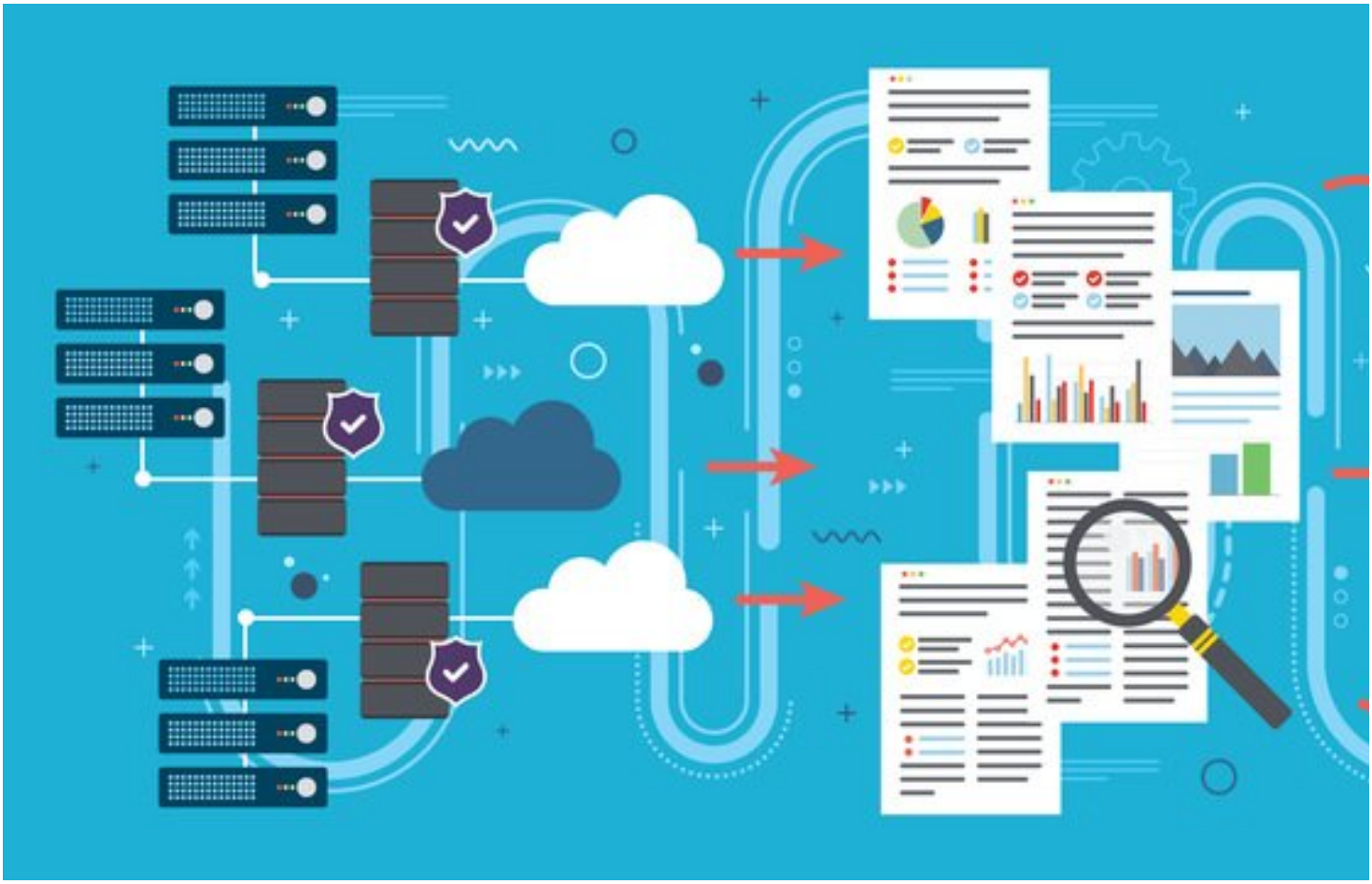
[Tutorial] Webscraping with BeautifulSoup4



🎓 The HyperText Transfer Protocol: HTTP

The core component in the exchange of messages consists of a HyperText Transfer Protocol (HTTP) request message to a web server, followed by an HTTP response (also oftentimes called an HTTP reply), which can be rendered by the browser. Since all of our web scraping will build upon HTTP, we do need to take a closer look at HTTP messages to learn what they look like.

HTTP is, in fact, a rather simple networking protocol. It is text based, which at least makes its messages somewhat readable to end users (compared to raw binary messages that have no textual structure at all) and follow a simple request-reply-based communication scheme. That is, contacting a web server and receiving a reply simply involves two HTTP messages: a request and a reply. In case your browser wants to download or fetch additional resources (such as images), this will simply entail additional request-reply messages being sent. In Bref, the client sends requests to the server and the server sends responses, or replies.

1️⃣ Request Message

A request message consists of the following:

- A request line;
- A number of request headers, each on their own line;
- An empty line;
- An optional message body, which can also take up multiple lines.

Each line in an HTTP message must end with <CR><LF> (the ASCII characters 0D and 0A). <CR> and <LF> are two special characters to indicate that a new line should be started.

The following code fragment shows a full HTTP request message as executed by a web browser (we don't show the "<CR><LF>" after each line, except for the last, blank line):

```
GET / HTTP/1.1
Host: example.com
Connection: keep-alive
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
User- Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
  (KHTML, like Gecko) Chrome/60.0.3112.90 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: https://www.google.com/
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.8,nl;q=0.6
<CR><LF>
```

2️⃣ Response Message

If all goes well, the web server will process our request and send back an HTTP reply.
These look very similar to HTTP requests and contain:

- A status line that includes the status code and a status message;
- A number of response headers, again all on the same line;
- An empty line;
- An optional message body.

As such, we might get the following response following our request above:

```
HTTP/1.1 200 OK
Connection: keep-alive
Content-Encoding: gzip
Content-Type: text/html;charset=utf-8
Date: Mon, 28 Aug 2017 10:57:42 GMT
Server: Apache v1.3
Vary: Accept-Encoding
Transfer-Encoding: chunked
<CR><LF>
<html>
```

```
<body>Welcome to My Web Page</body>
</html>
```

The first line indicates the status result of the request. It opens by listing the HTTP version the server understands (“HTTP/1.1”), followed by a status code (“200”), and a status message (“OK”). If all goes well, the status will be 200. If the page is not found, the status code is 404.

💡 Note: you can find more information on http protocol on **An overview of HTTP (https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview)**

🎓 Hypertext Markup Language: HTML

Basic components of the web:

- HTML : the structure and the main content of the page
- CSS : some styling to make the page look nicer
- Javascript : adds interactivity to web pages
- Medias: images, music, videos. etc

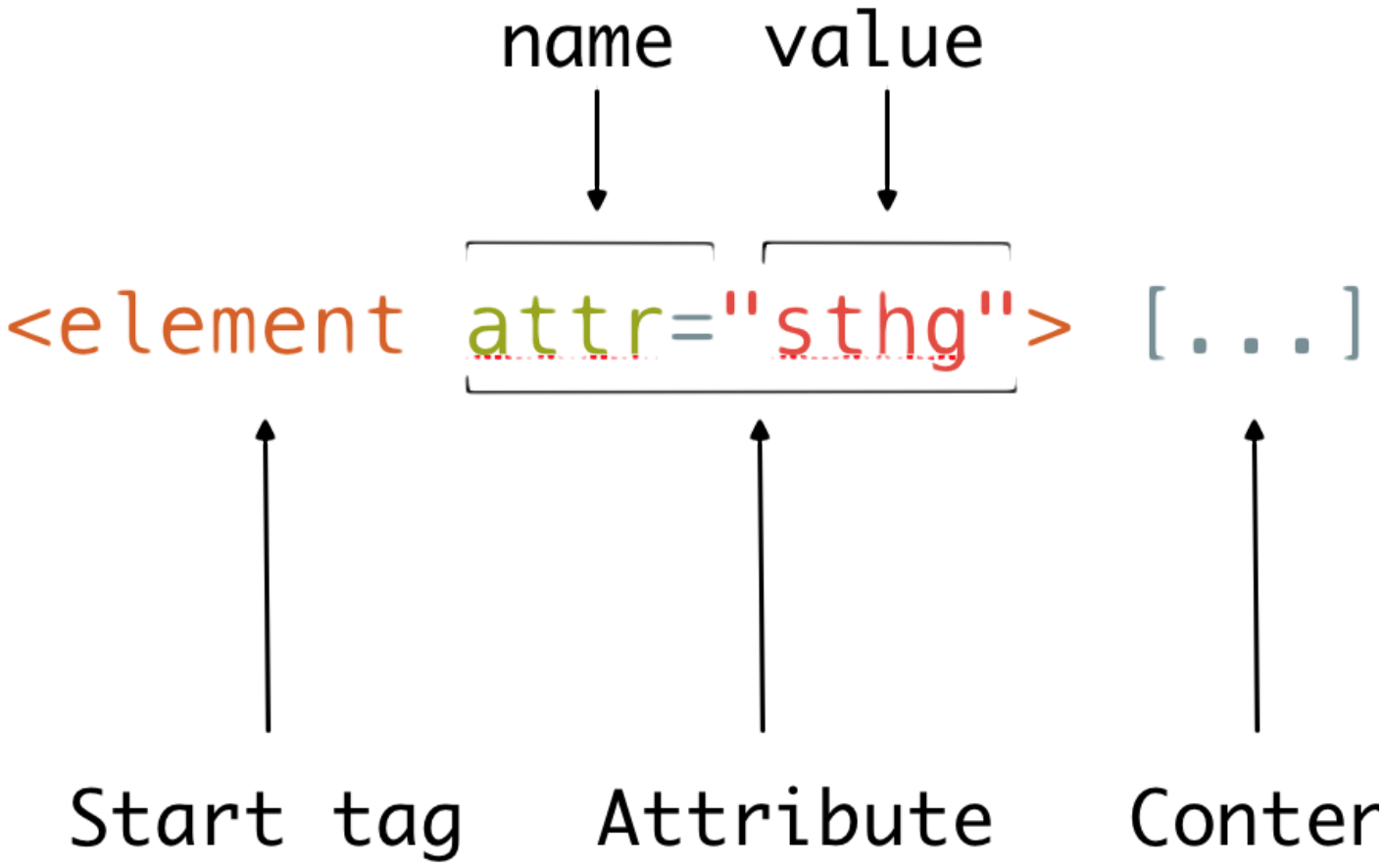
1️⃣ HTML

Hyper Text Markup Language (HTML) is the language used to tell computers how to structure webpages.

```
<html>
    <head>
    </head>
    <body>
        <h1>My first HTML page</h1>
        <p>
            Here's a first paragraph of text!
            <a href="https://www.griffith.edu.au/">Griffith University</a>
        </p>
        <p>
            Here's a second paragraph of text!
            <a href="https://www.python.org">Python</a>
        </p>
    </body>
</html>
```

2️⃣ HTML tags

Every piece of information is stored in **HTML tags**. A tag usually has to be opened with angle brackets `<>` and then closed `</>`. The combination of an opening tag and its corresponding closing tag and the content in between is called an **element**.



While inspecting the source code of a HTML page, you will mostly encounter the following tags:

- `h1`, `h2`, `h3`, `h4`, `h5`, `h6` — titles (or *headers*) from larger to smaller
- `p` — paragraph
- `a` — hyperlink
- `div` — group of content that belong to the same context (mostly use for vertical content, spread on several lines)
- `span` — same as `div`, but most often used for horizontal content (one-liner)
- `img` — image (a single tag = it does not require an end tag)
- `ul` — list of elements
  - `li` — element of the list (always **inside** the `<ul>...</ul>` tag)
- `form` — input form
- `table` — table (elements stored in rows and columns)
  - `th` — header of the table (usually stores the names of columns)
  - `tr` — row of the table
  - `td` — cell of the table (intersection of a single row and a single column)

The first tag of a web page is `<html>`. The interesting content is located in the `<body>`.

You can find a list of all HTML tags **here** **(https://developer.mozilla.org/en-US/docs/Web/HTML/Element)**

3️⃣ HTML attributes

HTML attributes are included in the opening tag. They provide extra details about the tag.

You will mainly encounter the following attributes:

- `href` —> the URL for a link
- `class` —> CSS class(es) for an element (see below)
- `id` —> a unique id for an element
- `src` —> the URL for an image
- `style` —> CSS properties for an element (see below)

4️⃣ Viewing a page's source in Chrome

You can open up Chrome's "Developer Tools." To do so, either select the Chrome Menu at the top right of your browser window, then select "Tools," "Developer Tools," or press `Control+Shift+I`. Alternatively, you can also right-click on any page element and select "Inspect Element." Other browsers such as Firefox and Microsoft
Edge have similar tools built in.

🎓Cascading Style Sheets: CSS

1️⃣ CSS

While HTML gives a webpage **structure and meaning**, CSS is in charge of adding **style attributes** to elements.

```
<html>
    <head>
    </head>
    <body>
        <h1 style="color: red; background: pink">My first HTML page</h1>
        <p class="bold-paragraph extra-large">
            Here's a first paragraph of text!
            <a href="https://vivadata.org">VIVADATA - AI Programming School</a>
        </p>
        <p class="bold-paragraph extra-large">
            Here's a second paragraph of text!
            <a href="https://www.python.org" id="headline">Python</a>
        </p>
    </body>
</html>
```

These style declarations can be included in a document in three different ways:

- Inside a regular HTML **style** attribute, for instance as in: `<p style="color:'red';">...</p>`.
- Inside of HTML `<style>...</style>` tags, placed in inside the `<head>` tag of a page.
- Inside a separate file, which is then referred to by means of a `<link>` tag inside the `<head>` tag of a page. This is the cleanest way of working. When loading a web page, your browser will perform an additional HTTP request to download this CSS file and apply its defined styles to the document.

2️⃣ CSS Selectors

In case style declarations are placed inside a "style" attribute, it is clear to which element the declarations should be applied: the HTML tag itself. In the other two cases, the style definition needs to incorporate information regarding the HTML element or elements a styling should be applied to. This is done by placing the style declarations inside curly brackets to group them, and putting a **CSS selector** at the beginning of each group, for example:

```
h1 {
  color: red;
}
div.box {
  border: 1px solid black;
}
#intro-paragraph {
  font-weight: bold;
}
```

CSS selectors define the patterns used to "select" the HTML elements you want to style. They are quite comprehensive in terms of syntax. The following list provides a full reference:

- `tagname` selects all elements with a particular tag name. For instance, "h1" simply matches with all "<h1>" tags on a page.
- `.classname` (note the dot) selects all elements having a particular class defined in the HTML document. This is exactly where the "class" attribute comes in. For instance, .intro will match with both `<p class="intro">`. Note that HTML elements can have multiple classes, for example, `<p class="intro highlight">`.
- `#idname` matches elements based on their "id" attribute. Contrary to classes, proper HTML documents should ensure that each "id" is unique and only given to one element only (though don't be surprised if some particularly messy HTML page breaks this convention and used the same id value multiple times). These selectors can be combined in all sorts of ways. div.box, for instance, selects all `<div class="box">` tags, but not `<div class="circle">` tags.
- Multiple selector rules can be specified by using a comma, ",", for example, `h1, h2, h3`.
- `selector1 selector2` defines a chaining rule (note the space) and selects all elements matching selector2 inside of elements matching selector1. Note that it is possible to chain more than two selectors together.
- `selector1 > selector2` selects all elements matching `selector2` where the parent element matches `selector1`. Note the subtle difference here with the previous line. A "parent" element refers to the

"direct parent." For instance, `div > span` will not match with the `span` element inside `"<div> <p> <span> </span> </p> </div>"` (as the parent element here is a `"<p>"` tag), whereas `div span` will.

- `selector1 + selector2` selects all elements matching selector2 that are placed directly after (i.e., on the same level in the HTML hierarchy) elements matching `selector1`.
- `selector1 ~ selector2` selects all elements matching selector2 that are placed after (on the same level in the HTML hierarchy) selector1. Again, there's a subtle difference here with the previous rule: the precedence here does not need to be "direct": there can be other tags in between.
- It is also possible to add more fine-tuned selection rules based on attributes of elements. `tagname[attributename]` selects all tagname elements where an attribute named `attributename` is present. Note that the tag selector is optional, and simply writing `[title]` selects all elements with a "title" attribute.
- The attribute selector can be further refined. `[attributename=value]` checks the actual value of an attribute as well. If you want to include spaces, wrap the value in double quotes.
- `[attributename~=value]` does something similar, but instead of performing an exact value comparison, here all elements are selected whose attributename attribute's value is a space-separated list of words, one of them being equal to value.
- `[attributename|=value]` selects all elements whose attributename attribute's value is a space-separated list of words, with any of them being equal to "value" or starting with "value" and followed by a hypen ("-").
- `[attributename^=value]` selects all elements whose attribute value starts with the provided value. If you want to include spaces, wrap the value in double quotes.
- `[attributename$=value]` selects all elements whose attribute value ends with the provided value. If you want to include spaces, wrap the value in double quotes.
- `[attributename*=value]` selects all elements whose attribute value contains the provided value. If you want to include spaces, wrap the value in double quotes.
- Finally, there are a number of "colon" and "double-colon" "pseudo-classes" that can be used in a selector rule as well. `p:first-child` selects every `<p>` tag that is the first child of its parent element, and `p:last-child` and `p:nth-child(10)` provide similar functionality.

🎓 Web Scraping with BeautifulSoup

1️⃣ Creating a BeautifulSoup object

We're now ready to start working with HTML pages using Python. Recall the following lines of code:

```
import requests
url = 'https://en.wikipedia.org/w/index.php' + \
      '?title=List_of_Game_of_Thrones_episodes&oldid=802553687'
r = requests.get(url)
print(r.text)
```

How do we deal with the HTML contained in html_contents? To properly parse and tackle this "soup," we'll bring in a library, called "Beautiful Soup."

Just as was the case with requests, installing Beautiful Soup is easy with pip:

```
pip install -U beautifulsoup4
```

Let get started and create a BeautifulSoup object. Try these codes:

```
import requests
url = 'https://en.wikipedia.org/w/index.php' + \
      '?title=List_of_Game_of_Thrones_episodes&oldid=802553687'
r = requests.get(url)
html_contents = r.text
html_soup = BeautifulSoup(html_contents, 'html.parser')
```

The Beautiful Soup library itself depends on an HTML parser to perform most of the bulk parsing work. In Python, multiple parsers exist to do so:

- `"html.parser"`: a built-in Python parser that is decent (especially when using recent versions of Python 3) and requires no extra installation.
- `"lxml"`: which is very fast but requires an extra installation.
- `"html5lib"`: which aims to parse web page in exactly the same way as a web browser does, but is a bit slower.

Since there are small differences between these parsers, Beautiful Soup warns you if you don't explicitly provide one, this might cause your code to behave slightly different when executing the same script on different machines. We'll stick with the default Python parser here:

```
html_soup = BeautifulSoup(html_contents, 'html.parser')
```

2 Working with BeautifulSoup object

Beautiful Soup's main task is to take HTML content and transform it into a tree-based representation. Once you've created a BeautifulSoup object, there are two methods you'll be using to fetch data from the page:

- `find(name, attrs, recursive, string, **keywords)`;
- `find_all(name, attrs, recursive, string, limit, **keywords)`.

Let's discuss the arguments of these two methods step by step

name	The ***name*** argument defines the tag names you wish to `find` on the page. You can pass a string, or a list of tags. Leaving this argument as an empty string simply selects all elements.
attrs	The ***attrs*** argument takes a Python dictionary of attributes and matches HTML elements that match those attributes.

recursive	The ***recursive*** argument is a Boolean and governs the depth of the search. If set to True — the default value, the find and find_all methods will look into children, children's children, and so on... for elements that match your query. If it is `False`, it will only look at direct child elements.
string	The ***string*** argument is used to perform matching based on the text content of elements.
limit	The ***limit*** argument is only used in the find_all method and can be used to limit the number of elements that are retrieved. Note that find is functionally equivalent to calling find_all with the limit set to 1, with the exception that the former returns the retrieved element directly, and that the latter will always return a list of items, even if it just contains a single element. Also important to know is that, when `find_all` cannot find anything, it returns an empty list, whereas if `find` cannot find anything, it returns `None`.
****keywords***	****keywords*** is kind of a special case. Basically, this part of the method signature indicates that you can add in as many extra named arguments as you like, which will then simply be used as attribute filters. Writing `find(id='myid')` is hence the same as `find(attrs={'id': 'myid'})`. If you define both the attrs argument and extra keywords, all of these will be used together as filters. This functionality is mainly offered as a convenience in order to write easier-to-read code.

Both find and find_all return Tag objects. Using these, there are a number of interesting things you can do:

- Access the name attribute to retrieve the tag name.
- Access the contents attribute to get a Python list containing the tag's children (its direct descendant tags) as a list.
- The `children` attribute does the same but provides an iterator instead; the `descendants` attribute also returns an iterator, now including all the tag's descendants in a recursive manner. These attributes are used when you call find and find_all.
- Similarly, you can also go "up" the HTML tree by using the parent and parents attributes. To go sideways (i.e., find next and previous elements at the same level in the hierarchy), `next_sibling`, `previous_sibling` and `next_siblings`, and `previous_siblings` can be used.
- Converting the Tag object to a string shows both the tag and its HTML content as a string. This is what happens if you call print out the Tag object, for instance, or wrap such an object in the str function.
- Access the attributes of the element through the attrs attribute of the Tag object. For the sake of convenience, you can also directly use the Tag object itself as a dictionary.
- Use the text attribute to get the contents of the Tag object as clear text (without HTML tags).
- Alternatively, you can use the `get_text` method as well, to which a strip Boolean argument can be given so that `get_text(strip=True)` is equivalent to `text.strip()`. It's also possible to specify a string to be used to join the bits of text enclosed in the element together, for example, `get_text('--')`.

- If a tag only has one child, and that child itself is simply text, then you can also use the string attribute to get the textual content. However, in case a tag contains other HTML tags nested within, string will return None whereas text will recursively fetch all the text.
- Finally, not all find and find_all searches need to start from your original BeautifulSoup objects. Every Tag object itself can be used as a new root from which new searches can be started.

Let's show off these concepts through example code:

```
import requests
from bs4 import BeautifulSoup
url = 'https://en.wikipedia.org/w/index.php' + \
      '?title=List_of_Game_of_Thrones_episodes&oldid=802553687'
r = requests.get(url)
html_contents = r.text
html_soup = BeautifulSoup(html_contents, 'html.parser')
# Find the first h1 tag
first_h1 = html_soup.find('h1')
print(first_h1.name) # h1
print(first_h1.contents) # ['List of ', [...], ' episodes']
print(str(first_h1))
# Prints out: <h1 class="firstHeading" id="firstHeading" lang="en">List of
#             <i>Game of Thrones</i> episodes</h1>
print(first_h1.text) # List of Game of Thrones episodes
print(first_h1.get_text()) # Does the same
print(first_h1.attrs)
# Prints out: {'id': 'firstHeading', 'class': ['firstHeading'], 'lang': 'en'}
print(first_h1.attrs['id']) # firstHeading
print(first_h1['id']) # Does the same
print(first_h1.get('id')) # Does the same
print('------------ CITATIONS ------------')
# Find the first five cite elements with a citation class
cites = html_soup.find_all('cite', class_='citation', limit=5)
for citation in cites:
  print(citation.get_text())
  # Inside of this cite element, find the first a tag
  link = citation.find('a')
  # ... and show its URL
  print(link.get('href'))
  print()
```

3 Chain of tag shorthand

If you find yourself traversing a chain of tag names as follows:

```
tag.find('div').find('table').find('thead').find('tr')
tag.find_all('h1')
```

You can rewrite it like this:

```
tag.div.table.thead.tr
tag('h1')
```

Let us now try to work out the following use case. You'll note that our Game of Thrones Wikipedia page has a number of well-maintained tables listing the episodes with their directors, writers, air date, and

number of viewers. Let's try to fetch all of this data at once using Chain of Tag shorthand:

```
import requests
from bs4 import BeautifulSoup
url = 'https://en.wikipedia.org/w/index.php' + \
'?title=List_of_Game_of_Thrones_episodes&oldid=802553687'
r = requests.get(url)
html_contents = r.text
html_soup = BeautifulSoup(html_contents, 'html.parser')
# We'll use a list to store our episode list
episodes = []
ep_tables = html_soup.find_all('table', class_='wikiepisodetable')
for table in ep_tables:
  headers = []
  rows = table.find_all('tr')
  # Start by fetching the header cells from the first row to determine
  # the field names
  for header in table.find('tr').find_all('th'):
    headers.append(header.text)
    # Then go through all the rows except the first one
    for row in table.find_all('tr')[1:]:
      values = []
      # And get the column cells, the first one being inside a th-tag
      for col in row.find_all(['th','td']):
        values.append(col.text)
      if values:
        episode_dict = {headers[i]: values[i] for i in
        range(len(values))}
        episodes.append(episode_dict)
# Show the results
for episode in episodes:
  print(episode)
```

4 More on Beautiful Soup

👉 Using Regex as argument

We have already seen how you can filter on a simple tag name or a list of them:

```
html_soup.find('h1')
html_soup.find(['h1', 'h2'])
```

However, these methods can also take other kinds of objects, such as a regular expression object. The following line of code will match with all tags that start with the letter "h" by constructing a regular expression using Python's "re" module:

```
import re
html_soup.find(re.compile('^h'))
```

👉 Using Function as argument

Apart from strings, lists, and regular expressions, you can also pass a function. This is helpful in complicated cases where other approaches wouldn't work:

```
def has_classa_but_not_classb(tag):
  cls = tag.get('class', [])
  return 'classa' in cls and not 'classb' in cls

html_soup.find(has_classa_but_not_classb)
```

👉 Other methods for searching the HTML tree

Apart from `find` and `find_all`, there are also a number of other methods for searching the HTML tree, which are very similar to `find` and `find_all`. The difference is that they will search different parts of the HTML tree:

- `find_parent` and `find_parents` work their way up the tree, looking at a tag's parents using its parents attribute. Remember that find and find_all work their way down the tree, looking at a tag's descendants.
- `find_next_sibling` and `find_next_siblings` will iterate and match a tag's siblings using `next_siblings` attribute.
- `find_previous_sibling` and `find_previous_siblings` do the same but use the previous_siblings attribute.
- `find_next` and `find_all_next` use the `next_elements` attribute to iterate and match over whatever comes after a tag in the document.
- `find_previous` and `find_all_previous` will perform the search backward using the `previous_elements` attribute instead.
- Remember that `find` and `find_all work` on the children attribute in case the recursive argument is set to `False`, and on the descendants attribute in case recursive is set to `True`.

👉 Extremely useful function: `select`

Using this method, you can simply pass a CSS selector rule as a string.

```
# Find all <a> tags
html_soup.select('a')
# Find the element with the info id
html_soup.select('#info')
# Find <div> tags with both classa and classb CSS classes
html_soup.select(div.classa.classb)
# Find <a> tags with an href attribute starting with http://example.com/
html_soup.select('a[href^="http://example.com/"]')
# Find <li> tags which are children of <ul> tags with class lst
html_soup.select(ul.lst > li')
```

Once you start getting used to CSS selectors, this method can be very powerful indeed. For instance, if we want to find out the citation links from our Game of Thrones Wikipedia page, we can simply run:

```
for link in html_soup.select('ol.references cite a[href]'):
  print(link.get('href'))
```

However, the CSS selector rule engine in Beautiful Soup is not as powerful as the one found in a modern web browser. The following rules are valid selectors, but will not work in Beautiful Soup:

```
# This will not work:
# cite a[href][rel=nofollow]
# Instead, you can use:
tags = [t for t in html_soup.select('cite a[href]') \
          if 'nofollow' in t.get('rel', [])]
# This will not work:
# cite a[href][rel=nofollow]:not([href*="archive.org"])
# Instead, you can use:
tags = [t for t in html_soup.select('cite a[href]') \
          if 'nofollow' in t.get('rel', []) and 'archive.org' not in t.get('href')]
```

Cases where you need to resort to such complex selectors are rare, and remember that you can still use `find`, `find_all`.