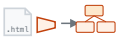
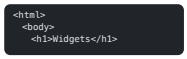
Beautiful Soup

A quick reference for pulling data out of messy HTML with Beautiful Soup, parsing a document and searching it by tag and CSS selector.

blog.thecoatlessprofessor.com
updated 06/18/2026

Parse a Document

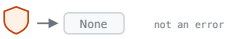
Turn an HTML string into a navigable tree; pick a parser.

PURPOSE	COMMAND	RESULT
Parse with the built-in parser.	`soup = BeautifulSoup(html, "html.parser")`	
Parse fast and lenient (lxml).	`BeautifulSoup(html, "lxml")`	
Parse browser-style (html5lib).	`BeautifulSoup(html, "html5lib")`	
Parse a file object.	`BeautifulSoup(open("page.html"), "html.parser")`	
Pretty-print the parsed tree.	`print(soup.prettify())`	
Parse only XML.	`BeautifulSoup(xml, "xml")`	

name the parser explicitly: html.parser is built in, lxml is fast, html5lib parses exactly like a browser

Read Tags & Attributes

A Tag is a dict of attributes plus a node you can descend into.

PURPOSE	COMMAND	RESULT
Reach the first matching tag.	`soup.h1` `soup.title`	
Get a tag's name.	`tag.name`	
Get one attribute (dict-style).	`tag["href"]` `tag["class"]`	
Safe attribute lookup.	`tag.get("href")` `tag.get("x", default)`	
All attributes as a dict.	`tag.attrs`	
Test for an attribute.	`tag.has_attr("class")`	

tag["href"] raises if missing, tag.get("href") returns None; class comes back as a list (multiple classes allowed)

Search: find / find_all

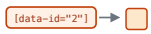
The two workhorses: one match or a list, filtered by name, attrs, text.

PURPOSE	COMMAND	RESULT
First match (or None).	`soup.find("a")`	
All matches as a list.	`soup.find_all("li")`	
Filter by CSS class.	`soup.find_all("a", class_="name")`	
Filter by any attribute.	`soup.find_all("li", attrs={"data-id": "2"})`	
Match attribute by regex.	`soup.find_all("a", href=re.compile(r"^/p/"))`	
Cap the number of results.	`soup.find_all("li", limit=2)`	

find returns the first Tag or None; find_all returns a list; filter by name, class_=, attrs={...}, regex, or limit=

Search: CSS Selectors

select() / select_one(): the same selectors you'd use in a browser.

PURPOSE	COMMAND	RESULT
All matches for a selector.	`soup.select("li.product")`	
First selector match.	`soup.select_one("a.name")`	
Descendant combinator.	`soup.select("ul.products .price")`	
Attribute selector.	`soup.select('li[data-id="2"]')`	
Direct child / grouping.	`soup.select("ul > li") ("h1, h2")`	
Positional pseudo-class.	`soup.select("li:nth-of-type(2)")`	

select() accepts browser-style selectors (Soup Sieve): descendant chains, > children, [attr], grouping, and pseudo-classes

Walk the Tree

Move up to parents, down to children, sideways to siblings.

PURPOSE	COMMAND	RESULT
Direct children (tags + strings).	`tag.children` `tag.contents`	
All descendants (deep).	`tag.descendants`	
Go up to the parent.	`tag.parent` `tag.parents`	
Next / previous sibling tag.	`tag.find_next_sibling("li")`	
Find an ancestor by filter.	`tag.find_parent("li")`	
Find next match anywhere.	`tag.find_next("span")`	

prefer find_next_sibling and find_parent over raw .next_sibling/.parent when whitespace text nodes get in the way

Pull the Data Out

Get text and attributes, clean and collected.

PURPOSE	COMMAND	RESULT
All visible text, joined.	`tag.get_text()`	
Text with a separator, trimmed.	`tag.get_text(separator=" | ", strip=True)`	
One tag's own string.	`tag.string`	
Iterate cleaned text pieces.	`list(tag.stripped_strings)`	
Collect an attribute across matches.	`[a["href"] for a in soup.select("a[href]")]`	
Collect text across matches.	`[li.get_text(strip=True) for li in soup.select("li")]`	

get_text() flattens a subtree; a list comprehension over a search result collects text or an attribute from many nodes

Edit the Tree

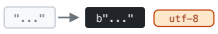
Add, change, replace, and remove nodes in place.

PURPOSE	COMMAND	RESULT
Change a tag's text.	`tag.string= "Gadgets"`	
Set or change an attribute.	`tag["href"]= "/changed"`	
Build and append a new node.	`soup.new_tag("span"); parent.append(t)`	
Insert next to a node.	`tag.insert_before(n) / insert_after(n)`	
Replace one node with another.	`tag.replace_with(new)`	
Remove a node from the tree.	`tag.decompose() tag.extract()`	

Beautiful Soup edits in place; decompose() destroys a node, extract() detaches and returns it for reuse

Serialize Back to a String

Turn the tree (or a piece of it) back into HTML text.

PURPOSE	COMMAND	RESULT
Whole tag back to HTML.	`str(soup) str(tag)`	
Indented, human-readable HTML.	`soup.prettify()`	
Inner HTML only (no wrapper).	`tag.decode_contents()`	
Bytes with an encoding.	`soup.encode("utf-8")`	
Strip all tags, keep text.	`soup.get_text()`	
Save to a file.	`Path("out.html").write_text(str(soup))`	

str(soup) renders HTML, prettify() indents it, decode_contents() gives inner markup, get_text() strips to plain text