

WEBSCRAPING WITH PYTHON

CSC 109 - Introduction to programming in Python - Summer 2023

Marcus Birkenkrahe

June 21, 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 BeautifulSoup) (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:]) # ' ' is inserted between args

    # 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 <code>mapit</code>
Open <code>maps.google.com</code>	
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
```

```
Requirement already satisfied: requests in c:\311-packages (2.29.0)
Requirement already satisfied: charset-normalizer<4,>=2 in c:\311-packages (from
Requirement already satisfied: idna<4,>=2.5 in c:\311-packages (from
Requirement already satisfied: urllib3<1.27,>=1.21.1 in c:\311-packages (from
Requirement already satisfied: certifi>=2017.4.17 in c:\311-packages (from
```

- 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'
```

```
<<url>>
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:

```
<<res>>
print(type(res1))
```

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

```
<<res>>
print(f'Page exists: {res1.status_code == requests.codes.ok:}')
```

- 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))
```

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

```
<<res>>
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:

```
<<res>>
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:

```
'i»¿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 `requestsN.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)
    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))
```

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)

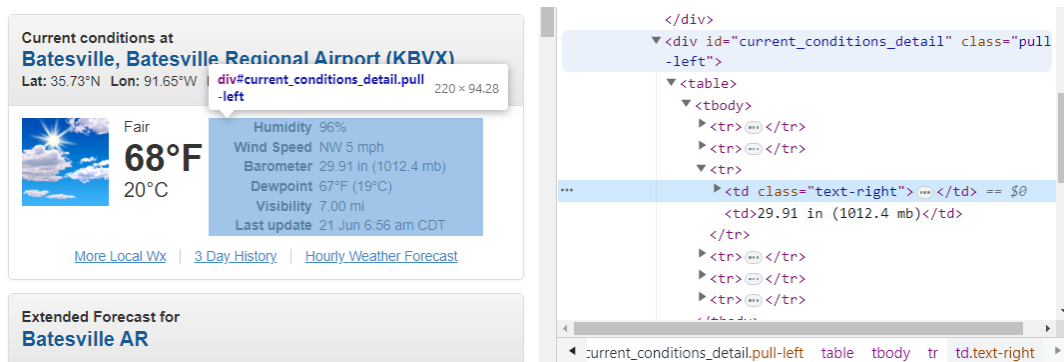
# Open the file in the web browser
webbrowser.open('hello.html')
```

- Many tags have attributes within the angle brackets. For example, open this page to an article on [aeaweb.org](http://aeaweb.org/articles?id=10.1257/jel.20201482), 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

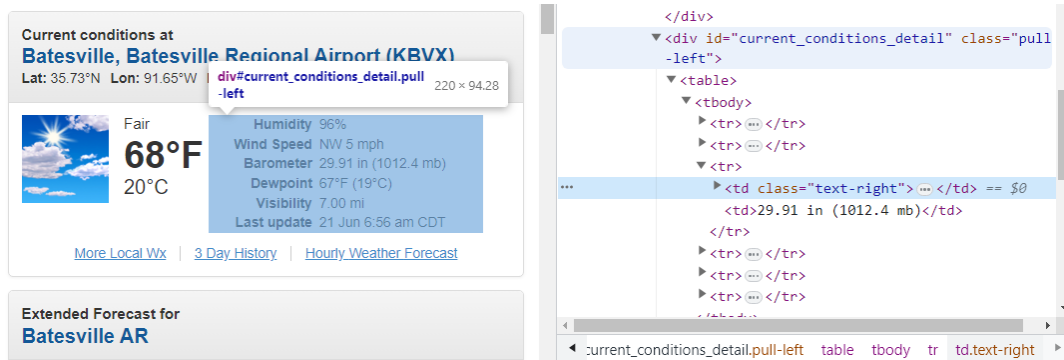
- Why would you look at the developer tools?
- Let's say you want to pull weather forecast data from <https://weather.gov/>.
- Enter the Batesville ZIP code 72501 in the search field at the top.



- 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">
  
  <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 also right click and **inspect** any element and the inspector will jump to the respective element:



- 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>
```

- You can also **Copy selector** directly, which will usually result in a much longer string which you may have to analyze to get the element and attribute or attribute value that you really want.

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 BeautifulSoup):

```
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. Get this file from GitHub: bit.ly/beautifulBook.

```
<!-- 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>
  <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` (you'll have to do this on a Python console other than Colab, e.g. IDLE):

```
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)
```

```
# check if the file is there
```

```
import os
```

```
print(os.path.isfile('example.html'))
```

```
# check where you are - get current working directory
```

```
print(os.getcwd())
```

```
# Open the file in the web browser
```

```
webbrowser.open('example.html')
```

```
True
```

```
c:\Users\birkenkrahe\Documents\GitHub\py\org
```

- Enter this in IDLE on the interactive shell/console and see if it works.

11 Creating a BeautifulSoup 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 (you can do this in Colab):

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')
res.raise_for_status() # check request status
res # prints status
res.text # prints downloaded text

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

- Now download `example.html` from here: bit.ly/beautifulBook - in Colab, you can upload it to the temporary directory (see sidebar):

```
%cat example.html # 'magic' command to display the file in Colab
```

- 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') # open file to stdio
exampleSoup = bs4.BeautifulSoup(exampleFile, 'html.parser')
print(type(exampleSoup))
exampleSoup
```

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

- The `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 (Gist link - bit.ly/3qUy8nb):

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

elements = soup.select('p') # look for all paragraph elements
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])))
```

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 Gutenberg</a>
    </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))
```

- 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]))
```

14 Getting data from an element's attributes

- You can use the `get` method to access attribute values from an element: you pass a string of an attribute name, e.g. `'id'`, and get the value in return:

```
import bs4

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

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

# select the first 'span' element
spanElement = exampleSoup.select('span')[0]

print(str(spanElement))
```

```
print(spanElement.get('id'))
print(spanElement.get('some_nonexistent_address') == None)
print(spanElement.attrs)
```

15 Summary

15.1 Webbrowser

- Python has a built-in `webbrowser`.
- You can open URLs with the `webbrowser.open` function.
- You can pass command line arguments as a `sys.argv` string
- The function `str.join` concatenates strings and inserts `str`

15.2 Downloading web pages and files

- The `Requests` module is a third-party module for downloading web pages and files.
- `requests.get()` returns a `Response` object.
- The `raise_for_status()` `Response` method will raise an exception if the download failed.
- You can save a downloaded file to your hard drive with calls to the `iter_content()` method.

15.3 HTML and CSS

- HTML files are text files that contain HTML and CSS elements and JavaScript (or similar) for dynamic layout creation
- For `Response.select` from `requests`, you need to identify CSS selectors like classes, attributes and attribute values
- You can inspect the source code of a web page and copy code elements or CSS selectors corresponding to the page elements that interest you.

15.4 Parsing web pages

- Web pages are plaintext files formatted as HTML.
- HTML can be parsed with the BeautifulSoup module.
- BeautifulSoup is imported with the name `bs4`.
- Pass the string with the HTML to the `bs4.BeautifulSoup` function to get a `Soup` object.
- The `Soup` object has a `select` method that can be passed a string of the CSS selector for an HTML tag.
- You can get a CSS selector string from the browser's developer tools by right-clicking the element and selecting `Copy CSS Path`.
- The `select` method will return a list of matching Element objects.
- The simpler `find` method returns the first occurrence of a specific element, e.g. `soup.find('div')`, so you have to loop to find more.

16 Project: opening all search results

- This script accepts a search term on the command line and opens a browser with the top search results in new tabs.
- The script uses the URL for the Python Package Index at `https://pypi.org` but you can adapt it to any website.
- Program flow:
 1. Get search keywords from command line arguments.
 2. Retrieves the search results page.
 3. Opens a browser tab for each result.
- Python script requirements:
 1. Read command line arguments from `sys.argv`.
 2. Fetch the result page with the `requests` module.
 3. Find the links to each search result with `bs4`.
 4. Call `webbrowser.open` to open the web browser.

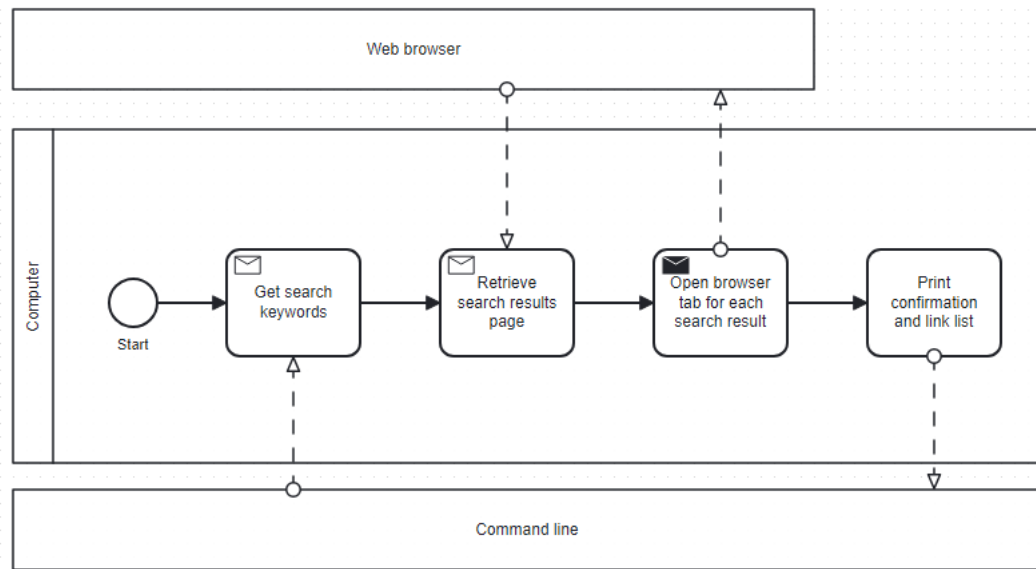


Figure 1: BPMN model of the program flow for search & open

17 Get cmd line arguments and request search page

- The result page of a search at `https://pypi.org` has the URL format:

`https://pypi.org/search/?q=[search_term]`

- We use the trick from `mapit.py` to attach the command line argument to the argument for `requests.get` with `str.join` and `sys.argv` :

```

#!/ python3
# searchpypi.py - opens several search results

# import required modules
import requests, sys, webbrowser, bs4

# download search result page
print('Searching...')
res = requests.get('https://pypi.org/?q=' + ' '.join(sys.argv[1:]))
res.raise_for_status()

```

- Remember that `res` holds the return value from `requests.get`, while `res.text` is a string attribute holding the HTML text of that page.

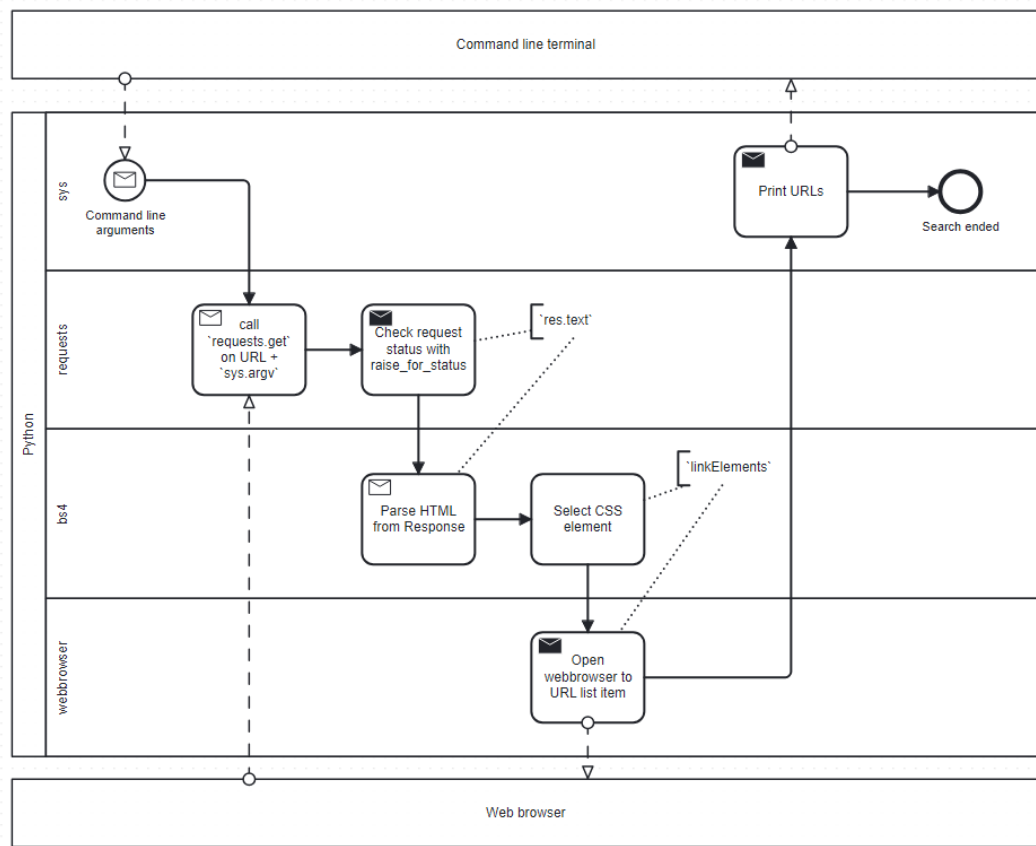


Figure 2: BPMN model of the program flow for `searchpypi.py`

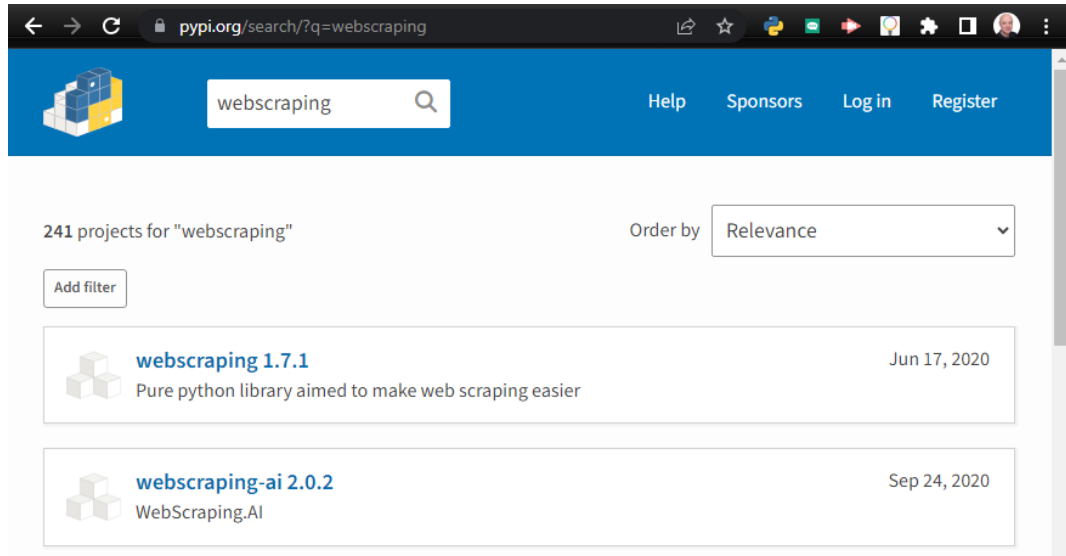


Figure 3: search string for a Google search at pypi.org for 'webscraping'

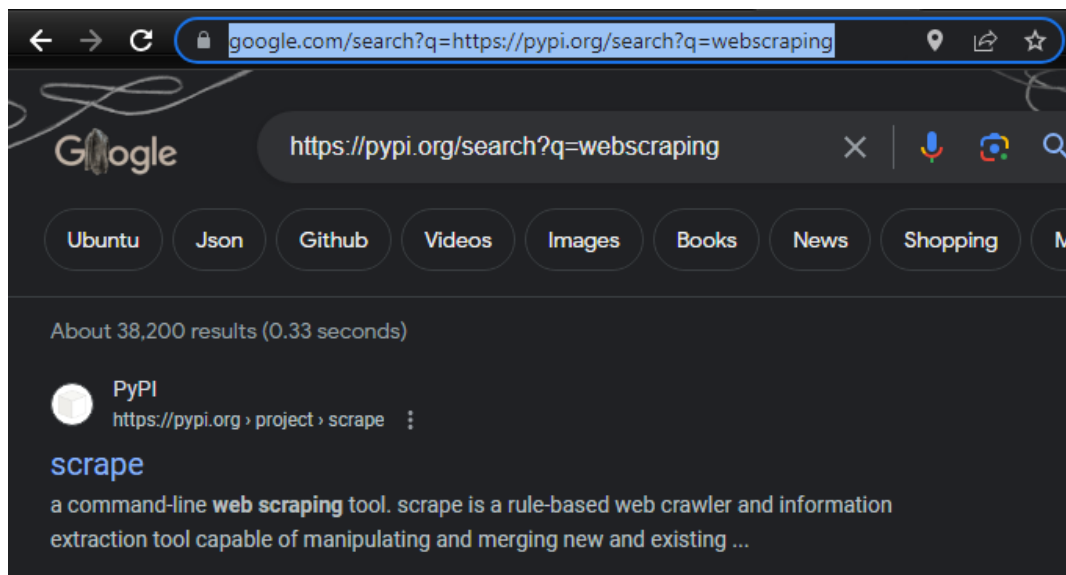


Figure 4: search string for a Google search at pypi.org for 'webscraping'

```
<<request>>
print(res) # return value from requests.get
print(type(res))
print(res.text)
print(type(res.text))
```

- In `'+'.join('ab')`, `+` is inserted between `'a'` and `'b'`:

```
print('http://pypi.org/search/?q=' + ''.join('webscraping'))
```

- When the program is launched, the user specifies the search terms as command line arguments. These arguments will be stored as a list in `sys.argv`.
- Why do we start with the index 1 in `sys.argv`? Because `sys.argv[0]` stores the name of the list: when you enter

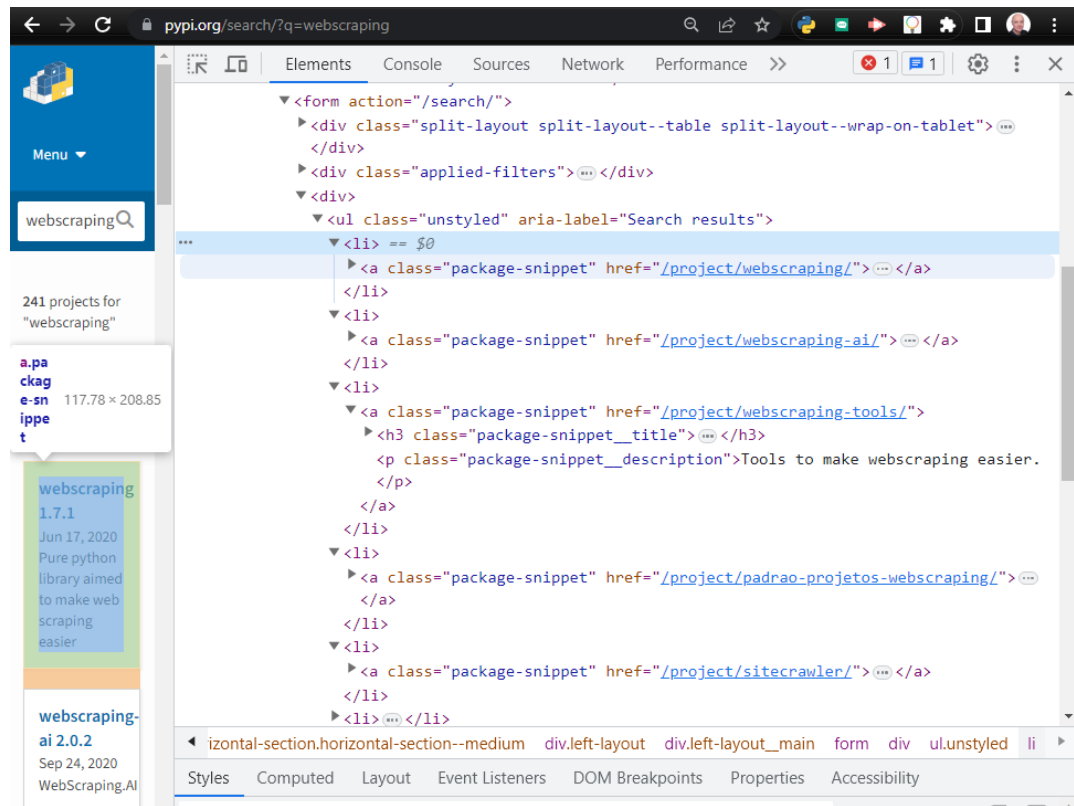
```
python myscript.py arg1 arg2 arg3
```

the list will be:

```
['myscript.py', 'arg1', 'arg2', 'arg3']
```

18 Find all the results

- To apply `select` from `Beautiful Soup`, you need to identify the right selector. But using the `<a>` tag element returns all links and not just the search links.
- You need to open the page inspector and find the pattern that all search results share: it is the CSS class `.package-snippet` in the `<main id="content">` block. The link itself is the value of the `'href'` attribute:



- Nice: you don't have to know what this CSS class is or what it does. You don't need to know how to write pages like these. You're only going to use it as a marker for the `<a>` element.
- Add this code to the earlier block:

```
<<request>>
```

```
# Parse the search page HTML code from the Response object
soup = bs4.BeautifulSoup(res.text, 'html.parser')
print(soup)
```

```
# Select link elements with the CSS class 'package-snippet'
linkElements = soup.select('.package-snippet')
print(linkElements)
```

- If the PyPi website changes its layout, you may need to update the script with the appropriate CSS selector string for `soup.select`.

- What about the content of `linkElements` and the element attributes?

```
<<soup>>
print(str(linkElements))
print(linkElements.attrs)
```

```
Searching...
[]
```

- The last code block returns an empty list and raises an `AttributeError` - can you think why?

Our search depends on passing a search term as a command line argument to `requests` - but there's no default search term. We have not done that so `requests.get` downloads the page without results.

19 Open web browsers for each result

- Finally, we open web browsers for each result:
 1. loop over (at most 5 of) the list `linkElements`.
 2. build a URL string `urlToOpen` from the value of the `href` attribute of each link element.
 3. feed the URL `urlToOpen` to `webbrowser.open`.

```
<<soup>>
# We want at most 5 search results
numOpen = min(5, len(linkElements))

for i in range(numOpen):
    urlToOpen = 'https://pypi.org' + linkElements[i].get('href')
    print('Opening', urlToOpen)
    webbrowser.open(urlToOpen)
```

- The `min` function takes care of the possibility that our search resulted in fewer than 5 links: take 5 or whatever is smaller.
- The `href` attribute value does not have the full URL (you can only see this after looking at the inspector), which is why we add it.

- To run the program download `searchpypi.py` from GitHub's `py/src/` repo and run it on a command line like a shell script or with python:

```
$ ./searchpypi webscraping # run as script using the shebang line
$ python searchpypi.py webscraping # run as python program
```

- Sample output for command `python searchpypi.py webscraping` addition to 5 windows being opened):

```
C:\Users\birkenkrahe\Downloads>python searchpypi.py webscraping
Searching...
Opening https://pypi.org/project/webscraping/
Opening https://pypi.org/project/webscraping-tools/
Opening https://pypi.org/project/webscraping-ai/
Opening https://pypi.org/project/padrao-projetos-webscraping/
Opening https://pypi.org/project/sitecrawler/
C:\Users\birkenkrahe\Downloads>
```

- This Colab notebook contains the minimal code (no exception handling): <https://bit.ly/3NExnaC>
- Here is the full program with exception handling for export (you can download it from GitHub in the `py/src` repository):

```
#!/ python3
# searchpypi.py - opens several search results

# import required modules
import requests, sys, webbrowser, bs4

# Check if argument is provided
if len(sys.argv) < 2:
    print("Please provide a search term")
    sys.exit()

# download search result page
print('Searching...')
res = requests.get('https://pypi.org/search?q=' + ''.join(sys.argv[1:]), headers=
res.raise_for_status()
```

```

# Print the response text for debugging
#print(res.text)

# Retrieve top search result links
soup = bs4.BeautifulSoup(res.text, 'html.parser')

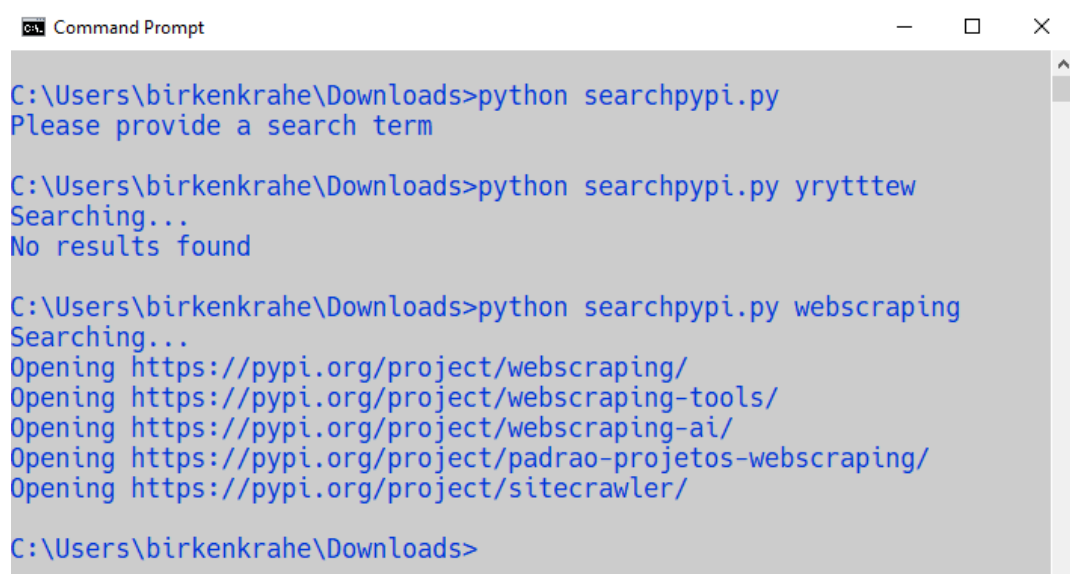
# Replace with the correct CSS selector for search results on PyPI
linkElements = soup.select('.package-snippet') # This might need to be updated

# We want at most 5 search results
numOpen = min(5, len(linkElements))

if numOpen == 0:
    print("No results found")
else:
    for i in range(numOpen):
        try:
            href = linkElements[i].get('href')
            if href:
                urlToOpen = 'https://pypi.org' + href
                print('Opening', urlToOpen)
                webbrowser.open(urlToOpen)
            else:
                print("No href found for result", i)
        except AttributeError as e:
            print("Error processing result", i, ":", str(e))

```

- A search without search term and with a nonsense search term is covered by this code:



```
Command Prompt

C:\Users\birkenkrahe\Downloads>python searchpypi.py
Please provide a search term

C:\Users\birkenkrahe\Downloads>python searchpypi.py yryttew
Searching...
No results found

C:\Users\birkenkrahe\Downloads>python searchpypi.py webscraping
Searching...
Opening https://pypi.org/project/webscraping/
Opening https://pypi.org/project/webscraping-tools/
Opening https://pypi.org/project/webscraping-ai/
Opening https://pypi.org/project/padrao-projetos-webscraping/
Opening https://pypi.org/project/sitecrawler/

C:\Users\birkenkrahe\Downloads>
```

20 Ideas for similar programs

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

21 IN PROGRESS Controlling the browser with selenium

- The **selenium** module controls the browser by clicking links and filling in login information.
- To do this, a web browser must be launched, so **selenium** is slower and harder to run in the background.
- You must do this if your interaction with a web page depends on the `<script>` JavaScript code that dynamically updates the page.
- Commercial sites run security programs to stop people from harvesting their information or creating multiple free accounts. They change often and will break your non-**selenium** Python scripts.

- Security programs identify you with a *user-agent* string that is served by the web browser and is included in all HTTP requests.
- For example the user-agent string for Python's **requests** module v.2.29 is **Python-requests/2.29.0**. But for **selenium**, user-agent is the same as a regular browser (see whatsmyua.info) and it mimicks the browser's traffic patterns (downloads, ads, cookies etc.).
- Since **selenium** is not entirely undetectable, however, there are sites which do not allow it to scrape their web pages.

22 IN PROGRESS Starting a selenium controlled browser

- You can install **selenium** on the command line via **pip**:

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

- Check installation:

```
import selenium
```

23 IN PROGRESS Extended example: downloading all xkcd cartoons

- See bs4's online documentation.
- Code:

```
#!/ python3
# downloadXkcd.py - Downloads every single XKCD comic.
import requests, os, bs4
url = 'https://xkcd.com'          # starting url
os.makedirs('xkcd', exist_ok=True) # store comics in ./xkcd
while not url.endswith('#'):
    # Download the page.
    print('Downloading page %s...' % url)
    res = requests.get(url)
    res.raise_for_status()
    soup = bs4.BeautifulSoup(res.text, 'html.parser')
```

```

# Find the URL of the comic image.
comicElem = soup.select('#comic img')
if comicElem == []:
    print('Could not find comic image.')
else:
    comicUrl = 'https:' + comicElem[0].get('src')
# Download the image.
print('Downloading image %s...' % (comicUrl))
res = requests.get(comicUrl)
res.raise_for_status()
# Save the image to ./xkcd.
imageFile = open(os.path.join('xkcd', os.path.basename(comicUrl)), 'wb')
for chunk in res.iter_content(100000):
    imageFile.write(chunk)
    imageFile.close()
# Get the Prev button's url.
prevLink = soup.select('a[rel="prev"]')[0]
url = 'https://xkcd.com' + prevLink.get('href')
print('Done.')

```

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

24 TODO Working with JSON APIs

25 TODO Tokenize text with nltk

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

26 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/>.