Siddharth Singha Roy

Beautiful Soup

A web scraper based on python

Installing Beautiful Soup

```
pip install bs4
```

In a python file

```
from bs4 import BeatifulSoup
soup = BeautifulSoup(html_doc, ‘html.parser’)
```

Useful Commands

1. print(soup.prettify())
2. print(soup.title)
3. print(soup.a)
4. print(soup.title.name)
5. print(soup.title.string)
6. print(soup.a.string)
7. print(soup.title.parent.name)
8. print(soup.find_all(‘a’))
9. print(soup.find(id=‘link3’))
10. print(soup.get_text())

Code Snippets

```
# print(soup.prettify())
# ***** Prints the entire document

# print(soup.title)
# *** <title> ........ </title>

# print(soup.title.name)
# *** Prints title

# print(soup.title.string)
# *** Prints the title of the document

# print(soup.prettify())

# print(soup.a)
# ******** Returned the first occurence of a

# print(soup.find_all('a'))
# *** Returns all occurence of a

# print(soup.find(id='link3'))
# Get the tag having id = link3

# print(soup.title.parent.name)
# Get the parent of the tag title

# print(soup.a.string)
# Get the string enclosed between the first tags
# print(soup.find_all('a')[0].string)
# find_all returns a list of all matching cases

# print(soup.a.parent.string)

# for link in soup.find_all('a'):
#     print(link.get('href'))

# To get all the links starting with a tag

# print(soup.get_text())
# Get the text of the html_doc
```

Making the soup

We will be using the lxml parser since it provides better speed as compared to other parsers.

```
from bs4 import BeautifulSoup
soup = BeautifulSoup("<html>Hi</html>")
 print(soup)
```

This will produce a warning. This is because the system will automatically use the parser present in the system. This may change from environment to environment. To eliminate this use

```
soup = BeautifulSoup("<html>Hi</html>", "lxml")
```

In BeautifulSoup we will only be dealing with 4 kinds of objects

1. Tag
2. NavigableString
3. BeautifulSoup
4. Comment

TAG

Tag corresponds to a HTML Tag like <a>, <p>, <h1> etc.

We can get the tag from a HTML file using

```
soup.tagname
```

Two of the most important features of tag is the tag name and its attributes.

Each tag belongs to the class **bs4.element.tag**

Every tag has a name that can be accessible using the **.name** attribute

We can even change the tag name and that change will be reflected in the changes made to the HTML Markup generated by Beautiful Soup

To get all the attributes of a tag use

```
tagvar.attrs
```

This will return a dictionary of all the attributes of that particular tag.

We can access individual attributes just like the way we access elements of an array. Any changes made to the attributes will also change the original soup.

Say, we want to access the classname, we use

```
tagvar[‘class’]
```

Change value of the class

```
tagvar[‘class’] = newClass
```

To delete an attribute from the tag

```
del tagvar[‘class’]
```

OR

```
tagvar.attrs.pop(‘class’)
```

Now it may happen that a particular attribute does not exist. So instead of directly accessing the values in the dictionary we use

```
tagvar.get(‘class’)
```

MultiValued Attributes

There maybe certain attributes that can hold multiple values for example classes.
If we have multiple values for a single attribute, we will get a list of the values instead of a single value for the same key in the dictionary.

For example if want to get the values of the classes, we will always be returned a list. If we do the same for id, we will get a specific string.

However some attributes like id cannot take multiple values. If specify multiple values for id, then the returned value still will be a string only.

We can also change the class values and give it multiple values by passing them in the form of list.

If we want our application to only care about single valued attributes then use

```
soup = BeautifulSoup(‘html_doc’, ‘lxml’, multi_valued_attributes=None)
```

This will prevent our application from considering multiple values in an attribute.

We can also check if the given attribute has multiple values or not by using the get_attribute_list()

```
soup.h1.get_attribute_list(‘id’)
```

This will always return a list.

NavigableString

The string inside the tag is stored in the NavigableString class.

```
tag.string
```

We cannot mutate this value as string in python is immutable. Although we can replace it with another value

```
tag.string.replace_with('Hiiii')
```

If we need to use that value of the string somewhere else, we need to call unicode() on it to prevent wastage of BeautifulSoup memory space.

However unicode() support was ended in python 3.x . In the new version it has been incorporated into str()

```
str(ptag..string)
```

This will ensure that now no reference to the original BeautifulSoup object remains unaltered.

BeautifulSoup

It refers to the entire html document passed as a whole.

To check if a particular text exists inside the document, we can use

soup.find(text=“hi”).replace_with(“Something”)

If we use soup.name we get the result ‘document’

Comment

It is a special type of NavigableString. It will give the comment line return in the HTML document.

Comment lines in HTML are of the form

<!—   —>

For a comment if we do soup.tagname.string this actually returns Comment object.

Navigating the tree

Going Down

Navigating a tree can be done by using the tag name directly.

For example we want to find the h1 or the head element of the tree.

```
soup.h1
soup.head
```

We can do this again and again until we reach the required destination of the html file.

We can also do nested queries

```
soup.head.h1.b.a
```

However using soup.a will only give the first matching case. To get all the tags use find_all

```
soup.find_all('a')
```

Using .contents we can get the tag's children in the form of a list.

```
soup.head.contents
```

This will return a list containing all html tags. If a tag is contained inside a tag, it will be kept as nested tag.

If we call contents on a tag which does not have any nesting inside it, we get the string.

Contents simply return what is contained inside the tag. A string does not have anything enclosed within it. So an error will be raised when we try to use .contents on a string.

.contents generates a list. Instead of generating a list, we can iterate over a tag's children using the .children generator.

```
for child in title_tag.children:
    print(child)
```

Both the .contents and .children only consider the tags direct children ie. the children of children is not considered.

For example the head tag will only have one direct child —— <title>.

The title also has a child which is the string which is not considered.

To overcome this we have the .descendants => This allows us to iterate recursively all the tag along with the children of the tags.

.string

If a tag has only one children and that is a NavigableString then we can get that string using

```
tagname.string
```

If a tag has another child tag which has only children ie. a NavigableString then we can still use .string to get the string.

```
parent_tag_name.string
```

However if there multiple children then this will return a None.

If there is more than one thing inside the tag then we can only look at the only the strings using .strings

However this will result in a lot of extra whitespace. To counter this use .stripped_strings

Going Up

You can access an element's parent using the .parent

soup.title.parent

This returns the entire tag of the parent including its children.

The parent of a BeautifulSoup object is None.

To get the tag of all the parents, use the .parents

This will recursively iterate until it becomes None.

Going Sideways

If two or more tags are on the same level then we can use next_sibling and previous_sibling.

If no such tag exists, then it returns a None.

By same level we mean that it must have the same parent.

Many time the next_sibling may return a whitespace character, so to get the next_sibling sometimes we may need to use

next_sibling.next_sibling

We can even recursively iterate over the next and previous siblings using .next_siblings and .previous_siblings

.next_element and .previous_element can be used to get the next element or the previous element that was parsed by the parser.

If we want to recursively get all the next or previous elements use

.next_elements

.previous_elements

Searching the tree

find_all

Returns a list of all matching strings

We can also use regular expressions to get the matching string

```
find_all(re.compile(reg_exxpr))
```

We can also use a list as an argument for find_all to get all the matching tags of the element included in that list.

To get all the tags present in the html document use

```
for i in soup.find_all(True):
        print(i.name)
```

This will match with everything that is present in the html doc.

If the above other methods do not work, we can always define our own function that takes an element as an argument and returns true if the argument matches else returns false.

```
def class_but_no_id(tag):
    return tag.has_attr('class') and not tag.has_attr('id')

print(soup.find_all(class_but_no_id))
```

tag.has_attr is used to check if the given class object has the attribute ‘class’ defined or not.

We can search by CSS Class (Since class is a keyword in python, this issue arises) using class_ = “classname”

We can also use regular expression while searching

soup.find_all(class_=re.compile("itl"))

The String Argument

This can be used to match strings instead of tag. String can be matched with another string or a list of strings or a regex. We can even define our own function which takes a string as an argument and either returns a true or false.

We can even pass multiple arguments like the tag name, a specific string to the find_all function.

Suppose we just want two tags instead of all the tags. We can use limit = 2 as an extra argument.

The find_all tag will also look at the children of each tag. To stop and this (just look at the direct children) use recursive=False

soup.find("head").find("title")

This is similar to soup.head.title

find() and find_all() both look at the descendants.

find_parent() and find_parents() both look at the parents.

Similar to this we have the

1. find_next_sibling()
2. find_next_siblings()
3. find_previous_sibling()
4. find_previous_siblings()

find_all_next()

This will find all the matching cases from the next part of the document

find_next(), find_all_previous(), find_previous()