

# Chapter 8 Web Scraping

Most webpages are designed for humans to look and read. But sometimes we do not want to look and read, but collect the data from the pages instead. This is called *web scraping*. The challenge with web scraping is getting the data out of pages that are not designed for this purpose.

## 8.1 Before you begin

Web scraping means extracting data from the “web”. However, web is not just an anonymous internet “out there” but a conglomerate of servers and sites, built and maintained by individuals, businesses and governments. Extracting data from there inevitably means using the resources and knowledge someone else has put into the websites. So we have to be careful from both legal and ethical perspective.

From the ethical side, you should try to minimize the problems you cause to the websites you are scraping. This involves the following steps:

- limit the number of queries to the necessary minimum. For instance, when developing your code, download the webpage once, and use the cached version for developing and debugging. Do not download more before you actually need more for further development. Do the full scrape only after the code has been well-enough tested. Store the final results in a local file.
- limit the frequency of queries to something the server can easily handle. For a small non-profit, consider to send only a handful of requests per minute, while a huge business like google can easily handle thousands of requests per second (but they may recognize you scraping and block you).
- consult the *robots.txt* file and understand what is allowed, what is not allowed. Do not download pages that the file does not allow to scrape.

*robots.txt* is a text file with simple commands for web crawlers, describing what the robots should and should not do. The file can be obtained by adding *robots.txt* at the end of the base url of the website. For instance, the *robots.txt* for the web address <https://ischool.uw.edu/events> is at <https://ischool.uw.edu/robots.txt> as the base url is <https://ischool.uw.edu>. A *robots.txt* file may look like:

```
User-agent: *  
Allow: /core/*.css$  
Disallow: /drawer/
```

This means all crawlers (user agent `*`) are allowed to read all files ending with *css* in *core*, e.g. <https://ischool.uw.edu/core/main.css>. But they are not allowed to read anything from *drawer*, e.g. <https://ischool.uw.edu/drawer/schedule.html>. There are various simple introductions to *robots.txt*, see for instance [moz.com](https://moz.com).

A related issue is legality. You should only scrape websites and services where it is legal. But in recent years it is getting more and more common for the sites to explicitly ban it. For instance, [allrecipes.com](https://allrecipes.com) states in [Terms of Service](#) that:

```
(e) you shall not use any manual or automated software, devices or  
other processes (including but not limited to spiders, robots,  
scrapers, crawlers, avatars, data mining tools or the like) to  
"scrape" or download data from the Services ...
```

Some websites permit downloading for “personal non-commercial use”. GitHub states in its [Acceptable Use Policies](#) that

You may scrape the website for the following reasons:

- \* Researchers may scrape public, non-personal information from the Service for research purposes, only if any publications resulting from that research are open access.
- \* Archivists may scrape the Service for public data for archival purposes.

You may not scrape the Service for spamming purposes, including for the purposes of selling User Personal Information (as defined in the GitHub Privacy Statement), such as to recruiters, headhunters, and job boards.

All use of data gathered through scraping must comply with the GitHub Privacy Statement.

There is also a plethora of websites that do not mention downloading, robots and scraping. Scraping such pages is a legally gray area. Other websites that are concerned with what happens to the scraped data. [Feasting at home](#) states:

You may NOT republish my recipe(s). I want Feasting at Home to remain the exclusive source for my recipes. Do not republish on your blog, printed materials, website, email newsletter or even on social media- always link back to the recipe.

Again, my recipes are copyrighted material and may not be published elsewhere.

While the legal issues may feel like a nuisance for a technology enthusiast, web scraping touches genuine questions about property rights, privacy, and free-riding. After all, many website creators have done a real effort and spent non-trivial resources to build and maintain the website. They may make the data available for browsers (not scrapers!) to support their business plan. But scrapers do not help with their business plan, and in some case may forward the data to a competitor instead.

In certain cases it also rises questions of privacy. Scraping even somewhat personal data (say, public social media profiles) for a large number of people and connecting this data with other resources may be a privacy violation. If you do this for research purposes, you should store the scraped data in a secure location, and not attempt to identify the real people in data!

In conclusion, before starting your first scraping project, you should answer these questions:

- Is it ethical to download and use the data for the purpose I have in mind?
- What can I do to minimize burden to the service providers?
- Is it legal?
- How should I store and use data?

## 8.2 HTML Basics

This section introduces basics of HTML. If you are familiar with HTML then you can safely skip forward to [Beautiful Soup](#) section.

HTML (hypertext markup language) is a way to write text using tags and attributes to mark the text structure. HTML is the standard language of internet, by far the most webpages are written in html and hence one needs to understand some HTML in order to be able to process the web. Below we briefly discuss the most important structural elements from the web scraping point of view.

### 8.2.1 Tags, elements and attributes

HTML pages are made of HTML *elements*. Most elements denote parts of page structure, such as header, paragraph, or image. Elements are enclosed inside *tags*. In common cases there are two versions of the tags—*beginning tag* and *end tag*. For instance, text beginning tag for a paragraph of text is `<p>` and the corresponding end tag is `</p>`, so an html document may contain

```
<p>This is a paragraph of text.</p>
```

In this case the `p`-tags enclose a paragraph, so `p`-s are tags and the paragraph is the element.

Tags can be nested, i.e. an element, such as paragraph, can contain other tags. Here is an example of emphasized text ( `<em>` ) inside of paragraph:

```
<p>This paragraph contains an <em>important</em> message.</p>
```

In this case `p`-tags enclose a paragraph element, and inside of this element we have another element, enclosed in `em`-tags.

Not all elements are enclosed between beginning and end tag. One such example is image, tagged by `<img>`. It has not closing tag, all the information is given inside of the image tag as *attributes*. An webpage may contain image using

```

```

This tag adds an image to the text. It has two attributes: *src* tells where on the web (or on the disk) is the actual image file, and *alt* gives an alternative text that is displayed in case the image cannot be displayed (e.g. the browser is a screen reader).

Two attributes that are very important from scraping purpose are *class* and *id*. *class* is used extensively to define certain layout elements, and *id* to uniquely identify the elements. For instance, a html page may contain

```
<div id="first-choice">
  <p><span class="red-warning">Do not</span> jump into water!</p>
</div>
```

It is up to the webpage to define how exactly should browser render both *div* (vertical block of text) and *span* (horizontal block of text), but here you may look for both elements with given id and class. The document may contain many other span-s, but here we may want to catch a *span* with class “red-warning”. In a similar fashion, we may be interested one *div* that is identified as “first-choice”.

## 8.2.2 Overall structure

A valid html document contains the doctype declaration, followed by `<html>` tag (see the example below). Everything that is important from the scraping perspective is embedded in the html-element. Html-element in turn contains two elements: *head* and *body*. *Head* contains various header information, including the page title (note—it is not the title that is displayed on page), stylesheets and other general information. *Body* contains all the text and other visual elements that the browser actually displays. So a minimalistic html-file might look like:

```

<!DOCTYPE html>
<html>
  <head>
    <title>Mad Monk Ji Gong</title>
  </head>
  <body>
    <h1>Ji Visits Buddha</h1>
    <p>Ji Gong went to visit Buddha</p>
  </body>
</html>

```

This tiny file demonstrates all the structure we have discussed so far:

- The first declaration in the file is *DOCTYPE* entry.
- All the content is embedded in the *html*-element.
- The *html*-element contains *head* and *body* elements.
- *head* includes element *title*, the title of the webpage. This is not what you see on the page, but browser may show it on the window title bar, and it may use it as the file name when you download and save the page.
- The *body*-element contains two elements: *h1*, this is the top title that is actually rendered on screen (typically big and bold), and a paragraph of text.

Because HTML elements are nested inside each other, we can depict an HTML page as a tree. The *html* element is the trunk that branches into two, *head* and *body*; and *body* in turn branches into other elements. Thinking about the page as a tree is a extremely useful way when designing code to navigate the it.

## 8.2.3 Important tags

Below is a list of some of the tags that are important from the webscraping perspective. The exact definition of tags is often not necessary to know, and often you can have good enough guess by just comparing the html code with the visual layout. But it definitely helps to be familiar with the most important tags.

- **h1, h2, h3, ...** these are headers with *h1* being the top-level header, *h2* is a subsection header and so on. Headers are typically rendered big and bold with *h1* being the biggest and boldest, and all lower-level headers successively less emphasized.
- **p** is a paragraph of text

- **a** is “anchor”, hypertext link. The link address is embedded in the *href* attribute, the link text (where you click) is between beginning tag `<a>` and end tag `</a>` tags. An anchor element may look like

```
<a href="www.example.com">click here</a>
```

- **ol** and **ul** or “ordered list” (numbered list) and “unordered list” (bullet list without numbers).
- **li** is the list item, a line inside of *ol* or *ul* list. Here is an example of ordered list:

```
<ol>
  <li>Wake up</li>
  <li>Snooze your alarm</li>
  <li>Fall asleep again</li>
</ol>
```

- **div** is a vertical block of text that typically contains multiple lines. These do not have any default visual layout but are typically important part of page structure. They usually have attributes like *class* or *id*.
- **span** is a horizontal block of text (in a single line). As in case of **div**, these do not have default visual properties but are important elements in page structure.

This basic introduction is enough to help you to understand the webpages from the scraping perspective. There are many good sources for further information, you may consider [W3Schools](#) for html and css tutorials, and [W3C](#) for detailed specifications.

## 8.3 Beautiful Soup

This section discusses the basics of webscraping using Beautiful Soup (BS) library. It is the most popular web scraping library in python. In essence it is an HTML parser with added functionality to search tags, classes and attributes, and move up and down in the HTML tree. These notes only contain the very basic introduction, the complete documentation is available at [crummy.com](#).

Normally one scrapes web by downloading the pages, using for instance `requests` library and thereafter parsing the pages using BS as the latter is devoted to parsing in-memory html pages.

## 8.3.1 Example html file

In the following we demonstrate the usage of library on an example page

```
<!DOCTYPE html>
<html>
  <head>
    <title>Mad Monk Ji Gong</title>
  </head>
  <body>
    <h1>Li Visits Buddha</h1>
    <p>This happened
      during <a href="https://en.wikipedia.org/wiki/Song_dynasty">Song Dynasty</a>.
      The patchwork robe made for <strong>Guang Liang</strong>...</p>
    <h2>Li Begs for a Son</h2>
    <p class="quote">When I wa strolling in the street,
      <footnote>They lived in Linan</footnote> almost
      everyone was calling me
      <span class="nickname">Virtuous Li</span> ...</p>
  </body>
</html>
```

This file includes a number of common html tags and attributes and a suitably rich structure for explaining the basics of Beautiful Soup. If you want to follow the examples, then you may copy the file from here and save it as “scrape-example.html”, or download it directly [from the repo](#).

## 8.3.2 Loading Beautiful Soup and opening the data

Beautiful Soup is located in `bs4` module, so normally one loads it as

```
from bs4 import BeautifulSoup
```

The first task after importing the module is to load the html page. When scraping data from web, we can do it using `requests` module:

```
import requests

htmlPage = requests.get("www.example.com")
```

However, for now we work with a local file so we have to load the file instead:



```
htmlPage = open("../files/scrape-example.html").read()
```

This loads the example file into a string variable `htmlPage` . If we print the string, we will see the original html text. Next, we parse it through BS:

```
soup = BeautifulSoup(htmlPage, "html.parser")  
type(soup)
```

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

This results in a parsed object of class `bs4.BeautifulSoup` . “html.parser” tells to parse the object as html (not as xml or something else). If we leave the parser unspecified in the call then Beautiful Soup can usually figure it out, but this may occasionally create issues, so better to always specify the desired parser.

We are done with parsing! What is left is to navigate the resulting parsed structure and extract the elements of interest.

### 8.3.3 The hard part: navigating the soup and extracting data

The next, and typically the most complex part of web scraping is to navigate the html tree and extract the data of interest. What makes it hard is the fact that each web page is different, even more, the pages occasionally change their structure and hence the code that worked just yesterday may suddenly stop working. The data we are interested may be stored in more or in a less structured form, making it sometimes easy and sometimes hard to find in an automated fashion.

Below, we list a number of navigation and data extraction methods in BS and demonstrate their usage on the example webpage.

#### 8.3.3.1 Finding tags

The simplest way is to use the tags as attributes: `soup.tag` returns the element in the tags. We can extract the first *head* as

```
HTML = soup.html
HEAD = HTML.head
print(HEAD)
```

```
## <head>
## <title>Mad Monk Ji Gong</title>
## </head>
```

Note that the result `head` is not a character string despite being printed like this, but a BeautifulSoup object:

```
type(HEAD)

## <class 'bs4.element.Tag'>
```

This multi-step approach can be shortened by chaining the methods:

```
head = soup.html.head
print(HEAD)

## <head>
## <title>Mad Monk Ji Gong</title>
## </head>
```

or, as there is only a single head, we can just leave out the `html` tag in the middle:

```
print(soup.head)

## <head>
## <title>Mad Monk Ji Gong</title>
## </head>
```

This makes BS to look through the parsed tree and return the first element with the requested tag. So in a similar fashion we do not have to specify all the tags in `soup.html.head.title` as there is only a single title:

```
print(soup.title)
```

```
## <title>Mad Monk Ji Gong</title>
```

If there is more than one element with this tag, this approach will only return the first one:

```
print(soup.p)
```

```
## <p>This happened
```

```
##   during <a href="https://en.wikipedia.org/wiki/Song_dynasty">Song Dynasty</a>.
```

```
##   The patchwork robe made for <strong>Guang Liang</strong>...</p>
```

Alternatively, one can use the `find(tag)` method to achieve the same result. If called without any other arguments `soup.find("title")` is equivalent to `soup.title`. But `find` method supports a number of additional options that are extremely useful for more complex pages. For instance, we can search for a paragraph ( `p` ) with class “quote”:

```
print(soup.find("p", class_ = "quote"))
```

```
## <p class="quote">When I was strolling in the street,
```

```
##   <footnote>They lived in Linan</footnote> almost
```

```
##   everyone was calling me
```

```
##   <span class="nickname">Virtuous Li</span> ...</p>
```

This gives us the first paragraph of the class “quote” which happens to be the second paragraph in the file. Note that as `class` is a python keyword, the corresponding argument is called `class_`.

It is not necessary to specify the tag when using `find`, we can also just search by class only (or by any other attribute):

```
print(soup.find(class_ = "nickname"))
```

```
## <span class="nickname">Virtuous Li</span>
```

Finally, if you want to extract all tags, you can use `find_all` method. It returns all elements that match the query in a list:

```
Ps = soup.find_all("p")
print(Ps)

## [<p>This happened
##   during <a href="https://en.wikipedia.org/wiki/Song_dynasty">Song Dynasty</a>.
##   The patchwork robe made for <strong>Guang Liang</strong>...</p>, <p class="quote">
##   <footnote>They lived in Linan</footnote> almost
##   everyone was calling me
##   <span class="nickname">Virtuous Li</span> ...</p>]
```



This list can be used as any other list, e.g. we can query the class of the first `p` object in the list:

```
type(Ps[0])

## <class 'bs4.element.Tag'>
```

### 8.3.3.2 Extracting content

The extracted elements, although they appear as text lines from the file, are not text but belong to BS internal classes. If we want to extract the text inside of the element as text, we can do this with the `.text` attribute. Here is the text of the second paragraph that we extracted above:

```
print(Ps[1].text)

## When I was strolling in the street,
##   They lived in Linan almost
##   everyone was calling me
##   Virtuous Li ...
```

Note that it also strips the text from inner tags but preserves eventual line breaks and spacing:

Html attributes (such as “class” or “href”) can be extracted using square brackets (like in case of dicts). Let’s extract the hyperlink inside of the first “a” tag:

```
soup.a["href"]

## 'https://en.wikipedia.org/wiki/Song_dynasty'
```

And here is the class of the second paragraph:

```
Ps[1]["class"]

## ['quote']
```

We can also explicitly get the tag using `name` attribute. By inverting the previous example, let’s find the first element of class “quote” and print it’s tag:

```
soup.find(class_ = "quote").name

## 'p'
```

We see that the element we found is a paragraph.

### 8.3.3.3 Moving up and down the tree

Beautiful Soup uses concepts *children*, *siblings*, *descendants*, and *parents* to navigate the html tree. *Children* are elements that are directly inside of a parent element and not inside other elements that are inside parent elements (those are grandchildren). *Parents* are elements that children are directly inside

One can get an iterable collection<sup>5</sup> of all first-level children of a tag with the `children` attribute. The following example prints the corresponding all the identified tags inside the html body:

```
children = soup.body.children
for child in children:
    print(child.name)
```

```
## None
## h1
## None
## p
## None
## h2
## None
## p
## None
## h2
## None
## div
## None
```

We see that the result contains a large number of `None` -s. These are line breaks, spaces, and other text between html elements:

```
children = list(soup.body.children)
print(children[0]) # Line break after body, before header
```

```
print(children[1]) # h1 header
```

```
## <h1>Li Visits Buddha</h1>
```

```
print(children[2]) # Line break after header
```

If you want to list all descendants of an element, i.e. to include grandchildren and such, you can use `descendants` instead of `children` :

```
children = soup.body.descendants
for child in children:
    print(child.name)
```

## None  
## h1  
## None  
## None  
## p  
## None  
## a  
## None  
## None  
## strong  
## None  
## None  
## None  
## h2  
## None  
## None  
## p  
## None  
## footnote  
## None  
## None  
## span  
## None  
## None  
## None  
## h2  
## None  
## None  
## div  
## None  
## table  
## None  
## thead  
## None  
## tr  
## None  
## th  
## None  
## None  
## th  
## None  
## None  
## None  
## tbody

```

## None
## tr
## None
## td
## None
## None
## td
## None
## None
## None
## tr
## None
## td
## None
## None
## td
## None
## None
## None
## None
## None
## None

```

Moving horizontally among the same level elements proceeds with `find_next_sibling` , `find_next_siblings` , `find_previous_sibling` and `find_previous_siblings` . Moving back-and forth disregarding the nesting level can be done with `find_next` , `find_all_next` , `find_previous` and `find_all_previous` . The *siblings* and *all* versions find all the occurrences, the others find just the first occurrence. All these functions accept various parameters so one can specify tag names and attributes.

Let us extract second “p” inside of “body”, and thereafter move back to first preceding “h2”:

```

ps = soup.body.find_all("p") # List of both b-s
p = ps[1] # 2nd p
h2 = p.find_previous_sibling("h2") # h2 before the 2nd p
h2.text # print the text inside h2

## 'Li Begs for a Son'

```

### 8.3.3.4 Removing elements from the tree

Let us extract the second “p” structure from the document:



```
p = soup.body.find("p", class_ = "quote").text
print(p)
```

```
## When I was strolling in the street,
##     They lived in Linan almost
##     everyone was calling me
##     Virtuous Li ...
```

The problem here is that the extracted text also includes the footnote that is placed out of context in the middle of the extracted text. In particular, *Linan* in the second line belongs to the footnote and the following *almost* belongs to the main text, but there is no way to tell this based on the printout. Sometimes this is what we want, e.g. when we strip text from its attributes like “i” and “b” (for italic and bold respectively). But here it is clearly undesirable. As a solution, we can delete the “footnote” element using `extract` method. However, if we extract an element, it will be deleted from the whole source tree and we cannot access it later. So we may want to make a copy of the element:

```
import copy
p = soup.body.find("p", class_ = "quote") # find the quote paragraph
p = copy.copy(p) # make copy of p (and forget the original)
footnote = p.footnote.extract() # find the first footnote there and remove it
print(p.text) # print the final text
```

```
## When I was strolling in the street,
##     almost
##     everyone was calling me
##     Virtuous Li ...
```

Indeed, the annoying footnote is not printed. However, it is still preserved in the original source tree:

```
p = soup.body.find("p", class_ = "quote")
print(p.text)
```

```
## When I was strolling in the street,
##     They lived in Linan almost
##     everyone was calling me
##     Virtuous Li ...
```

`copy.copy` ensures that all the deep structure of complex BS objects will be copied correctly.

### 8.3.3.5 Following links on the page

Most HTML pages are full of links and often we want to follow the links to wherever they are linking to. This is fairly simple, and involves the following steps:

- find the link. HTML links are stored as *href* attributes of *a* elements, so one can extract those as:

```
link = a["href"]
```

given you have extracted the *a* element and named it *a*.

- ensure that the link is *absolute*, not relative. Namely, websites typically encode links in relative manner by leaving out the base url address. For instance, wikipedia entry about China links to Taiwan as

```
For the Republic of China, see  
<a href="/wiki/Taiwan" title="Taiwan">Taiwan</a>
```

The *href* attribute is relative: `/wiki/Taiwan` is not a valid web address. One has to add the base url *en.wikipedia.org* in front of it so the final address would be *en.wikipedia.org/wiki/Taiwan*. The safest way to achieve this is to use *urllib* that takes care of various corner cases:

```
from urllib.request import urljoin  
  
base = "https://en.wikipedia.org"  
link = "/wiki/Taiwan.html"  
address = urljoin(base, relativeLink) # en.wikipedia.org/wiki/Taiwan
```

- Now we have the web address and we can retrieve the corresponding page using `requests.get`, exactly as before.


## 8.4 Finding elements on webpage

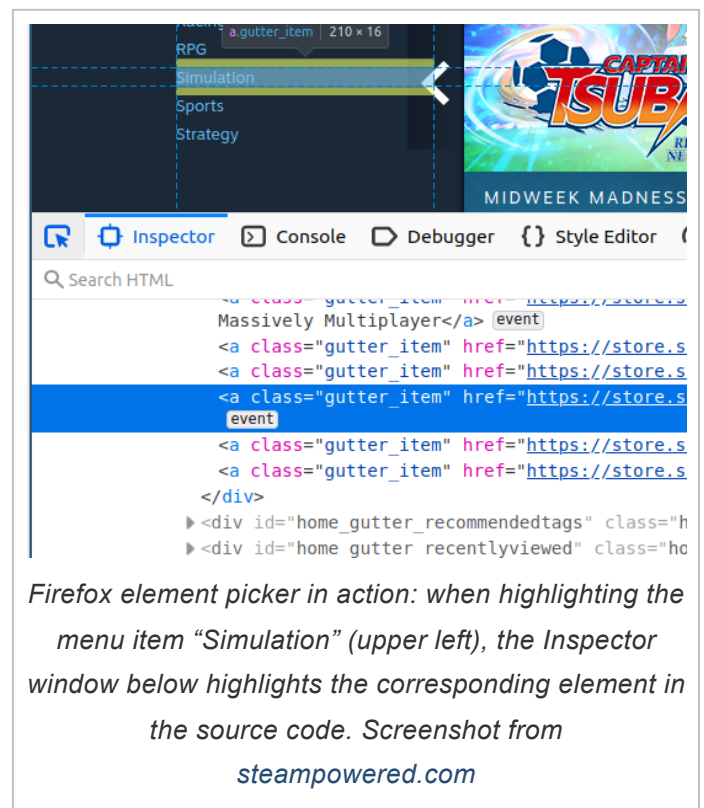
All the examples above were done for a very simple webpage. But modern website are often much more complex. How can one locate the elements of interest in the source?

For simple webpages one can just look at the page source in the browser, or open the html document in a text editor. The source can be easily translated into the parse tree navigation algorithm. But modern complex pages are often very complicated and hard to understand by just consulting the source.

For complex pages, the easiest approach is to use browsers' developer tools. Modern browser, such as Firefox and Chrome, contain web developers' tools. These can be invoked by *Ctrl-Shift-I* in both Firefox and Chromium on Linux and Windows, *Cmd-option-I* (⌘ - ⌥ - I) on Mac. However, these tools are also excellent means to locate elements of interest on the page.

A particularly useful tool is element picker

(labeled  both in Firefox and Chromium) that highlights the element in the html source that you point on the webpage. In the figure at right one can see that the menu elements are contained in "a" tags with class "gutter\_item". If we are interested in scraping the menu, we may try to locate such elements in the parsed html tree.



However, this approach only works if both browser and scraper actually download the same page. If browser shows you the javascript-enabled version targeted for your powerful new browser, but scraper gets a different version for non-javascript crawlers, one has to dive deeper into the developer tools and fool the website to send the crawler version to the browser too.

Finally, one can always just download the page, parse its html and walk through the various elements in a search for the data in question. But this is often tedious even for small pages.

5. Iterable collection is something we can loop over↔