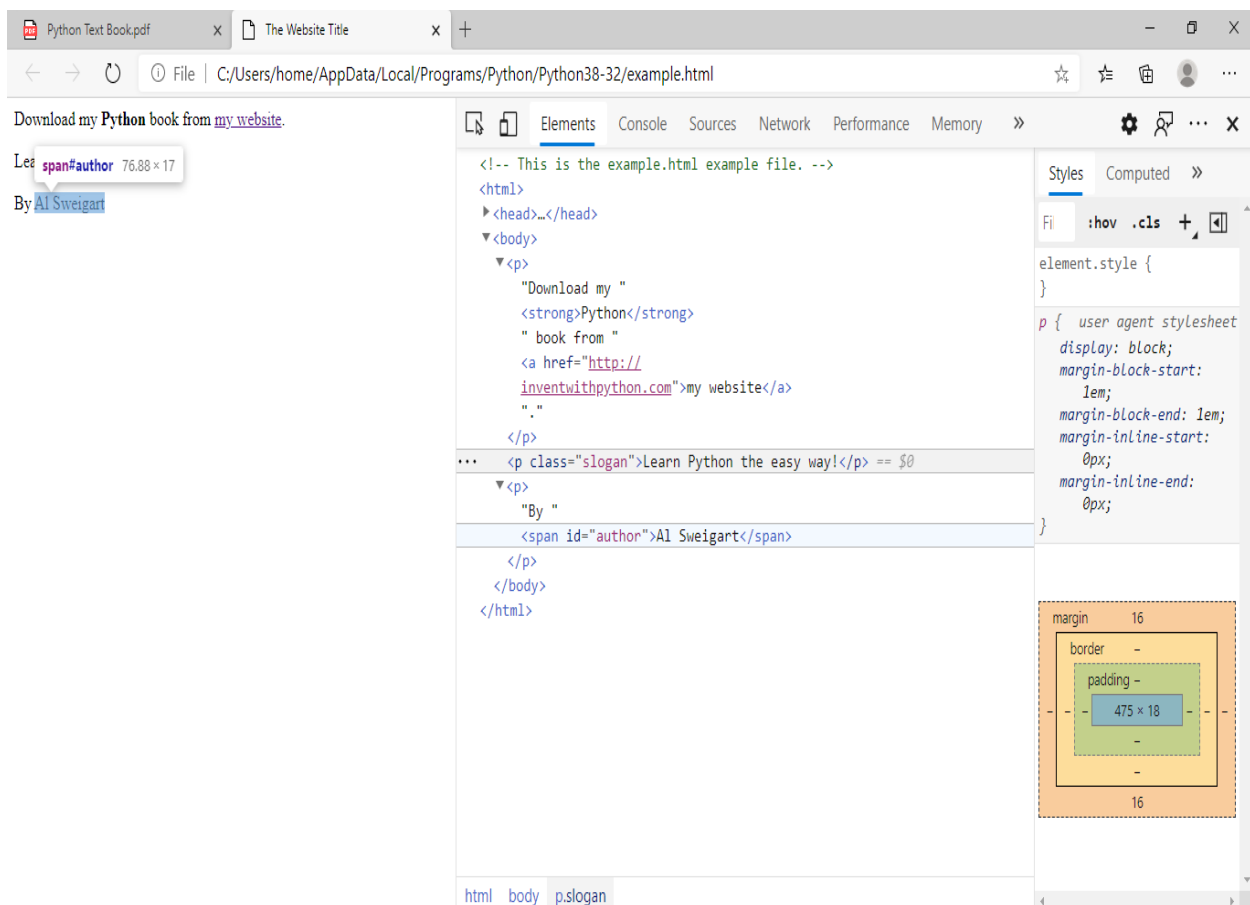
```
Python 3.8.1 Shell
File Edit Shell Debug Options Window Help
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:39:24) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:/Users/home/AppData/Local/Programs/Python/Python38-32/bsoupprog.py
<class 'bs4.BeautifulSoup'>
>>> |
```

## 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.
- The various selector patterns can be combined to make sophisticated matches.

## Examples of CSS Selectors

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



- `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 BeautifulSoup 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.

```
import bs4

exampleFile = open('example.html')

exampleSoup = bs4.BeautifulSoup(exampleFile.read(),features="html.parser")

elems = exampleSoup.select('#author')

print(type(elems))

print(len(elems))

print(type(elems[0]))

print(elems[0].getText())

print(str(elems[0]))

print(elems[0].attrs)
```

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'

```
= RESTART: C:/Users/home/AppData/Local/Programs/Python/Python38-32/bsoupselect.p
Y
<class 'bs4.element.ResultSet'>
1
<class 'bs4.element.Tag'>
Al Sweigart
<span id="author">Al Sweigart</span>
{'id': 'author'}
```

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'.

### **Pull all the <p> elements from the BeautifulSoup object**

```
import bs4

exampleFile = open('example.html')

exampleSoup = bs4.BeautifulSoup(exampleFile.read(),features="html.parser")

pElems = exampleSoup.select('p')

print(str(pElems[0]))

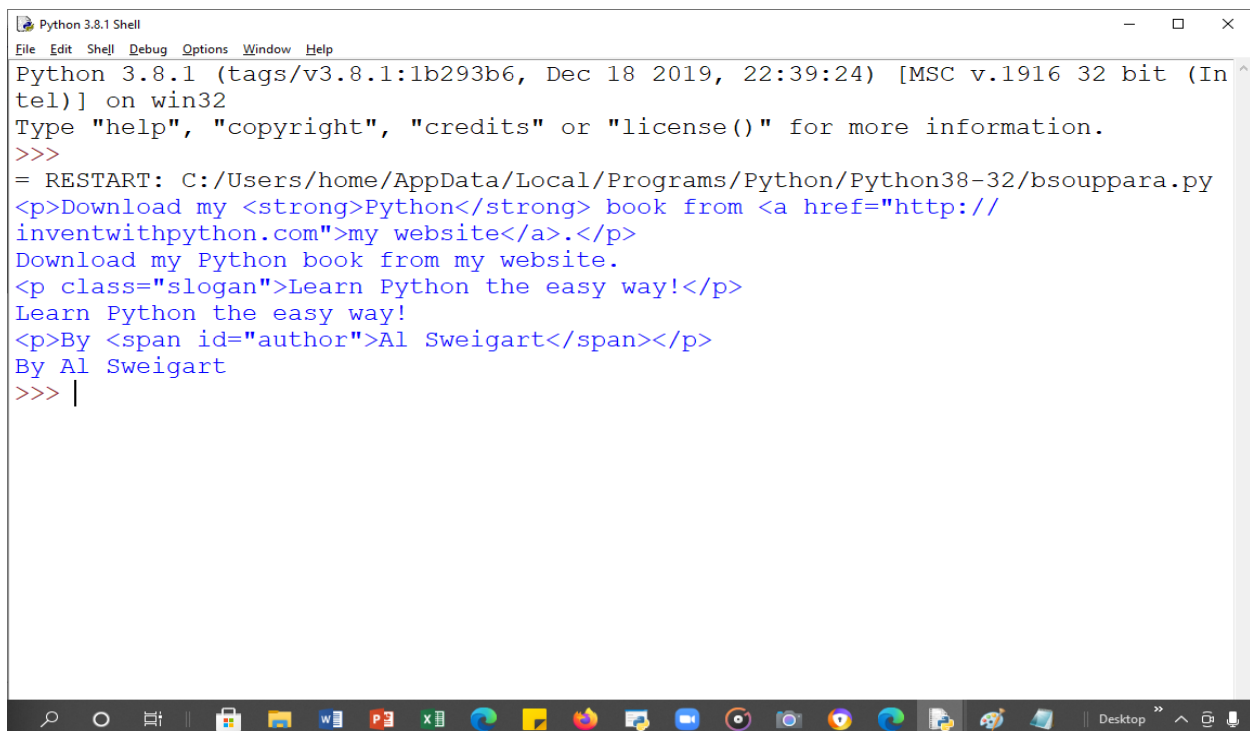
print(pElems[0].getText())

print(str(pElems[1]))

print(pElems[1].getText())

print(str(pElems[2]))

print(pElems[2].getText())
```



```
Python 3.8.1 Shell
File Edit Shell Debug Options Window Help
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:39:24) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:/Users/home/AppData/Local/Programs/Python/Python38-32/bsouppara.py
<p>Download my <strong>Python</strong> book from <a href="http://
inventwithpython.com">my website</a>.</p>
Download my Python book from my website.
<p class="slogan">Learn Python the easy way!</p>
Learn Python the easy way!
<p>By <span id="author">Al Sweigart</span></p>
By Al Sweigart
>>> |
```

his 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. 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'.

```
import bs4

soup = bs4.BeautifulSoup(open('example.html'))

spanElem = soup.select('span')[0]

print(str(spanElem))

print(spanElem.get('id'))

print(spanElem.get('some_nonexistent_addr') == None)

print(spanElem.attrs)

o/p

<span id="author">Al Sweigart</span>
author
True
{'id': 'author'}
```

## 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.

### Starting a Selenium-Controlled Browser

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/>.

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

```

Command Prompt
Microsoft Windows [Version 10.0.18363.1256]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\home>cd..
C:\Users>cd..
C:\>cd Users\home\AppData\Local\Programs\Python\Python38-32
C:\Users\home\AppData\Local\Programs\Python\Python38-32>cd Scripts
C:\Users\home\AppData\Local\Programs\Python\Python38-32\Scripts>pip install selenium
Collecting selenium
  Downloading https://files.pythonhosted.org/packages/80/d6/4294f0b4bce4de0abf13e17190289f9d0613b0a44e5dd6a7f5ca98459853/selenium-3.141.0-py2.py3-none-any.whl (904kB)
    |#####| 911kB 1.3MB/s
Requirement already satisfied: urllib3 in c:\users\home\AppData\Local\Programs\Python\Python38-32\lib\site-packages (from selenium) (1.26.2)
Installing collected packages: selenium
Successfully installed selenium-3.141.0
WARNING: You are using pip version 19.2.3, however version 20.3.3 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.
C:\Users\home\AppData\Local\Programs\Python\Python38-32\Scripts>

```

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/>.



### Finding Elements on the Page

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.

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 text 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 text 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.

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 name attribute
<code>text</code>	The text within the element, such as 'hello' in <span>hello</span>
<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

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

**o/p**

Found <img> element with that class name!

## Clicking the Page

WebElement objects returned from the find\_element\_\* and find\_elements\_\* methods have a click() method that simulates a mouse click on that element. This method can be used to follow a link, make a selection on a radio button, click a Submit button, or trigger whatever else might happen when the element is clicked by the mouse. 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
```

## 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.

```
>>> from selenium import webdriver
>>> browser = webdriver.Firefox()
>>> browser.get('https://mail.yahoo.com')
>>> emailElem = browser.find_element_by_id('login-username')
>>> emailElem.send_keys('not_my_real_email')
>>> passwordElem = browser.find_element_by_id('login-passwd')
>>> passwordElem.send_keys('12345')
>>> passwordElem.submit()
```

Calling the submit() method on any element will have the same result as clicking the Submit button for the form that element is in.

## 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.

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

Commonly Used Variables in the selenium.webdriver.common.keys Module

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

```
>>> 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

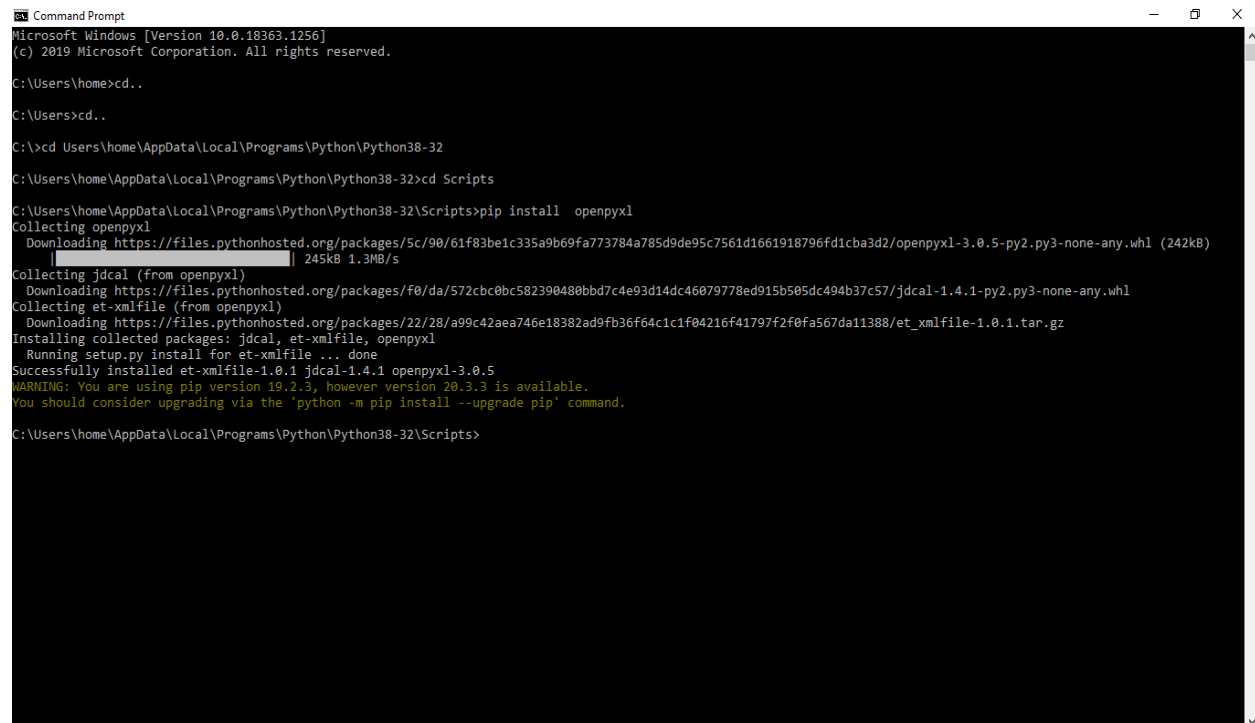
Selenium can do much more beyond the functions described here. It can modify your browser's cookies, take screenshots of web pages, and run custom JavaScript. To learn more about these features, you can visit the **Selenium documentation** at <http://selenium-python.readthedocs.org/>

## Working with Excel Spread Sheets

Excel is a popular and powerful spreadsheet application for Windows. The openpyxl module allows your Python programs to read and modify Excel spreadsheet files.

<http://openpyxl.readthedocs.org/>

### Installation of openpyxl module



```
Microsoft Windows [Version 10.0.18363.1256]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\home>cd..

C:\Users>cd..

C:\>cd Users\home\AppData\Local\Programs\Python\Python38-32

C:\Users\home\AppData\Local\Programs\Python\Python38-32>cd Scripts

C:\Users\home\AppData\Local\Programs\Python\Python38-32\Scripts>pip install openpyxl
Collecting openpyxl
  Downloading https://files.pythonhosted.org/packages/5c/90/61f83be1c335a9b69fa773784a785d9de95c7561d1661918796fd1c3a3d2/openpyxl-3.0.5-py2.py3-none-any.whl (242kB)
    |#####| 245kB 1.3MB/s
Collecting jdcal (from openpyxl)
  Downloading https://files.pythonhosted.org/packages/f0/da/572cbc0bc582398480bbd7c4e93d14dc46079778ed915b505dc494b37c57/jdcal-1.4.1-py2.py3-none-any.whl
Collecting et_xmlfile (from openpyxl)
  Downloading https://files.pythonhosted.org/packages/22/28/a99c42aea746e18382ad9fb36f64c1c1f04216f41797f2f0fa567da11388/et_xmlfile-1.0.1.tar.gz
Installing collected packages: jdcal, et_xmlfile, openpyxl
  Running setup.py install for et_xmlfile ... done
Successfully installed et_xmlfile-1.0.1 jdcal-1.4.1 openpyxl-3.0.5
WARNING: You are using pip version 19.2.3, however version 20.3.3 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.

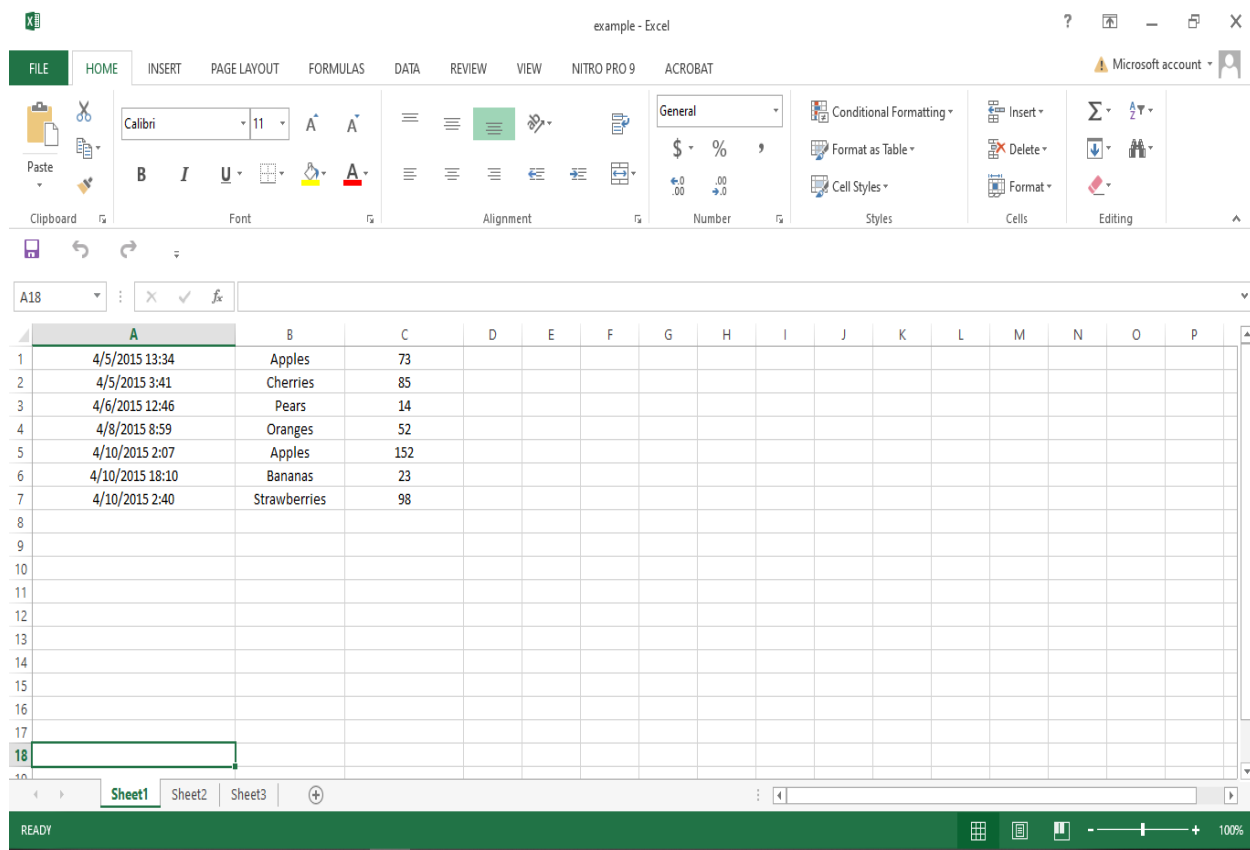
C:\Users\home\AppData\Local\Programs\Python\Python38-32\Scripts>
```

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.

### Excel Documents

- An Excel spreadsheet document is called a workbook. A single workbook is saved in a file with the .xlsx extension. Each workbook can contain multiple sheets (also called worksheets). The sheet the user is currently viewing (or last viewed before closing Excel) is called the active sheet. Each sheet has columns (addressed by letters starting at A) and rows (addressed by numbers starting at 1).

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



## Opening Excel Documents with OpenPyXL

```
>>> 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.

```
>>> 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.

```
>>> 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
```

```
import openpyxl
import os
os.chdir('E:\\')
workbook=openpyxl.load_workbook('example.xlsx')
print(type(workbook))
sheet=workbook.get_sheet_by_name('Sheet1')
print(type(sheet))
sheetx=workbook['Sheet1']
print(type(sheetx))
print(type(sheet))
print(workbook['Sheet1'])
print(workbook.get_sheet_names())
print(workbook.sheetnames)
print(sheet['A1'])
cell=sheet['A1']
print(cell.value)
print(sheet['B1'].value)
print(sheet['C1'].value)
sheet.cell(row=1,column=2)
for i in range(1,8):
    print(i,sheet.cell(row=i,column=2).value)
for i in range(1,8):
    print(i,sheet.cell(row=i,column=1).value)
for i in range(1,8):
    print(i,sheet.cell(row=i,column=3).value)
```



```
Python 3.8.1 Shell
File Edit Shell Debug Options Window Help
> print(workbook.get_sheet_names())
['Sheet1', 'Sheet2', 'Sheet3']
> print(workbook.sheetnames)
['Sheet1', 'Sheet2', 'Sheet3']
> print(sheet['A1'])
<Cell 'Sheet1'.A1>
2015-04-05 13:34:02
Apples
73
1 Apples
2 Cherries
3 Pears
4 Oranges
5 Apples
6 Bananas
7 Strawberries
1 2015-04-05 13:34:02
2 2015-04-05 03:41:23
3 2015-04-06 12:46:51
4 2015-04-08 08:59:43
5 2015-04-10 02:07:00
6 2015-04-10 18:10:37
7 2015-04-10 02:40:46
1 73
2 85
3 14
4 52
Ln: 152 Col: 0
```

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.

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.

```
>>> 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
```

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.

You can determine the size of the sheet with the Worksheet object's get\_highest\_row() and get\_highest\_column() methods.

```
>>> 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
```

### 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.

```
>>> 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
```

### 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.

```
import openpyxl
wb = openpyxl.load_workbook('example.xlsx')
sheet = wb.get_sheet_by_name('Sheet1')
print(tuple(sheet['A1':'C3']))
for rowOfCellObjects in sheet['A1':'C3']:
    for cellObj in rowOfCellObjects:
        print(cellObj.coordinate, cellObj.value)
print('--- END OF ROW ---')
```

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

- 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. Then, for each row, the nested for loop goes through each cell in that row

To access the values of cells in a particular row or column, you can also use a Worksheet object's rows and columns attribute.

```
>>> 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 cellObj in sheet.columns[1]:
    print(cellObj.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

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.

## 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.

```
>>> 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.

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

```
>>> wb.save('example_copy.xlsx')
```

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.

```
>>> 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.

```
>>> 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.

```
>>> 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.

## Setting the Font Style of Cells

Styling certain cells, rows, or columns can help you emphasize important areas in your spreadsheet. To customize font styles in cells, import the `Font()` and `Style()` functions from the `openpyxl.styles` module.

```
>>> import openpyxl
>>> from openpyxl.styles import Font, Style
>>> wb = openpyxl.Workbook()
>>> sheet = wb.get_sheet_by_name('Sheet')
>>> italic24Font = Font(size=24, italic=True)
>>> styleObj = Style(font=italic24Font)
>>> sheet['A1'].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`. 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`. And when `styleObj` is assigned to the cell's `style` attribute all that font styling information gets applied to cell A1.

## Font Objects

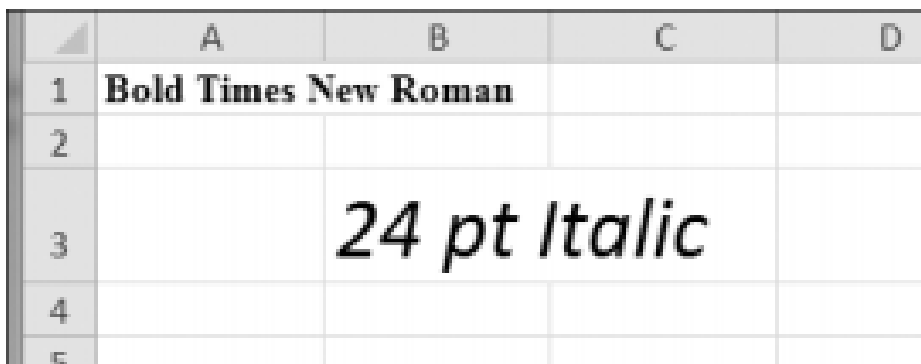
The style attributes in Font objects affect how the text in cells is displayed. To set font style attributes, you pass keyword arguments to Font().

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.

```
>>> 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/styleObj
>>> sheet['A1'] = 'Bold Times New Roman'
>>> fontObj2 = Font(size=24, italic=True)
>>> styleObj2 = Style(font=fontObj2)
>>> sheet['B3'].style/styleObj
>>> sheet['B3'] = '24 pt Italic'
>>> wb.save('styles.xlsx')
```



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

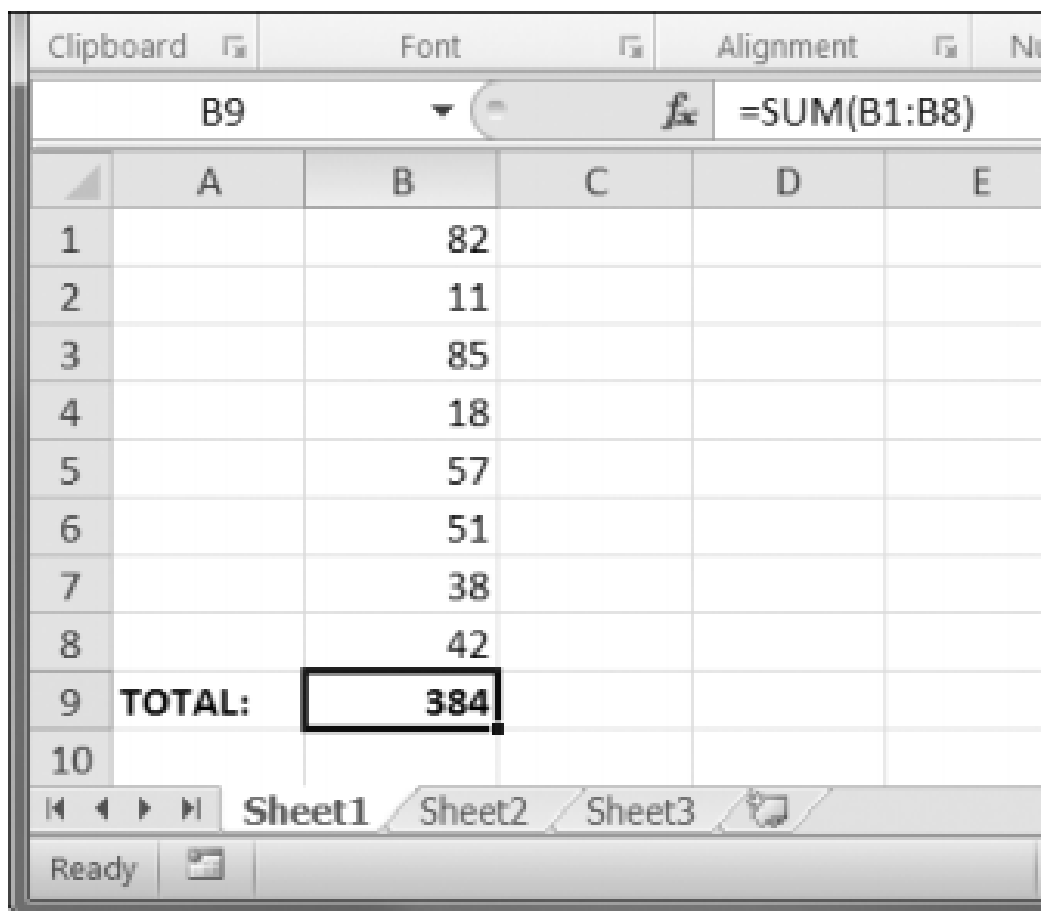
For cell A1, we set the font name to 'Times New Roman' and set bold to true, so our text appears in bold Times New Roman. We didn't specify a size, so the openpyxl default, 11, is used. In cell B3, our text is italic, with a size of 24; we didn't specify a font name, so the openpyxl default, Calibri,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

```
>>> 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.



The screenshot shows a spreadsheet application window. The formula bar at the top displays the formula `=SUM(B1:B8)` for cell B9. The spreadsheet grid shows columns A through E and rows 1 through 10. Column B contains the values 82, 11, 85, 18, 57, 51, 38, and 42 in rows 1 through 8 respectively. Cell B9 is highlighted with a thick black border and contains the value 384. The status bar at the bottom indicates the application is 'Ready'.

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

A formula is set just like any other text value in a cell.

```
>>> 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.

```
>>> 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
```

### Adjusting Rows and Columns

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).

	A	B
1	Tall row	
2		Wide column
3		

*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.

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

*Merged cells in a spreadsheet*

```
>>> 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')
```

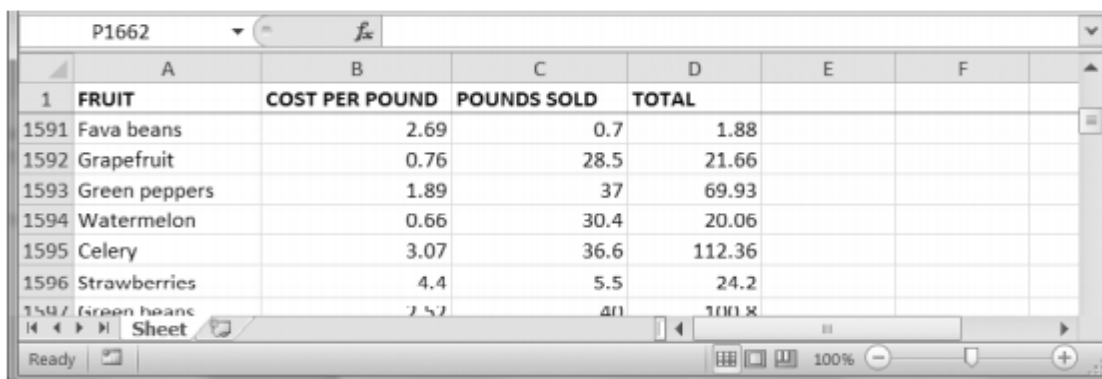
### Freeze Panes

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'.

#### Frozen Pane Examples

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

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



*: With freeze\_panes set to 'A2', row 1 is always visible even as the user scrolls down.*

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
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												
12	Spinach	4.12	30	123.6												
13	Cucumber	1.07	36	38.52												
14	Apricots	3.71	29.4	109.07												
15	Okra	2.26	9.5	21.47												
16	Fava beans	2.69	5.3	14.26												
17	Watermelon	0.66	35.4	23.36												
18	Ginger	5.13	14.4	73.87												
19	Corn	1.07	12.2	13.05												
20	Grapefruit	0.76	35.7	27.12												

## Charts

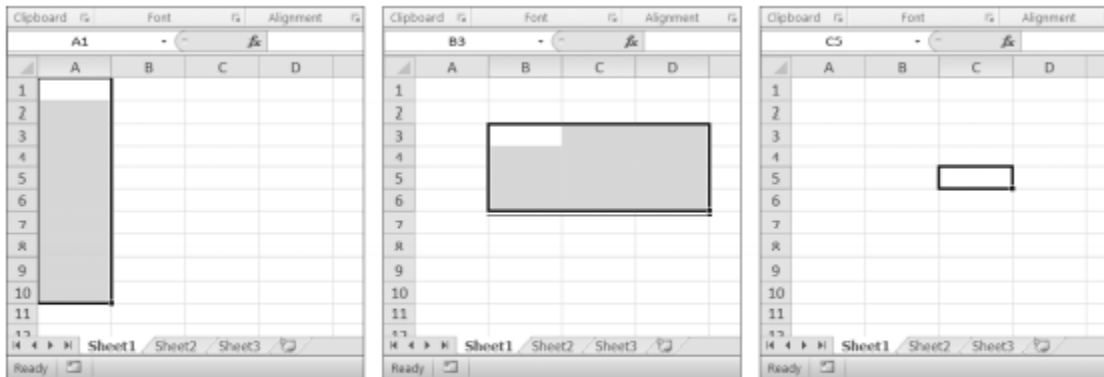
OpenPyXL supports creating bar, line, scatter, and pie charts using the data in a sheet's cells. To make a chart, you need to do the following:

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:

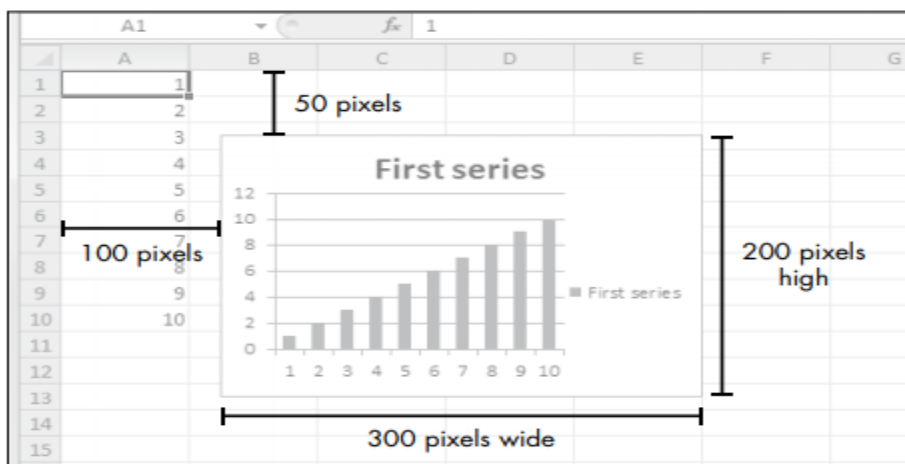
1. The Worksheet object containing your chart data.
2. A tuple of two integers, representing the top-left cell of the rectangular selection of cells containing your chart data: The first integer in the tuple is the row, and the second is the column. Note that 1 is the first row, not 0.
3. A tuple of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





From left to right: (1, 1), (10, 1); (3, 2), (6, 4); (5, 3), (5, 3)

```
>>> 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
>>> refObj = openpyxl.charts.Reference(sheet, (1, 1), (10, 1))
>>> seriesObj = openpyxl.charts.Series(refObj, title='First series')
>>> chartObj = openpyxl.charts.BarChart()
>>> chartObj.append(seriesObj)
>>> 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')
```



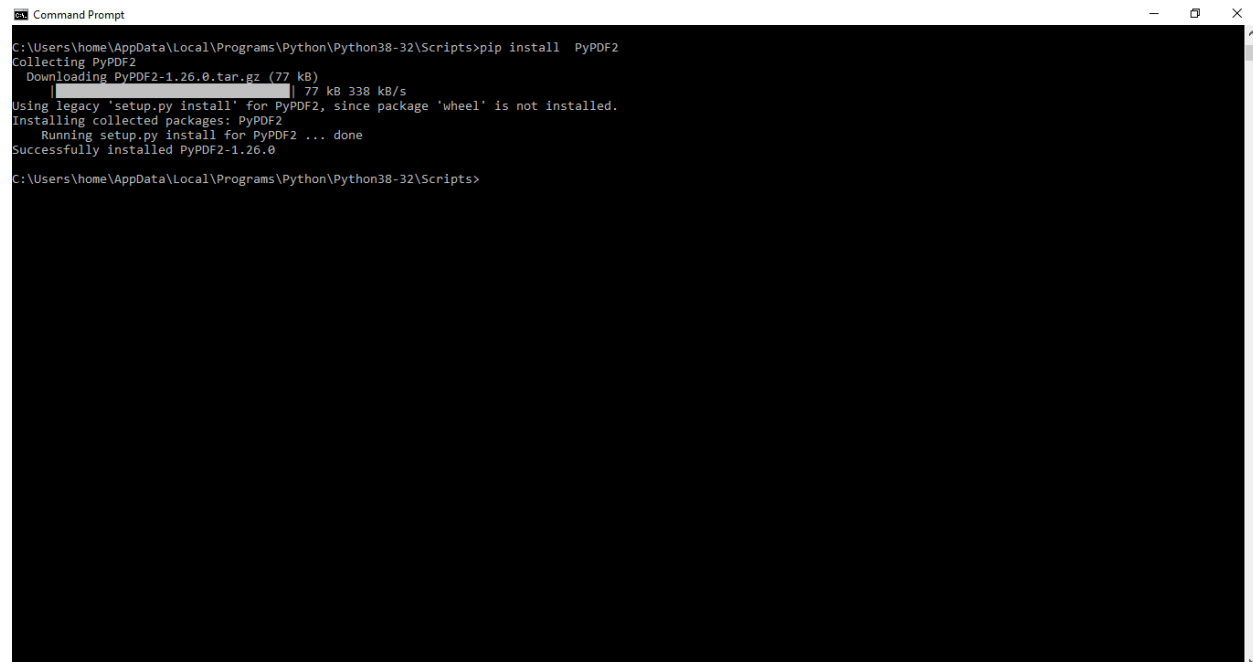
We've created a bar chart by calling `openpyxl.charts.BarChart()`. You can also create line charts, scatter charts, and pie charts by calling `openpyxl.charts.LineChart()`, `openpyxl.charts.ScatterChart()`, and `openpyxl.charts.PieChart()`.

## 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()`.

**PDF Documents** PDF stands for **Portable Document Format** and uses the **.pdf** file extension.

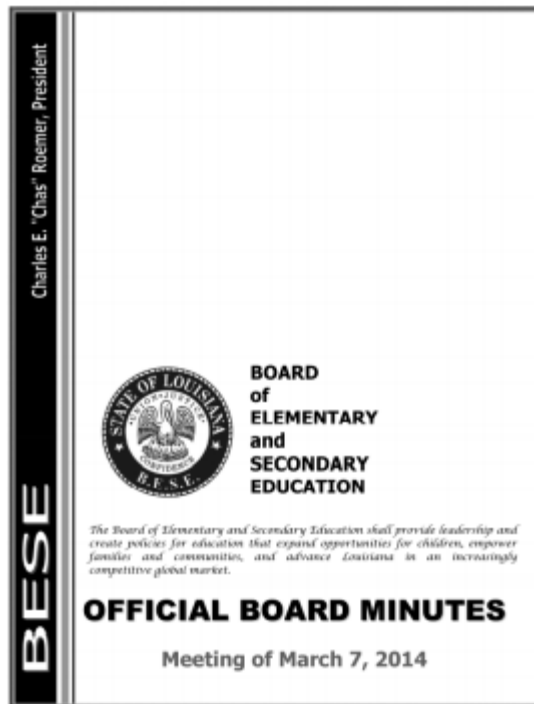
The module you'll use to work with PDFs is PyPDF2. To install it, run `pip install PyPDF2` from the command line. This module name is case sensitive, so make sure the `y` is lowercase and everything else is uppercase.



```
C:\Users\home\AppData\Local\Programs\Python\Python38-32\Scripts>pip install PyPDF2
Collecting PyPDF2
  Downloading PyPDF2-1.26.0.tar.gz (77 kB)
    77 kB 338 kB/s
Using legacy 'setup.py install' for PyPDF2, since package 'wheel' is not installed.
Installing collected packages: PyPDF2
  Running setup.py install for PyPDF2 ... done
Successfully installed PyPDF2-1.26.0
C:\Users\home\AppData\Local\Programs\Python\Python38-32\Scripts>
```

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.

```
>>> import PyPDF2
>>> pdfFileObj = open('meetingminutes.pdf', 'rb')
>>> pdfReader = PyPDF2.PdfFileReader(pdfFileObj)
>>> pdfReader.numPages
19
>>> pageObj = pdfReader.getPage(0)
>>> pageObj.extractText()
```



```
Python 3.8.1 Shell
File Edit Shell Debug Options Window Help
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:39:24) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:/Users/home/AppData/Local/Programs/Python/Python38-32/meetingpdf.py
19
>>> pageObj.extractText()
'OFFFFFFIICIIAALL BBOOAARRDD MMIINNUUTTEESS Meeting of \nMarch 7\n, 2014\n
\n The Board of Elementary and Secondary Education shall provide leadership and \ncreate policies for education that expand opportunities for children , empower \nfamilies and communities, and advance Louisiana in an increasingly \ncompetitive glob\nal market.\n BOARD \n of ELEMENTARY\n and \n SECONDARY\n EDUCATION\n '
>>> |
```

- 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 . 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 on a PdfFileReader object and passing it the page number of the page you're interested in.
- 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 w. 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.

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

```
>>> pdfReader.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.
- Any attempt to call a function that reads the file before it has been decrypted with the correct password will result in an error v.
- To read an encrypted PDF, call the decrypt() function and pass the password as a string w. 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.
- 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.

## 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.

```
>>> import PyPDF2
>>> pdf1File = open('meetingminutes.pdf', 'rb')
>>> pdf2File = open('meetingminutes2.pdf', 'rb')
>>> pdf1Reader = PyPDF2.PdfFileReader(pdf1File)
>>> pdf2Reader = PyPDF2.PdfFileReader(pdf2File)
>>> pdfWriter = PyPDF2.PdfFileWriter()
>>> for pageNum in range(pdf1Reader.numPages):
>>>     pageObj = pdf1Reader.getPage(pageNum)
>>>     pdfWriter.addPage(pageObj)
>>> for pageNum in range(pdf2Reader.numPages):
>>>     pageObj = pdf2Reader.getPage(pageNum)
>>>     pdfWriter.addPage(pageObj)
>>> 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 .
- Call it again and pass it pdf2File to get a PdfFileReader object for meetingminutes2.pdf. Then create a new PdfFileWriter object, which represents a blank PDF document.
- 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 .
- Then pass that Page object to your PdfFileWriter's addPage() method.
- 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.

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

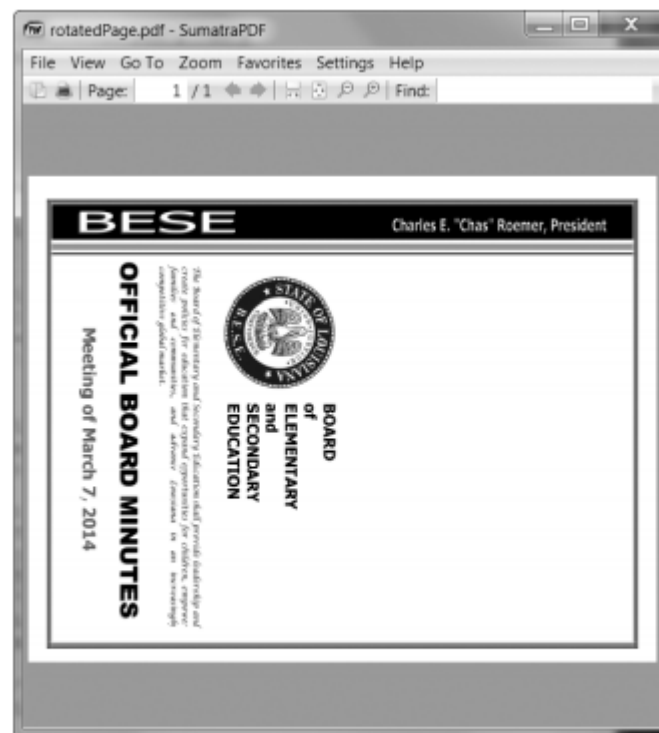
You have now created a new PDF file that combines the pages from meetingminutes.pdf and meetingminutes2.pdf into a single document. Remember that the File object passed to PyPDF2.PdfFileReader() needs to be opened in read-binary mode by passing 'rb' as the second argument to open(). Likewise, the File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

```
>>> import PyPDF2
>>> minutesFile = open('meetingminutes.pdf', 'rb')
>>> pdfReader = PyPDF2.PdfFileReader(minutesFile)
>>> page = pdfReader.getPage(0)
>>> page.rotateClockwise(90)
>>> pdfWriter = PyPDF2.PdfFileWriter()
>>> pdfWriter.addPage(page)
>>> 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 , and then we call rotateClockwise(90) on that page . We write a new PDF with the rotated page and save it as rotatedPage.pdf .



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.

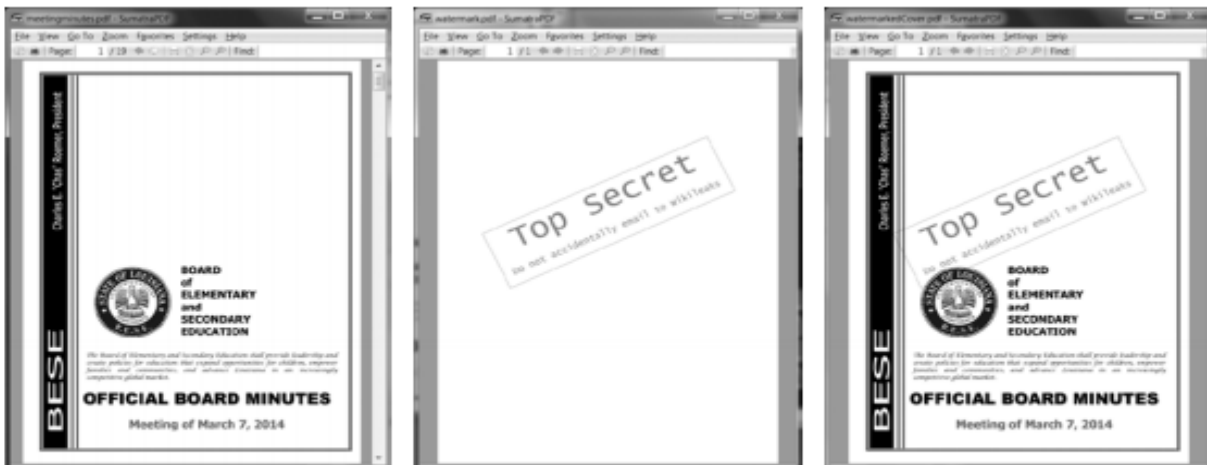
```
>>> import PyPDF2
>>> minutesFile = open('meetingminutes.pdf', 'rb')
❶ >>> pdfReader = PyPDF2.PdfFileReader(minutesFile)
❷ >>> minutesFirstPage = pdfReader.getPage(0)
❸ >>> pdfWatermarkReader = PyPDF2.PdfFileReader(open('watermark.pdf', 'rb'))
❹ >>> minutesFirstPage.mergePage(pdfWatermarkReader.getPage(0))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

❽ >>> 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()
```



- We call `getPage(0)` to get a `Page` object for the first page and store this object in `minutesFirstPage`. We then make a `PdfFileReader` object for `watermark .pdf`
- call `mergePage()` on `minutesFirstPage` .
- 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 and add the watermarked first page .
- Then we loop through the rest of the pages in `meetingminutes.pdf` and add them to the `PdfFileWriter` object .
- Finally, we open a new PDF called `watermarkedCover.pdf` and write the contents of the `PdfFileWriter` to the new PDF.



*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.

```
>>> 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))

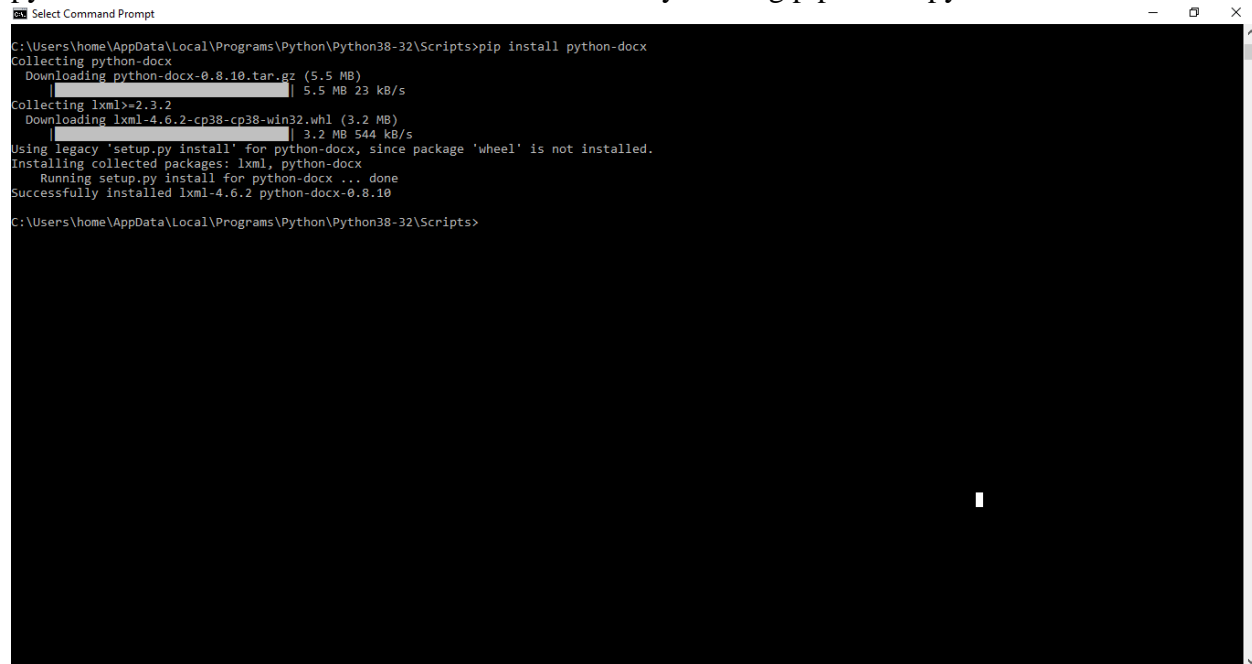
❶ >>> pdfWriter.encrypt('swordfish')
>>> resultPdf = open('encryptedminutes.pdf', 'wb')
>>> pdfWriter.write(resultPdf)
>>> resultPdf.close()
```

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.

## 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`

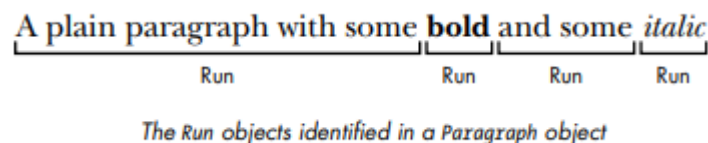


```

C:\Users\home\AppData\Local\Programs\Python\Python38-32\Scripts>pip install python-docx
Collecting python-docx
  Downloading python-docx-0.8.10.tar.gz (5.5 MB)
    |#####| 5.5 MB 23 kB/s
Collecting lxml>=2.3.2
  Downloading lxml-4.6.2-cp38-cp38-win32.whl (3.2 MB)
    |#####| 3.2 MB 544 kB/s
Using legacy 'setup.py install' for python-docx, since package 'wheel' is not installed.
Installing collected packages: lxml, python-docx
  Running setup.py install for python-docx ... done
Successfully installed lxml-4.6.2 python-docx-0.8.10
C:\Users\home\AppData\Local\Programs\Python\Python38-32\Scripts>
  
```

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 has four runs.



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

```

>>> 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
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'

```

1. 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.
2. Each of these Paragraph objects has a `text` attribute that contains a string of the text in that paragraph (without the style information).
3. Here, the first text attribute contains 'DocumentTitle' w, and the second contains 'A plain paragraph with some bold and some italic'

4. 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
5. The first run object contains 'A plain paragraph with some '
6. Then, the text change to a bold style, so 'bold' starts a new Run object
7. The text returns to an unbolded style after that, which results in a third Run object, ' and some
8. Finally, the fourth and last Run object contains 'italic' in an italic style
9. 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.

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

```
>>> 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)

### Styling Paragraph and Run Objects

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'.

### 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. 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.

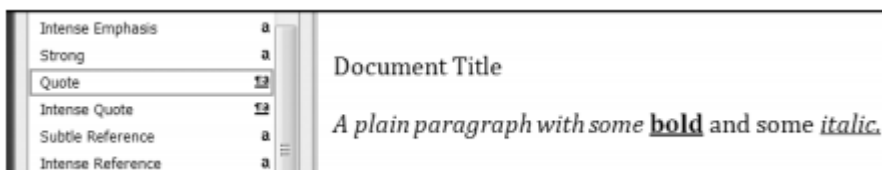
### Run Attributes

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)

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.

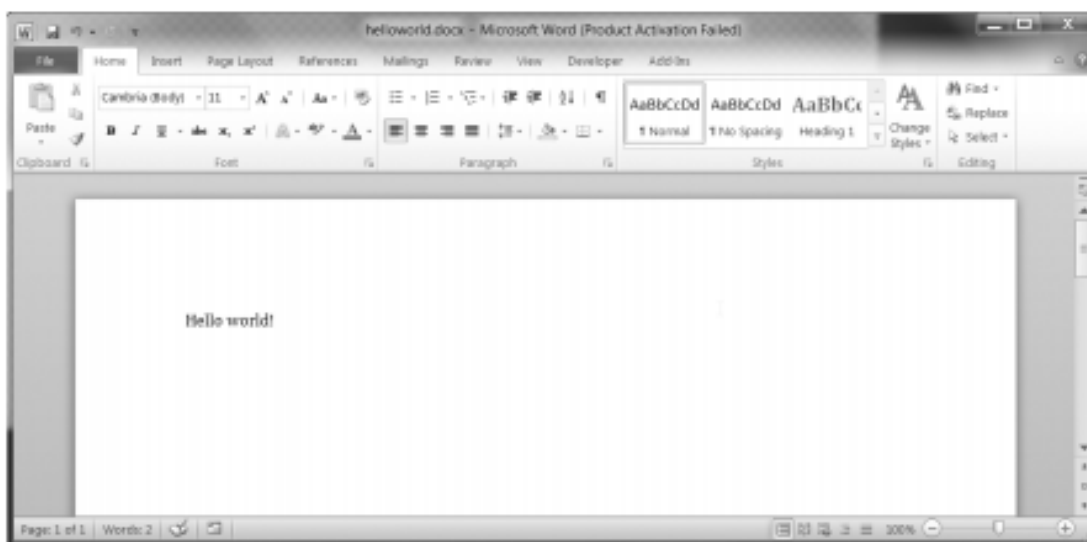
```
>>> doc = docx.Document('demo.docx')
>>> doc.paragraphs[0].text
'Document Title'
>>> doc.paragraphs[0].style
'Title'
>>> doc.paragraphs[0].style = 'Normal'
>>> doc.paragraphs[1].text
'A plain paragraph with some bold and some italic'
>>> (doc.paragraphs[1].runs[0].text, doc.paragraphs[1].runs[1].text,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



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

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



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

```
>>> 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')
```

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

### Adding Headings

Calling `add_heading()` adds a paragraph with one of the heading styles.

```
import docx
doc = docx.Document()
doc.add_heading('Header 0', 0)
doc.add_heading('Header 1', 1)
doc.add_heading('Header 2', 2)
doc.add_heading('Header 3', 3)
doc.add_heading('Header 4', 4)
doc.save('headings.docx')
```

o/p

## Header 0

---

Header 1

Header 2

Header 3

Header 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!")
>>> doc.paragraphs[0].runs[0].add_break(docx.text.WD_BREAK.PAGE)
>>> doc.add_paragraph("This is on the second page!")
>>> doc.save('twoPage.docx')
```

### 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)

```
>>> doc.add_picture('zophie.png', width=docx.shared.Inches(1),
height=docx.shared.Cm(4))
<docx.shape.InlineShape object at 0x00000000036C7D30>
```

The first argument is a string of the image's filename. The optional width and height keyword arguments will set the width and height of the image in the document. If left out, the width and height will default to the normal size of the image. You'll probably prefer to specify an image's height and width in familiar units such as inches and centimeters, so you can use the `docx.shared.Inches()` and `docx.shared.Cm()` functions when you're specifying the width and height keyword arguments.

## Working with CSV Files and JSON Data

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

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

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

### Sample .csv file

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
1	NAICS	NAICS Description	Tax Status	Employer	2012 Revenue	2011 Revenue	2010 Revenue	2012 Coef	2011 Coef	2010 Coef	Coefficient of Variation										
2	561599	All Other Commis	All Establi	Employer	191	179	168	10.6	14.5	14											
3	51222	Integrated Licensing	All Establi	Employer	D	668	791	D	1	0.9											
4	56292	Material R Total oper	All Establi	Employer	S	5,068	5,842	4,854	5.3	5.1	5										
5	6239	Other Res Patient o	All Establi	Employer	S		26	S		29.2	S										
6	62133	Offices of Total oper	All Establi	Employer	S	8,040	7,350	6,702	4.5	4.3	4.4										
7	51112	Periodical Sale or lic	All Establi	Employer	S	506	482	475	13.4	15.2	15.3										
8	56132	Temporar All other c	All Establi	Employer	S	9,400	8,736	7,693	6.1	7.8	7.8										
9	621498	All Other All other r	All Establi	Employer	S	2,010	2,494	2,375	4.5	3.7	4.1										
10	6231	Nursing Ci Patient o	All Establi	Employer	S	11,659	11,298	11,247	2.8	2.7	2.2										
11	51911	News Syn Total oper	All Establi	Employer	S	2,348	2,188	1,960	4.5	3.2	3.1										
12	5415	Computer All other c	All Establi	Employer	S	51,910	46,927	46,931	5.1	4.4	3.4										
13	711211	Sports Tec Total oper	Establishm	Employer	S	22,581	22,421	21,504	4.8	3.8	2.9										
14	51114	Directory All other c	All Establi	Employer	S		545	610	S	11.6	11.1										
15	5622	Waste Tre Nonhazari	All Establi	Employer	S	7,565	6,847	6,623	4.5	2.5	2.4										
16	62132	Offices of Other pati	All Establi	Employer	S	858	832	781	11.8	8.3	8.6										
17	62141	Outpatient Patient o	All Establi	Employer	ZZ	S	S		NA	S											
18	51112	Periodical Periodical	All Establi	Employer	S	14,027	12,790	12,608	7.5	7.4	5.9										
19	711211	Sports Tec Total oper	Establishm	Employer	S	22,581	22,421	21,504	4.8	3.8	2.9										

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

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.

```
>>> import csv
>>> exampleFile = open('example.csv')
>>> exampleReader = csv.reader(exampleFile)
>>> exampleData = list(exampleReader)
>>> 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 without having to install it first.
- To read a CSV file with the csv module, first open it using the open() function 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
- 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().
- 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.

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.

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

### 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.

```
>>> 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']

### Writer Objects

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

```
>>> import csv
>>> outputFile = open('output.csv', 'w', newline='')
>>> 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 u. This will create the object you can then pass to `csv.writer()` v to create a Writer object.

	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							

*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).

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='')
>>> 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'` 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
```

## 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. 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.

## JSON Syntax Rules

JSON syntax is derived from JavaScript object notation syntax:

- Data is in name/value pairs
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays

### JSON Values

In JSON, values must be one of the following data types:

- a string
- a number
- an object (JSON object)
- an array
- a boolean
- null

The following JSON example defines an employee's object, with an array of 3 employees:

```
{"employees":[  
  { "firstName":"Bill", "lastName":"Gates" },  
  { "firstName":"Satya", "lastName":"Nadella" },  
  { "firstName":"Sundar", "lastName":"Pichai" }  
]}
```

### 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 BeautifulSoup.)
- Automatically download new posts from one of your social network accounts and post them to another account. For example, you could take your posts and post them to Facebook.
- Create a “movie encyclopedia” for your personal movie collection by pulling data from IMDb, Rotten Tomatoes, and Wikipedia and putting it into a single text file on your computer.

### The json Module

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

### Reading JSON with the loads() Function

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

```
>>> 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 json Data As Python Value.

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

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