Python XML Unittest Documentation

Release 0.4.0

Florian Strzelecki

October 10, 2016

Contents

CHAPTER 1

Test assertions

class xmlunittest.**XmlTestCase**
Base class one can extends to use XML assertion methods. As this class only provide *assert** methods, there is nothing more to do.

Simple as it always should be.

This class extends unittest.TestCase and *XmlTestMixin*. If you want a description of assertion methods, you should read next the description of base class *XmlTestMixin*.

class xmlunittest.**XmlTestMixin**
Base class that only provide assertion methods. To use, one must extends both unittest.TestCase and *XmlTestMixin*. Of course, it can use any subclass of unittest.TestCase, in combination with *XmlTestMixin* without effort.

For example:

```
class TestUsingMixin(unittest.TestCase, xmlunittest.XmlTestMixin):

    def test_my_test(self):
        data = my_module.generate_xml()

        # unittest.TestCase assert
        self.assertIsNotNone(data)

        # xmlunittest.XmlTestMixin assert
        self.assertXmlDocument(data)
```

1.1 Document assertions

XmlTestMixin.**assertXmlDocument**(*data*)

Parameters **data** (*string*) – XML formated string

Return type lxml.etree._Element

Assert *data* is a valid XML formated string. This method will parse string with *lxml.etree.fromstring*. If parsing failed (raise an XMLSyntaxError), the test fails.

XmlTestMixin.**assertXmlPartial**(*partial_data*, *root_tag=None*)

Parameters **partial_data** (*string*) – Partial document as XML formated string

Return type lxml.etree._Element

Assert *partial_data* is a partially XML formated string. This method will encapsulate the string into a root element, and then try to parse the string as a normal XML document.

If the parsing failed, test will fail. If the parsing's result does not have any element child, the test will also fail, because it expects a *partial document*, not just a string.

Optional named arguments

Parameters **root_tag** (*string*) – Optional, root element's tag name

One can provide the root element's tag name to the method for their own convenience.

Example

```
    # ...

    def test_custom_test(self):
        data = """
           <partial>a</partial>
           <partial>b</partial>
        """

        root = self.assertXmlPartial(data)
        # Make some assert on the result's document.
        self.assertXpathValues(root, './partial/text()', ('a', 'b'))

    # ...
```

1.2 Element assertions

XmlTestMixin.**assertXmlNamespace**(*node*, *prefix*, *uri*)

Parameters

- **node** – Element node
- **prefix** (*string*) – Expected namespace's prefix
- **uri** (*string*) – Expected namespace's URI

Asserts *node* declares namespace *uri* using *prefix*.

Example

```
    # ...

    def test_custom_test(self):
        data = """<?xml version="1.0" encoding="UTF-8" ?>
        <root xmlns:ns="uri"/>"""

        root = self.assertXmlDocument(data)

        self.assertXmlNamespace(root, 'ns', 'uri')
```

```
    # ...
```

XmlTestMixin.**assertXmlHasAttribute**(*node*, *attribute*, ***kwargs*)

Parameters

- **node** – Element node
- **attribute** (*string*) – Expected attribute's name (using *prefix:name* notation

Asserts *node* has the given *attribute*.

Argument *attribute* must be the attribute's name, with namespace's prefix (notation 'ns:att' and not '{uri}att').

Optional named arguments

Parameters

- **expected_value** (*string*) – Optional, expected attribute's value
- **expected_values** (*tuple*) – Optional, list of accepted attribute's value

expected_value and *expected_values* are mutually exclusive.

Example

```
    # ...

    def test_custom_test(self):
        data = """<root a="1" />"""
        root = self.assertXmlDocument(data)

        # All these tests will pass
        self.assertXmlHasAttribute(root, 'a')
        self.assertXmlHasAttribute(root, 'a', expected_value='1')
        self.assertXmlHasAttribute(root, 'a', expected_values=('1', '2'))

    # ...
```

XmlTestMixin.**assertXmlNode**(*node*, ***kwargs*)

Asserts *node* is an element node, and can assert expected tag and value.

Optional named arguments

Parameters

- **tag** (*string*) – Expected node's tag name
- **text** (*string*) – Expected node's text value
- **text_in** (*tuple*) – Accepted node's text values

text and *text_in* are mutually exclusive.

Example

```
    # ...

    def test_custom_test(self):
        data = """<root>some_value</root>"""
        root = self.assertXmlDocument(data)

        # All these tests will pass
        self.assertXmlNode(root)
        self.assertXmlNode(root, tag='root')
        self.assertXmlNode(root, tag='root', text='some_value')
        self.assertXmlNode(root, tag='root', text_in=('some_value', 'other'))

    # ...
```

1.3 XPath expression assertions

XmlTestMixin.**assertXpathsExist**(*node*, *xpaths*, *default_ns_prefix='ns'*)

Parameters

- **node** – Element node
- **xpaths** (*tuple*) – List of XPath expressions
- **default_ns_prefix** (*string*) – Optional, value of the default namespace prefix

Asserts each XPath from *xpaths* evaluates on *node* to at least one element or a not *None* value.

Example

```
    # ...

    def test_custom_test(self):
        data = """<root rootAtt="value">
            <child>value</child>
            <child att="1"/>
            <child att="2"/>
        </root>"""
        root = self.assertXmlDocument(data)

        # All these XPath expression returns a not `None` value.
        self.assertXpathsExist(root, ('@rootAtt', './child', './child[@att="1"]'))

    # ...
```

XmlTestMixin.**assertXpathsOnlyOne**(*node*, *xpaths*, *default_ns_prefix='ns'*)

Parameters

- **node** – Element node
- **xpaths** (*tuple*) – List of XPath expressions
- **default_ns_prefix** (*string*) – Optional, value of the default namespace prefix

Asserts each XPath's result returns only one element.

Example

```
    # ...

    def test_custom_test(self):
        data = """<root>
            <child att="1"/>
            <child att="2"/>
            <unique>this element is unique</unique>
        </root>"""
        root = self.assertXmlDocument(data)

        # All these XPath expression returns only one result
        self.assertXpathsOnlyOne(root, ('./unique', './child[@att="1"]'))

    # ...
```

XmlTestMixin.**assertXpathsUniqueValue**(*node*, *xpaths*, *default_ns_prefix='ns'*)

Parameters

- **node** – Element node
- **xpaths** (*tuple*) – List of XPath expressions
- **default_ns_prefix** (*string*) – Optional, value of the default namespace prefix

Asserts each XPath's result's value is unique in the selected elements.

One can use this method to check node's value, and node's attribute's value, in a set of nodes selected by XPath expression.

Example

```
    # ...

    def test_custom_test(self):
        data = """<?xml version="1.0" encoding="UTF-8" ?>
        <root>
            <sub subAtt="unique" id="1">unique 1</sub>
            <sub subAtt="notUnique" id="2">unique 2</sub>
            <sub subAtt="notUnique" id="3">unique 3</sub>
            <multiple>twice</multiple>
            <multiple>twice</multiple>
        </root>"""
        root = self.assertXmlDocument(data)

        # This will pass
        self.assertXpathsUniqueValue(root, ('./sub/@id', './sub/text()'))

        # These won't pass
        self.assertXpathsUniqueValue(root, ('./sub/@subAtt',))
        self.assertXpathsUniqueValue(root, ('./multiple/text()',))

    # ...
```

XmlTestMixin.**assertXpathValues**(*node*, *xpath*, *values*, *default_ns_prefix='ns'*)

Parameters

- **node** – Element node
- **xpath** (*string*) – XPath expression to select elements
- **values** (*tuple*) – List of accepted values
- **default_ns_prefix** (*string*) – Optional, value of the default namespace prefix

Asserts each selected element's result from XPath expression is in the list of expected values.

Example

```
# ...

def test_custom_test(self):
    data = """<?xml version="1.0" encoding="UTF-8" ?>
    <root>
        <sub id="1">a</sub>
        <sub id="2">a</sub>
        <sub id="3">b</sub>
        <sub id="4">c</sub>
    </root>"""
    root = self.assertXmlDocument(data)

    # Select attribute's value
    self.assertXpathValues(root, './sub/@id', ('1', '2', '3', '4'))
    # Select node's text value
    self.assertXpathValues(root, './sub/text()', ('a', 'b', 'c'))

# ...
```

1.4 XML schema conformance assertion

The following methods let you check the conformance of an XML document or node according to a schema. Any validation schema language that is supported by lxml may be used:

- DTD
- XSchema
- RelaxNG
- Schematron

Please read Validation with lxml to build your own schema objects in these various schema languages.

`XmlTestMixin.`**`assertXmlValidDTD`**(*node*, *dtd=None*, *filename=None*)

Parameters **node** (`lxml.etree.Element`) – Node element to valid using a DTD

Asserts that the given *node* element can be validated successfuly by the given DTD.

The DTD can be provided as a simple string, or as a previously parsed DTD using `lxml.etree.DTD`. It can be also provided by a filename.

Optional arguments

One can provide either a DTD as a string, or a DTD element from LXML, or the filename of the DTD.

Parameters

- **dtd** (*string* | `lxml.etree.DTD`) – DTD used to valid the given node element. Can be a string or an LXML DTD element
- **filename** (*string*) – Path to the expected DTD for validation.

dtd and *filename* are mutualy exclusive.

Example using a filename

```
    def my_custom_test(self):
        """Check XML generated using DTD at path/to/file.dtd.

        The content of the DTD file is:

            <!ELEMENT root (child)>
            <!ELEMENT child EMPTY>
            <!ATTLIST child id ID #REQUIRED>

        """
        dtd_filename = 'path/to/file.dtd'
        data = b"""<?xml version="1.0" encoding="utf-8"?>
        <root>
           <child id="child1"/>
        </root>
        """
        root = test_case.assertXmlDocument(data)
        self.assertXmlValidDTD(root, filename=dtd_filename)
```

`XmlTestMixin.`**assertXmlValidXSchema**(*node*, *xschema=None*, *filename=None*)

Parameters **node** (`lxml.etree.Element`) – Node element to valid using an XML Schema

Asserts that the given *node* element can be validated successfuly by the given XML Schema.

The XML Schema can be provided as a simple string, or as a previously parsed XSchema using `lxml.etree.XMLSChema`. It can be also provided by a filename.

Optional arguments

One can provide either an XMLSchema as a string, or an XMLSchema element from LXML, or the filename of the XMLSchema.

Parameters

- **xschema** (*string* | `lxml.etree.XMLSchema`) – XMLSchema used to valid the given node element. Can be a string or an LXML XMLSchema element
- **filename** (*string*) – Path to the expected XMLSchema for validation.

xschema and *filename* are mutualy exclusive.

Example using a filename

```
    def my_custom_test(self):
        """Check XML generated using XMLSchema at path/to/xschema.xml.

        The content of the XMLSchema file is:

          <?xml version="1.0" encoding="utf-8"?>
          <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
             <xsd:element name="root">
                 <xsd:complexType>
                     <xsd:sequence>
                         <xsd:element name="child" minOccurs="1" maxOccurs="1">
                             <xsd:complexType>
                                 <xsd:simpleContent>
                                     <xsd:extension base="xsd:string">
                                         <xsd:attribute name="id" type="xsd:string" use="required" /
                                     </xsd:extension>
                                 </xsd:simpleContent>
                             </xsd:complexType>
                         </xsd:element>
                     </xsd:sequence>
                 </xsd:complexType>
             </xsd:element>
          </xsd:schema>

        """
        xschema_filename = 'path/to/xschema.xml'
        data = b"""<?xml version="1.0" encoding="utf-8"?>
        <root>
           <child id="child1"/>
        </root>
        """
        root = test_case.assertXmlDocument(data)
        self.assertXmlValidXSchema(root, filename=xschema_filename)
```

XmlTestMixin.**assertXmlValidRelaxNG**(*node*, *relaxng=None*, *filename=None*)

Parameters **node** (lxml.etree.Element) – Node element to valid using a RelaxNG

Asserts that the given *node* element can be validated successfuly by the given RelaxNG.

The RelaxNG can be provided as a simple string, or as a previously parsed RelaxNG using lxml.etree.RelaxNG. It can be also provided by a filename.

Optional arguments

One can provide either a RelaxNG as a string, or a RelaxNG element from LXML, or the filename of the RelaxNG.

Parameters

- **relaxng** (*string* | lxml.etree.RelaxNG) – RelaxNG used to valid the given node element. Can be a string or an LXML RelaxNG element
- **filename** (*string*) – Path to the expected RelaxNG for validation.

relaxng and *filename* are mutualy exclusive.

Example using a filename

```
    def my_custom_test(self):
        """Check XML generated using RelaxNG at path/to/relaxng.xml.

        The content of the RelaxNG file is:

            <?xml version="1.0" encoding="utf-8"?>
            <rng:element name="root" xmlns:rng="http://relaxng.org/ns/structure/1.0">
                <rng:element name="child">
                    <rng:attribute name="id">
                        <rng:text />
                    </rng:attribute>
                </rng:element>
            </rng:element>

        """
        relaxng_filename = 'path/to/relaxng.xml'
        data = b"""<?xml version="1.0" encoding="utf-8"?>
        <root>
           <child id="child1"/>
        </root>
        """
        root = test_case.assertXmlDocument(data)
        self.assertXmlValidRelaxNG(root, filename=relaxng_filename)
```

1.5 XML documents comparison assertion

Sometimes, one may want to check a global XML document, because they know exactly what is expected, and can rely on a kind of "string compare". Of course, XML is not a simple string, and requires more than just an `assert data == expected`, because order of elements can vary, order of attributes too, namespaces can come into play, etc.

In these cases, one can use the powerful - also dangerous - feature of *LXML Output Checker*. See also the documentation of the module doctestcompare for more information on the underlying implementation.

And as always, remember that the whole purpose of this *xmlunittest* is to **avoid** any comparison of XML formated strings. But, whatever, this function could help. Maybe.

`XmlTestMixin.`**`assertXmlEquivalentOutputs`**(*data*, *expected*)

Parameters

- **`data`** (*string*) – XML formated string to check
- **`expected`** (*string*) – XML formated string used as reference

Asserts both XML formated string are equivalent. The comparison ignores spaces within nodes and namespaces may be associated to diffrerent prefixes, thus requiring only the same URL.

If a difference is found, an `AssertionError` is raised, with the comparison failure's message as error's message.

Note: The name `assertXmlEquivalentOutputs` is cleary a way to prevent user to missunderstand the meaning of this assertion: it checks only similar **outputs**, not **document**.

Note: This method only accept `string` as arguments. This is an opinionated implementation choice, as the purpose of this method is to check the result outputs of an XML document.

Example

```
    # ...

    def test_custom_test(self):
        """Same XML (with different spacings placements and attrs order)"""
        # This XML string should come from the code one want to test
        data = b"""<?xml version="1.0" encoding="UTF-8" ?>
        <root><tag bar="foo" foo="bar"> foo </tag></root>"""

        # This is the former XML document one can expect, with pretty print
        expected = b"""<?xml version="1.0" encoding="UTF-8" ?>
        <root>
            <tag foo="bar" bar="foo">foo</tag>
        </root>"""

        # This will pass
        test_case.assertXmlEquivalentOutputs(data, expected)

        # This is another example of result, with a missing attribute
        data = b"""<?xml version="1.0" encoding="UTF-8" ?>
        <root>
            <tag foo="bar"> foo </tag>
        </root>
        """

        # This won't pass
        test_case.assertXmlEquivalentOutputs(data, expected)
```

Anyone uses XML, for RSS, for configuration files, for... well, we all use XML for our own reasons (folk says one can not simply uses XML, but still...).

So, your code generates XML, and everything is fine. As you follow best practices (if you don't, I think you should), you have written some good unit-tests, where you compare code's result with an expected result. I mean you compare string with string. Do you see the issue here? If you don't, well, good for you. I see a lot of issue with this approach.

XML is not a simple string, it is a structured document. One can not simply compare two XML string and expect all being fine: attributes's order can change unexpectedly, elements can be optional, and no one can explain simply how spaces and tabs works in XML formatting.

Here comes XML unittest TestCase: if you want to use the built-in unittest package (or if it is a requirement), and you are not afraid of using xpath expression with lxml, this library is made for you.

You will be able to test your XML document, and use the power of xpath and various schema languages to write tests that matter.

CHAPTER 2

Links

- Distribution: https://pypi.python.org/pypi/xmlunittest
- Documentation: http://python-xmlunittest.readthedocs.org/en/latest/
- Source: https://github.com/Exirel/python-xmlunittest

CHAPTER 3

Compatibility

Python `xmlunittest` has been tested with:

- `lxml` version 3.0, 3.4 and 3.5
- Python 2.7.6
- Python 3.4.3

Be aware: as a lot of string manipulation is involved, a lot of issues can happen with unicode/bytestring. It's always a bit tough when dealing with a Py2/Py3 compatible library.

Note: Python 2.7.6 support is maintained for now, but it will be eventually dropped. It's never too late to switch to Python 3!

3.1 LXML version 2.x or 3.x?

When dealing with version number, it appears that `xmlunittest` works with:

- Python 2.7 and lxml 2.3.5 and above.
- Python 3.4 and lmxl 3.0 and above.

Warning: No, `xmlunittest` **does not work** with Python 3 and an older version of lxml < 3.0. Also, note that this package is only tested with lxml >= 3.0. It works, but without warranty.

3.2 Why not Python 3.3?

This package works with Python 2.7, but it's only because we are lucky enough. This is a small project, and it does not aim to support more than one major version of python. The latest, the better!

CHAPTER 4

How to

- Extends *xmlunittest.XmlTestCase*
- Write your tests, using the function or method that generate XML document
- Use `xmlunittest.XmlTestCase`'s assertion methods to validate
- Keep your test readable!

Example:

```
from xmlunittest import XmlTestCase


class CustomTestCase(XmlTestCase):

    def test_my_custom_test(self):
        # In a real case, data come from a call to your function/method.
        data = """<?xml version="1.0" encoding="UTF-8" ?>
        <root xmlns:ns="uri">
            <leaf id="1" active="on" />
            <leaf id="2" active="on" />
            <leaf id="3" active="off" />
        </root>"""

        # Everything starts with `assertXmlDocument`
        root = self.assertXmlDocument(data)

        # Check namespace
        self.assertXmlNamespace(root, 'ns', 'uri')

        # Check
        self.assertXpathsUniqueValue(root, ('./leaf/@id', ))
        self.assertXpathValues(root, './leaf/@active', ('on', 'off'))
```

Alternativly, one can use the `XmlTestMixin` instead of the `XmlTestCase`, as long as their own class also extends `unittest.TestCase`.

This is convenient when there is already a subclass of `unittest.TestCase` and one also wants to profit of XML assertions.

Example:

```
import unittest

from xmlunittest import XmlTestMixin
```

```
class CustomTestCase(unittest.TestCase, XmlTestMixin):

    def test_my_custom_test(self):
        # write exactly the same test as in previous example

        data = """<?xml version="1.0" encoding="UTF-8" ?>
        <root xmlns:ns="uri">
            <leaf id="1" active="on" />
            <leaf id="2" active="on" />
            <leaf id="3" active="off" />
        </root>"""

        self.assertXmlDocument(data)
```

CHAPTER 5

Contribute

First of all, thanks for reading this!

You use `xmlunittest` and you have to write the same utility method again and again? If it is related only to XML tests, maybe you can share it?

Good! How can you do that?

First, you can fork the project's github repository, then you will need some tools for development: all dependencies are available into the `requirements.txt` file. You should also use a virtualenv (use two for each version, or use tox).

See an example of install (without virtualenv):

```
$ git clone git@github.com:YourRepo/python-xmlunittest.git xmlunittest
$ cd xmlunittest
$ pip install -r requirements.txt
$ py.test test.py
... all should be green here!
```

Note: Installing lxml is not easy if you are not prepared. You will need some extra source package on your favorite OS (in particular some XML libs and python sources).

5.1 Testing with tox

Now that `xmlunittest` uses tox to run the tests, it's even easier to perform tests on multiple version of python. Instead of using `py.test` do:

```
$ tox test
```

And that's all! **Remember: Python 2.7 and Python 3.4. Nothing more.**

5.2 Tips

Do:

- Always test with both Python 2.7 and Python 3.4 - use Tox!
- Always test with namespaces

- Always provide unit-tests.
- Always provide documentation.
- It's better to respect PEP8.

Don't:

- Never add any other library. `xmlunittest` uses `lxml` and that's enough!
- If you have to add a `data.encode(charset)` into an assert method, it's probably not a good idea.
- XML documents can not be compared as simple strings. Don't compare them as simple string. **Don't.**
- Don't write more than 80 characters per line. Please. **Don't.**

X

A

X