**WWN Development Document**

Tools Used to Construct and Edit Web-Page Contents:

Beautiful Soup and Jinja2

Word’s Navigation pane shows the table-of-contents (View : Show : Navigation pane).

* Contents:
  + Beautiful Soup:
    - The Word HTML is parsed and edited using Beautiful Soup.
    - Links to references and tutorials
    - Info on how to use Beautiful Soup, in WWN
  + Jinja2:
    - The WWN web-page is constructed using a Jinja2 template
    - Links to references and tutorials

This document was created by the WWN author for his own use in developing WWN. It is included in the WWN repo, as other developers may find it useful.

# Beautiful Soup

## Docs and tutorials

* Beautiful Soup Documentation — Beautiful Soup 4.9.0 documentation
  + <https://www.crummy.com/software/BeautifulSoup/bs4/doc/>
* Tutorialspoint
  + Beautiful Soup Tutorial | Tutorialspoint
    - <https://www.tutorialspoint.com/beautiful_soup/index.htm>
    - PDF: <https://www.tutorialspoint.com/beautiful_soup/beautiful_soup_tutorial.pdf>
  + Beautiful Soup - Quick Guide | Tutorialspoint
    - <https://www.tutorialspoint.com/beautiful_soup/beautiful_soup_quick_guide.htm>
* Intro to Beautiful Soup | Programming Historian
  + <https://programminghistorian.org/en/lessons/intro-to-beautiful-soup>

## Functions

### Getting the elements and text-content within a tag

#### Summary

* Example, getting tags within a paragraph tag: <p...> ... </p>
* Summary
  + <Soup element.Tag object>.contents
    - Returns a list whose elements are: the child elements and the newlines before and after child-elements (if any)
    - In the html, there can be multiple newlines before or after an child-element
      * In this case, BS just puts one newline in the returned list
  + <Soup element.Tag object>.find\_all(True, recursive=False)
    - returns a list that just includes: child elements
      * It does not include the newlines before and after child-elements

#### Examples

##### Paragraph with 2 children, and no newlines before or after

* Summary
  + The two techniques for getting children-elements return the same thing
    - <Soup element.Tag object>.contents
    - <Soup element.Tag object>.find\_all(True, recursive=False)
* The paragraph shown contains:
  + Two children elements, and the first child-element has a child-element
  + There are no newlines between child-elements
* Child-elements:
  + <span...>
    - <span ...></span>
  + </span>
  + <a...>...</a>
* HTML:

**<p** class=MsoListParagraphCxSpLast style='margin-left:.75in;text-indent:-.25in'>**<span**

style='font-family:"Courier New"'>o**<span** style='font:7.0pt "Times New Roman"'>&nbsp;&nbsp;

</span></span>**<a** href="https://docs.python.org/3/library/argparse.html">https://docs.python.org/3/library/argparse.html</a></p>

* Using <Soup element.Tag object>.contents
  + Variable p:
    - type(p): <class 'bs4.element.Tag'>
    - Contains the paragraph shown in the HTML, above
  + p.contents
    - Returns a list, with two elements

[<span style='font-family:"Courier New"'>o<span style='font:7.0pt "Times New Roman"'>

</span></span>, <a href="https://docs.python.org/3/library/argparse.html">https://docs.python.org/3/library/argparse.html</a>]

* Using <Soup element.Tag object> .find\_all(True, recursive=False)
  + Variable p:
    - type(p): <class 'bs4.element.Tag'>
    - Contains the paragraph shown in the HTML, above
  + p.find\_all(True, recursive=False)
    - Returns a list, with two elements

[<span style='font-family:"Courier New"'>o<span style='font:7.0pt "Times New Roman"'>

</span></span>, <a href="https://docs.python.org/3/library/argparse.html">https://docs.python.org/3/library/argparse.html</a>]

##### Paragraph with 1 child, and newlines before and after it

* HTML:

<p

class=MsoToc1>

<span class=MsoHyperlink>

<a href="#\_Toc69205835">

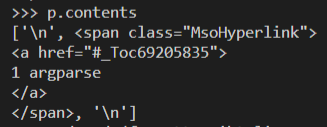
1 argparse

</a>

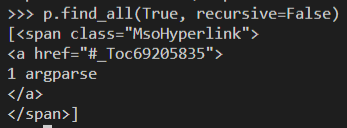
</span>

</p>

* Using <Soup element.Tag object>.contents
  + Variable p:
    - type(p): <class 'bs4.element.Tag'>
    - Contains the paragraph shown in the HTML, above
  + p.contents
    - Returns a list, with three elements
    - HTML:
      * Has two newlines before the span, but BS reduces it to one newline, in the list
      * The newline after the </span> is in the list



* + p.find\_all(True, recursive=False)
    - Returns a list with one element
    - No newlines are returned as list elements
    - Newlines within the grand-child <a> element are not removed



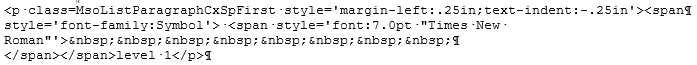
##### Paragraph, with 2 children, text and newline

* Summary
* HTML
  + Note:
    - In the second span: "&nbsp;"s, new-line
  + Paragraph structure: two children, first child has a child
    - <p...>
      * <span...>
        + <span...>
        + "&nbsp;"s
        + </span>
      * </span>
      * [paragraph text]

<p class=MsoListParagraphCxSpFirst style='margin-left:.25in;text-indent:-.25in'><span

style='font-family:Symbol'>·<span style='font:7.0pt "Times New Roman"'>&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;

</span></span>level 1</p>



* p (a BS variable)
  + NOTE: the "&nbsp;"s in the child span are converted to spaces

<p class="MsoListParagraphCxSpFirst" style="margin-left:.25in;text-indent:-.25in"><span style="font-family:Symbol">·<span style='font:7.0pt "Times New Roman"'>

</span></span>level 1</p>



* p.decode(formatter='html')
  + NOTE:
    - the contents "·" were converted to "&middot;"
      * "&middot;" does not fix the display problem in Firefox
    - the "&nbsp;"s are included, and the newline was converted to \n

'<p class="MsoListParagraphCxSpFirst" style="margin-left:.25in;text-indent:-.25in"><span style="font-family:Symbol">&middot;<span style=\'font:7.0pt "Times New Roman"\'>&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;\n</span></span>level 1</p>'



* p.contents
  + NOTE:
    - the paragraph's child-contents are a list-element, i.e., "level 1"
    - the span's child-contents ('·') are in span's list-element

[<span style="font-family:Symbol">·<span style='font:7.0pt "Times New Roman"'>

</span></span>, 'level 1']

* p.find\_all(True, recursive=False)

[<span style="font-family:Symbol">·<span style='font:7.0pt "Times New Roman"'>

</span></span>]

* + NOTE:
    - the paragraph's child-contents are NOT a list-element, i.e., "level 1"
    - the grandchildren contents are in the list
* span (a BS variable), has first child of the paragraph

<span style="font-family:Symbol">·<span style='font:7.0pt "Times New Roman"'>

</span></span>

* span.contents
  + Note: "&nbsp;"s converted to spaces

['·', <span style='font:7.0pt "Times New Roman"'>

</span>]



* span.string (returns None, since the span has multiple children)
* span.find\_all(True, recursive=False)
  + NOTE:
    - the span contents are left out
    - "&nbsp;"s converted to spaces

[<span style='font:7.0pt "Times New Roman"'>

</span>]



### .append

* .append will move a tag from one BS object (tree?) to another

>>> from bs4 import BeautifulSoup

>>> soup\_main = BeautifulSoup('<html><head></head><body><a>FooBar</a></body></html>', 'html.parser')

>>> soup\_append = BeautifulSoup('<html><head></head><body><a>Meh</a></body></html>', 'html.parser')

>>> soup\_main.body.append(soup\_append.a)

>>> print(soup\_main)

<html><head></head><body><a>FooBar</a><a>Meh</a></body></html>

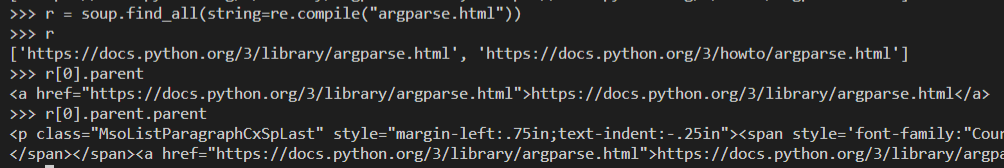
>>> print(soup\_append)

<html><head></head><body></body></html>

## Cheat sheet

### find\_all, for a particular string

* find element with a particular string, and find its ancestors (parent, etc.)
  + "string=" by itself may just look within the page's text, e.g., paragraph text; and not with the attributes



* + "string=" didn't find the href value sought
    - HTML:

<p

class=MsoToc1>

<span class=MsoHyperlink>

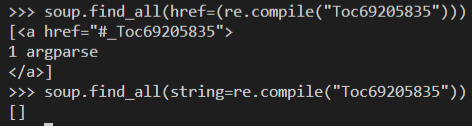
<a href="#\_Toc69205835">

1 argparse

</a>

</span>

</p>



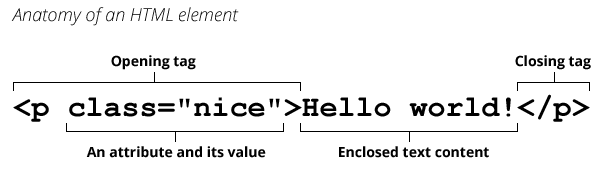
## My BS documentation

### HTML terminology

* HTML terminology from Mozilla:
  + Getting started with HTML - Learn web development | MDN
    - <https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Getting_started>



* + HTML - MDN Web Docs Glossary: Definitions of Web-related terms | MDN
    - <https://developer.mozilla.org/en-US/docs/Glossary/HTML>



* + - *There are a few* ***empty******or void tags*** *that cannot enclose any text, for instance <img>.*
* HTML terminology from BeautifulSoup
  + **Tag**: a whole element, from opening-tag to closing-tag
  + **Content**:
    - Made-up of strings and/or tags
    - There can be zero or more strings and/or tags, in any order
  + **String**: a text string in the tag's content, but it's not within a tag that's in the content.
    - The string is between two tag-delimeters ">" and "<".
  + **Element**: a string or a tag
  + Tags may contain strings and other tags. These elements are the **tag’s children**.
    - A string can’t have children.
    - Every tag and every string has a **parent**: the tag that contains it.
    - **Siblings**: elements that are direct children of the same tag
* HTML terminology for this tutorial
  + A ***whitespace string*** is a string made-up only of whitespace characters.
* Example:
  + A **paragraph** **tag**:

<p>dog, <i>cat, </i>frog, <span class=a><b>toad, </b>snake,</span>lizard</p>

* + The tag's **conent**:

dog, <i>cat, </i>frog, <span class=a><b>toad, </b>snake,</span>lizard

* + - It contains **5 elements**
    - 3 **strings**: "dog,", "frog,", "lizard"
    - 2 **tags**:
      * <i>cat, </i>
      * <span class=a><b>toad, </b>snake,</span>
  + For the **paragraph tag**, its **content** consists of 5 **elements**
    - Those **elements** are the **paragraph tag's** **child elements**.
  + One of the **child elements** is a **span tag**: <span class=a><b>toad, </b>snake,</span>
    - The **span tag's** content contains 2 **elements**:
      * 1 **tag**: <b>toad, </b>
      * 1 **string**: "snake,"

### BS's whitespace compression, in a tag's content

* Summary of BS's whitespace-compression in a tag's contents
  + A tag's content is made-up of tags and/or strings
  + If a string is made-up only of whitespace, the string will be compressed by replacing it with a single whitespace character.
  + Example strings (whole strings are shown):
    - For a string of three newlines ("\n\n\n"), it is replaced with one newline ("\n").
    - For a string of three newlines followed by three spaces ("\n\n\n "), it is replaced with one newline ("\n").
    - For a string with six newlines and a letter (e.g., "\n\n\nx\n\n\n"), it will be left as-is, and not compressed.
  + BeautifulSoup compresses contiguous whitespace, but only under certain conditions.
    - Apparently, BeautifulSoup does not document the conditions for whitespace compression
    - The conditions described here were determined empirically, so the info may not be complete or not fully accurate.
* Whitespace
  + *Whitespace is any string of text composed only of spaces, tabs or line breaks (to be precise, CRLF sequences, carriage returns or line feeds)....*
  + *whitespace is largely ignored — whitespace in between words is treated as a single character, and whitespace at the start and end of elements [contents?] and outside elements is ignored.*
    - How whitespace is handled by HTML, CSS, and in the DOM - Web APIs | MDN
      * <https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Whitespace>
  + In a string, *whitespace-compression* involves replacing a contiguous set of whitespace characters with a single whitespace-character.
* BeautifulSoup compresses contiguous whitespace, but only under certain conditions.
  + Apparently, BeautifulSoup does not document the conditions for whitespace compression
  + Here are some examples of when whitespace is compressed, and not compressed, for an HTML paragraph's contents (<p>[contents]</p>).
  + **Example:** When the contents are only newlines, they are compressed to a single newline:

>>> html = '<p>\n\n\n</p>'

>>> soup = BeautifulSoup(html, 'html.parser')

>>> soup

<p>

</p>

>>> p = soup.p

>>> p

<p>

</p>

>>> p.contents

['\n']

>>> p.find\_all(True, recursive=False)

[]

>>> p.decode(formatter='html')

'<p>\n</p>'

* + **Example:** When the contents are multiple newlines, and one non-whitespace character, there is no compression:

>>> html1 = '<p>\n\n\nx\n\n\n</p>'

>>> soup1 = BeautifulSoup(html1, 'html.parser')

>>> soup1

<p>

x

</p>

>>> p1 = soup1.p

>>> p1

<p>

x

</p>

>>> p1.contents

['\n\n\nx\n\n\n']

>>> p1.find\_all(True, recursive=False)

[]

>>> p1.decode(formatter='html')

'<p>\n\n\nx\n\n\n</p>'

* + **Example:** When a string is only newlines, bordered by a tag, the newlines are compressed to one newline

>>> html2 = '<p>\n\n\n<i>x</i>\n\n\n</p>'

>>> soup2 = BeautifulSoup(html2, 'html.parser')

>>> soup2

<p>

<i>x</i>

</p>

>>> p2 = soup2.p

>>> p2

<p>

<i>x</i>

</p>

>>> p2.contents

['\n', <i>x</i>, '\n']

>>> p2.contents[0]

'\n'

>>> type(p2.contents[0])

<class 'bs4.element.NavigableString'>

>>> p2.contents[1]

<i>x</i>

>>> type(p2.contents[1])

<class 'bs4.element.Tag'>

>>> p2.find\_all(True, recursive=False)

[<i>x</i>]

>>> p2.decode(formatter='html')

'<p>\n<i>x</i>\n</p>'

* **Example:** 
  + The whitespace after the <p> is 3 newlines, followed by 3 spaces
  + The whitespace before the </p> is 3 newlines, followed by 3 spaces

>>> html3 = '<p>\n\n\n <i>x</i>\n\n\n </p>'

>>> soup3 = BeautifulSoup(html3, 'html.parser')

>>> p = soup3.p

>>> p

<p>

<i>x</i>

</p>

>>> p.contents

['\n', <i>x</i>, '\n']

>>> p.find\_all(True, recursive=False)

[<i>x</i>]

>>> p.decode(formatter='html')

'<p>\n<i>x</i>\n</p>'

* **Example:** 
  + The whitespace after the <p> is 3 spaces
  + The whitespace before the </p> is 3 spaces

>>> html4 = '<p> <i>x</i> </p>'

>>> soup4 = BeautifulSoup(html4, 'html.parser')

>>> p4 = soup4.p

>>> p4

<p> <i>x</i> </p>

>>> p4.contents

[' ', <i>x</i>, ' ']

>>> p4.find\_all(True, recursive=False)

[<i>x</i>]

>>> p4.decode(formatter='html')

'<p> <i>x</i> </p>'

* **Example:** 
  + The three strings are each 3 spaces

>>> html5 = '<p> <i>x</i> <b>y</b> </p>'

>>> soup5 = BeautifulSoup(html5, 'html.parser')

>>> p = soup5.p

>>> p

<p> <i>x</i> <b>y</b> </p>

>>> p.contents

[' ', <i>x</i>, ' ', <b>y</b>, ' ']

>>> p.find\_all(True, recursive=False)

[<i>x</i>, <b>y</b>]

>>> p.decode(formatter='html')

'<p> <i>x</i> <b>y</b> </p>'

### Unexpected whitespace in paragraph contents

* Summary
  + In a paragraph tag's contents, whitespace-strings can be created by having one or more newlines between tags.
* **Example:** a paragraph tag with no whitespace strings

>>> html7 = "<p><span class=a>text1<span class=b>text2</span></span></p>"

>>> soup7 = BeautifulSoup(html7, 'html.parser')

>>> p7 = soup7.p

>>> p7

<p><span class="a">text1<span class="b">text2</span></span></p>

>>> p7.contents

[<span class="a">text1<span class="b">text2</span></span>]

>>> p7.find\_all(True, recursive=False)

[<span class="a">text1<span class="b">text2</span></span>]

>>> p7.decode(formatter='html')

'<p><span class="a">text1<span class="b">text2</span></span></p>'

* **Example:**

>>> html8 = """<p>

...

... <span class=a>text1

... <span class=b>text2</span>

... </span>

... </p>

... """

>>> soup8 = BeautifulSoup(html8, 'html.parser')

>>> p8 = soup8.p

>>> p8

<p>

<span class="a">text1

<span class="b">text2</span>

</span>

</p>

>>> p8.contents

['\n', <span class="a">text1

<span class="b">text2</span>

</span>, '\n']

>>> p8.find\_all(True, recursive=False)

[<span class="a">text1

<span class="b">text2</span>

</span>]

>>> p8.decode(formatter='html')

'<p>\n<span class="a">text1\n<span class="b">text2</span>\n</span>\n</p>'

### Getting an opening tag

#### Using BeautifulSoup

* My post:
  + python - How to get the opening and closing tag in beautiful soup from HTML string? - Stack Overflow
    - <https://stackoverflow.com/questions/41597854/how-to-get-the-opening-and-closing-tag-in-beautiful-soup-from-html-string/67691907#67691907>

There is a way to do this with BeautifulSoup and a simple reg-ex:

* Put the paragraph in a BeautifulSoup object, e.g., soupParagraph.
* For the contents between the opening (<p>) and closing (</p>) tags, move the contents to another BeautifulSoup object, e.g., soupInnerParagraph. (By moving the contents, they are not deleted).
* Then, soupParagraph will just have the opening and closing tags.
* Convert soupParagraph to HTML text-format and store that in a string variable
* To get the opening tag, use a regular expression to remove the closing tag from the string variable.

In general, parsing HTML with a regular-expression is problematic, and usually best avoided. However, it may be reasonable here.

* A closing tag is simple. It does not have attributes defined for it, and a comment is not allowed within it.
  + <https://stackoverflow.com/questions/4138006/can-i-have-attributes-on-closing-tags>
  + <https://stackoverflow.com/questions/5926580/html-comments-inside-opening-tag-of-the-element>
* This code gets the opening tag from a <body...> ... </body> section. The code has been tested.

# The variable "body" is a BeautifulSoup object that contains a <body> section.

bodyInnerHtml = BeautifulSoup("", 'html.parser')

bodyContentsList = body.contents

for i in range(0, len(bodyContentsList)):

# .append moves the HTML element from body to bodyInnerHtml

bodyInnerHtml.append(bodyContentsList[0])

# Convert the <body> opening and closing tags to HTML text format

bodyTags = body.decode(formatter='html')

# Extract the opening tag, by removing the closing tag

regex = r"(\s\*<\/body\s\*>\s\*$)\Z"

substitution = ""

bodyOpeningTag, substitutionCount = re.subn(regex, substitution, bodyTags, 0, re.M)

if (substitutionCount != 1):

print("")

print("ERROR. The expected HTML </body> tag was not found.")

#### Using html.parser

* Info on getting an opening tag, using Python’s html.parser
* <https://docs.python.org/3/library/html.parser.html>
  + With html.parser though you can listen to "start" and "end" tag "events".

class MyHTMLParser(HTMLParser):

def \_\_init\_\_(self,tagToGet):

super().\_\_init\_\_()

self.tagReturned = None

self.openingTagToGet = tagToGet

self.tagFound = False

# When the tag-to-get is found, apparently there's no way to stop the parsing.

# \* So, self.tagFound is used to ignore subsequent tags

# \* More info: https://stackoverflow.com/questions/30285039/python-htmlparser-stop-parsing/49385685

def handle\_starttag(self, tag, attrs):

if (self.tagFound == False) and (tag == self.openingTagToGet):

self.tagReturned = self.get\_starttag\_text()

self.tagFound = True

-------

bodyHtml = body.decode(formatter='html')

htmlParser = MyHTMLParser("body")

htmlParser.feed(bodyHtml)

if htmlParser.tagFound == True:

bodyOpeningTag = htmlParser.tagReturned

else:

print("")

print("ERROR. For the input Word-HTML, the expected body opening-tag was not found.")

return 1

# Delete variable. This copy of the body HTML isn't needed now

del bodyHtml

# Jinja

* Jinja is a web template engine for the Python programming language.
* Jinja (template engine) - Wikipedia
  + <https://en.wikipedia.org/wiki/Jinja_(template_engine)>

## Tutorials

* Jinja2 Tutorial - Part 1 - Introduction and variable substitution |
  + <https://ttl255.com/jinja2-tutorial-part-1-introduction-and-variable-substitution/>
* Primer on Jinja Templating – Real Python
  + <https://realpython.com/primer-on-jinja-templating/>
* Jinja2 Explained in 5 Minutes!. (Part 4: Back-end Web Framework: Flask) | by Diva Dugar | codeburst
  + <https://codeburst.io/jinja-2-explained-in-5-minutes-88548486834e>
* Jinja2 Templating Engine Tutorial | by Jason Rigden | Medium
  + <https://medium.com/@jasonrigden/jinja2-templating-engine-tutorial-4bd31fb4aea3>
* Jinja tutorial - creating templates in Python with Jinja module
  + <https://zetcode.com/python/jinja/>

## Package

* Jinja2 · PyPI
  + <https://pypi.org/project/Jinja2/>

## Reference info

* Jinja — Jinja Documentation (2.11.x)

<https://jinja.palletsprojects.com/en/2.11.x/>