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?

BeatifulSoup 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:

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

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

```html
<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:

```python
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 in 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:

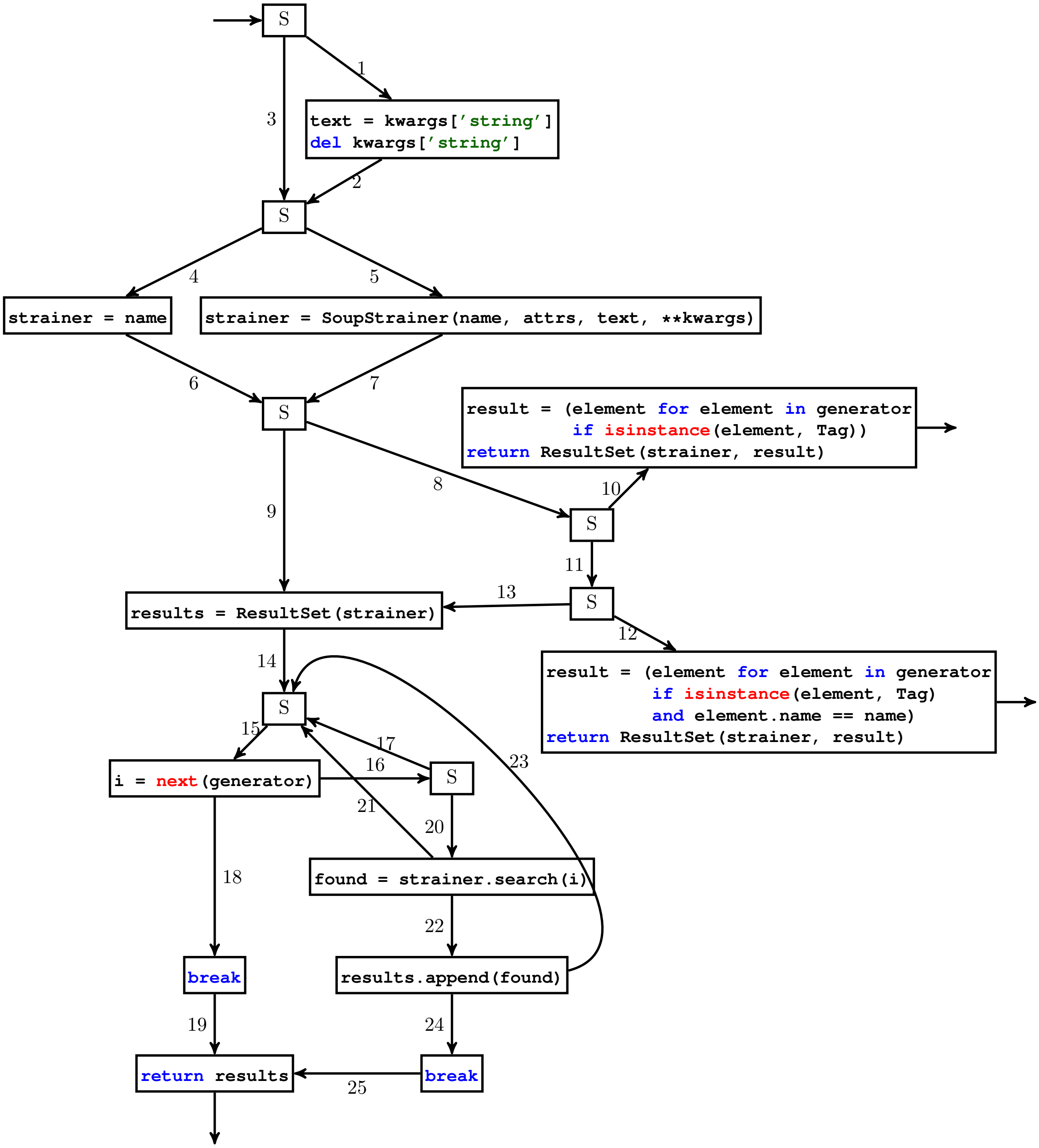
```
def _find_all(self, name, attrs, text, limit, generator, **kwargs):
      "Iterates over a generator looking for things that match."

      if text is None and ’string’ in kwargs:
          text = kwargs[’string’]
          del kwargs[’string’]

      if isinstance(name, SoupStrainer):
          strainer = name
      else:
          strainer = SoupStrainer(name, attrs, text, **kwargs)

      if text is None and not limit and not attrs and not kwargs:
          if name is True or name is None:
              # Optimization to find all tags.
              result = (element for element in generator
                          if isinstance(element, Tag))
              return ResultSet(strainer, result)
          elif isinstance(name, basestring):
              # Optimization to find all tags with a given name.
              result = (element for element in generator
                        if isinstance(element, Tag)
                          and element.name == name)
              return ResultSet(strainer, result)
      results = ResultSet(strainer)
      while True:
          try:
              i = next(generator)
          except StopIteration:
              break
          if i:
              found = strainer.search(i)
              if found:
                  results.append(found)
                  if limit and len(results) >= limit:
                      break
      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.



```
text = kwargs['string']
del kwargs['string']
```

```
strainer = name
```

```
strainer = SoupStrainer(name, attrs, text, **kwargs)
```

```
result = (element for element in generator
          if isinstance(element, Tag))
return ResultSet(strainer, result)
```

```
results = ResultSet(strainer)
```

```
result = (element for element in generator
          if isinstance(element, Tag)
          and element.name == name)
return ResultSet(strainer, result)
```

```
i = next(generator)
```

```
found = strainer.search(i)
```

```
break
```

```
results.append(found)
```

```
return results
```

```
break
```

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 egde has no condition on when it is taken, then it is taken unconditionally.

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

[1] Although 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 egde, and the node at the start of the next edge, which are equal.

The line

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

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

The line

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

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

The line

```
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

```
strainer = SoupStrainer("a")
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

```
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 egde. This resulted in the following line of code, and its corresponding execution path.

The lines

```
iterator = EvilIter(3)
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.

[3] Copied 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**:

```
## File for black box testing of the BS4 web scraping library.


## Erik Bertse
## Sara Gustavsson
## Moa Marklund
## Henrik Thorsell

## Software Testing, 5c
## Autumn 2017
## Uppsala University

import unittest
from bs4 import BeautifulSoup

```

```

class TestBS(unittest.TestCase):

  @classmethod
  def setUpClass(cls):
    #Sets up various soups for tests

    cls.soup = BeautifulSoup("", "html.parser")

    #How to get soupify html files: (this is not used in the tests below)
    #cls.html_file = open("html_testfiles/html_test.html")
    #cls.html_file_soup = BeautifulSoup(cls.html_file, "html.parser")


    #Directly defined soups.

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

    cls.a_nested_soup = BeautifulSoup(
    '''
    <html>
      <head>
      </head>
        <body>
          <p id = "p_lvl1">
            <p id = "p_lvl2">
              <p id = "p_lvl3">
    <p id = "p_lvl4">
      <a>First anchor tag</a>
      <a>Second anchor tag</a>
      <a>Third anchor tag</a>
    </p>
              </p>
            </p>
          </p>
        </body>
    </html>
    ''', "html.parser")

    cls.css_select_soup = BeautifulSoup(
    '''
    <html>
      <head>
      </head>
        <body>
          <p class = "my_CSS_class">Here is my styled paragraph</p>
          <a id = "my_link"></a>
        </body>
    </html>
    ''', "html.parser")



    ##Directly defined tags, to compare with BS output.

    #a generic anchor tag for append testing
    cls.an_append_tag = cls.soup.new_tag("a")
    cls.an_append_tag.string = "Foo"
    cls.an_appended_tag = cls.soup.new_tag("a")
    cls.an_appended_tag.string = "FooBar"

    #a generic anchor tag
    cls.a_tag = cls.soup.new_tag("a")
    cls.a_tag.string = "An anchor tag"

    #First, second, third anchor
    cls.a_tag_first = cls.soup.new_tag("a")
    cls.a_tag_first.string = "First anchor tag"

    cls.a_tag_second = cls.soup.new_tag("a")
    cls.a_tag_second.string = "Second anchor tag"

    cls.a_tag_third = cls.soup.new_tag("a")
    cls.a_tag_third.string = "Third anchor tag"

    #a-tag with id
    cls.a_tag_id = cls.soup.new_tag("a", id="my_link")

    #p-tag with class attribute
    cls.p_tag_css = cls.soup.new_tag("p", **{'class':'my_CSS_class'})
    cls.p_tag_css.string = "Here is my styled paragraph"

    #tag with no name
    cls.no_name_tag = cls.soup.new_tag("")


  def test_soup_contents(self):
    ''' Test the contents attribute, which returns the contents of a tag.
    Note the line break elements
    A string should throw error on contents'''

    tags = ["\n", self.a_tag_first, "\n", self.a_tag_second, "\n", self.a_tag_third, "\n"]

    self.assertEqual(self.a_nested_soup.p.p.p.p.contents, tags)
    self.assertEqual(self.a_nested_soup.p.p.p.p.a.contents, [self.a_tag_first.string])

    with self.assertRaises(AttributeError):
            self.assertEqual(self.a_nested_soup.p.p.p.p.a.string.contents, 0)


  def test_soup_string(self):
    ''' Test the string attribute for finding the string in a tag '''

```

```python
    markup = '<a>A tag string</a>'
    string = 'A tag string'
    string_soup = BeautifulSoup(markup, "html.parser")
    self.assertEqual(string_soup.a.string, string)

  def test_soup_stringless(self):
    ''' Test soup constructor for nonexisting string '''

    markup = "<a></a>"
    stringless_soup = BeautifulSoup(markup, "html.parser")
    self.assertEqual(stringless_soup.a.string, None)

  def test_soup_strings(self):
    ''' Test soup constructor for strings '''

    markup = '<html><head>String0</head><body>String1</body>String2</html>'
    soup = BeautifulSoup(markup, "html.parser")

    strings = ["String0", "String1", "String2"]

    soup_strings = []
    for string in soup.strings:
            soup_strings.append(string)

    self.assertEqual(soup_strings, strings)

    with self.assertRaises(IndexError):
            soup_strings[3]

  def test_soup_no_strings(self):
    '''Test soup constructor for strings without strings'''

    markup = '<html><head></head><body></body></html>'
    no_strings_soup = BeautifulSoup(markup, "html.parser")
    strings = []
    for string in no_strings_soup.strings:
            strings.append(string)
    self.assertEqual(strings, [])


  def test_soup_head(self):
    ''' Test attribute navigation for head tag '''

    markup = '<html><head>head test</head><body><p>paragraph</p></body></html>'

    head_content_markup = ['head test']
    soup = BeautifulSoup(markup, "html.parser")

    head_tag = soup.new_tag("head")
    head_tag.string = "head test"

    self.assertEqual(soup.head, head_tag)
    self.assertEqual(soup.head.contents, head_tag.contents)

  def test_soup_headless(self):
    ''' Test attribute navigation without head tag '''

    markup = '<html><body><p>paragraph</p></body></html>'
    headless_soup = BeautifulSoup(markup, "html.parser")

    self.assertEqual(headless_soup.head, None)

  def test_soup_title(self):
    ''' Test attribute navigation for title tag '''

    markup = '<html><head><title>A title</title></head><body></body></html>'
    title_soup = BeautifulSoup(markup, "html.parser")

    title_tag = title_soup.new_tag("title")
    title_tag.string = 'A title'

    self.assertEqual(title_soup.title, title_tag)
    self.assertEqual(title_soup.title.string, title_tag.string)

  def test_soup_titleless(self):
    ''' Test attribute navigation for nonexisting title tag '''

    markup = '<html><head>asd</head><body>dsa</body></html>'
    title_soup = BeautifulSoup(markup, "html.parser")
    self.assertEqual(title_soup.title, None)


  def test_soup_children(self):
    ''' The children attribute returns a generator, for iterating over children
    Note that the children attribute returns linebreaks'''

    tags = ["\n", self.a_tag_first, "\n", self.a_tag_second, "\n", self.a_tag_third, "\n"]

    index = 0
    for c in self.a_nested_soup.p.p.p.p.children:
            self.assertEqual(c, tags[index])
            index += 1


  def test_soup_descendants(self):
    ''' The descentants attribute returns the all descentants of a tag, including strings '''

    markup = '<a><b>bold tag 1</b><b>bold tag 2</b></a>'
    soup = BeautifulSoup(markup, "html.parser")

    a1 = soup.new_tag("a")
    b1 = soup.new_tag("b")
    b1.string = "bold tag 1"

    b2 = soup.new_tag("b")
    b2.string = "bold tag 2"

    a1.append(b1)
    a1.append(b2)

    desc = [a1, b1, b1.string, b2, b2.string]

    index = 0
    for c in soup.descendants:
```

```
             self.assertEqual(c, desc[index])
             index += 1

  def test_soup_stripped_strings(self):
    ''' Test stripped_strings, which should remove whitespaces and linebreaks before and after strings '''

    markup = '''<html><head>
                 String0
             </head><body>   String1  </body> String2
             </html>'''

    soup = BeautifulSoup(markup, "html.parser")

    strings = ["String0", "String1", "String2"]

    soup_strings = []
    for string in soup.stripped_strings:
            soup_strings.append(string)

    self.assertEqual(strings, soup_strings)

    with self.assertRaises(IndexError):
            strings[3]

  def test_soup_stripped_strings_empty(self):
    ''' Test stripping an soup with just whitespace'''

    markup = '''<a>     </a>   <a>
      </a>'''
    soup = BeautifulSoup(markup, "html.parser")

    strings = []
    for string in soup.stripped_strings:
            strings.append(string)

    self.assertEqual(strings, [])

  def test_parent(self):
    '''Parent attribute finds the direct parent. This applies to strings as well.
    The parent of a soup is None'''

    markup = '<a><b>bold tag 1</b><b>bold tag 2</b></a>'
    soup = BeautifulSoup(markup, "html.parser")

    self.assertEqual(soup.a.b.parent, soup.a)
    self.assertEqual(soup.a.parent, soup)
    self.assertEqual(soup.parent, None)

    self.assertEqual(soup.a.b.string.parent, soup.a.b)

  def test_parents(self):
    '''The parents attribute finds all parents of a given tag'''

    markup = '<a><b>bold tag 1</b><b>bold tag 2</b></a>'
    soup = BeautifulSoup(markup, "html.parser")

    parents = [soup.a.b, soup.a, soup]

    ##Small irragularity here, parents does not go as high as None, although
    ##the documentation states that it should.

    index = 0
    for c in soup.a.b.string.parents:
            self.assertEqual(c, parents[index])
            index += 1

  def test_find(self):
    '''Tests finding tags with a given name '''

    self.assertEqual(self.a_simple_soup.find("a"), self.a_tag)
    self.assertEqual(self.a_simple_soup.p.find("a"), self.a_tag)
    self.assertEqual(self.a_nested_soup.find("a"), self.a_tag_first)

  def test_find_empty(self):
    ''' Test finding a tag with empty string name '''

    self.assertEqual(self.a_nested_soup.find(""), None) #finds html tag
    #self.assertEqual(self.a_nested_soup.find(""), self.a_nested_soup) #this fails also

  def test_find_fail(self):
    '''Test not finding a tag '''

    self.assertEqual(self.a_nested_soup.find("b"), None)

  def test_find_all(self):
    '''Find all tags, returns a list'''

    result = [self.a_tag_first,self.a_tag_second,self.a_tag_third]
    self.assertEqual(self.a_nested_soup.find_all("a"), result)

  def test_find_all_not_found(self):
    '''Search for non-existing tags'''

    self.assertEqual(self.a_nested_soup.find_all("b"), [])
    self.assertEqual(self.a_nested_soup.find_all(""), [])  #finds nothing

  def test_css_select(self):
    '''Find tags based on CSS selectors, returns a list'''

    self.assertEqual(self.css_select_soup.select(".my_CSS_class"), [self.p_tag_css])
    self.assertEqual(self.css_select_soup.select("a#my_link"), [self.a_tag_id])
    self.assertEqual(self.css_select_soup.select('a[id="my_link"]'), [self.a_tag_id])
    self.assertEqual(self.css_select_soup.select('#my_link'), [self.a_tag_id])
    self.assertEqual(self.css_select_soup.select('a[id~="my_link"]'), [self.a_tag_id])

  def test_css_select_empty(self):
    ''' Test the select function with an empty string '''
    self.assertEqual(self.css_select_soup.select(""), None)
    #Crashes with IndexError

  def test_find_next(self):
    '''Use find_next to find the next anchor tag, until there are no more'''

    self.assertEqual(self.a_nested_soup.a, self.a_tag_first)
    self.assertEqual(self.a_nested_soup.a.find_next(), self.a_tag_second)
    self.assertEqual(self.a_nested_soup.a.find_next().find_next(), self.a_tag_third)
    self.assertEqual(self.a_nested_soup.a.find_next().find_next().find_next(), None)

  def test_find_all_next(self):
    '''Find all following tags using two different usages of find_all_next'''

    self.next_tags = [self.a_tag_second, self.a_tag_third]
    self.first_link = self.a_nested_soup.a

    self.assertEqual(self.first_link.find_all_next(), self.next_tags)
    self.assertEqual(self.first_link.find_all_next("a"), self.next_tags)
    self.assertEqual(self.first_link.find_all_next("b"), [])

  def test_find_previous(self):
    '''Finds the previous tag of a given name '''

    self.assertEqual(self.a_simple_soup.a.find_previous("html"), self.a_simple_soup.html)

  def test_find_all_previous(self):
    '''Finds all the previous tags of a given name '''

    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]
    self.assertEqual(self.a_nested_soup.a.find_all_previous("p"), result)

    '''find previous which is not parent '''
    self.assertEqual(self.a_nested_soup.a.find_all_previous("head"), [self.a_nested_soup.head])

  def test_find_parent(self):
    ''' Finds the parent of a given name'''

    self.assertEqual(self.a_nested_soup.a.find_parent("p"), self.a_nested_soup.p.p.p.p)
    self.assertEqual(self.a_nested_soup.p.p.p.p.find_parent("p"), self.a_nested_soup.p.p.p)



  def test_find_parents(self):
    '''Finds all the parents with a given name '''

    p_lvl1 = self.a_nested_soup.p
    p_lvl2 = self.a_nested_soup.p.p
    p_lvl3 = self.a_nested_soup.p.p.p
    p_lvl4 = self.a_nested_soup.p.p.p.p

    self.assertEqual(self.a_nested_soup.a.find_parents("p"), [p_lvl4, p_lvl3, p_lvl2, p_lvl1])
    self.assertEqual(self.a_nested_soup.p.find_parents("body"), [self.a_nested_soup.body])



  def test_append(self):
    ''' Test the append function by appending an <b>Bar</b> tag'''

    soup = BeautifulSoup("<a>Foo</a>", "html.parser")
    tag = soup.new_tag("b")
    tag.string = "Bar"

    soup.a.append(tag)

    self.assertEqual(soup.a.b, tag)
    self.assertEqual("<a>Foo<b>Bar</b></a>", str(soup.a))

    ''' Test the append function by appending empty string '''

    soup_2 = BeautifulSoup("<a>Foo</a>", "html.parser")
    soup_2_app = BeautifulSoup("<a>Foo</a>", "html.parser")
    soup_2.append("")
    self.assertEqual(soup_2.contents, soup_2_app.contents  + [""])


  def test_append_raise(self):
    ''' Assert that append without argument raises a TypeError due to too few arguments '''

    append_raise_soup = BeautifulSoup("<a>Foo</a>", "html.parser")
    with self.assertRaises(TypeError):
            append_raise_soup.a.append()

  def test_insert_before(self):
    ''' Insert_before() test with a <i>Don't</i> tag inserted before string '''

    soup = BeautifulSoup("<b><i>Don't</i>stop</b>", "html.parser")
    b_soup = BeautifulSoup("<b>stop</b>", "html.parser")

    tag = soup.new_tag("i")
    tag.string = "Don't"

    b_soup.b.string.insert_before(tag)

    self.assertEqual(soup, b_soup)

    ''' Insert_before test with string argument before tag. '''
    soup_2 = BeautifulSoup("<b>tester soup</b>", "html.parser")
    b_soup_2 = BeautifulSoup("Foo<b>tester soup</b>", "html.parser")

    soup_2.b.insert_before("Foo")
    self.assertEqual(soup_2, b_soup_2)

  def test_insert_before_raise(self):
    ''' Insert_before() without argument test.'''

    soup = BeautifulSoup("<b>testing soup</b>", "html.parser")
    with self.assertRaises(TypeError):
            soup.insert_before()

  def test_insert_after(self):
    ''' Insert_after() test with an <i>stop</i> tag after string '''

    soup = BeautifulSoup("<b>Don't </b>", "html.parser")
    soup_r = BeautifulSoup("<b>Don't <i>stop</i></b>", "html.parser")

    tag = soup.new_tag("i")
    tag.string = "stop"

    soup.b.string.insert_after(tag)

    self.assertEqual(soup, soup_r)

    ’’’ Insert_after() test with an string argument after tag. ’’’
    soup_2 = BeautifulSoup("<b>Don’t</b>", "html.parser")
    soup_2_r = BeautifulSoup("<b>Don’t</b>TestThis", "html.parser")
    soup_2.b.insert_after("TestThis")

    self.assertEqual(soup_2, soup_2_r)



  def test_insert_after_raise(self):
    ’’’ Insert_after () test with no argument, to raise exception. ’’’

    soup = BeautifulSoup("<b>Don’t </b>", "html.parser")
    with self.assertRaises(TypeError):
            soup.insert_after()

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

    markup = ’<a href="http://example.com/">I linked to <i>example.com</i></a>’
    soup = BeautifulSoup(markup, "html.parser")

    soup.a.clear()

    cleared_markup = ’<a href="http://example.com/"></a>’
    cleared_soup = BeautifulSoup(cleared_markup, "html.parser")

    self.assertEqual(soup, cleared_soup)

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

    markup = ’<a href="http://example.com/"></a>’
    soup = BeautifulSoup(markup, "html.parser")
    soup_2 = BeautifulSoup(markup, "html.parser")

    soup.a.clear()

    self.assertEqual(soup, soup_2)

  def test_clear_with_arg(self):
    ’’’ Call clear with argument, works just as clear(). ’’’

    markup = ’<a href="http://example.com/"></a>’
    soup = BeautifulSoup(markup, "html.parser")
    soup_2 = BeautifulSoup(markup, "html.parser")

    soup.a.clear("argument")

    self.assertEqual(soup, soup_2)

  def test_extract(self):
    ’’’ Extract a tag and see if it is correctly returned, and that the original is changed accordingly ’’’

    markup = ’<a href="http://example.com/">I linked to <i>example.com</i></a>’
    extracted_markup = '<a href="http://example.com/">I linked to </a>'

    soup = BeautifulSoup(markup, "html.parser")
    extracted_soup = BeautifulSoup(extracted_markup, "html.parser")

    tag = soup.new_tag("i")
    tag.string = "example.com"

    extracted_tag = soup.i.extract()

    self.assertEqual(extracted_tag.parent, None)
    self.assertEqual(extracted_tag, tag)
    self.assertEqual(soup, extracted_soup)

  def test_extract_raises(self):
    ''' Test extracting a non-existing tag, ensuring that the original isn't changed. '''

    markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
    extr_soup = BeautifulSoup(markup, "html.parser")
    extr_null_soup = extr_soup.extract()

    self.assertEqual(extr_soup, extr_null_soup)

  def test_extract_with_arg(self):
    ''' Test extracting by calling extract() with a random argument. '''

    markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
    extr_arg_soup = BeautifulSoup(markup, "html.parser")
    with self.assertRaises(TypeError):
            extr_arg_soup_extracted = extr_arg_soup.extract("arg")

  def test_decompose(self):
    ''' Test the decompose function by removing and destroying a tag. '''

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

    soup = BeautifulSoup(markup, "html.parser")
    dec_soup = BeautifulSoup(markup_decomposed, "html.parser")

    soup.i.decompose()

    self.assertEqual(soup, dec_soup)

  def test_decompose_empty(self):
    ''' Test the decompose function by removing an empty tag. '''

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

    soup = BeautifulSoup(markup, "html.parser")
    dec_soup = BeautifulSoup(markup_decomposed, "html.parser")

    soup.i.decompose()

    self.assertEqual(soup, dec_soup)

  def test_decmpose_arg(self):
    ''' Test the decompose function by calling decompose with arg. '''

    markup = '<a href="http://example.com/">I linked to <i> test </i></a>'
    markup_decomposed = '<a href="http://example.com/">I linked to </a>'
    dec_soup = BeautifulSoup(markup, "html.parser")

    with self.assertRaises(TypeError):
            dec_soup.i.decompose("test_arg")

  def test_replace_with(self):
    ''' Test the replace with function '''

    markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
    rep_markup = '<a href="http://example.com/">I linked to <b>new_example.com</b></a>'

    soup = BeautifulSoup(markup, "html.parser")
    rep_soup = BeautifulSoup(rep_markup, "html.parser")

    new_tag = soup.new_tag("b")
    new_tag.string = "new_example.com"

    soup.a.i.replace_with(new_tag)

    self.assertEqual(soup, rep_soup)

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

    markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
    rep_markup = '<a href="http://example.com/">I linked to testText</a>'

    soup = BeautifulSoup(markup, "html.parser")
    rep_soup = BeautifulSoup(rep_markup, "html.parser")

    soup.i.replace_with("testText")

    self.assertEqual(soup, rep_soup)
    self.assertEqual(str(soup), str(rep_soup))

    #soup.a.contents <--- this gives a list containing 2 elements, one for each string.

  def test_replace_with_no_tag(self):
    ''' Test replace_with() called without argument. '''

    markup = '<a href="http://example.com/"I linked to <i>example.com</i></a>'
    no_soup = BeautifulSoup(markup, "html.parser")
    a_tag = no_soup.a

    with self.assertRaises(AttributeError):
            a_tag.i.replace_with()

  def test_wrap(self):
    ''' Test wrap() by wrapping a b tag around an a tag '''

    markup = '<a>Text to be wrapped</a>'
    soup = BeautifulSoup(markup, "html.parser")
    wr_markup = '<b><a>Text to be wrapped</a></b>'
    wr_soup = BeautifulSoup(wr_markup, "html.parser")

    tag = soup.new_tag("b")

    soup.a.wrap(tag)

    self.assertEqual(soup, wr_soup)

  def test_wrap_with_string(self):
    ''' Test wrap() by wrapping a b tag around an a tag, where b contains a string'''

    markup = '<a>Text to be wrapped</a>'

    soup = BeautifulSoup(markup, "html.parser")

    wr_markup = '<b>string<a>Text to be wrapped</a></b>'
    wr_soup = BeautifulSoup(wr_markup, "html.parser")

    tag = soup.new_tag("b")
    tag.string = "string"

    soup.a.wrap(tag)

    self.assertEqual(soup, wr_soup)

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

    markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
    soup = BeautifulSoup(markup, "html.parser")

    unwrapped_markup = '<a href="http://example.com/">I linked to example.com</a>'
    unwrapped_soup = BeautifulSoup(unwrapped_markup, "html.parser")

    soup.a.i.unwrap()

    #The soups are not equal, but as strings they are equal
    self.assertEqual(str(soup), str(unwrapped_soup))
    self.assertEqual(soup, unwrapped_soup)

    #soup.a.string <-- This is None at this point
    #soup.a.contents <--- this gives a list containing 2 elements, one for each string.

  def test_unwrap_with_arg(self):
    ''' Test unwrap() with arg. '''

    markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
    unwrap_soup = BeautifulSoup(markup, "html.parser")
    a_tag = unwrap_soup.a

    with self.assertRaises(TypeError):
            a_tag.i.unwrap("a")


  def test_unwrap_no_tag(self):
    ''' Test unwrap() on non-exisiting tag. '''

    markup = 'htadsasdtp/example.codssdd'
    unwrap_soup = BeautifulSoup(markup, "html.parser")

    with self.assertRaises(ValueError):
            unwrap_soup.unwrap()


  def test_wrap_unwrap(self):
    ''' Test wrapping and then unwrapping '''

    markup = '<a>Text to be wrapped</a>'
    soup = BeautifulSoup(markup, "html.parser")
    soup_2 = BeautifulSoup(markup, "html.parser")

    tag = soup.new_tag("b")

    soup.a.wrap(tag)
    soup.b.unwrap()

    self.assertEqual(soup, soup_2)

  def test_no_name_tag(self):
    ''' test returning a name of a tag with no name'''

    self.assertEqual(self.no_name_tag.name, "")



if __name__ == '__main__':
  unittest.main()
```

The file **unittest_cov_tests.py**

```
## File for white box testing of the BS4 web scraping library.
## This code provides full branch coverage of the functions find and find_all
## as well as the internal function _find_all


## Erik Bertse
## Sara Gustavsson
## Moa Marklund
## Henrik Thorsell

## Software Testing, 5c
## Autumn 2017
## Uppsala University


import unittest
import itertools
from bs4 import BeautifulSoup
from bs4 import SoupStrainer


## WARNING ## This is a badly written iterator, specifically for breaking _find_all ###
## DONT USE THIS CODE IF YOU NEED AN ITERATOR
class EvilIter():
  def __init__(self,breaker):
    self.current = breaker

  def __iter__(self):
    return self

  def next(self):
    if self.current == 0:
      raise StopIteration
    else:
      self.current = self.current - 1
      return self

  def  __len__(self):
    return 0


class TestBS(unittest.TestCase):

  def test_soup_branch(self):

    self.a_simple_soup = BeautifulSoup(
    '''
    <html>
      <head>
      </head>
    <body>
      <p>
        <a href="http://example.com"><b>A bold anchor tag</b></a>
      </p>
    </body>
    </html>
                ''', "html.parser")

    self.tiny_soup = BeautifulSoup(
    '''
    <html></html>
                ''', "html.parser")

    a_tag = self.a_simple_soup.a

    #Here we call find directly
    tag = self.a_simple_soup.find("a")
    self.assertEqual(tag, a_tag)

    #Here we call find all directly
    tag = self.a_simple_soup.find_all("a")
    self.assertEqual(tag, [a_tag])

    #Here we call find all directly, with no argument
    html = self.a_simple_soup.html
    head = self.a_simple_soup.head
    body = self.a_simple_soup.body
    p = self.a_simple_soup.p
    a = self.a_simple_soup.a
    b = self.a_simple_soup.b

    tag = self.a_simple_soup.find_all()
    self.assertEqual(tag, [html, head, body, p, a, b])

    #here kwarg contains "string", and there is no text argument.
    tag = self.a_simple_soup.find_all(string = "a")
    self.assertEqual(tag, [])

    #here we define a custom SoupStrainer
    strainer = SoupStrainer("a")
    tag = self.a_simple_soup.find_all(strainer)
    self.assertEqual(tag, [a_tag])

    #here we find with limit
    tag = self.a_simple_soup.find_all("a", limit = 1)
    self.assertEqual(tag, [a_tag])

    #Putting recursive to False makes BS only look one level down from the
    #tag from which the search started.
    tag = self.tiny_soup.find_all("a", recursive = False)
    self.assertEqual(tag, [])

    #This is far outside of normal BS usage
    iterator = EvilIter(3)
    tag = self.tiny_soup._find_all("a", None, None, None, recursive = False, generator = iterator)
    self.assertEqual(tag, [])

if __name__ == ’__main__’:
  unittest.main()
```