

Testing BeautifulSoup

A software testing project
Uppsala University 2017

Erik Bertse
Sara Gustavsson
Moa Marklund
Henrik Thorsell

Abstract

The BeautifulSoup library is a Python library designed to easily search and manipulate data associated with HTML/XML markup. In this project, we test the functionality of this library. Most of the available API functions have been subject to black-box testing. One of the functions was selected to undergo white-box testing. Our testing has resulted in the discovery on some irregularities.

1 Introduction

1.1 What is BeautifulSoup?

BeautifulSoup is a Python library which provides functionality for navigating and extracting data from HTML and XML markup. It does so by using external parsers, specifiable by the users of the library, whose output is assembled by BeautifulSoup into a tree structure, the nodes of which are tags in the HTML/XML markup. Thus, children of any given tag are nested tags.

The BeautifulSoup library is compatible with Python 2.7 and Python 3.2, but we have limited the scope of our testing to using version 2.7. The most recent version of BeautifulSoup is 4, abbreviated BS4. Previous (and no longer supported) versions of BeautifulSoup are available, but are not subject to any tests in this project.

Given some markup, the library creates a *Soup* object, containing all the information in the markup. In the following example, we call the BeautifulSoup constructor with some markup, using the HTML parser that comes with Python:

```

a_simple_soup = BeautifulSoup(
    '''
    <html>
    <head>
    </head>
    <body>
    <p>
    <a>An anchor tag</a>
    </p>
    </body>
    </html>
    ''', "html.parser")

```

The soup object created is a rather complex. Each tag in the markup gives rise to an object inside the soup, simply referred to as a *Tag* object. The text inside a tag gives rise to a *NavigableString* object, which, for our purposes, works like a Python string. Tags are then organized in a tree structure, such that any inner tag is a child of its parent tag. The soup class itself inherits from the tag class, and therefore all soup objects are also tag objects.

The following example show how to use class attributes to access tag objects in the tree structure. To access the tag object which corresponds to the head tag, and assign it to a variable, we would write the following code:

```
tag = a_simple_soup.head
```

Note that we are not required to first access its parent, i.e. we do not have to specify

```
a_simple_soup.html.head
```

although this is perfectly valid also. If the tree structure contains several tags with the same name, then the first one in the markup will be accessed. Please refer to the Discussion section for more information on how this works.

The library contains a number of functions for navigating and modifying the tree, many of which were tested in this project. The following call exemplifies searching the tree. The call:

```
lst = a_simple_soup.find_all("a")
```

will return a list of all tag objects, whose name is **a**, in the tree structure.

We can also modify the tree in various ways. The following call clears the contents of the **html** tag:

```
a_simple_soup.html.clear()
```

Many other functions are available. We refer the curious reader to the BeautifulSoup documentation, for a complete list. The documentation is available here: <https://www.crummy.com/software/BeautifulSoup/bs4/doc/>

2 Method

We have tested a subset of the total functionality of the BeautifulSoup library. The testing was organized in two stages, black box testing and white box testing.

2.1 Black box testing

We started by looking over the total functionality of the BeautifulSoup library. This included reading all the documentation, and quickly going over the source code. We finally decided to perform black box testing on the following functionality, divided into three sections.

- Navigation with attributes.

The following attributes were selected:

- **contents**
- **children**
- **descendants**
- **string**
- **strings**
- **stripped_strings**
- **parent**
- **parents**

There are also tests provided for two of the following “attributes”

- **soup.head**
- **soup.title**

There were originally plans to provide more tests for accessing specific tags in this way, but these were cut. Please refer the Discussion section for more information on this.

- Searching the tree.

The following search functions were selected:

- **find()**
- **find_all()**
- **find_parents()**
- **find_parent()**
- **find_all_next()**
- **find_next()**
- **find_all_previous()**
- **find_previous()**
- **select()**

- Modifying the tree.
The following modifying functions were selected:

- `append()`
- `insert_before()`
- `insert_after()`
- `clear()`
- `extract()`
- `decompose()`
- `replace_with()`
- `wrap()`
- `unwrap()`

The criterion we aimed at satisfying could be expressed in the following way: for each function listed above, execute the function, and verify its output. Because of the state of the documentation, detailed in the Discussion section, this involved some subjective assessment of what constitutes correct output.

As an example, consider the function `find()`. For this function, three tests were written.

```
def test_find(self):
    '''Tests finding tags with a given name'''
def test_find_empty(self):
    ''' Test finding a tag with empty string name'''
def test_find_fail(self):
    '''Test not finding a tag'''
```

The first of these tests use the soup objects `a_nested_soup` and `a_simple_soup`, defined in the initialization code, and tests whether the `find()` function outputs the correct result. The `find()` function supposed to return the first tag of it finds, whose name is the string argument fed to the function. Since `a_simple_soup` only contains a single anchor tag, calling

```
a_simple_soup.find("a")
```

is expected to return the single found tag, and nothing else. Since `a_nested_soup` contains several anchor tags, we test whether the function returns the first tag to lexically appear in the markup.

The example above exemplifies the basic layout of each test. Specifically, each test consists of

- The reason for the existence of the test, documented as a Python docstring.
- A soup object to either search or modify.
- The function call itself.
- Assertion of the equality of the output and the expected output.

Since all the functions and attributes listed above require soup objects or tag objects to operate on, we proceeded to define test data that could be used in several tests. This data consists of several soup objects and tag objects. This was done in the initialization code in the file containing the black box tests. Here we initialized several soup objects, by feeding some markup into the soup constructor. The markup was selected to be small and easy to read, yet provided enough structure to write non-trivial tests.

Many, but not all, of the tests that pertain to navigation and searching use the test data defined in the initialization code.

Unfortunately, all of the functions that modify the tree could not be run on this test data. If a test modifies the tree structure in some way, then the next test would obviously use the modified test data, which was not our intention.

The solution was to define a new soup object for each test that modifies the tree structure. The markup used is generally very short, only as much as necessary to test the modification under consideration.

For example, the test:

```
def test_clear(self):  
    ''' Clear the contents of a HTML <a href> tag. '''
```

which tests the `clear()` function, operates on the following simple markup

```
<a href="http://example.com/">I linked to <i>example.com</i></a>
```

The expected output of navigation and searching is almost always a tag object. Given a soup object to be tested, we define free-standing tag using the `new_tag()` factory function. We use these tags to compare with the output of the function under consideration. See the Discussion section for details on Tag and Soup equality.

For functions that modify the tree structure of the soup object, the expected output is a soup object, and therefore these tests begin with an instantiation of a soup object that is used for comparison.

We used the documentation as a specification on what each function should return. We let the documentation inform the tests, i.e. the documentation specifies the what is supposed to be the correct behavior of a given function, and we test whether the function behaves as stated in the documentation. Where applicable, there are tests to test negative results (e.g. searching for a tag that is not present), or edge cases (e.g. clearing the contents of an empty tag).

For example, the test:

```
def test_clear_empty(self):  
    ''' Clear the contents of an empty HTML <a href> tag. '''
```

executes the **clear()** function on a tag that is already empty, and verifies that the input remains unchanged.

Towards the end of the project, the black box tests underwent considerable consistency rework, so as to conform the same notion of soup and tag equality.

There are 53 tests in total. The purpose of each test is specified as a docstring.

2.2 White box testing

The following functions were selected for white box testing:

- **find()** Public function
- **find_all()** Public function
- **_find_all()** Private function

The goal of the white box testing was to attain full branch coverage of these functions. Since **find()** and **find_all()** are mostly wrappers for the private function **_find_all()**, attaining full branch coverage of these functions was easy. Covering each branch in **_find_all()** proved much more challenging.

We started by studying the internal structure of the function. This allowed us to write calls to the **_find()** and **_find_all()** functions, in such a way, that each call allowed for different statements to be executed. This let us achieve full statement coverage.

Unfortunately, this code was not sufficient to achieve branch coverage, and we had to extend it.

To achieve full branch coverage, we had to step far outside the normal usage the BeautifulSoup library, and break a few Python principles along the way. See the Discussion section for how this was done.

While we will use the term *branch coverage* throughout this document, the concept is often referred to as *edge coverage*. Our working definition is as follows:

Given the control flow graph of some code, we satisfy the edge coverage criterion if we execute each edge in the graph.

Equivalently, we could state that we achieve branch coverage if we execute each path in the graph of length (up to) one.

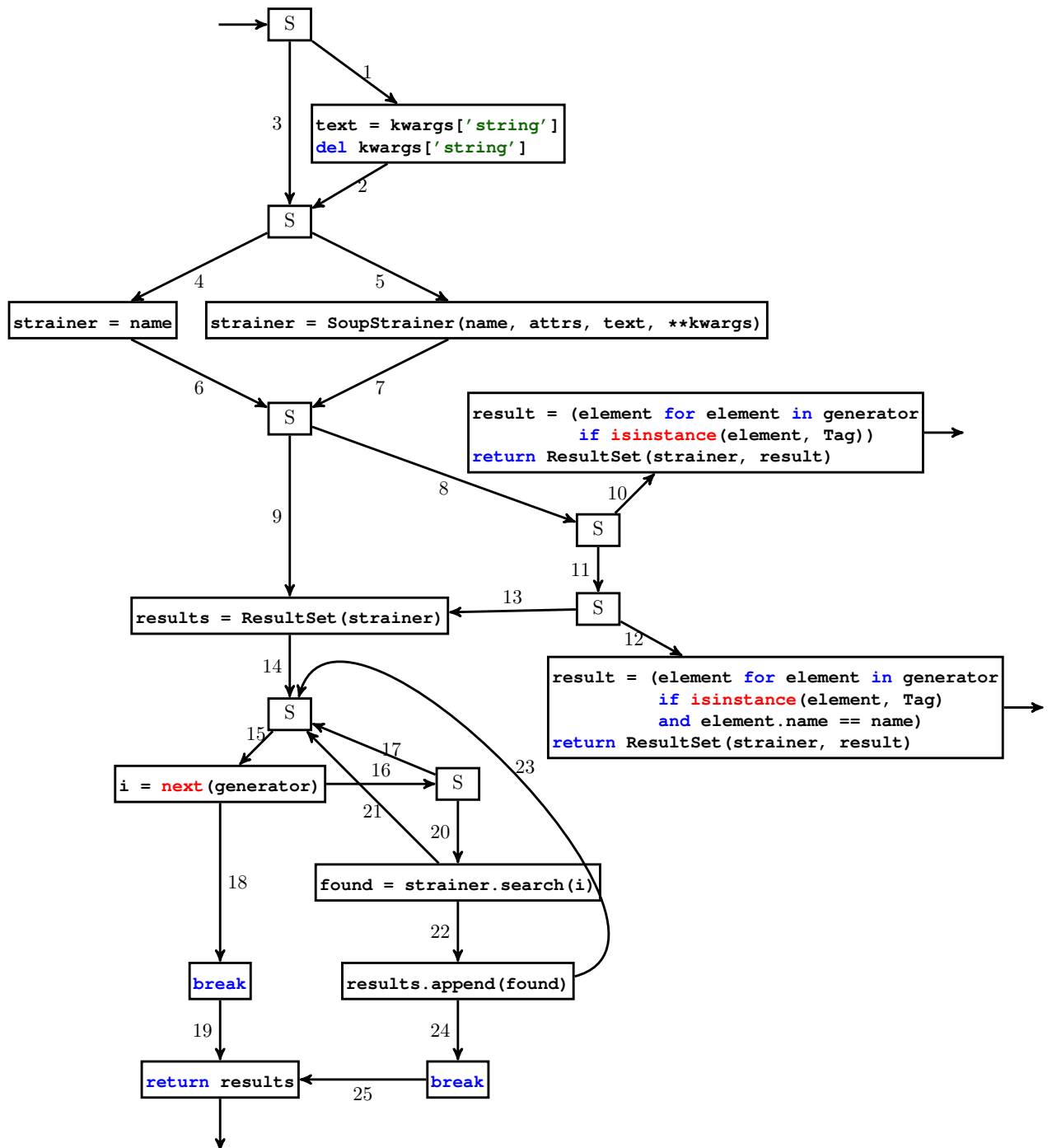
On the following page, we present the source code of the **_find_all** function in full:

```

1  def _find_all(self, name, attrs, text, limit, generator, **kwargs):
2      "Iterates over a generator looking for things that match."
3
4      if text is None and 'string' in kwargs:
5          text = kwargs['string']
6          del kwargs['string']
7
8      if isinstance(name, SoupStrainer):
9          strainer = name
10     else:
11         strainer = SoupStrainer(name, attrs, text, **kwargs)
12
13     if text is None and not limit and not attrs and not kwargs:
14         if name is True or name is None:
15             # Optimization to find all tags.
16             result = (element for element in generator
17                       if isinstance(element, Tag))
18             return ResultSet(strainer, result)
19         elif isinstance(name, basestring):
20             # Optimization to find all tags with a given name.
21             result = (element for element in generator
22                       if isinstance(element, Tag)
23                       and element.name == name)
24             return ResultSet(strainer, result)
25     results = ResultSet(strainer)
26     while True:
27         try:
28             i = next(generator)
29         except StopIteration:
30             break
31         if i:
32             found = strainer.search(i)
33             if found:
34                 results.append(found)
35                 if limit and len(results) >= limit:
36                     break
37     return results

```

We consider each code block as a node. There is a (directed) edge between two blocks, if execution can flow from one block to the other. Thus, we constructed the following control flow graph, presented on the next page.



Each node corresponds to a block of code. The nodes marked by S are symbolic nodes. They do not correspond to a block of code, rather, they represent only a decision point in the control flow. These are sometimes called *dummy nodes* in the literature. To keep the diagram not too cluttered, we have not marked each edge with the condition that is satisfied when that edge is taken, in the graph. Instead, these are presented in the following table. If an edge has no condition on when it is taken, then it is taken unconditionally.

Edge number	Taken when...
1	<code>text is None and 'string' in kwargs</code> is true
2	-
3	<code>text is None and 'string' in kwargs</code> is false
4	<code>isinstance(name, SoupStrainer)</code> is true
5	<code>isinstance(name, SoupStrainer)</code> is false
6	-
7	-
8	<code>text is None and not limit and not attrs and not kwargs</code> is true
9	<code>text is None and not limit and not attrs and not kwargs</code> is false
10	<code>name is True or name is None</code> is true
11	<code>name is True or name is None</code> is false
12	<code>isinstance(name, basestring)</code> is true
13	<code>isinstance(name, basestring)</code> is false
14	-
15	- (This edge corresponds to the condition <code>while True</code>)
16	<code>StopIteration</code> exception is not thrown
17	<code>i</code> is false
18	<code>StopIteration</code> exception is thrown
19	-
20	<code>i</code> is true
21	<code>found</code> is false
22	<code>found</code> is true
23	<code>limit and len(results) >= limit</code> is false
24	<code>limit and len(results) >= limit</code> is true
25	-

Thus, our goal was to execute each edge in the graph. The following summarizes what execution paths various lines of code take in our file for white box testing.¹ Note that we use the notation `[a,b,c]*`, to signify repeatedly executing this path (looping).

The path `[15, 16, 20, 21]*` is found in all but the last of the paths listed below. The exact number this path executes varies, but generally, it will execute once for every object (*Tag* or *NavigableString*) in the soup object.

Although, formally, a path is a sequence of nodes, if we state that a path consists of edges, we mean a path consisting of nodes in the edges, only accou-

¹Although this exposition belongs in the Results section, we put it here to allow the reader to more easily trace out execution paths in the graph

ting once for the node at the end of an edge, and the node at the start of the next edge, which are equal.

The line

```
1 tag = self.a_simple_soup.find_all("a")
```

executes the path: 3, 5, 7, 8, 11, 12

The line

```
1 tag = self.a_simple_soup.find_all()
```

executes the path: 3, 5, 7, 8, 10

The line

```
1 tag = self.a_simple_soup.find_all(string = "a")
```

executes the path: 1, 2, 5, 7, 9, 14, [15, 16, 20, 21]*, 15, 18, 19

The lines

```
1 strainer = SoupStrainer("a")
2 tag = self.a_simple_soup.find_all(strainer)
```

executes the path: 3, 4, 6, 8, 11, 13, 14, [15, 16, 20, 21]*, 15, 16, 20, 22, 23, [15, 16, 20, 21]*, 15, 18, 19

The line

```
1 tag = self.a_simple_soup.find_all("a", limit = 1)
```

executes the path: 3, 5, 7, 9, 14, [15, 16, 20, 21]*, 15, 16, 20, 22, 24, 25

Note that, at this point, all edges, except the edge labelled 17, are executed at least once. Section 4.5 details the tricks that were required to execute this edge. This resulted in the following line of code, and its corresponding execution path.

The lines

```
1 iterator = EvilIter(3)
2 tag = self.tiny_soup._find_all("a", None, None, None, recursive = False, generator = iterator)
```

executes the path: 3, 5, 7, 9, 14, [15, 16, 17]*, 15, 18, 19

3 Results

3.1 Black box testing

The following tests do not pass:

```

test_css_select_empty(self):
    ''' Test the select function with an empty string '''

test_replace_tag_with_string(self):
    ''' Test replacing tag with string '''

test_unwrap(self):
    ''' Test unwrapping and compare to soup after unwrapping.'''

test_find_empty(self):
    ''' Test finding a tag with no name '''

```

The first test does not pass because the an inner function throws an `IndexError`.

It is debatable whether the second, third and fourth test should be considered bugs, or whether this is normal functionality of BeautifulSoup. See Discussion section.

3.2 White box testing

We successfully achieved full branch coverage of the `find()`, `find.all()` and `_find.all()` function, but only through bizarre usage of both BeautifulSoup and Python.

Coverage was measured with the tool called Coverage ²

4 Discussion

4.1 The documentation

The documentation available for the library is presented in a relatively informal style, with small and informal examples. There does not seem to be any rigorous documentation available, and therefore, it is not always clear what functions should return (or what modifications will be made to the tree structure) in edge cases. For example, for functions that take strings as parameters, the library's behavior on an empty string is unclear. We have therefore been forced on make subjective decisions as to what makes a test pass.

This has resulted in some failing tests, although, in all cases but one, the results are debatable. Please see the failing black box testing subsection further down.

4.2 Class attributes and navigation

Originally, the intention was to perform more extensive testing on navigation using the following syntax:

```
a_simple_soup.html.head
```

²<https://coverage.readthedocs.io/en/coverage-4.4.2/>

Tests for finding **html** and **title** tags were written, and is still part of the black box testing file. After further study of the how Python attributes work, the following property of Python language was discovered: Given an object, if an attribute is not defined (as a function, for example), calling that attribute will invoke the function

```
__getattr__().
```

If this function is defined as

```
__getattr__(self, arg),
```

then the name of the attribute is passed as the **arg** parameter.

Looking at the BeautifulSoup source code revealed the following function:

```
def __getattr__(self, tag):
    ... #Code omitted here
    return self.find(tag)
```

Thus, using attributes for navigation is subsumed by tests that use the **find()** function, and we therefore scrapped plans for testing this further. Please note that this is not true for the first set of attributes (such as **children** and **contents**), which are defined as separate functions and are therefore tested.

4.3 Tag and Soup equality

For quite some time during the the development of the tests, there were inconsistencies regarding how to compare two tags. Some tests compared tags for equality by comparing their string representations. Although this is not a bad idea, ultimately, it did not capture the nested structure of the tree.

The documentation states:

*Beautiful Soup says that two NavigableString or Tag objects are equal when they represent the same HTML or XML markup.*³

A more rigorous definition was found inside the BeautifulSoup source:

```
def __eq__(self, other):
    '''Returns true iff this tag has the same name, the same attributes,
    and the same contents (recursively) as the given tag.'''
```

The Soup class contains a factory function called **new_tag()** that generates a new tag. This tag is not placed in the tree structure. Thus, to compare the output of a function, we define a new tag, and compare the expected output with this tag.

The tests underwent consistency rework, so that all tests conformed with this notion of tag equality. It is also on the basis of this notion of equality that some tests fail.

³Copied verbatim from the documentation

4.4 The failing black box tests

Here we provide details for the tests that fail. For convenience, we provide line numbers of where the tests can be found in the black box testing file.

The test at line 349:

```
test_css_select_empty(self):  
    ''' Test the select function with an empty string '''
```

fails. The function `select()` is used to find tags based on CSS selectors. For example, calling this function like so:

```
soup.select("#my_id")
```

will return a list of items tags whose `id` attribute equals `my_id`.

The documentation does not specify the behavior of this function when the input is an empty string. We made the subjective assumption that the call should return `None`, but instead, BeautifulSoup crashes with an uncaught `IndexError` exception, so we consider this a proper bug.

The tests at lines 612 and 670

```
test_replace_tag_with_string(self):  
    ''' Test replacing tag with string '''  
test_unwrap(self):  
    ''' Test unwrapping and compare to soup after unwrapping.'''
```

fail. They both fail for similar reasons. The `replace()` function should replace the object (tag) that called it with the function input. The `unwrap()` function should remove a tag, but leave the tag's contents untouched.

The issue here is that the tree structure is not updated accordingly. Consider a soup created from the following markup:

```
<a href="http://example.com/">I linked to <i>example.com</i></a>
```

One would expect that after unwrapping the `i`-tag, the contents of the `a`-tag should be a string. This is not the case, instead, the contents of the `a`-tag is a list containing two strings, one of which is what used to be the contents of the `i`-tag. If we modify the tree using the `unwrap()`-function, the value of `tagstring` after the call

```
tagstring = soup.a.string
```

is `None`. Again, the functionality at this level of detail is not specified in the documentation, so it is a subjective assessment that this is erroneous behavior.

The `replace()` function does not pass for the same reason, the modification of the tree is not updated properly, when replacing a tag with a string.

The test at line 317

```
test_find_empty(self):  
    ''' Test finding a tag with no name '''
```

fails. Here we run the `find()` function with an empty string. Again, the behavior of this call is not specified in the documentation. Our assumption is that the result should be `None`, especially since it is possible to define a tag with an empty name. This function returns the `html` tag (whose name is certainly not empty). One could suspect that this call will return the soup object itself, but this is also not true.

4.5 The white box hacks

The goal of the white box testing was to achieve full branch coverage of the functions `find()`, `find_all()`, and the internal function `_find_all()`. It turned out that `find()` calls the function `find_all()`, which in turn calls `_find_all()`. This internal function is relatively complicated, and the test file for branch coverage contains various calls so as to execute every branch. Please refer to the Method section for the full code for the function `_find_all()`, we provide only the relevant excerpt here .

A section of `_find_all()` is the following code:

```
1 #...Code omitted here  
2 while True:  
3     try:  
4         i = next(generator)  
5     except StopIteration:  
6         break  
7     if i:  
8         found = strainer.search(i)  
9         if found:  
10             results.append(found)  
11             if limit and len(results) >= limit:  
12                 break  
13 return results
```

Making line 7 in the above excerpt false required some trickery. A Python iterator is an object used for iteration. Iterators must implement (among other functions) a `next()` function that returns an iterator (usually itself). Python generators are a type of Python iterators. Thus, `i`, at line 7, is an iterator object. Please refer to the Python documentation for more information on iterators and generators.

Objects in Python generally return true when evaluated as a boolean (but not always). Even if the iterator was empty, calling next on it would throw a `StopIteration` exception, and the loop would break at line 6, before ever reaching line 7. So, given normal usage, the line 7 will always return true.

The solution was to write a custom iterator class, and override its `__len__`

function, to always return 0, while making sure that it would iterate a number of times first. This way, we can ensure that the condition on line 7 evaluates to false a certain number of times before the iterator throws the `StopIteration` exception. We do this to prevent an infinite loop. Note that this iterator was written specifically to break BeautifulSoup, and that this is not how iterators are supposed to work. The code for this custom iterator can be found in the white box testing file.

After this, we feed this custom iterator directly into `_find_all()`, since feeding the iterator into both `find()` and `find_all()` causes BeautifulSoup to throw a `TypeError` exception. Note that this is not the intended usage of BeautifulSoup, but the only way we found to get the false branch at line 7 to execute.

Through this method we achieved 100% branch coverage of these three functions.

5 Conclusion

We performed black box testing on a selection of function in the BeautifulSoup library. We found some discrepancies, but most of the functions performed as expected. We also performed white box testing on a selected function, which entailed achieving full branch coverage of this function. Here, we succeeded, although we had to use ad hoc solutions to do this.

6 Appendix

The file `unittest_tests.py` contains the code for black box testing.

The file `unittest_cov_tests.py` contains the code for white box testing.

The file `unittest_tests.py`:

```
1  ## File for black box testing of the BS4 web scraping library.
2
3
4  ## Erik Bertse
5  ## Sara Gustavsson
6  ## Moa Marklund
7  ## Henrik Thorsell
8
9  ## Software Testing, 5c
10 ## Autumn 2017
11 ## Uppsala University
12
13 import unittest
14 from bs4 import BeautifulSoup
15
```

```

16
17 class TestBS(unittest.TestCase):
18
19     @classmethod
20     def setUpClass(cls):
21         #Sets up various soups for tests
22
23         cls.soup = BeautifulSoup("", "html.parser")
24
25         #How to get soupify html files: (this is not used in the tests below)
26         #cls.html_file = open("html_testfiles/html_test.html")
27         #cls.html_file_soup = BeautifulSoup(cls.html_file, "html.parser")
28
29
30         #Directly defined soups.
31
32         cls.a_simple_soup = BeautifulSoup(
33             '''
34             <html>
35                 <head>
36                 </head>
37                 <body>
38                     <p>
39                         <a>An anchor tag</a>
40                     </p>
41                 </body>
42             </html>
43             ''', "html.parser")
44
45         cls.a_nested_soup = BeautifulSoup(
46             '''
47             <html>
48                 <head>
49                 </head>
50                 <body>
51                     <p id = "p_lv11">
52                         <p id = "p_lv12">
53                             <p id = "p_lv13">
54                                 <p id = "p_lv14">
55                                     <a>First anchor tag</a>
56                                     <a>Second anchor tag</a>
57                                     <a>Third anchor tag</a>
58                                 </p>
59                             </p>
60                         </p>
61                     </p>
62                 </body>
63             </html>
64             ''', "html.parser")
65
66         cls.css_select_soup = BeautifulSoup(
67             '''
68             <html>
69                 <head>
70                 </head>
71                 <body>
72                     <p class = "my_CSS_class">Here is my styled paragraph</p>

```



```

73         <a id = "my_link"></a>
74     </body>
75 </html>
76 ''' , "html.parser")
77
78
79
80 ##Directly defined tags, to compare with BS output.
81
82 #a generic anchor tag for append testing
83 cls.an_append_tag = cls.soup.new_tag("a")
84 cls.an_append_tag.string = "Foo"
85 cls.an_appended_tag = cls.soup.new_tag("a")
86 cls.an_appended_tag.string = "FooBar"
87
88 #a generic anchor tag
89 cls.a_tag = cls.soup.new_tag("a")
90 cls.a_tag.string = "An anchor tag"
91
92 #First, second, third anchor
93 cls.a_tag_first = cls.soup.new_tag("a")
94 cls.a_tag_first.string = "First anchor tag"
95
96 cls.a_tag_second = cls.soup.new_tag("a")
97 cls.a_tag_second.string = "Second anchor tag"
98
99 cls.a_tag_third = cls.soup.new_tag("a")
100 cls.a_tag_third.string = "Third anchor tag"
101
102 #a-tag with id
103 cls.a_tag_id = cls.soup.new_tag("a", id="my_link")
104
105 #p-tag with class attribute
106 cls.p_tag_css = cls.soup.new_tag("p", **{'class': 'my_CSS_class'})
107 cls.p_tag_css.string = "Here is my styled paragraph"
108
109 #tag with no name
110 cls.no_name_tag = cls.soup.new_tag("")
111
112
113 def test_soup_contents(self):
114     ''' Test the contents attribute, which returns the contents of a tag.
115     Note the line break elements
116     A string should throw error on contents'''
117
118     tags = ["\n", self.a_tag_first, "\n", self.a_tag_second, "\n", self.a_tag_third, "\n"]
119
120     self.assertEqual(self.a_nested_soup.p.p.p.p.contents, tags)
121     self.assertEqual(self.a_nested_soup.p.p.p.p.a.contents, [self.a_tag_first.string])
122
123     with self.assertRaises(AttributeError):
124         self.assertEqual(self.a_nested_soup.p.p.p.p.a.string.contents, 0)
125
126
127 def test_soup_string(self):
128     ''' Test the string attribute for finding the string in a tag '''
129

```

```

130     markup = '<a>A tag string</a>'
131     string = 'A tag string'
132     string_soup = BeautifulSoup(markup, "html.parser")
133     self.assertEqual(string_soup.a.string, string)
134
135     def test_soup_stringless(self):
136         ''' Test soup constructor for nonexisting string '''
137
138         markup = "<a></a>"
139         stringless_soup = BeautifulSoup(markup, "html.parser")
140         self.assertEqual(stringless_soup.a.string, None)
141
142     def test_soup_strings(self):
143         ''' Test soup constructor for strings '''
144
145         markup = '<html><head>String0</head><body>String1</body>String2</html>'
146         soup = BeautifulSoup(markup, "html.parser")
147
148         strings = ["String0", "String1", "String2"]
149
150         soup_strings = []
151         for string in soup.strings:
152             soup_strings.append(string)
153
154         self.assertEqual(soup_strings, strings)
155
156         with self.assertRaises(IndexError):
157             soup_strings[3]
158
159     def test_soup_no_strings(self):
160         '''Test soup constructor for strings without strings'''
161
162         markup = '<html><head></head><body></body></html>'
163         no_strings_soup = BeautifulSoup(markup, "html.parser")
164         strings = []
165         for string in no_strings_soup.strings:
166             strings.append(string)
167         self.assertEqual(strings, [])
168
169
170     def test_soup_head(self):
171         ''' Test attribute navigation for head tag '''
172
173         markup = '<html><head>head test</head><body><p>paragraph</p></body></html>'
174
175         head_content_markup = ['head test']
176         soup = BeautifulSoup(markup, "html.parser")
177
178         head_tag = soup.new_tag("head")
179         head_tag.string = "head test"
180
181         self.assertEqual(soup.head, head_tag)
182         self.assertEqual(soup.head.contents, head_tag.contents)
183
184     def test_soup_headless(self):
185         ''' Test attribute navigation without head tag '''
186

```

```

187     markup = '<html><body><p>paragraph</p></body></html>'
188     headless_soup = BeautifulSoup(markup, "html.parser")
189
190     self.assertEqual(headless_soup.head, None)
191
192     def test_soup_title(self):
193         ''' Test attribute navigation for title tag '''
194
195         markup = '<html><head><title>A title</title></head><body></body></html>'
196         title_soup = BeautifulSoup(markup, "html.parser")
197
198         title_tag = title_soup.new_tag("title")
199         title_tag.string = 'A title'
200
201         self.assertEqual(title_soup.title, title_tag)
202         self.assertEqual(title_soup.title.string, title_tag.string)
203
204     def test_soup_titleless(self):
205         ''' Test attribute navigation for nonexistent title tag '''
206
207         markup = '<html><head>asd</head><body>dsa</body></html>'
208         title_soup = BeautifulSoup(markup, "html.parser")
209         self.assertEqual(title_soup.title, None)
210
211
212     def test_soup_children(self):
213         ''' The children attribute returns a generator, for iterating over children
214         Note that the children attribute returns linebreaks'''
215
216         tags = ["\n", self.a_tag_first, "\n", self.a_tag_second, "\n", self.a_tag_third, "\n"]
217
218         index = 0
219         for c in self.a_nested_soup.p.p.p.p.children:
220             self.assertEqual(c, tags[index])
221             index += 1
222
223
224     def test_soup_descendants(self):
225         ''' The descendants attribute returns the all descendants of a tag, including strings '''
226
227         markup = '<a><b>bold tag 1</b><b>bold tag 2</b></a>'
228         soup = BeautifulSoup(markup, "html.parser")
229
230         a1 = soup.new_tag("a")
231         b1 = soup.new_tag("b")
232         b1.string = "bold tag 1"
233
234         b2 = soup.new_tag("b")
235         b2.string = "bold tag 2"
236
237         a1.append(b1)
238         a1.append(b2)
239
240         desc = [a1, b1, b1.string, b2, b2.string]
241
242         index = 0
243         for c in soup.descendants:

```

```

244         self.assertEqual(c, desc[index])
245         index += 1
246
247     def test_soup_stripped_strings(self):
248         ''' Test stripped_strings, which should remove whitespaces and linebreaks before and after strings '''
249
250         markup = '''<html><head>
251             String0
252         </head><body>    String1  </body> String2
253         </html>'''
254
255         soup = BeautifulSoup(markup, "html.parser")
256
257         strings = ["String0", "String1", "String2"]
258
259         soup_strings = []
260         for string in soup.stripped_strings:
261             soup_strings.append(string)
262
263         self.assertEqual(strings, soup_strings)
264
265         with self.assertRaises(IndexError):
266             strings[3]
267
268     def test_soup_stripped_strings_empty(self):
269         ''' Test stripping an soup with just whitespace'''
270
271         markup = '''<a>        </a>    <a>
272         </a>'''
273         soup = BeautifulSoup(markup, "html.parser")
274
275         strings = []
276         for string in soup.stripped_strings:
277             strings.append(string)
278
279         self.assertEqual(strings, [])
280
281     def test_parent(self):
282         '''Parent attribute finds the direct parent. This applies to strings as well.
283         The parent of a soup is None'''
284
285         markup = '<a><b>bold tag 1</b><b>bold tag 2</b></a>'
286         soup = BeautifulSoup(markup, "html.parser")
287
288         self.assertEqual(soup.a.b.parent, soup.a)
289         self.assertEqual(soup.a.parent, soup)
290         self.assertEqual(soup.parent, None)
291
292         self.assertEqual(soup.a.b.string.parent, soup.a.b)
293
294     def test_parents(self):
295         '''The parents attribute finds all parents of a given tag'''
296
297         markup = '<a><b>bold tag 1</b><b>bold tag 2</b></a>'
298         soup = BeautifulSoup(markup, "html.parser")
299
300         parents = [soup.a.b, soup.a, soup]

```

```

301
302     ##Small irregularity here, parents does not go as high as None, although
303     ##the documentation states that it should.
304
305     index = 0
306     for c in soup.a.b.string.parents:
307         self.assertEqual(c, parents[index])
308         index += 1
309
310     def test_find(self):
311         '''Tests finding tags with a given name'''
312
313         self.assertEqual(self.a_simple_soup.find("a"), self.a_tag)
314         self.assertEqual(self.a_simple_soup.p.find("a"), self.a_tag)
315         self.assertEqual(self.a_nested_soup.find("a"), self.a_tag_first)
316
317     def test_find_empty(self):
318         ''' Test finding a tag with empty string name'''
319
320         self.assertEqual(self.a_nested_soup.find(""), None) #finds html tag
321         #self.assertEqual(self.a_nested_soup.find(""), self.a_nested_soup) #this fails also
322
323     def test_find_fail(self):
324         '''Test not finding a tag'''
325
326         self.assertEqual(self.a_nested_soup.find("b"), None)
327
328     def test_find_all(self):
329         '''Find all tags, returns a list'''
330
331         result = [self.a_tag_first, self.a_tag_second, self.a_tag_third]
332         self.assertEqual(self.a_nested_soup.find_all("a"), result)
333
334     def test_find_all_not_found(self):
335         '''Search for non-existing tags'''
336
337         self.assertEqual(self.a_nested_soup.find_all("b"), [])
338         self.assertEqual(self.a_nested_soup.find_all(""), []) #finds nothing
339
340     def test_css_select(self):
341         '''Find tags based on CSS selectors, returns a list'''
342
343         self.assertEqual(self.css_select_soup.select(".my_CSS_class"), [self.p_tag_css])
344         self.assertEqual(self.css_select_soup.select("#my_link"), [self.a_tag_id])
345         self.assertEqual(self.css_select_soup.select('a[id="my_link"]'), [self.a_tag_id])
346         self.assertEqual(self.css_select_soup.select('#my_link'), [self.a_tag_id])
347         self.assertEqual(self.css_select_soup.select('a[id~="my_link"]'), [self.a_tag_id])
348
349     def test_css_select_empty(self):
350         ''' Test the select function with an empty string'''
351         self.assertEqual(self.css_select_soup.select(""), None)
352         #Crashes with IndexError
353
354     def test_find_next(self):
355         '''Use find_next to find the next anchor tag, until there are no more'''
356
357         self.assertEqual(self.a_nested_soup.a, self.a_tag_first)

```

```

358     self.assertEqual(self.a_nested_soup.a.find_next(), self.a_tag_second)
359     self.assertEqual(self.a_nested_soup.a.find_next().find_next(), self.a_tag_third)
360     self.assertEqual(self.a_nested_soup.a.find_next().find_next().find_next(), None)
361
362     def test_find_all_next(self):
363         '''Find all following tags using two different usages of find_all_next'''
364
365         self.next_tags = [self.a_tag_second, self.a_tag_third]
366         self.first_link = self.a_nested_soup.a
367
368         self.assertEqual(self.first_link.find_all_next(), self.next_tags)
369         self.assertEqual(self.first_link.find_all_next("a"), self.next_tags)
370         self.assertEqual(self.first_link.find_all_next("b"), [])
371
372     def test_find_previous(self):
373         '''Finds the previous tag of a given name '''
374
375         self.assertEqual(self.a_simple_soup.a.find_previous("html"), self.a_simple_soup.html)
376
377     def test_find_all_previous(self):
378         '''Finds all the previous tags of a given name '''
379
380         result = [self.a_nested_soup.p.p.p.p, self.a_nested_soup.p.p.p, self.a_nested_soup.p.p, self.a_nested_soup.p]
381         self.assertEqual(self.a_nested_soup.a.find_all_previous("p"), result)
382
383         '''find previous which is not parent '''
384         self.assertEqual(self.a_nested_soup.a.find_all_previous("head"), [self.a_nested_soup.head])
385
386     def test_find_parent(self):
387         ''' Finds the parent of a given name'''
388
389         self.assertEqual(self.a_nested_soup.a.find_parent("p"), self.a_nested_soup.p.p.p.p)
390         self.assertEqual(self.a_nested_soup.p.p.p.p.find_parent("p"), self.a_nested_soup.p.p.p)
391
392
393
394     def test_find_parents(self):
395         '''Finds all the parents with a given name '''
396
397         p_lvl1 = self.a_nested_soup.p
398         p_lvl2 = self.a_nested_soup.p.p
399         p_lvl3 = self.a_nested_soup.p.p.p
400         p_lvl4 = self.a_nested_soup.p.p.p.p
401
402         self.assertEqual(self.a_nested_soup.a.find_parents("p"), [p_lvl4, p_lvl3, p_lvl2, p_lvl1])
403         self.assertEqual(self.a_nested_soup.p.find_parents("body"), [self.a_nested_soup.body])
404
405
406
407     def test_append(self):
408         ''' Test the append function by appending an <b>Bar</b> tag'''
409
410         soup = BeautifulSoup("<a>Foo</a>", "html.parser")
411         tag = soup.new_tag("b")
412         tag.string = "Bar"
413
414         soup.a.append(tag)

```

```

415
416     self.assertEqual(soup.a.b, tag)
417     self.assertEqual("<a>Foo<b>Bar</b></a>", str(soup.a))
418
419     ''' Test the append function by appending empty string '''
420
421     soup_2 = BeautifulSoup("<a>Foo</a>", "html.parser")
422     soup_2_app = BeautifulSoup("<a>Foo</a>", "html.parser")
423     soup_2.append("")
424     self.assertEqual(soup_2.contents, soup_2_app.contents + [""])
425
426
427 def test_append_raise(self):
428     ''' Assert that append without argument raises a TypeError due to too few arguments '''
429
430     append_raise_soup = BeautifulSoup("<a>Foo</a>", "html.parser")
431     with self.assertRaises(TypeError):
432         append_raise_soup.a.append()
433
434 def test_insert_before(self):
435     ''' Insert_before() test with a <i>Don't</i> tag inserted before string '''
436
437     soup = BeautifulSoup("<b><i>Don't</i>stop</b>", "html.parser")
438     b_soup = BeautifulSoup("<b>stop</b>", "html.parser")
439
440     tag = soup.new_tag("i")
441     tag.string = "Don't"
442
443     b_soup.b.string.insert_before(tag)
444
445     self.assertEqual(soup, b_soup)
446
447     ''' Insert_before test with string argument before tag. '''
448     soup_2 = BeautifulSoup("<b>tester soup</b>", "html.parser")
449     b_soup_2 = BeautifulSoup("Foo<b>tester soup</b>", "html.parser")
450
451     soup_2.b.insert_before("Foo")
452     self.assertEqual(soup_2, b_soup_2)
453
454 def test_insert_before_raise(self):
455     ''' Insert_before() without argument test. '''
456
457     soup = BeautifulSoup("<b>testing soup</b>", "html.parser")
458     with self.assertRaises(TypeError):
459         soup.insert_before()
460
461 def test_insert_after(self):
462     ''' Insert_after() test with an <i>stop</i> tag after string '''
463
464     soup = BeautifulSoup("<b>Don't </b>", "html.parser")
465     soup_r = BeautifulSoup("<b>Don't <i>stop</i></b>", "html.parser")
466
467     tag = soup.new_tag("i")
468     tag.string = "stop"
469
470     soup.b.string.insert_after(tag)
471

```

```

472     self.assertEqual(soup, soup_r)
473
474     ''' Insert_after() test with an string argument after tag. '''
475     soup_2 = BeautifulSoup("<b>Don't</b>", "html.parser")
476     soup_2_r = BeautifulSoup("<b>Don't</b>TestThis", "html.parser")
477     soup_2.b.insert_after("TestThis")
478
479     self.assertEqual(soup_2, soup_2_r)
480
481
482
483     def test_insert_after_raise(self):
484         ''' Insert_after () test with no argument, to raise exception. '''
485
486         soup = BeautifulSoup("<b>Don't </b>", "html.parser")
487         with self.assertRaises(TypeError):
488             soup.insert_after()
489
490     def test_clear(self):
491         ''' Clear the contents of a HTML <a href> tag. '''
492
493         markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
494         soup = BeautifulSoup(markup, "html.parser")
495
496         soup.a.clear()
497
498         cleared_markup = '<a href="http://example.com/"></a>'
499         cleared_soup = BeautifulSoup(cleared_markup, "html.parser")
500
501         self.assertEqual(soup, cleared_soup)
502
503     def test_clear_empty(self):
504         ''' Clear the contents of an empty HTML <a href> tag. '''
505
506         markup = '<a href="http://example.com/"></a>'
507         soup = BeautifulSoup(markup, "html.parser")
508         soup_2 = BeautifulSoup(markup, "html.parser")
509
510         soup.a.clear()
511
512         self.assertEqual(soup, soup_2)
513
514     def test_clear_with_arg(self):
515         ''' Call clear with argument, works just as clear(). '''
516
517         markup = '<a href="http://example.com/"></a>'
518         soup = BeautifulSoup(markup, "html.parser")
519         soup_2 = BeautifulSoup(markup, "html.parser")
520
521         soup.a.clear("argument")
522
523         self.assertEqual(soup, soup_2)
524
525     def test_extract(self):
526         ''' Extract a tag and see if it is correctly returned, and that the original is changed accordingly '''
527
528         markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'

```



```

529     extracted_markup = '<a href="http://example.com/">I linked to </a>'
530
531     soup = BeautifulSoup(markup, "html.parser")
532     extracted_soup = BeautifulSoup(extracted_markup, "html.parser")
533
534     tag = soup.new_tag("i")
535     tag.string = "example.com"
536
537     extracted_tag = soup.i.extract()
538
539     self.assertEqual(extracted_tag.parent, None)
540     self.assertEqual(extracted_tag, tag)
541     self.assertEqual(soup, extracted_soup)
542
543     def test_extract_raises(self):
544         ''' Test extracting a non-existing tag, ensuring that the original isn't changed. '''
545
546         markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
547         extr_soup = BeautifulSoup(markup, "html.parser")
548         extr_null_soup = extr_soup.extract()
549
550         self.assertEqual(extr_soup, extr_null_soup)
551
552     def test_extract_with_arg(self):
553         ''' Test extracting by calling extract() with a random argument. '''
554
555         markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
556         extr_arg_soup = BeautifulSoup(markup, "html.parser")
557         with self.assertRaises(TypeError):
558             extr_arg_soup.extract("arg")
559
560     def test_decompose(self):
561         ''' Test the decompose function by removing and destroying a tag. '''
562
563         markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
564         markup_decomposed = '<a href="http://example.com/">I linked to </a>'
565
566         soup = BeautifulSoup(markup, "html.parser")
567         dec_soup = BeautifulSoup(markup_decomposed, "html.parser")
568
569         soup.i.decompose()
570
571         self.assertEqual(soup, dec_soup)
572
573     def test_decompose_empty(self):
574         ''' Test the decompose function by removing an empty tag. '''
575
576         markup = '<a href="http://example.com/">I linked to <i></i></a>'
577         markup_decomposed = '<a href="http://example.com/">I linked to </a>'
578
579         soup = BeautifulSoup(markup, "html.parser")
580         dec_soup = BeautifulSoup(markup_decomposed, "html.parser")
581
582         soup.i.decompose()
583
584         self.assertEqual(soup, dec_soup)
585

```

```

586 def test_decmpose_arg(self):
587     ''' Test the decompose function by calling decompose with arg. '''
588
589     markup = '<a href="http://example.com/">I linked to <i> test </i></a>'
590     markup_decomposed = '<a href="http://example.com/">I linked to </a>'
591     dec_soup = BeautifulSoup(markup, "html.parser")
592
593     with self.assertRaises(TypeError):
594         dec_soup.i.decompose("test_arg")
595
596 def test_replace_with(self):
597     ''' Test the replace with function '''
598
599     markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
600     rep_markup = '<a href="http://example.com/">I linked to <b>new_example.com</b></a>'
601
602     soup = BeautifulSoup(markup, "html.parser")
603     rep_soup = BeautifulSoup(rep_markup, "html.parser")
604
605     new_tag = soup.new_tag("b")
606     new_tag.string = "new_example.com"
607
608     soup.a.i.replace_with(new_tag)
609
610     self.assertEqual(soup, rep_soup)
611
612 def test_replace_tag_with_string(self):
613     ''' Test replacing tag with string '''
614
615     markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
616     rep_markup = '<a href="http://example.com/">I linked to testText</a>'
617
618     soup = BeautifulSoup(markup, "html.parser")
619     rep_soup = BeautifulSoup(rep_markup, "html.parser")
620
621     soup.i.replace_with("testText")
622
623     self.assertEqual(soup, rep_soup)
624     self.assertEqual(str(soup), str(rep_soup))
625
626     #soup.a.contents <--- this gives a list containing 2 elements, one for each string.
627
628 def test_replace_with_no_tag(self):
629     ''' Test replace_with() called without argument. '''
630
631     markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
632     no_soup = BeautifulSoup(markup, "html.parser")
633     a_tag = no_soup.a
634
635     with self.assertRaises(AttributeError):
636         a_tag.i.replace_with()
637
638 def test_wrap(self):
639     ''' Test wrap() by wrapping a b tag around an a tag '''
640
641     markup = '<a>Text to be wrapped</a>'
642     soup = BeautifulSoup(markup, "html.parser")

```

```

643
644     wr_markup = '<b><a>Text to be wrapped</a></b>'
645     wr_soup = BeautifulSoup(wr_markup, "html.parser")
646
647     tag = soup.new_tag("b")
648
649     soup.a.wrap(tag)
650
651     self.assertEqual(soup, wr_soup)
652
653 def test_wrap_with_string(self):
654     ''' Test wrap() by wrapping a b tag around an a tag, where b contains a string'''
655
656     markup = '<a>Text to be wrapped</a>'
657
658     soup = BeautifulSoup(markup, "html.parser")
659
660     wr_markup = '<b>string<a>Text to be wrapped</a></b>'
661     wr_soup = BeautifulSoup(wr_markup, "html.parser")
662
663     tag = soup.new_tag("b")
664     tag.string = "string"
665
666     soup.a.wrap(tag)
667
668     self.assertEqual(soup, wr_soup)
669
670 def test_unwrap(self):
671     ''' Test unwrapping and compare to soup after unwrapping.'''
672
673     markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
674     soup = BeautifulSoup(markup, "html.parser")
675
676     unwrapped_markup = '<a href="http://example.com/">I linked to example.com</a>'
677     unwrapped_soup = BeautifulSoup(unwrapped_markup, "html.parser")
678
679     soup.a.i.unwrap()
680
681     #The soups are not equal, but as strings they are equal
682     self.assertEqual(str(soup), str(unwrapped_soup))
683     self.assertEqual(soup, unwrapped_soup)
684
685     #soup.a.string <-- This is None at this point
686     #soup.a.contents <--- this gives a list containing 2 elements, one for each string.
687
688 def test_unwrap_with_arg(self):
689     ''' Test unwrap() with arg. '''
690
691     markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
692     unwrap_soup = BeautifulSoup(markup, "html.parser")
693     a_tag = unwrap_soup.a
694
695     with self.assertRaises(TypeError):
696         a_tag.i.unwrap("a")
697
698
699 def test_unwrap_no_tag(self):

```

```

700     ''' Test unwrap() on non-exisiting tag. '''
701
702     markup = 'htadsasdtp/example.codssdd'
703     unwrap_soup = BeautifulSoup(markup, "html.parser")
704
705     with self.assertRaises(ValueError):
706         unwrap_soup.unwrap()
707
708
709     def test_wrap_unwrap(self):
710         ''' Test wrapping and then unwrapping '''
711
712         markup = '<a>Text to be wrapped</a>'
713         soup = BeautifulSoup(markup, "html.parser")
714         soup_2 = BeautifulSoup(markup, "html.parser")
715
716         tag = soup.new_tag("b")
717
718         soup.a.wrap(tag)
719         soup.b.unwrap()
720
721         self.assertEqual(soup, soup_2)
722
723     def test_no_name_tag(self):
724         ''' test returning a name of a tag with no name'''
725
726         self.assertEqual(self.no_name_tag.name, "")
727
728
729
730 if __name__ == '__main__':
731     unittest.main()

```

The file `unittest_cov_tests.py`

```

1  ## File for white box testing of the BS4 web scraping library.
2  ## This code provides full branch coverage of the functions find and find_all
3  ## as well as the internal function _find_all
4
5
6  ## Erik Bertse
7  ## Sara Gustavsson
8  ## Moa Marklund
9  ## Henrik Thorsell
10
11 ## Software Testing, 5c
12 ## Autumn 2017
13 ## Uppsala University
14
15
16 import unittest
17 import itertools
18 from bs4 import BeautifulSoup
19 from bs4 import SoupStrainer
20
21
22 ## WARNING ## This is a badly written iterator, specifically for breaking _find_all ###

```

```

23  ## DONT USE THIS CODE IF YOU NEED AN ITERATOR
24  class EvilIter():
25      def __init__(self,breaker):
26          self.current = breaker
27
28      def __iter__(self):
29          return self
30
31      def next(self):
32          if self.current == 0:
33              raise StopIteration
34          else:
35              self.current = self.current - 1
36              return self
37
38      def __len__(self):
39          return 0
40
41
42  class TestBS(unittest.TestCase):
43
44      def test_soup_branch(self):
45
46          self.a_simple_soup = BeautifulSoup(
47              '''
48              <html>
49              <head>
50              </head>
51              <body>
52              <p>
53              <a href="http://example.com"><b>A bold anchor tag</b></a>
54              </p>
55              </body>
56              </html>
57              ''', "html.parser")
58
59          self.tiny_soup = BeautifulSoup(
60              '''
61              <html></html>
62              ''', "html.parser")
63
64          a_tag = self.a_simple_soup.a
65
66          #Here we call find directly
67          tag = self.a_simple_soup.find("a")
68          self.assertEqual(tag, a_tag)
69
70          #Here we call find all directly
71          tag = self.a_simple_soup.find_all("a")
72          self.assertEqual(tag, [a_tag])
73
74          #Here we call find all directly, with no argument
75          html = self.a_simple_soup.html
76          head = self.a_simple_soup.head
77          body = self.a_simple_soup.body
78          p = self.a_simple_soup.p
79          a = self.a_simple_soup.a

```

```

80     b = self.a_simple_soup.b
81
82     tag = self.a_simple_soup.find_all()
83     self.assertEqual(tag, [html, head, body, p, a, b])
84
85     #here kwarg contains "string", and there is no text argument.
86     tag = self.a_simple_soup.find_all(string = "a")
87     self.assertEqual(tag, [])
88
89     #here we define a custom SoupStrainer
90     strainer = SoupStrainer("a")
91     tag = self.a_simple_soup.find_all(strainer)
92     self.assertEqual(tag, [a_tag])
93
94     #here we find with limit
95     tag = self.a_simple_soup.find_all("a", limit = 1)
96     self.assertEqual(tag, [a_tag])
97
98     #Putting recursive to False makes BS only look one level down from the
99     #tag from which the search started.
100    tag = self.tiny_soup.find_all("a", recursive = False)
101    self.assertEqual(tag, [])
102
103    #This is far outside of normal BS usage
104    iterator = EvilIter(3)
105    tag = self.tiny_soup._find_all("a", None, None, None, recursive = False, generator = iterator)
106    self.assertEqual(tag, [])
107
108    if __name__ == '__main__':
109        unittest.main()

```