WEBSCRAPING WITH PYTHON

CSC 109 - Introduction to programming in Python - Summer 2023

Marcus Birkenkrahe

June 20, 2023

Contents

1 README

- This is an outline of several webscraping packages:
  1. `webbrowser` - to `open` web pages
  2. `requests` - to download files and web pages
  3. `bs4` - to parse HTML source files
  4. `selenium` - to launch and control a web browser
- The development follows chapter 12 (pp. 267-299) in Sweigart (2019) and the publicly available documentation for the packages:
  1. webbrowser (standard library)
- There are some good tutorials from the DataCamp blog:
  1. Web Scraping & NLP in Python (DataCamp tutorial) (2017).
  2. Web Scraping using Python (and Beautiful Soup) (2018).
  3. Making Web Crawlers Using Scrapy for Python (2019)
- However, web development is a highly volatile field, with many different languages, technologies and standards involved, and I would not expect the code from older tutorials to work out of the box.
- Making it work nevertheless, however, is a great way to learn about a package.

2 Using `webbrowser` to open a URL

- The `webbrowser` module provides an interface to displaying web-based documents to users.

- You can call the `open` function on the URL to open the page:

```
import webbrowser
url = 'https://www.gutenberg.org/files/2701/2701-h/2701-h.htm'
webbrowser.open(url)
url = 'https://lyon.edu'
webbrowser.open(url)
url = 'https://www.python.org'
webbrowser.open(url)
```

- The script `webbrowser` can also be used on the command line. Enter this in a terminal window:

```
python -m webbrowser -t "https://www.python.org"
```

- These will not work in Colab but they work on the terminal or in a Python script.
- As long as you have the URL, `webbrowser` lets users cut out the step of opening the browser. Sample applications (scripts) include:
  1. open all links on a page in separate browser tabs
  2. open the browser to the URL for your local weather
  3. open several social network sites that you regularly check.

3 Example: open Google map with an address only

- We create a script that is run on the command line by the shell program (`bash`) though it is a Python file.
- The shell passes an address argument to the script where it is received as a list of strings `sys.argv`.
- To turn the list into a single string value `address` (a URL for the browser), use `str.join`, then feed the `address` to `webbrowser.open`.
- You find this script in GitHub in `py/src` as `mapit` (link):

```
#! python3
# launch map in browser using an address from the command line
# import pyperclip and use address = pyperclip.paste for clipboard use
```

```
import webbrowser, sys

# If there is at least one command line argument
if len(sys.argv) > 1:
   address = ' '.join(sys.argv[1:])

   # Open the web browser with the constructed URL
   webbrowser.open('https://www.google.com/maps/place/' + address)

   # Write sys.argv to a file and print to the screen
   filename = "address.txt"
   with open(filename, "w") as file:
   print("Contents of sys.argv:")
   for arg in sys.argv:
              # Write each argument to the file
              file.write(arg + "\n")
              # Print confirmation message
              print(f"sys.argv has been written to {filename}")
```

- Download it, open a terminal and run it with an address like this:

  ```
  ./mapit 1014 E Main St, Batesville, AR 72501
  ```

- This will open Google maps to the address and save the list values to a file `address.txt` which you can view with the command `cat`.

4 Savings!

This is what getting a map with or without Python has cost you:

MANUALLY	PYTHON
Highlight address	Highlight address
Copy address	Copy address
Open web browser	Run `mapit`
Open `maps.google.com`	
Click the address text field	
Paste the address	
Press enter	

5 Using `requests` to download files from the web

- With `requests`, you can download files without having to worry about network errors, connection problems or data compression.
- This is the equivalent of the `wget` Unix command (similar to `curl`, which supports a wide variety of protocols not just HTTP and FTP)
- This package is not part of the standard Python library and must be installed (not on Colab or DataCamp):

```
pip install --user requests  # installs for current user only
```

- Test that `requests` installed alright:

```
import requests
```

6 Download a web page with `requests.get`

- The `requests.get` function takes a string of a URL to download:

```
# a CSV file: gapminder dataset
url1 = 'https://raw.githubusercontent.com/birkenkrahe/py/main/data/gapminder.csv'
# a TXT file: Henry James, The American
url2 = 'https://www.gutenberg.org/files/177/177-0.txt'
```

```
# a CSV file: gapminder dataset
url1 = 'https://raw.githubusercontent.com/birkenkrahe/py/main/data/gapminder.csv'
# a TXT file: Henry James, The American
url2 = 'https://www.gutenberg.org/files/177/177-0.txt'
import requests
res1 = requests.get(url1)
res2 = requests.get(url2)
```

- Check out the `type` of the return value of this function. Remember that to check the return value, you need to save the function call itself in a variable and print it:

```
# a CSV file: gapminder dataset
url1 = 'https://raw.githubusercontent.com/birkenkrahe/py/main/data/gapminder.csv'
```

```
# a TXT file: Henry James, The American
url2 = 'https://www.gutenberg.org/files/177/177-0.txt'
import requests
res1 = requests.get(url1)
res2 = requests.get(url2)
print(type(res1))
```

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

- Before reaching out to the file, let's check if the page exists - `requests.status.codes` contains HTTP status codes:

```
# a CSV file: gapminder dataset
url1 = 'https://raw.githubusercontent.com/birkenkrahe/py/main/data/gapminder.csv'
# a TXT file: Henry James, The American
url2 = 'https://www.gutenberg.org/files/177/177-0.txt'
import requests
res1 = requests.get(url1)
res2 = requests.get(url2)
print(f'Page exists: {res1.status_code == requests.codes.ok:}')
```

```
Page exists: True
```

- Look at the (standardized) list of status codes: you'll see 200 for "OK", 404 for "not found" etc. (here is the complete list):

```
import requests
print(help(requests.status_codes))
```

```
Help on module requests.status_codes in requests:

NAME
    requests.status_codes

DESCRIPTION
    The ``codes`` object defines a mapping from common names for HTTP statuses
    to their numerical codes, accessible either as attributes or as dictionary
    items.
```

```
    Example::

>>> import requests
>>> requests.codes['temporary_redirect']
307
>>> requests.codes.teapot
418
>>> requests.codes['\o/']
200

    Some codes have multiple names, and both upper- and lower-case versions of
    the names are allowed. For example, ``codes.ok``, ``codes.OK``, and
    ``codes.okay`` all correspond to the HTTP status code 200.

    * 100: ``continue``
    * 101: ``switching_protocols``
    * 102: ``processing``
    * 103: ``checkpoint``
    * 122: ``uri_too_long``, ``request_uri_too_long``
    * 200: ``ok``, ``okay``, ``all_ok``, ``all_okay``, ``all_good``, ``\o/``, ``\u
    * 201: ``created``
    * 202: ``accepted``
    * 203: ``non_authoritative_info``, ``non_authoritative_information``
    * 204: ``no_content``
    * 205: ``reset_content``, ``reset``
    * 206: ``partial_content``, ``partial``
    * 207: ``multi_status``, ``multiple_status``, ``multi_stati``, ``multiple_stat
    * 208: ``already_reported``
    * 226: ``im_used``
    * 300: ``multiple_choices``
    * 301: ``moved_permanently``, ``moved``, ``\o-``
    * 302: ``found``
    * 303: ``see_other``, ``other``
    * 304: ``not_modified``
    * 305: ``use_proxy``
    * 306: ``switch_proxy``
    * 307: ``temporary_redirect``, ``temporary_moved``, ``temporary``
    * 308: ``permanent_redirect``, ``resume_incomplete``, ``resume``
    * 400: ``bad_request``, ``bad``
    * 401: ``unauthorized``
```

```
* 402: ``payment_required``, ``payment``
* 403: ``forbidden``
* 404: ``not_found``, ``-o-``
* 405: ``method_not_allowed``, ``not_allowed``
* 406: ``not_acceptable``
* 407: ``proxy_authentication_required``, ``proxy_auth``, ``proxy_authenticati
* 408: ``request_timeout``, ``timeout``
* 409: ``conflict``
* 410: ``gone``
* 411: ``length_required``
* 412: ``precondition_failed``, ``precondition``
* 413: ``request_entity_too_large``
* 414: ``request_uri_too_large``
* 415: ``unsupported_media_type``, ``unsupported_media``, ``media_type``
* 416: ``requested_range_not_satisfiable``, ``requested_range``, ``range_not_s
* 417: ``expectation_failed``
* 418: ``im_a_teapot``, ``teapot``, ``i_am_a_teapot``
* 421: ``misdirected_request``
* 422: ``unprocessable_entity``, ``unprocessable``
* 423: ``locked``
* 424: ``failed_dependency``, ``dependency``
* 425: ``unordered_collection``, ``unordered``
* 426: ``upgrade_required``, ``upgrade``
* 428: ``precondition_required``, ``precondition``
* 429: ``too_many_requests``, ``too_many``
* 431: ``header_fields_too_large``, ``fields_too_large``
* 444: ``no_response``, ``none``
* 449: ``retry_with``, ``retry``
* 450: ``blocked_by_windows_parental_controls``, ``parental_controls``
* 451: ``unavailable_for_legal_reasons``, ``legal_reasons``
* 499: ``client_closed_request``
* 500: ``internal_server_error``, ``server_error``, ``/o\``, ``\u2717``
* 501: ``not_implemented``
* 502: ``bad_gateway``
* 503: ``service_unavailable``, ``unavailable``
* 504: ``gateway_timeout``
* 505: ``http_version_not_supported``, ``http_version``
* 506: ``variant_also_negotiates``
* 507: ``insufficient_storage``
* 509: ``bandwidth_limit_exceeded``, ``bandwidth``
```

```
    * 510: ‘‘not_extended‘‘
    * 511: ‘‘network_authentication_required‘‘, ‘‘network_auth‘‘, ‘‘network_authen

DATA
    codes = <lookup 'status_codes'>

FILE
    c:\users\birkenkrahe\appdata\local\programs\python\python311\lib\site-packages


None
```

- Print the number of characters of the targeted web page, which is now stored as one long string:

```
# a CSV file: gapminder dataset
url1 = 'https://raw.githubusercontent.com/birkenkrahe/py/main/data/gapminder.csv'
# a TXT file: Henry James, The American
url2 = 'https://www.gutenberg.org/files/177/177-0.txt'
import requests
res1 = requests.get(url1)
res2 = requests.get(url2)
print(len(res1.text))
print(len(res2.text))
```

```
7862
794196
```

- Strings are sequence data (indexed), so we can look at the top of the text files like this:

```
# a CSV file: gapminder dataset
url1 = 'https://raw.githubusercontent.com/birkenkrahe/py/main/data/gapminder.csv'
# a TXT file: Henry James, The American
url2 = 'https://www.gutenberg.org/files/177/177-0.txt'
import requests
res1 = requests.get(url1)
res2 = requests.get(url2)
print(res1.text[:250])
print(---------------)
print(res2.text[:250])
```

- Microsoft Windows (inside Emacs) renders the text file (not the CSV) with additional control characters. On the Python console, and in Colab, it looks worse:

```
'ï»¿The Project Gutenberg eBook of The American, by Henry James\r\n\r\nThis eBook is for
the use of anyone anywhere in the United States and\r\nmost other parts of the world at
no cost and with almost no restrictions\r\nwhatsoever. You may copy it, give it aw'
```

- Connection issues are rampant. Another way to check if the download succeeded is to call `raise_for_status` on the `response` object: if there was an error, then an exception will be raised.
- Raise a 404 exception with a non-existent page (incomplete name):

```
import requests
bad_url = 'https://www.gutenberg.org/files/177/177'
res = requests.get(bad_url)
res.raise_for_status()
```

- You can wrap the `raise_for_status()` line with `try...except` to handle the exception:

```
import requests
bad_url = 'https://www.gutenberg.org/files/177/177'
res = requests.get(bad_url)
try:
    res.raise_for_status()
except Exception as exc:
    print(f'There was a problem: {exc}')
```

- Always call `raise_for_status()` after calling `requests.get()` before continuing.

7 Save downloaded files

- To save the file from the `response` object in Python to a file, use the standard library functions `open` and `write`:
  1. `open` the file in `write binary` mode (parameter `'wb'`) to maintain the Unicode encoding of the text.

2. `write` the web page to a file using `requests.Response.iter_content`:

```
import requests
url2 = 'https://www.gutenberg.org/files/177/177-0.txt'
res = requests.get(url2)
try:
    res.raise_for_status()
except Exception as exc:
    print(f'There was a problem: {exc}')

# open file in write binary mode
jamesFile = open('TheAmerican.txt','wb')

# write the web page to file
for chunk in res.iter_content(100000):
    jamesFile.write(chunk, 'wb')
    bytes_written = jamesFile.write(chunk)
    print(f'Written {bytes_written} bytes')

# close the output stream to file
jamesFile.close()
```

- The `iter_content` method returns chunks of the content on each iteration. The chunks are of the `bytes` data type and the argument specifies how many bytes a chunk can contain (100kB). This prevents loading the entire file into memory at once. The `close()` function flushes all data to disk and frees resources.
- The `requests` module contains many more methods for users:

```
import requests
print(len(dir(requests)))
print(dir(requests))

69
['ConnectTimeout', 'ConnectionError', 'DependencyWarning', 'FileModeWarning', 'HT
```

8 HTML code and CSS classes

- HTML (HyperText Markup Language) files are plaintext `.html` files

- Though tempting to the initiated, you cannot parse HTML using regular expressions (see here) (which is why I left regex out).
- Text in an HTML file is surrounded by *tags*, which are enclosed in angle brackets:

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

- CSS (Cascading Style Sheets) are essentially functions to change the layout without having to change every single HTML file. Example:

```
<div class="container">
    <p class="text">Hello</p>
    <p id="special">World</p>
</div>
<p class="text">Outside</p>
```

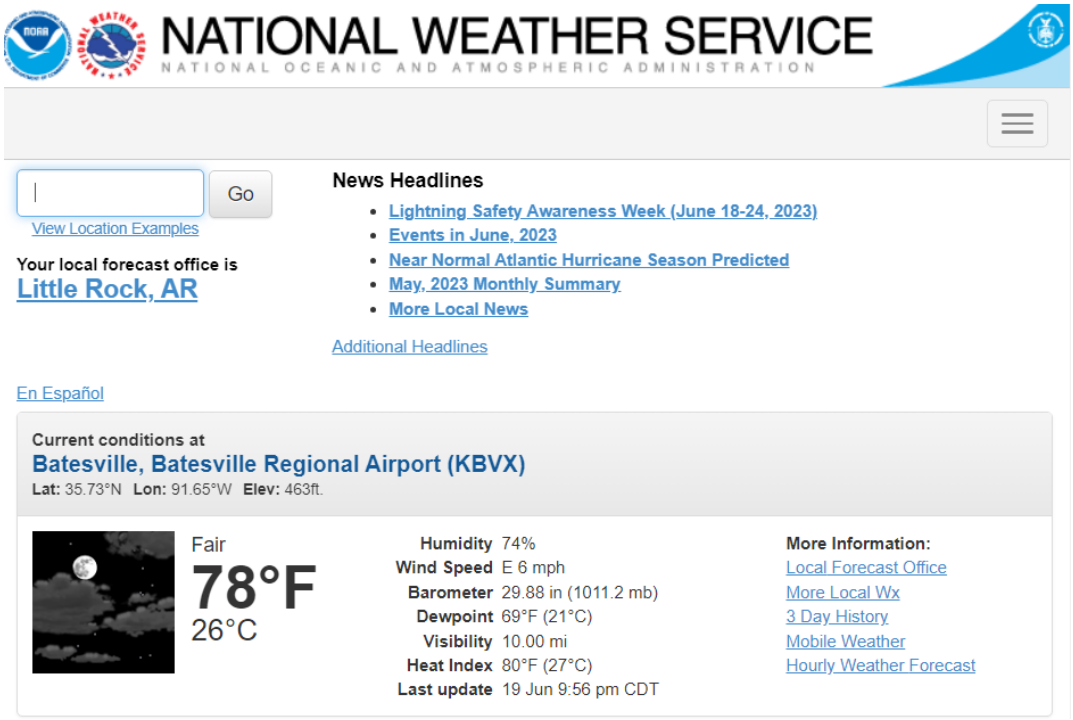
- This code contains:
  1. one `div` divider element
  2. two CSS classes (aka functions), `.container` and `.text`
  3. three `p` elements (new paragraph)
  4. one `id` attribute with the value `special` inside a `p` element (attributes can be linked to like `#special` or `?id=special`)
- You can open this code from within an HTML file with the `webbrowser` module - run this in IDLE as a file `html.py`:

```
import webbrowser

# HTML content: whitespace is irrelevant here
html_content = '<strong>Hello</strong>, world!'

# Write HTML content to a file
with open('hello.html', 'w') as file:
    file.write(html_content)

# Open the file in the web browser
webbrowser.open('hello.html')
```

- Many tags have attributes within the angle brackets. For example, open this page to an article on `aeaweb.org`, aeaweb.org/articles?id=10.1257/jel.20201482, right-click and select `view page source` (CTRL + u): after some HTML comments (`<!-- ... -->`) follows the `<html>` tag, which brackets the entire page: this tag has a language attribute `lang='en'`.
- But you can see that most of the meta information about this paper is contained within a page of `<meta>` tags with the attribute `name`.
- To see even more hidden information, you can right click and select `Inspect` (or open the `More tools > Developer tools` browser menu).

9 Viewing HTML/CSS source: weather data

- Why would you look at the developer tools?
- Let's say you want to pull weather forecast data from `https://weather.gov/`.
- Enter the Batesville ZIP code `72501` in the search field at the top.



- Open the `Inspect` panel and after some searching, you'll find that the current weather conditions for example are included in one `<div>` block:

```
▼<div id="current_conditions-summary" class="pull-left">
    <img src="newimages/large/nsct.png" alt class="pull-left">
    <p class="myforecast-current">Fair</p>
    <p class="myforecast-current-lrg">78°F</p>
    <p class="myforecast-current-sm">26°C</p>
  </div>
```

- You can copy any element with right-click and selecting `Copy > Copy Element`, and later use this information for scraping:

```
<div id="current_conditions-summary" class="pull-left">
          ...
          <p class="myforecast-current">Fair</p>
```

```
            <p class="myforecast-current-lrg">78°F</p>
            <p class="myforecast-current-sm">26°C</p>
        </div>
```

10 Parsing HTML with the `bs4` module

- 'Beautiful Soup' is a module for extracting information from an HTML page. The module's real name is `bs4` (version 4).
- To install (if not in Colab or DataCamp, or on Python 3.11 which comes with Beautiful Soup):

  ```
  pip install --user beautifulsoup4
  ```

- Import the module (there should be no complaints):

  ```
  import bs4
  ```

- For our example, we'll use `bs4` to parse (i.e. analyze + identify the parts of) a simple HTML file on the hard drive:

  ```
  <!-- This is the example.html file. -->
  <html><head><title>The Website Title</title></head>
  <body>
    <p>Download the book <strong>The American</strong> from
    <a href="https://www.gutenberg.org/files/177/177-0.txt">Project Gutenberg</a>.</
    <p class="slogan">Read more 19th century fiction!</p>
    <p>By <span id="author">Henry James</span></p>
  </body></html>
  ```

- Use `webbrowser` to render the file in your browser as `example.html`:

  ```
  import webbrowser

  # HTML content - whitespace is irrelevant
  html_content = '  <!-- This is the example.html file. --> <html><head><title>The W

  # Write HTML content to a file
  with open('example.html', 'w') as file:
      file.write(html_content)

  # Open the file in the web browser
  webbrowser.open('example.html')
  ```

11 Creating a Beautiful Soup Object from HTML

- The `bs4.BeautifulSoup` function is called with a string containing the HTML it will parse. It returns a `BeautifulSoup` object (on which various methods will work).
- Example:
  1. get a HTML page
  2. raise an status exception check
  3. pass response text to bs4.Beautiful Soup
  4. show the type of the bs4 object

```
import requests, bs4

# download the main page from Project Gutenberg
res = requests.get('https://gutenberg.org')
try:
    res.raise_for_status()
except Exception as exc:
    print(f'There was a problem: {exc}')

# Pass the text attribute of the response to bs4.BeautifulSoup
gutenbergSoup = bs4.BeautifulSoup(res.text, 'html.parser')
print(type(gutenbergSoup))
```

```
<class 'bs4.BeautifulSoup'>
```

- Download `example.html` from here: bit.ly/beautifulBook - in Colab, you can upload it to the temporary directory.
- Use `requests` to load the HTML file from your hard drive and pass a `File` object instead of a `requests.Response` to `bs4.BeautifulSoup`:

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

```
<class 'bs4.BeautifulSoup'>
```

- The `html.parser` comes with Python (there is a faster parser in the `lxml` module - see here).

12 Finding an element with `select`

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

- The `select` function uses CSS selectors to match elements or tags, like classes, IDs etc. It returns a list of elements matching the selector.
- Try this yourself with this HTML code:

```
from bs4 import BeautifulSoup

html = """
<div class="container">
    <p class="text">Hello</p>
    <p id="special">World</p>
</div>
<p class="text">Outside</p>
```

```
"""
soup = BeautifulSoup(html, 'html.parser')

# Example: all elements that use a CSS `class` attribute named `.container`
elements = soup.select('.container .text')
for element in elements:
    print(f'container text: {element.text}')

elements = soup.select('p')
for element in elements:
    print(f'p elements: {element.text}')

# result values are stored in a list (sequence data)
print(elements)

# the data type is specific to bs4
print(type(elements))

container text: Hello
p elements: Hello
p elements: World
p elements: Outside
[<p class="text">Hello</p>, <p id="special">World</p>, <p class="text">Outside</p>
<class 'bs4.element.ResultSet'>
```

- Use this code to select the following elements:
  1. Elements in the CSS class 'text'
  2. Elements named 'div'
  3. Elements named p with id value
  4. Elements named p with id value 'special'
- Solution:

```
from bs4 import BeautifulSoup

html = """
<div class="container">
    <p class="text">Hello</p>
```

```
    <p id="special">World</p>
</div>
<p class="text">Outside</p>
"""
soup = BeautifulSoup(html, 'html.parser')

elements = soup.select('.container .text') # 'text' in '.container' class
for element in elements:
    print(f'container text: {element.text}')

elements = soup.select('.text') # elements in the '.text' class
for element in elements:
    print(f"'.text' element: {element.text}")

elements = soup.select('div')  # elements named 'div'
for element in elements:
    print(f'div elements: {element.text}')

elements = soup.select('p')  # elements named 'p'
for element in elements:
    print(f'p elements {element.text}')

elements = soup.select('p[id]')  # elements named 'p' w/id' value
for element in elements:
    print(f"p elements with 'id' value: {element.text}")

elements = soup.select('p[id="special"]')  # p elements with id = 'special'
for element in elements:
    print(f"p elements with 'class=special' value: {element.text}")

container text: Hello
'.text' element: Hello
'.text' element: Outside
div elements:
Hello
World

p elements Hello
p elements World
p elements Outside
```

```
p elements with 'id' value: World
p elements with 'class=special' value: World
```

- Selector patterns can be combined to make sophisticated matches: this pattern will match any element that has an `id` attribute of `author` as long as it is also inside a `p` element

```
soup.select('p #author') # selects <p id="author">THIS</p>
```

- What will this pattern select?

```
soup.select('span .text')
```

  Any element of the CSS class 'text' inside a `span` element: THIS

- The tag values of the `soup.select` result `list` can be passed to `str` to show the HTML tags they represent, and an `attrs` attribute that shows all HTML attributes of the tag as a `dictionary`.

```
from bs4 import BeautifulSoup

html = """
<div class="container">
    <p class="text">Hello</p>
    <p id="special">World</p>
</div>
<p class="text">Outside</p>
"""
soup = BeautifulSoup(html, 'html.parser')

elements = soup.select('.text')
print(elements[0])
print(type(elements[0]))
print(type(str(elements[0])))

<p class="text">Hello</p>
<class 'bs4.element.Tag'>
<class 'str'>
```

13 Example HTML file: extract text and attributes

- Here is the example HTML with mostly HTML and one CSS class element:

```
<!-- This is the example.html file. -->
<html><head>
        <title>The Website Title</title>
      </head>
      <body>
        <p>Download the book <strong>The American</strong> from
           <a href="https://www.gutenberg.org/files/177/177-0.txt">Project Gutenbe
        </p>
        <p class="slogan">Read more 19th century fiction!</p>
        <p>By <span id="author">Henry James</span>
        </p>
      </body>
</html>
```

- Pull the element with `id="author"` out of the example HTML:

```
import bs4

# open the HTML file
exampleFile = open('example.html')

# parse HTML from file
exampleSoup = bs4.BeautifulSoup(exampleFile.read(), 'html.parser')

# select all 'id' attributes with the value 'author':
elements = exampleSoup.select('#author')

print(isinstance(elements,list))
print(elements)     # Output: list
print(len(elements))  # Output: 1
print(type(elements[0]))  # Output: bs4.element.Tag

# the tag object as a string
print(str(elements[0]))
```

```
# get the tag text string
print(elements[0].getText())
print(isinstance(elements[0].getText(),str))

# get the tag attributes dictionary
print(elements[0].attrs)
print(isinstance(elements[0].attrs,dict))

True
[<span id="author">Henry James</span>]
1
<class 'bs4.element.Tag'>
<span id="author">Henry James</span>
Henry James
True
{'id': 'author'}
True
```

- As an exercise, pull the <p> elements from the example HTML and
  1. get the text of the 1st list item with `getText()`
  2. turn the 2nd list item into a string with `str`
  3. get the text of the 2nd list item with `getText()`
  4. turn the 3rd list item into a string with `str`
  5. get the text of the 3rd list item with `getText()`

```
import bs4
exampleFile = open('example.html')
exampleSoup = bs4.BeautifulSoup(exampleFile.read(), 'html.parser')
# select all 'p' elements
pElements = exampleSoup.select('p')

for i in range(3):
    print(pElements[i].getText())
    print(str(pElements[i]))

Download the book The American from Project Gutenberg.
<p>Download the book <strong>The American</strong> from <a href="https://www.gute
```

```
Read more 19th century fiction!
<p class="slogan">Read more 19th century fiction!</p>
By Henry James
<p>By <span id="author">Henry James</span></p>
```

14 Getting data from an element's attributes

15 TODO Extended example: opening all search results

16 TODO Extended example: downloading all xkcd cartoons

17 TODO Controlling the browser with `selenium`

18 TODO Tokenize text with `nltk`

Source: `https://www.datacamp.com/tutorial/web-scraping-python-nlp`

19 TODO Working with APIs

20 References

- Sweigart, A. (2019). Automate the Boring Stuff with Python. NoStarch. URL: automatetheboringstuff.com
- Van Rossum, G., Drake, F. L. (2009). Python 3 Reference Manual. URL: `https://docs.python.org/3/reference/`.