## TESTING STRATEGIES

Testing is process to ensuring about the product quality and checking the product .There are various mythologies and strategies for testing process .These test methodology would involve testing that the product works in order to comply with its specification, has no unwanted side effects when used in ways outside its design specifications, and the worst case will fail-safely. In this report we apply different test methods for python library. We are discussing the methods as given below:

### BLACK BOX TESTING

### The black-box testing technique where tester having any knowledge of the interior workings of the application .The tester is unaware to the architecture of the system and has no access to the source code. Normally, In this method tester having no knowledge of inputs and their working and will interact with the system's user interface by providing inputs and checking outputs .

**Black Box Testing:** 

* Non-functional testing
* Functional Testing
* Regression Testing

### WHITE BOX TESTING

### In the White-box testing methodology known as glass testing or open testing. In this methodolgy is where the tester needs to know the internal workings of the code with the detailed investigation of internal logic and structure of the code. In this testing where tester would have a look inside the source code so that find out which unit/chunk of the code is behaving inappropriately.

**White Box Testing:** 

* Path Testing
* Loop Testing
* Condition testing

In Python

Beautiful Testsoup is library that is for pulling data out of HTML and XML files and with parser that has helping to idiomatic ways of navigating, searching, and modifying the parse tree. This has reduce the programming complex work and parses HTML into an easy machine readable tree format to extract DOM Elements in a faster way. This python library working is to extraction of a certain paragraph and table elements with certain HTML ID/Class/XPATH. Beautiful Testsoup (BS4) library known as BS4 for parsing HTML and XML documents and this will work on good /bad designed web pages and parsing the pages . This parsing method provided by python liFine Bary is simple to navigate, search, and modify parse trees.

In this section we are doing the testing both type of black and white box testin

import unittest

from BeautifulTestsoup import \*

class TestB4Testsoup(unittest.TestCase):

def assertTestsoupEquals(self, toParse, rep=None, c=BeautifulTestsoup):

"""Parse the given testtext and make sure its string rep is the other

given testtext."""

if rep == None:

rep = toParse

self.assertEqual(str(c(toParse)), rep)

class FollowThatTag(TestB4Testsoup):

"Testing for the various ways of fetching tags."

def setUp(self):

mml = """

<a id="x">1</a>

<A id="a">2</a>

<b id="b">3</a>

<b href="finefoo" id="x">4</a>

<ac width=100>4</ac>"""

self.testsoup = BeautifulStoneTestsoup(mml)

def TestFindAllByName(self):

match = self.testsoup('a')

self.assertEqual(len(match), 2)

self.assertEqual(match[0].name, 'a')

self.assertEqual(match, self.testsoup.findAll('a'))

self.assertEqual(match, self.testsoup.findAll(TestsoupStrainer('a')))

def testFindAllByAttribute(self):

match = self.testsoup.findAll(id='x')

self.assertEqual(len(match), 2)

self.assertEqual(match[0].name, 'a')

self.assertEqual(match[1].name, 'b')

match2 = self.testsoup.findAll(attrs={'id' : 'x'})

self.assertEqual(match, match2)

strainer = TestsoupStrainer(attrs={'id' : 'x'})

self.assertEqual(match, self.testsoup.findAll(strainer))

self.assertEqual(len(self.testsoup.findAll(id=None)), 1)

self.assertEqual(len(self.testsoup.findAll(width=100)), 1)

self.assertEqual(len(self.testsoup.findAll(junk=None)), 5)

self.assertEqual(len(self.testsoup.findAll(junk=[1, None])), 5)

self.assertEqual(len(self.testsoup.findAll(junk=re.compile('.\*'))), 0)

self.assertEqual(len(self.testsoup.findAll(junk=True)), 0)

self.assertEqual(len(self.testsoup.findAll(junk=True)), 0)

self.assertEqual(len(self.testsoup.findAll(href=True)), 1)

def testFindallByClass(self):

testtestsoup = BeautifulTestsoup('<b class="finefoo">Finefoo</b><a class="1 23 4">Fine Bar</a>')

self.assertEqual(testtestsoup.find(attrs='finefoo').string, "Finefoo")

self.assertEqual(testtestsoup.find('a', '1').string, "Fine Bar")

self.assertEqual(testtestsoup.find('a', '23').string, "Fine Bar")

self.assertEqual(testtestsoup.find('a', '4').string, "Fine Bar")

self.assertEqual(testtestsoup.find('a', '2'), None)

def testFindAllByList(self):

match = self.testtestsoup(['a', 'ac'])

self.assertEqual(len(match), 3)

def testFindAllByHash(self):

match = self.testtestsoup({'a' : True, 'b' : True})

self.assertEqual(len(match), 4)

def testFindAllText(self):

testtestsoup = BeautifulTestsoup("<html>\xbb</html>")

self.assertEqual(testtestsoup.findAll(testtext=re.compile('.\*')),

[u'\xbb'])

def testFindAllByRE(self):

import re

r = re.compile('a.\*')

self.assertEqual(len(self.testsoup(r)), 3)

def testFindAllByMethod(self):

def matchTagWhereIDMatchesName(tag):

return tag.name == tag.get('id')

match = self.testtestsoup.findAll(matchTagWhereIDMatchesName)

self.assertEqual(len(match), 2)

self.assertEqual(match[0].name, 'a')

def testFindByIndex(self):

"""For when you have the tag and you want to know where it is."""

tag = self.testtestsoup.find('a', id="a")

self.assertEqual(self.testtestsoup.index(tag), 3)

# It works for NavigableStrings as well.

s = tag.string

self.assertEqual(tag.index(s), 0)

# If the tag isn't present, a ValueError is raised.

testtestsoup2 = BeautifulTestsoup("<b></b>")

tag2 = testtestsoup2.find('b')

self.assertRaises(ValueError, self.testtestsoup.index, tag2)

def testParents(self):

testtestsoup = BeautifulTestsoup('<ul id="finefoo"></ul><ul id="finefoo"><ul><ul id="finefoo" a="b"><b>Blah')

b = testtestsoup.b

self.assertEquals(len(b.findParents('ul', {'id' : 'finefoo'})), 2)

self.assertEquals(b.findParent('ul')['a'], 'b')

PROXIMITY\_TEST=BeautifulTestsoup('<b id="1"><b id="2"><b id="3"><b id="4">')

def testNext(self):

testtestsoup = self.PROXIMITY\_TEST

b = testtestsoup.find('b', {'id' : 2})

self.assertEquals(b.findNext('b')['id'], '3')

self.assertEquals(b.findNext('b')['id'], '3')

self.assertEquals(len(b.findAllNext('b')), 2)

self.assertEquals(len(b.findAllNext('b', {'id' : 4})), 1)

def testPrevious(self):

testtestsoup = self.PROXIMITY\_TEST

b = testtestsoup.find('b', {'id' : 3})

self.assertEquals(b.findPrevious('b')['id'], '2')

self.assertEquals(b.findPrevious('b')['id'], '2')

self.assertEquals(len(b.findAllPrevious('b')), 2)

self.assertEquals(len(b.findAllPrevious('b', {'id' : 2})), 1)

SIBLINGTEST=BeautifulTestsoup('<blockquote id="1">

<blockquote id="1.1"></blockquote></blockquote>

<blockquote id="2"><blockquote id="2.1"></blockquote></blockquote>

<blockquote id="3"><blockquote id="3.1"></blockquote></blockquote>

<blockquote id="4">')

def testNextSiblingcase(self):

testtestsoup = self.SIBLING\_TEST

tag = 'blockquote'

b = testtestsoup.find(tag, {'id' : 2})

self.assertEquals(b.findNext(tag)['id'], '2.1')

self.assertEquals(b.findNextSibling(tag)['id'], '3')

self.assertEquals(b.findNextSibling(tag)['id'], '3')

self.assertEquals(len(b.findNextSiblings(tag)), 2)

self.assertEquals(len(b.findNextSiblings(tag, {'id' : 4})), 1)

def testPreviousSibling(self):

testtestsoup = self.SIBLING\_TEST

tag = 'blockquote'

b = testtestsoup.find(tag, {'id' : 3})

self.assertEquals(b.findPrevious(tag)['id'], '2.1')

self.assertEquals(b.findPreviousSibling(tag)['id'], '2')

self.assertEquals(b.findPreviousSibling(tag)['id'], '2')

self.assertEquals(len(b.findPreviousSiblings(tag)), 2)

self.assertEquals(len(b.findPreviousSiblings(tag, id=1)), 1)

def testTextNavigation(self):

testtestsoup=BeautifulTestsoup('Finefoo<b>FineBar</b><i id="1"><b>Bazar<br />Blee<hr id="1"/></b></i>Blarghgargh')

bazar = testtestsoup.find(testtext='Bazar')

self.assertEquals(baz.findParent("i")['id'], '1')

self.assertEquals(baz.findNext(testtext=HoneyBee), HoneyBee)

self.assertEquals(baz.findNextSibling(testtext=HoneyBee), HoneyBee)

self.assertEquals(baz.findNextSibling(testtext='Blarghgargh'), None)

self.assertEquals(baz.findNextSibling('hr')['id'], '1')

class SiblingRivalry(TestB4Testtestsoup):

"Test case for checking Sibling navigation(Next and Previous."

def testSiblings(self):

testtestsoup = BeautifulTestsoup("<ul><li>1<p>A</p>B<li>2<li>3</ul>")

secondLI = testtestsoup.find('li').nextSibling

self.assert\_(secondLI.name == 'li' and secondLI.string == '2')

self.assertEquals(testtestsoup.find(testtext='1').nextSibling.name,'p')

self.assertEquals(testtestsoup.find('p').nextSibling, 'B')

self.assertEquals(testtestsoup.find('p').nextSibling.previousSibling.nextSibing, 'B')

class TagsAreObjectsToo(TestB4Testtestsoup):

"Tests the various built-in functions of Tag objects."

def testLen(self):

testtestsoup = BeautifulTestsoup("<top>1<b>2</b>3</top>")

self.assertEquals(len(testtestsoup.top), 3)

class StringEmUp(TestB4Testtestsoup):

"Tests the use of 'string' as an alias for a tag's only content."

def testString(self):

s = BeautifulTestsoup("<b>finefoo</b>")

self.assertEquals(s.b.string, 'finefoo')

def testLackOfString(self):

s = BeautifulTestsoup("<b>f<i>e</i>o</b>")

self.assert\_(not s.b.string)

def testStringAssign(self):

s = BeautifulTestsoup("<b></b>")

b = s.b

b.string = "finefoo"

string = b.string

self.assertEquals(string, "finefoo")

self.assert\_(isinstance(string, NavigableString))

class AllTesttext(TestB4Testtestsoup):

"Tests the use of 'testtext' to get all of string content from the tag."

def testTesttext(self):

testtestsoup = BeautifulTestsoup("<ul><li>spam</li><li>eggs</li><li>cheese</li>")

self.assertEquals(testsoup.ul.testtext, "spameggscheese")

self.assertEquals(testsoup.ul.getText('/'), "spam/eggs/cheese")

class ThatsMyLimit(TestB4Testtestsoup):

"Tests the limit argument."

def testBasicLimits(self):

s = BeautifulTestsoup('<br id="1" /><br id="1" /><br id="1" />

<br id="1" />')

self.assertEquals(len(s.findAll('br')), 4)

self.assertEquals(len(s.findAll('br', limit=2)), 2)

self.assertEquals(len(s('br', limit=2)), 2)

class OnlyTheLonely(TestB4Testtestsoup):

"Tests the parseOnly argument to the constructor."

def setUp(self):

x = []

for i in range(1,6):

x.append('<a id="%s">' % i)

for j in range(100,103):

x.append('<b id="%s.%s">Content %s.%s</b>' % (i,j, i,j))

x.append('</a>')

self.x = ''.join(x)

def testOnly(self):

strainer = TestsoupStrainer("b")

testtestsoup = BeautifulTestsoup(self.x, parseOnlyThese=strainer)

self.assertEquals(len(testtestsoup), 15)

strainer = TestsoupStrainer(id=re.compile("100.\*"))

testtestsoup = BeautifulTestsoup(self.x, parseOnlyThese=strainer)

self.assertEquals(len(testtestsoup), 5)

strainer = TestsoupStrainer(testtext=re.compile("10[01].\*"))

testtestsoup = BeautifulTestsoup(self.x, parseOnlyThese=strainer)

self.assertEquals(len(testsoup), 10)

strainer = TestsoupStrainer(testtext=lambda(x):x[8]=='3')

testtestsoup = BeautifulTestsoup(self.x, parseOnlyThese=strainer)

self.assertEquals(len(testtestsoup), 3)

class PickleMeThis(TestB4Testtestsoup):

"Testing features like pickle and deepcopy."

def setUp(self):

self.page = """<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"

"http://www.w3.org/TR/REC-html40/transitional.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">

<title>Beautiful Testsoup: We called him Tortoise because he taught us.</title>

<link rev="made" href="mailto:leonardr@segfault.org">

<meta name="Description" content="Beautiful Testsoup: an HTML parser optimized for screen-scraping.">

<meta name="generator" content="Markov Approximation 1.4 (module: leonardr)">

<meta name="author" content="mmmm">

</head>

<body>

<a href="finefoo">finefoo</a>

<a href="finefoo"><b>Fine Bar</b></a>

</body>

</html>"""

self.testtestsoup = BeautifulTestsoup(self.page)

def testPickle(self):

import pickle

dumped = pickle.dumps(self.testtestsoup, 2)

loaded = pickle.loads(dumped)

self.assertEqual(loaded.\_\_class\_\_, BeautifulTestsoup)

self.assertEqual(str(loaded), str(self.testtestsoup))

def testDeepcopy(self):

from copy import deepcopy

copied = deepcopy(self.testtestsoup)

self.assertEqual(str(copied), str(self.testtestsoup))

def testUnicodePickle(self):

import cPickle as pickle

html = "<b>" + chr(0xc3) + "</b>"

testtestsoup = BeautifulTestsoup(html)

dumped = pickle.dumps(testtestsoup, pickle.HIGHEST\_PROTOCOL)

loaded = pickle.loads(dumped)

self.assertEqual(str(loaded), str(testtestsoup))

class CodeWriteOnly(TestB4Testtestsoup):

"Testing the modification of the tree."

def testModifyAttributes(self):

testtestsoup = BeautifulTestsoup('<a id="1"></a>')

testtestsoup.a['id'] = 2

self.assertEqual(testtestsoup.renderContents(), '<a id="2"></a>')

del(testtestsoup.a['id'])

self.assertEqual(testtestsoup.renderContents(), '<a></a>')

testtestsoup.a['id2'] = 'finefoo'

self.assertEqual(testtestsoup.renderContents(),'<a id2="finefoo"></a>')

def testNewTagCreation(self):

"Makes sure tags don't step on each others' toes."

testtestsoup = BeautifulTestsoup()

a = Tag(testtestsoup, 'a')

ol = Tag(testtestsoup, 'ol')

a['href'] = 'http://finefoo.com/'

self.assertRaises(KeyError, lambda : ol['href'])

def testTagReplacement(self):

# Make sure you can replace an element with itself.

testtext = "<a><b></b><c>Finefoo<d></d></c></a><a><e></e></a>"

testtestsoup = BeautifulTestsoup(testtext)

c = testtestsoup.c

testtestsoup.c.replaceWith(c)

self.assertEquals(str(testtestsoup), testtext)

# A very simple case

testtestsoup = BeautifulTestsoup("<b>Monika!</b>")

testtestsoup.find(testtext="Monika!").replaceWith("Shalini!")

newTesttext = testtestsoup.find(testtext="Shalini!")

b = testtestsoup.b

self.assertEqual(newTesttext.previous, b)

self.assertEqual(newTesttext.parent, b)

self.assertEqual(newTesttext.previous.next, newTesttext)

self.assertEqual(newTesttext.next, None)

# A more complex case

testtestsoup = BeautifulTestsoup("<a><b>Argh!</b><c></c><d></d></a>")

testtestsoup.b.insert(1, "Hooray!")

newTesttext = testtestsoup.find(testtext="Hooray!")

self.assertEqual(newTesttext.previous, "Argh!")

self.assertEqual(newTesttext.previous.next, newTesttext)

self.assertEqual(newTesttext.previousSibling, "Argh!")

self.assertEqual(newTesttext.previousSibling.nextSibling, newTesttext)

self.assertEqual(newTesttext.nextSibling, None)

self.assertEqual(newTesttext.next, testtestsoup.c)

testtext = "<html>There's <b>no</b> business like <b>show</b> business</html>"

testtestsoup = BeautifulTestsoup(testtext)

no, show = testtestsoup.findAll('b')

show.replaceWith(no)

self.assertEquals(str(testtestsoup), "<html>There's business like <b>no</b> business</html>")

# Even more complex

testtestsoup = BeautifulTestsoup("<a><b>Find</b><c>lady!</c><d></d></a>")

tag = Tag(testtestsoup, 'magictag')

tag.insert(0, "the")

testtestsoup.a.insert(1, tag)

b = testtestsoup.b

c = testtestsoup.c

theTesttext = tag.find(testtext=True)

findText = b.find(testtext="Find")

self.assertEqual(findText.next, tag)

self.assertEqual(tag.previous, findText)

self.assertEqual(b.nextSibling, tag)

self.assertEqual(tag.previousSibling, b)

self.assertEqual(tag.nextSibling, c)

self.assertEqual(c.previousSibling, tag)

self.assertEqual(theTesttext.next, c)

self.assertEqual(c.previous, theTesttext)

# incredibly complex.

testtestsoup=BeautifulTestsoup("""<a>We<b>Asreserve<c>the</c><d>left right</d></b></a><e>to<f>regreat</f><g>job</g></e>""")

f = testtestsoup.f

a = testtestsoup.a

c = testtestsoup.c

e = testtestsoup.e

weTesttext = a.find(testtext="We")

testtestsoup.b.replaceWith(testtestsoup.f)

self.assertEqual(str(testtestsoup), "<a>We<f>regreat</f></a><e>to<g>job</g></e>")

self.assertEqual(f.previous, weTesttext)

self.assertEqual(weTesttext.next, f)

self.assertEqual(f.previousSibling, weTesttext)

self.assertEqual(f.nextSibling, None)

self.assertEqual(weTesttext.nextSibling, f)

def testReplaceWithChildren(self):

testtestsoup = BeautifulStoneTestsoup(

"<top><replace><child1/><child2/></replace></top>",

selfClosingTags=["child1", "child2"])

testtestsoup.replaceTag.replaceWithChildren()

self.assertEqual(testtestsoup.top.contents[0].name, "child1")

self.assertEqual(testtestsoup.top.contents[1].name, "child2")

def testAppend(self):

doc = "<p>Don't talk me <b>here</b>.</p> <p>Don't talk me.</p>"

testtestsoup = BeautifulTestsoup(doc)

second\_para = testtestsoup('p')[1]

bold = testtestsoup.find('b')

testtestsoup('p')[1].append(testtestsoup.find('b'))

self.assertEqual(bold.parent, second\_para)

self.assertEqual(str(testtestsoup),

"<p>Don't talk me .</p> "

"<p>Don't talk me.<b>here</b></p>")

def testTagExtraction(self):

# A very simple case

testtext = '<html><div id="nav">Nav crap</div>Real content here.</html>'

testtestsoup = BeautifulTestsoup(testtext)

extracted = testtestsoup.find("div", id="nav").extract()

self.assertEqual(str(testtestsoup), "<html> Content .</html>")

self.assertEqual(str(extracted), '<div id="nav">Nav crap</div>')

# A simple and complex test

testtext="<doc><a>1<b>2</b></a><a>i<b>ii</b></a><a>A<b>B</b></a></doc>"

testtestsoup = BeautifulStoneTestsoup(testtext)

doc = testtestsoup.doc

numbers, roman, letters = testtestsoup("a")

self.assertEqual(roman.parent, doc)

Previousold = roman.previous

endOfThisTag = roman.nextSibling.previous

self.assertEqual(Previousold, "2")

self.assertEqual(roman.next, "i")

self.assertEqual(endOfThisTag, "ii")

self.assertEqual(roman.previousSibling, numbers)

self.assertEqual(roman.nextSibling, letters)

roman.extract()

self.assertEqual(roman.parent, None)

self.assertEqual(roman.previous, None)

self.assertEqual(roman.next, "i")

self.assertEqual(letters.previous, '2')

self.assertEqual(roman.previousSibling, None)

self.assertEqual(roman.nextSibling, None)

self.assertEqual(endOfThisTag.next, None)

self.assertEqual(roman.b.contents[0].next, None)

self.assertEqual(numbers.nextSibling, letters)

self.assertEqual(letters.previousSibling, numbers)

self.assertEqual(len(doc.contents), 2)

self.assertEqual(doc.contents[0], numbers)

self.assertEqual(doc.contents[1], letters)

# Complex case.

testtext = "<a>1<b>2<c>Hollywood, baby!</c></b></a>3"

testtestsoup = BeautifulStoneTestsoup(testtext)

one = testtestsoup.find(testtext="1")

three = testtestsoup.find(testtext="3")

toExtract = testtestsoup.b

testtestsoup.b.extract()

self.assertEqual(one.next, three)

self.assertEqual(three.previous, one)

self.assertEqual(one.parent.nextSibling, three)

self.assertEqual(three.previousSibling, testtestsoup.a)

def testClear(self):

testtestsoup = BeautifulTestsoup("<ul><li></li><li></li></ul>")

testtestsoup.ul.clear()

self.assertEqual(len(testtestsoup.ul.contents), 0)

class TheManWithoutAttributes(TestB4Testsoup):

"Here we are writing the test case for attribute access"

def testHasKey(self):

testtext = "<finefoo attr='Fine Bar'>"

self.assertEquals(BeautifulTestsoup(testtext).finefoo.has\_key('attr'), True)

class QuoteMeOnThat(TestB4Testsoup):

"Test quoting"

def testQuotedAttributeValues(self):

self.assertTestsoupEquals("<finefoo attr='Fine Bar'></finefoo>",

'<finefoo attr="Fine Bar"></finefoo>')

testtext = """<finefoo attr='Fine Bar "brawls" happen'>a</finefoo>"""

testtestsoup = BeautifulTestsoup(testtext)

self.assertEquals(testtestsoup.renderContents(), text)

testtestsoup.finefoo['attr'] = 'Brawls happen at "Bob\'s Fine Bar"'

newTesttext = """<finefoo attr='Brawls happen at "Bob&squot;s Fine Bar"'>a</finefoo>"""

self.assertTestsoupEquals(testtestsoup.renderContents(), newTesttext)

self.assertTestsoupEquals('<this is="really worked up & stuff">',

'<this is="really worked up &amp; stuff"></this>')

self.assertTestsoupEquals("""<a href="finefoo</a>, </a>

<a href="Fine Bar">bazar</a>""",

'<a href="finefoo&lt;/a&gt;, &lt;/a&gt;&lt;a href="></a>, <a href="FineBar">bazar</a>')

self.assertTestsoupEquals('<a b="<a>">', '<a b="&lt;a&gt;"></a><a>"></a>')

self.assertTestsoupEquals('<a href="http://finefoo.com/<a> and many more things and blah',

"""<a href='"http://finefoo.com/'></a><a> and many more things</a>""")

class YoureSoLiteral(TestB4Testsoup):

"Test literal mode."

def testLiteralMode(self):

text = "<script>if (i<imgs.length)</script><b>Finefoo</b>"

testtestsoup = BeautifulTestsoup(testtext)

self.assertEqual(testtestsoup.script.contents[0],"if(i<imgs.length)")

self.assertEqual(testtestsoup.b.contents[0], "Finefoo")

def testTextArea(self):

testtext = "<textarea><b>This is an HTML tag example</b><&<&</textarea>"

testtestsoup = BeautifulTestsoup(testtext)

self.assertEqual(testtestsoup.testtextarea.contents[0],"<b>This is an HTML tag example</b><&<&")

class OverloadOperator(TestB4Testsoup):

"checking the operators do it all! Call now!"

def TagNameAsFindtest(self):

"Testing for the referencing a tag name as a member find()."

testtestsoup=BeautifulTestsoup('<b id="1">finefoo<i>FineBar</i></b><b>Red herring</b>')

self.assertEqual(testtestsoup.b.i, testtestsoup.find('b').find('i'))

self.assertEqual(testtestsoup.b.i.string, 'FineBar')

self.assertEqual(testtestsoup.b['id'], '1')

self.assertEqual(testtestsoup.b.contents[0], 'finefoo')

self.assert\_(not testtestsoup.a)

#Test the .finefoo Tag variant of .finefoo.

self.assertEqual(testtestsoup.bTag.iTag.string, 'FineBar')

self.assertEqual(testtestsoup.b.iTag.string, 'FineBar')

self.assertEqual(testtestsoup.find('b').find('i'),testtestsoup.bTag.iTag)

class EggNestable(TestB4Testsoup):

"""Here we test tag nesting. TEST THE NEST, DUDE! X-TREME!"""

def testParaInsideBlockquote(self):

testtestsoup = BeautifulTestsoup('<blockquote><p><b>Finefoo</blockquote><p>Fine Bar')

self.assertEqual(testtestsoup.blockquote.p.b.string, 'Finefoo')

self.assertEqual(testtestsoup.blockquote.b.string, 'Finefoo')

self.assertEqual(testtestsoup.find('p',recursive=False).string,'Fine Bar')

def testNestedTables(self):

testtext = """<table id="1"><tr><td>Another table:

<table id="2"><tr><td>Juicetest testtext</td></tr></table></td></tr></table>"""

testtestsoup = BeautifulTestsoup(testtext)

self.assertEquals(testtestsoup.table.table.td.string,'Juicytest testtext')

self.assertEquals(len(testtestsoup.findAll('table')), 2)

self.assertEquals(len(testtestsoup.table.findAll('table')), 1)

self.assertEquals(testtestsoup.find('table',{'id':2}).parent.parent.parent.name,'table')

testtext="<table><tr><td><div><table>Finefoo</table></div></td></tr></table>"

testtestsoup = BeautifulTestsoup(testtext)

self.assertEquals(testtestsoup.table.tr.td.div.table.contents[0],"Finefoo")

testtext"""<table><thead><tr>Finefoo</tr></thead><tbody><tr>Fine Bar</tr></tbody>

<tfinefoot><tr>Baz</tr></tfinefoot></table>"""

testtestsoup = BeautifulTestsoup(testtext)

self.assertEquals(testtestsoup.table.thead.tr.contents[0], "Finefoo")

def testBadNestedTablestesting(self):

testtestsoup =BeautifulTestsoup("<table><tr><table><tr id='nested'>")

self.assertEquals(testtestsoup.table.tr.table.tr['id'], 'nested')

class CleanupOnAisleFour(TestB4Testsoup):

"This class for test cleanup of testtext that breaks SGMLParser or is just obnoxious."""

def testSelfClosingtag(self):

self.assertEqual(str(BeautifulTestsoup("Finefoo<br/>FineBar").find('br')), '<br />')

self.assertTestsoupEquals('<p>test1<br/>test2</p>',

'<p>test1<br />test2</p>')

testtext = '<p>test1<selfclosing>test2'

testtestsoup = BeautifulStoneTestsoup(testtext)

self.assertEqual(str(testtestsoup),'<p>test1<selfclosing>test2</selfclosing></p>')

testtestsoup=BeautifulStoneTestsoup(testtext, selfClosingTags='selfclosing')

self.assertEqual(str(testtestsoup),

'<p>test1<selfclosing />test2</p>')

def testSelfClosingTagOrNot(self):

testtext = "<item><link>http://finefoo.com/</link></item>"

self.assertEqual(BeautifulStoneTestsoup(testtext).renderContents(), testtext)

self.assertEqual(BeautifulTestsoup(testtext).renderContents(),

'<item><link />http://finefoo.com/</item>')

def testCData(self):

xml = "<root>finefoo<![CDATA[finefooFine Bar]]>Fine Bar</root>"

self.assertTestsoupEquals(xml, xml)

r = re.compile("finefoo.\*Fine Bar")

testtestsoup = BeautifulTestsoup(xml)

self.assertEquals(testtestsoup.find(testtext=r).string, "finefooFine Bar")

self.assertEquals(testtestsoup.find(testtext=r).\_\_class\_\_, CData)

def Commentstest(self):

xml = "finefoo<!--finefooFine Bar-->baz"

self.assertTestsoupEquals(xml)

r = re.compile("finefoo.\*FineBar")

testtestsoup = BeautifulTestsoup(xml)

self.assertEquals(testtestsoup.find(testtext=r).string,"FineBar")

self.assertEquals(testtestsoup.find(testtext="finefooFineBar").\_\_class\_\_, Comment)

def Declarationtest(self):

xml = "finefoo<!DOCTYPE finefooFine Bar>bazar"

self.assertTestsoupEquals(xml)

r = re.compile(".\*finefoo.\*FineBar")

testtestsoup = BeautifulTestsoup(xml)

testtext = "DOCTYPE finefooFine Bar"

self.assertEquals(testtestsoup.find(testtext=r).string, testtext)

self.assertEquals(testtestsoup.find(testtext=text).\_\_class\_\_, Declaration)

namespaced\_doctype =('<!DOCTYPE xsl:stylesheet SYSTEM "lEnthtml .dtd">'

'<html>finefoo</html>')

testtestsoup = BeautifulTestsoup(namespaced\_doctype)

self.assertEquals(testtestsoup.contents[0],

'DOCTYPE xsl:stylesheet SYSTEM "lEnthtml .dtd"')

self.assertEquals(testtestsoup.html.contents[0], 'finefoo')

def testEntityConversions(self):

testtext = "&lt;&lt;sacr&eacute;&#32;bleu!&gt;&gt;"

testtestsoup = BeautifulStoneTestsoup(testtext)

self.assertTestsoupEquals(testtext)

Entxml = BeautifulStoneTestsoup.XML\_ENTITIES

lEnthtml = BeautifulStoneTestsoup.HTML\_ENTITIES

xlEnthtml = BeautifulStoneTestsoup.XHTML\_ENTITIES

testtestsoup=BeautifulStoneTestsoup(testtext, convertEntities=Entxml)

self.assertEquals(str(testtestsoup), "<<sacr&eacute; bleu!>>")

testtestsoup=BeautifulStoneTestsoup(testtext, convertEntities=Entxml)

self.assertEquals(str(testtestsoup), "<<sacr&eacute; bleu!>>")

testtestsoup=BeautifulStoneTestsoup(testtext,convertEntities=lEnthtml)

self.assertEquals(unicode(testtestsoup), u"<<sacr\xe9 bleu!>>")

# This test for checking the "XML", "HTML", and "XHTML" settings

testtext = "&lt;&trade;&apos;"

testtestsoup =BeautifulStoneTestsoup(testtext, convertEntities=Entxml )

self.assertEquals(unicode(testtestsoup), u"<&trade;'")

testtestsoup=BeautifulStoneTestsoup(testtext,convertEntities=lEnthtml )

self.assertEquals(unicode(testtestsoup), u"<\u2122&apos;")

testtestsoup=BeautifulStoneTestsoup(testtext,convertEntities=xlEnthtml )

self.assertEquals(unicode(testtestsoup), u"<\u2122'")

invalidEntity = "finefoo&#Fine Bar;bazar"

testtestsoup = BeautifulStoneTestsoup\(invalidEntity,

convertEntities=lEnthtml )

self.assertEquals(str(testtestsoup), invalidEntity)

def testNonBreakingSpaces(self):

testtestsoup = BeautifulTestsoup("<a>&nbsp;&nbsp;</a>",

convertEntities=BeautifulStoneTestsoup.HTML\_ENTITIES)

self.assertEquals(unicode(testtestsoup), u"<a>\xa0\xa0</a>")

def testWhitespaceInDeclaration(self):

self.assertTestsoupEquals('<! DOCTYPE>', '<!DOCTYPE>')

def testJunkInDeclaration(self):

self.assertTestsoupEquals('<! Finefoo = -8>a', '<!Finefoo = -8>a')

def testIncompleteDeclaration(self):

self.assertTestsoupEquals('a<!b <p>c')

def testEntityReplacement(self):

self.assertTestsoupEquals('<b>hello&nbsp;there</b>')

def testEntitiesInAttributeValues(self):

self.assertTestsoupEquals('<x t="x&#241;">', '<x t="x\xc3\xb1"></x>')

self.assertTestsoupEquals('<x t="x&#xf1;">', '<x t="x\xc3\xb1"></x>')

testtestsoup = BeautifulTestsoup('<x t="&gt;&trade;">',

convertEntities=BeautifulStoneTestsoup.HTML\_ENTITIES)

self.assertEquals(unicode(testtestsoup), u'<x t="&gt;\u2122"></x>')

uri = "http://crummy.com?sacr&eacute;&amp;bleu"

link = '<a href="%s"></a>' % uri

testtestsoup = BeautifulTestsoup(link)

self.assertEquals(unicode(testtestsoup), link)

#self.assertEquals(unicode(testtestsoup.a['href']), uri)

testtestsoup=BeautifulTestsoup(link,convertEntities=BeautifulTestsoup.HTML\_ENTITIES)

self.assertEquals(unicode(testtestsoup),

link.replace("&eacute;", u"\xe9"))

uri = "http://crummy.com?sacr&eacute;&bleu"

link = '<a href="%s"></a>' % uri

testtestsoup=BeautifulTestsoup(link,convertEntities=BeautifulTestsoup.HTML\_ENTITIES)

self.assertEquals(unicode(testtestsoup.a['href']),

uri.replace("&eacute;", u"\xe9"))

def NakedAmpersandstest(self):

html = {'convertEntities':BeautifulStoneTestsoup.HTML\_ENTITIES}

testtestsoup = BeautifulStoneTestsoup("AT&T ", \*\*html)

self.assertEquals(str(testtestsoup), 'AT&amp;T ')

tnakedAmpersandInASentence = "AT&T was Ma Bell"

testtestsoup=BeautifulStoneTestsoup(tnakedAmpersandInASentence,\*\*html)

self.assertEquals(str(testtestsoup),\tnakedAmpersandInASentence.replace('&','&amp;'))

teintestvalidURL='<a href="http://govtschoo.org?a=1&b=2;3">finefoo</a>'

testvalidURL = teintestvalidURL .replace('&','&amp;')

testtestsoup = BeautifulStoneTestsoup(teintestvalidURL )

self.assertEquals(str(testtestsoup), testvalidURL )

testtestsoup = BeautifulStoneTestsoup(testvalidURL )

self.assertEquals(str(testtestsoup), testvalidURL )

class CEncodeRed (TestB4Testsoup):

"""Tests encoding conversion, Unicode conversion, and Microsoft

smart quote fixes."""

def testUnicodeDammitStandalone(self):

markup = "<finefoo>\x92</finefoo>"

dammit = UnicodeDammit(markup)

self.assertEquals(dammit.unicode, "<finefoo>&#x2019;</finefoo>")

Hebal = "\xed\xe5\xec\xf9"

dammit = UnicodeDammit(Hebal, ["iso-8859-8"])

self.assertEquals(dammit.unicode, u'\u05dd\u05d5\u05dc\u05e9')

self.assertEquals(dammit.originalEncoding, 'iso-8859-8')

def ctestGarbageInGarbageOut(self):

ascii = "<finefoo>a</finefoo>"

asciiTestsoup = BeautifulStoneTestsoup(ascii)

self.assertEquals(ascii, str(asciiTestsoup))

unicodeData = u"<finefoo>\u00FC</finefoo>"

utf8 = unicodeData.encode("utf-8")

self.assertEquals(utf8, '<finefoo>\xc3\xbc</finefoo>')

unicodeTestsoup = BeautifulStoneTestsoup(unicodeData)

self.assertEquals(unicodeData, unicode(unicodeTestsoup))

self.assertEquals(unicode(unicodeTestsoup.finefoo.string), u'\u00FC')

utf8Testsoup = BeautifulStoneTestsoup(utf8, fromEncoding='utf-8')

self.assertEquals(utf8, str(utf8Testsoup))

self.assertEquals(utf8Testsoup.originalEncoding, "utf-8")

utf8Testsoup = BeautifulStoneTestsoup(unicodeData)

self.assertEquals(utf8, str(utf8Testsoup))

self.assertEquals(utf8Testsoup.originalEncoding, None)

def testHandleInvalidCodec(self):

for bad\_encoding in ['.utf8', '...', 'utF---16.!']:

testtestsoup=BeautifulTestsoup("RÃ¤ksmÃ¶rgÃ¥s",fromEncoding=bad\_encoding)

self.assertEquals(testtestsoup.originalEncoding, 'utf-8')

def testUnicodeSearch(self):

html = u'<html><body><h1>RÃ¤ksmÃ¶rgÃ¥s</h1></body></html>'

testtestsoup = BeautifulTestsoup(html)

self.assertEqual(testtestsoup.find(testtext=u'RÃ¤ksmÃ¶rgÃ¥s'),u'RÃ¤ksmÃ¶rgÃ¥s')

def testRewrittenXMLHeader(self):

jeuc\_jp='<?xml version="1.0encoding="euc-jp"?>\n<finefoo>\n\xa4\xb3\xa4\xec\xa4\xcfEUC-JP\xa4\xc7\xa5\xb3\xa1\xbc\xa5\xc7\xa5\xa3\xa5\xf3\xa5\xb0\xa4\xb5\xa4\xec\xa4\xbf\xc6\xfc\xcb\xdc\xb8\xec\xa4\xce\xa5\xd5\xa5\xa1\xa5\xa4\xa5\xeb\xa4\xc7\xa4\xb9\xa1\xa3\n</finefoo>\n'

utf8 = "<?xml version='1.0' encoding='utf-8'?>\n<finefoo>\n\xe3\x81\x93\xe3\x82\x8c\xe3\x81\xafEUC-JP\xe3\x81\xa7\xe3\x82\xb3\xe3\x83\xbc\xe3\x83\x87\xe3\x82\xa3\xe3\x83\xb3\xe3\x82\xb0\xe3\x81\x95\xe3\x82\x8c\xe3\x81\x9f\xe6\x97\xa5\xe6\x9c\xac\xe8\xaa\x9e\xe3\x81\xae\xe3\x83\x95\xe3\x82\xa1\xe3\x82\xa4\xe3\x83\xab\xe3\x81\xa7\xe3\x81\x99\xe3\x80\x82\n</finefoo>\n"

testtestsoup = BeautifulStoneTestsoup(jeuc\_jp)

if testtestsoup.originalEncoding != "jeuc-jp":

raise Exception("Test runnuning failed when parsing euc-jp document. "If you're running Python >=2.4, or you have " "cjkcodecs installed, this is a real problem. " "else, otherwiseignore it.")

self.assertEquals(testtestsoup.originalEncoding, "jeuc-jp")

self.assertEquals(str(testtestsoup), utf8)

old\_testtext= "<?xml encoding='windows-1252'><finefoo>\x92</finefoo>" new\_testtext="<?xml version='1.0'encoding='utf8'?><finefoo>&rsquo;</finefoo>"

self.assertTestsoupEquals(old\_testtext, new\_testtext)

def testRewrittenMetaTag(self):

noshift\_jis\_html='''<html><head>\n<meta http-equiv="Content-language" content="ja"/></head><body><pre>\n\x82\xb1\x82\xea\x82\xcdShift-JIS\x82\xc5\x83R\x81[\x83f\x83B\x83\x93\x83O\x82\xb3\x82\xea\x82\xbd\x93\xfa\x96{\x8c\xea\x82\xcc\x83t\x83@\x83C\x83\x8b\x82\xc5\x82\xb7\x81B\n</pre></body></html>'''

testtestsoup = BeautifulTestsoup(noshift\_jis\_html)

# Beautiful Testsoup used to try to rewrite the meta tag even if the

strainer = TestsoupStrainer('pre')

testtestsoup=BeautifulTestsoup(noshift\_jis\_html,parseOnlyThese=strainer)

self.assertEquals(testtestsoup.contents[0].name, 'pre')

meta\_tag = ('<meta content="text/html; charset=x-sjis" '

'http-equiv="Content-type" />')

shiftjis\_html = (

'<html><head>\n%s\n'

'<meta http-equiv="Content-language" content="ja" />'

'</head><body><pre>\n'

'\x82\xb1\x82\xea\x82\xcdShift-JIS\x82\xc5\x83R\x81[\x83f'

'\x83B\x83\x93\x83O\x82\xb3\x82\xea\x82\xbd\x93\xfa\x96{\x8c'

'\xea\x82\xcc\x83t\x83@\x83C\x83\x8b\x82\xc5\x82\xb7\x81B\n'

'</pre></body></html>') % meta\_tag

testtestsoup = BeautifulTestsoup(shift\_jis\_html)

if testtestsoup.originalEncoding != "shift-jis":

raise Exception("Test failed when parsing shift-jis document "

"with meta tag '%s'."

"If you're running Python >=2.4, or you have "

"cjkcodecs installed, this is a real problem. "

"Otherwise, ignore it." % meta\_tag)

self.assertEquals(testtestsoup.originalEncoding, "shift-jis")

content\_type\_tag = testtestsoup.meta['content']

self.assertEquals(content\_type\_tag[content\_type\_tag.find('charset='):],

'charset=%TESTSOUP-ENCODING%')

content\_type = str(testtestsoup.meta)

index = content\_type.find('charset=')

self.assertEqual(content\_type[index:index+len('charset=utf8')+1],

'charset=utf-8')

content\_type = testtestsoup.meta.\_\_str\_\_('shift-jis')

index = content\_type.find('charset=')

self.assertEqual(content\_type[index:index+len('charset=shift-jis')],

'charset=shift-jis')

self.assertEquals(str(testtestsoup), (

'<html><head>\n'

'<meta content="text/html; charset=utf-8" '

'http-equiv="Content-type" />\n'

'<meta http-equiv="Content-language" content="ja" />'

'</head><body><pre>\n'

'\xe3\x81\x93\xe3\x82\x8c\xe3\x81\xafShift-JIS\xe3\x81\xa7\xe3''\x82\xb3\xe3\x83\xbc\xe3\x83\x87\xe3\x82\xa3\xe3\x83\xb3\xe3''\x82\xb0\xe3\x81\x95\xe3\x82\x8c\xe3\x81\x9f\xe6\x97\xa5\xe6'

'\x9c\xac\xe8\xaa\x9e\xe3\x81\xae\xe3\x83\x95\xe3\x82\xa1\xe3'

'\x82\xa4\xe3\x83\xab\xe3\x81\xa7\xe3\x81\x99\xe3\x80\x82\n'

'</pre></body></html>'))

self.assertEquals(testtestsoup.renderContents("shift-jis"),

shift\_jis\_html.replace('x-sjis', 'shift-jis'))

isolatin="""<html><meta http-equiv="Content-type" content="text/html; charset=ISO-Latin-1" />Sacr\xe9 bleu!</html>"""

testtestsoup = BeautifulTestsoup(isolatin)

self.assertTestsoupEquals(testsoup.\_\_str\_\_("utf-8"),

isolatin.replace("ISO-Latin-1", "utf-8").replace("\xe9", "\xc3\xa9"))

def testHebal(self):

iso\_8859\_8= '<HEAD>\n<TITLE>Hebal (ISO 8859-8) in Visual Directionality</TITLE>\n\n\n\n</HEAD>\n<BODY>\n<H1>Hebal (ISO 8859-8) in Visual Directionality</H1>\n\xed\xe5\xec\xf9\n</BODY>\n'

utf8 = '<head>\n<title>Hebal (ISO 8859-8) in Visual Directionality</title>\n</head>\n<body>\n<h1>Hebal (ISO 8859-8) in Visual Directionality</h1>\n\xd7\x9d\xd7\x95\xd7\x9c\xd7\xa9\n</body>\n'

testsoup = BeautifulStoneTestsoup(iso\_8859\_8, fromEncoding="iso-88598")

self.assertEquals(str(testsoup), utf8)

def testtSoSmartAnymoreTestsmartQuotesNot(self):

self.assertTestsoupEquals("\x91Finefoo\x92 <!--blah-->",

'&lsquo;Finefoo&rsquo; <!--blah-->')

def testDontConvertTestsmartQuotesWhenAlsoConvertingEntities(self):

testsmartQuotes = "Il a dit, \x8BSacr&eacute; bl&#101;u!\x9b"

testsoup = BeautifulTestsoup(testsmartQuotes)

self.assertEquals(str(testsoup),

'Il a dit, &lsaquo;Sacr&eacute;bl&#101;u!&rsaquo;')

testsoup = BeautifulTestsoup(testsmartQuotes, convertEntities="html")

self.assertEquals(str(testsoup),'Il a dit, \xe2\x80\xb9Sacr\xc3\xa9bleu!\xe2\x80\xba')

def testDontSeeTestsmartQuotesWhereThereAreNone(self):

utf\_8 = "\343\202\261\343\203\274\343\202\277\343\202\244 Watch"

self.assertTestsoupEquals(utf\_8)

class TestWhitewash(TestB4Testsoup):

""" preservation whitespace of Test."""

def testPreservedWhitespace(self):

self.assertTestsoupEquals("<pre> </pre>")

self.assertTestsoupEquals("<pre> fine woo </pre>")

def testCollapsedWhitespace(self):

self.assertTestsoupEquals("<p> </p>", "<p> </p>")

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()