xmlschema Documentation

Release 1.0.18

Davide Brunato

Jan 07, 2020

Contents

CHAPTER 1

Introduction

The *xmlschema* library is an implementation of XML Schema for Python (supports Python 2.7 and Python 3.5+).

This library arises from the needs of a solid Python layer for processing XML Schema based files for MaX (Materials design at the Exascale) European project. A significant problem is the encoding and the decoding of the XML data files produced by different simulation software. Another important requirement is the XML data validation, in order to put the produced data under control. The lack of a suitable alternative for Python in the schema-based decoding of XML data has led to build this library. Obviously this library can be useful for other cases related to XML Schema based processing, not only for the original scope.

The full xmlschema documentation is available on "Read the Docs".

1.1 Features

This library includes the following features:

- Full XSD 1.0 and XSD 1.1 support
- Building of XML schema objects from XSD files
- Validation of XML instances against XSD schemas
- Decoding of XML data into Python data and to JSON
- Encoding of Python data and JSON to XML
- Data decoding and encoding ruled by converter classes
- An XPath based API for finding schema's elements and attributes
- Support of XSD validation modes *strict*/*lax*/*skip*
- Remote attacks protection by default using an XMLParser that forbids entities

Note: Currently the XSD 1.1 validator is provided by class *XMLSchema11* and the default *XMLSchema* class is still an alias of the XSD 1.0 validator, the class *XMLSchema10*. From version 1.1 of the package the default validator will

be linked to the XSD 1.1 validator, a version that will also removes support for Python 2.7.

1.2 Installation

You can install the library with *pip* in a Python 2.7 or Python 3.5+ environment:

```
pip install xmlschema
```

The library uses the Python's ElementTree XML library and requires elementpath additional package. The base schemas of the XSD standards are included in the package for working offline and to speed-up the building of schema instances.

CHAPTER 2

Usage

Import the library in your code with:

```
import xmlschema
```

The module initialization builds the dictionary containing the code points of the Unicode categories.

2.1 Create a schema instance

Import the library and then create an instance of a schema using the path of the file containing the schema as argument:

```
>>> import xmlschema
>>> schema = xmlschema.XMLSchema('xmlschema/tests/test_cases/examples/vehicles/
↪vehicles.xsd')
```

Otherwise the argument can be also an opened file-like object:

```
>>> import xmlschema
>>> schema_file = open('xmlschema/tests/test_cases/examples/collection/collection.xsd
↪')
>>> schema = xmlschema.XMLSchema(schema_file)
```

Alternatively you can pass a string containing the schema definition:

```
>>> import xmlschema
>>> schema = xmlschema.XMLSchema("""
... <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
... <xs:element name="block" type="xs:string"/>
... </xs:schema>
... """)
```

Strings and file-like objects might not work when the schema includes other local subschemas, because the package cannot knows anything about the schema's source location:

```
>>> import xmlschema
>>> schema_xsd = open('xmlschema/tests/test_cases/examples/vehicles/vehicles.xsd').
↪read()
>>> schema = xmlschema.XMLSchema(schema_xsd)
Traceback (most recent call last):
...
...
xmlschema.validators.exceptions.XMLSchemaParseError: unknown element '{http://example.
↪com/vehicles}cars':

Schema:

  <xs:element xmlns:xs="http://www.w3.org/2001/XMLSchema" ref="vh:cars" />

Path: /xs:schema/xs:element/xs:complexType/xs:sequence/xs:element
```

In these cases you can provide an appropriate *base_url* optional argument to define the reference directory path for other includes and imports:

```
>>> import xmlschema
>>> schema_file = open('xmlschema/tests/test_cases/examples/vehicles/vehicles.xsd')
>>> schema = xmlschema.XMLSchema(schema_file, base_url='xmlschema/tests/test_cases/
↪examples/vehicles/')
```

2.2 XSD declarations

The schema object includes XSD components of declarations (*elements*, *attributes* and *notations*) and definitions (*types*, *model groups*, *attribute groups*, *identity constraints* and *substitution groups*). The global XSD components are available as attributes of the schema instance:

```
>>> import xmlschema
>>> from pprint import pprint
>>> schema = xmlschema.XMLSchema('xmlschema/tests/test_cases/examples/vehicles/
↪vehicles.xsd')
>>> schema.types
NamespaceView({'vehicleType': XsdComplexType(name='vehicleType')})
>>> pprint(dict(schema.elements))
{'bikes': XsdElement(name='vh:bikes', occurs=[1, 1]),
 'cars': XsdElement(name='vh:cars', occurs=[1, 1]),
 'vehicles': XsdElement(name='vh:vehicles', occurs=[1, 1])}
>>> schema.attributes
NamespaceView({'step': XsdAttribute(name='vh:step')})
```

Global components are local views of *XSD global maps* shared between related schema instances. The global maps can be accessed through `XMLSchema.maps` attribute:

```
>>> from pprint import pprint
>>> pprint(sorted(schema.maps.types.keys())[:5])
['{http://example.com/vehicles}vehicleType',
 '{http://www.w3.org/2001/XMLSchema}ENTITIES',
 '{http://www.w3.org/2001/XMLSchema}ENTITY',
 '{http://www.w3.org/2001/XMLSchema}ID',
 '{http://www.w3.org/2001/XMLSchema}IDREF']
>>> pprint(sorted(schema.maps.elements.keys())[:10])
```

(continues on next page)

(continued from previous page)

```
['{http://example.com/vehicles}bikes',
 '{http://example.com/vehicles}cars',
 '{http://example.com/vehicles}vehicles',
 '{http://www.w3.org/2001/XMLSchema}all',
 '{http://www.w3.org/2001/XMLSchema}annotation',
 '{http://www.w3.org/2001/XMLSchema}any',
 '{http://www.w3.org/2001/XMLSchema}anyAttribute',
 '{http://www.w3.org/2001/XMLSchema}appinfo',
 '{http://www.w3.org/2001/XMLSchema}attribute',
 '{http://www.w3.org/2001/XMLSchema}attributeGroup']
```

Schema objects include methods for finding XSD elements and attributes in the schema. Those are methods ot the ElementTree's API, so you can use an XPath expression for defining the search criteria:

```
>>> schema.find('vh:vehicles/vh:bikes')
XsdElement(ref='vh:bikes', occurs=[1, 1])
>>> pprint(schema.findall('vh:vehicles/*'))
[XsdElement(ref='vh:cars', occurs=[1, 1]),
 XsdElement(ref='vh:bikes', occurs=[1, 1])]
```

2.3 Validation

The library provides several methods to validate an XML document with a schema.

The first mode is the method `XMLSchema.is_valid()`. This method returns `True` if the XML argument is validated by the schema loaded in the instance, returns `False` if the document is invalid.

```
>>> import xmlschema
>>> schema = xmlschema.XMLSchema('xmlschema/tests/test_cases/examples/vehicles/
↪vehicles.xsd')
>>> schema.is_valid('xmlschema/tests/test_cases/examples/vehicles/vehicles.xml')
True
>>> schema.is_valid('xmlschema/tests/test_cases/examples/vehicles/vehicles-1_error.xml
↪')
False
>>> schema.is_valid("""<?xml version="1.0" encoding="UTF-8"?><fancy_tag/>""")
False
```

An alternative mode for validating an XML document is implemented by the method `XMLSchema.validate()`, that raises an error when the XML doesn't conforms to the schema:

```
>>> import xmlschema
>>> schema = xmlschema.XMLSchema('xmlschema/tests/test_cases/examples/vehicles/
↪vehicles.xsd')
>>> schema.validate('xmlschema/tests/test_cases/examples/vehicles/vehicles.xml')
>>> schema.validate('xmlschema/tests/test_cases/examples/vehicles/vehicles-1_error.xml
↪')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/brunato/Development/projects/xmlschema/xmlschema/schema.py", line 220,␣
↪in validate
    raise error
xmlschema.exceptions.XMLSchemaValidationError: failed validating <Element ...
```

(continues on next page)

(continued from previous page)

```
Reason: character data between child elements not allowed!

Schema:

  <xs:sequence xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <xs:element maxOccurs="unbounded" minOccurs="0" name="car" type=
↪"vh:vehicleType" />
  </xs:sequence>

Instance:

  <ns0:cars xmlns:ns0="http://example.com/vehicles">
    NOT ALLOWED CHARACTER DATA
    <ns0:car make="Porsche" model="911" />
    <ns0:car make="Porsche" model="911" />
  </ns0:cars>
```

A validation method is also available at module level, useful when you need to validate a document only once or if you extract information about the schema, typically the schema location and the namespace, directly from the XML document:

```
>>> import xmlschema
>>> xmlschema.validate('xmlschema/tests/test_cases/examples/vehicles/vehicles.xml')
```

```
>>> import xmlschema
>>> os.chdir('xmlschema/tests/test_cases/examples/vehicles/')
>>> xmlschema.validate('vehicles.xml', 'vehicles.xsd')
```

2.4 Data decoding and encoding

Each schema component includes methods for data conversion:

```
>>> schema.types['vehicleType'].decode
<bound method XsdComplexType.decode of XsdComplexType(name='vehicleType')>
>>> schema.elements['cars'].encode
<bound method ValidationMixin.encode of XsdElement(name='vh:cars', occurs=[1, 1])>
```

Those methods can be used to decode the correspondents parts of the XML document:

```
>>> import xmlschema
>>> from pprint import pprint
>>> from xml.etree import ElementTree
>>> xs = xmlschema.XMLSchema('xmlschema/tests/test_cases/examples/vehicles/vehicles.
↪xsd')
>>> xt = ElementTree.parse('xmlschema/tests/test_cases/examples/vehicles/vehicles.xml
↪')
>>> root = xt.getroot()
>>> pprint(xs.elements['cars'].decode(root[0]))
{'{http://example.com/vehicles}car': [{'@make': 'Porsche', '@model': '911'},
                                      {'@make': 'Porsche', '@model': '911'}]}
>>> pprint(xs.elements['cars'].decode(xt.getroot()[1], validation='skip'))
None
>>> pprint(xs.elements['bikes'].decode(root[1], namespaces={'vh': 'http://example.com/
↪vehicles'}))
```

(continues on next page)

(continued from previous page)

```
{'@xmlns:vh': 'http://example.com/vehicles',
 'vh:bike': [{'@make': 'Harley-Davidson', '@model': 'WL'},
             {'@make': 'Yamaha', '@model': 'XS650'}]}
```

You can also decode the entire XML document to a nested dictionary:

```
>>> import xmlschema
>>> from pprint import pprint
>>> xs = xmlschema.XMLSchema('xmlschema/tests/test_cases/examples/vehicles/vehicles.
↪xsd')
>>> pprint(xs.to_dict('xmlschema/tests/test_cases/examples/vehicles/vehicles.xml'))
{'@xmlns:vh': 'http://example.com/vehicles',
 '@xmlns:xsi': 'http://www.w3.org/2001/XMLSchema-instance',
 '@xsi:schemaLocation': 'http://example.com/vehicles vehicles.xsd',
 'vh:bikes': {'vh:bike': [{'@make': 'Harley-Davidson', '@model': 'WL'},
                          {'@make': 'Yamaha', '@model': 'XS650'}]},
 'vh:cars': {'vh:car': [{'@make': 'Porsche', '@model': '911'},
                        {'@make': 'Porsche', '@model': '911'}]}}
```

The decoded values coincide with the datatypes declared in the XSD schema:

```
>>> import xmlschema
>>> from pprint import pprint
>>> xs = xmlschema.XMLSchema('xmlschema/tests/test_cases/examples/collection/
↪collection.xsd')
>>> pprint(xs.to_dict('xmlschema/tests/test_cases/examples/collection/collection.xml
↪'))
{'@xmlns:col': 'http://example.com/ns/collection',
 '@xmlns:xsi': 'http://www.w3.org/2001/XMLSchema-instance',
 '@xsi:schemaLocation': 'http://example.com/ns/collection collection.xsd',
 'object': [{'@available': True,
             '@id': 'b0836217462',
             'author': {'@id': 'PAR',
                        'born': '1841-02-25',
                        'dead': '1919-12-03',
                        'name': 'Pierre-Auguste Renoir',
                        'qualification': 'painter'},
             'estimation': Decimal('10000.00'),
             'position': 1,
             'title': 'The Umbrellas',
             'year': '1886'},
            {'@available': True,
             '@id': 'b0836217463',
             'author': {'@id': 'JM',
                        'born': '1893-04-20',
                        'dead': '1983-12-25',
                        'name': 'Joan Miró',
                        'qualification': 'painter, sculptor and ceramicist'},
             'position': 2,
             'title': None,
             'year': '1925'}]}
```

If you need to decode only a part of the XML document you can pass also an XPath expression using in the *path* argument.

```
>>> xs = xmlschema.XMLSchema('xmlschema/tests/test_cases/examples/vehicles/vehicles.
↪xsd')
```

(continues on next page)

(continued from previous page)

```
>>> pprint(xs.to_dict('xmlschema/tests/test_cases/examples/vehicles/vehicles.xml', '/
↪vh:vehicles/vh:bikes'))
{'vh:bike': [{'@make': 'Harley-Davidson', '@model': 'WL'},
             {'@make': 'Yamaha', '@model': 'XS650'}]}
```

Note: Decode using an XPath could be simpler than using subelements, method illustrated previously. An XPath expression for the schema *considers the schema as the root element with global elements as its children*.

All the decoding and encoding methods are based on two generator methods of the *XMLSchema* class, namely *iter_decode()* and *iter_encode()*, that yield both data and validation errors. See *Schema level API* section for more information.

2.5 Validating and decoding ElementTree's elements

Validation and decode API works also with XML data loaded in ElementTree structures:

```
>>> import xmlschema
>>> from pprint import pprint
>>> from xml.etree import ElementTree
>>> xs = xmlschema.XMLSchema('xmlschema/tests/test_cases/examples/vehicles/vehicles.
↪xsd')
>>> xt = ElementTree.parse('xmlschema/tests/test_cases/examples/vehicles/vehicles.xml
↪')
>>> xs.is_valid(xt)
True
>>> pprint(xs.to_dict(xt, process_namespaces=False), depth=2)
{'@{http://www.w3.org/2001/XMLSchema-instance}schemaLocation': 'http://...',
 '{http://example.com/vehicles}bikes': {'{http://example.com/vehicles}bike': [...]},
 '{http://example.com/vehicles}cars': {'{http://example.com/vehicles}car': [...]}}
```

The standard ElementTree library lacks of namespace information in trees, so you have to provide a map to convert URIs to prefixes:

```
>>> namespaces = {'xsi': 'http://www.w3.org/2001/XMLSchema-instance', 'vh': 'http://
↪example.com/vehicles'}
>>> pprint(xs.to_dict(xt, namespaces=namespaces))
{'@xmlns:vh': 'http://example.com/vehicles',
 '@xmlns:xsi': 'http://www.w3.org/2001/XMLSchema-instance',
 '@xsi:schemaLocation': 'http://example.com/vehicles vehicles.xsd',
 'vh:bikes': {'vh:bike': [{'@make': 'Harley-Davidson', '@model': 'WL'},
                          {'@make': 'Yamaha', '@model': 'XS650'}]},
 'vh:cars': {'vh:car': [{'@make': 'Porsche', '@model': '911'},
                        {'@make': 'Porsche', '@model': '911'}]}}
```

You can also convert XML data using the lxml library, that works better because namespace information is associated within each node of the trees:

```
>>> import xmlschema
>>> from pprint import pprint
>>> import lxml.etree as ElementTree
>>> xs = xmlschema.XMLSchema('xmlschema/tests/test_cases/examples/vehicles/vehicles.
↪xsd')
```

(continues on next page)

(continued from previous page)

```
>>> xt = ElementTree.parse('xmlschema/tests/test_cases/examples/vehicles/vehicles.xml
↪')
>>> xs.is_valid(xt)
True
>>> pprint(xs.to_dict(xt))
{'@xmlns:vh': 'http://example.com/vehicles',
 '@xmlns:xsi': 'http://www.w3.org/2001/XMLSchema-instance',
 '@xsi:schemaLocation': 'http://example.com/vehicles vehicles.xsd',
 'vh:bikes': {'vh:bike': [{'@make': 'Harley-Davidson', '@model': 'WL'},
                          {'@make': 'Yamaha', '@model': 'XS650'}]},
 'vh:cars': {'vh:car': [{'@make': 'Porsche', '@model': '911'},
                        {'@make': 'Porsche', '@model': '911'}]}}
>>> pprint(xmlschema.to_dict(xt, 'xmlschema/tests/test_cases/examples/vehicles/
↪vehicles.xsd'))
{'@xmlns:vh': 'http://example.com/vehicles',
 '@xmlns:xsi': 'http://www.w3.org/2001/XMLSchema-instance',
 '@xsi:schemaLocation': 'http://example.com/vehicles vehicles.xsd',
 'vh:bikes': {'vh:bike': [{'@make': 'Harley-Davidson', '@model': 'WL'},
                          {'@make': 'Yamaha', '@model': 'XS650'}]},
 'vh:cars': {'vh:car': [{'@make': 'Porsche', '@model': '911'},
                        {'@make': 'Porsche', '@model': '911'}]}}
```

2.6 Customize the decoded data structure

Starting from the version 0.9.9 the package includes converter objects, in order to control the decoding process and produce different data structures. Those objects intervene at element level to compose the decoded data (attributes and content) into a data structure.

The default converter produces a data structure similar to the format produced by previous versions of the package. You can customize the conversion process providing a converter instance or subclass when you create a schema instance or when you want to decode an XML document. For instance you can use the *Badgerfish* converter for a schema instance:

```
>>> import xmlschema
>>> from pprint import pprint
>>> xml_schema = 'xmlschema/tests/test_cases/examples/vehicles/vehicles.xsd'
>>> xml_document = 'xmlschema/tests/test_cases/examples/vehicles/vehicles.xml'
>>> xs = xmlschema.XMLSchema(xml_schema, converter=xmlschema.BadgerFishConverter)
>>> pprint(xs.to_dict(xml_document, dict_class=dict), indent=4)
{   '@xmlns': {   'vh': 'http://example.com/vehicles',
                  'xsi': 'http://www.w3.org/2001/XMLSchema-instance'},
    'vh:vehicles': {   '@xsi:schemaLocation': 'http://example.com/vehicles '
                                              'vehicles.xsd',
                       'vh:bikes': {   'vh:bike': [   {   '@make': 'Harley-Davidson',
                                                          '@model': 'WL'},
                                                      {   '@make': 'Yamaha',
                                                          '@model': 'XS650'}]},
                       'vh:cars': {   'vh:car': [   {   '@make': 'Porsche',
                                                        '@model': '911'},
                                                    {   '@make': 'Porsche',
                                                        '@model': '911'}]}}}
```

You can also change the data decoding process providing the keyword argument *converter* to the method call:

```
>>> pprint(xs.to_dict(xml_document, converter=xmlschema.ParkerConverter, dict_
↪class=dict), indent=4)
{'vh:bikes': {'vh:bike': [None, None]}, 'vh:cars': {'vh:car': [None, None]}}
```

See the *Customizing output data with converters* section for more information about converters.

2.7 Decoding to JSON

The data structured created by the decoder can be easily serialized to JSON. But if you data include *Decimal* values (for *decimal* XSD built-in type) you cannot convert the data to JSON:

```
>>> import xmlschema
>>> import json
>>> xml_document = 'xmlschema/tests/test_cases/examples/collection/collection.xml'
>>> print(json.dumps(xmlschema.to_dict(xml_document), indent=4))
Traceback (most recent call last):
  File "/usr/lib64/python2.7/doctest.py", line 1315, in __run
    compileflags, 1) in test.globs
  File "<doctest default[3]>", line 1, in <module>
    print(json.dumps(xmlschema.to_dict(xml_document), indent=4))
  File "/usr/lib64/python2.7/json/__init__.py", line 251, in dumps
    sort_keys=sort_keys, **kw).encode(obj)
  File "/usr/lib64/python2.7/json/encoder.py", line 209, in encode
    chunks = list(chunks)
  File "/usr/lib64/python2.7/json/encoder.py", line 434, in _iterencode
    for chunk in _iterencode_dict(o, _current_indent_level):
  File "/usr/lib64/python2.7/json/encoder.py", line 408, in _iterencode_dict
    for chunk in chunks:
  File "/usr/lib64/python2.7/json/encoder.py", line 332, in _iterencode_list
    for chunk in chunks:
  File "/usr/lib64/python2.7/json/encoder.py", line 408, in _iterencode_dict
    for chunk in chunks:
  File "/usr/lib64/python2.7/json/encoder.py", line 442, in _iterencode
    o = _default(o)
  File "/usr/lib64/python2.7/json/encoder.py", line 184, in default
    raise TypeError(repr(o) + " is not JSON serializable")
TypeError: Decimal('10000.00') is not JSON serializable
```

This problem is resolved providing an alternative JSON-compatible type for *Decimal* values, using the keyword argument *decimal_type*:

```
>>> print(json.dumps(xmlschema.to_dict(xml_document, decimal_type=str), indent=4))
{
    "object": [
        {
            "@available": true,
            "author": {
                "qualification": "painter",
                "born": "1841-02-25",
                "@id": "PAR",
                "name": "Pierre-Auguste Renoir",
                "dead": "1919-12-03"
            },
            "title": "The Umbrellas",
            "year": "1886",
```

(continues on next page)

(continued from previous page)

```
            "position": 1,
            "estimation": "10000.00",
            "@id": "b0836217462"
        },
        {
            "@available": true,
            "author": {
                "qualification": "painter, sculptor and ceramicist",
                "born": "1893-04-20",
                "@id": "JM",
                "name": "Joan Mir\u00f3",
                "dead": "1983-12-25"
            },
            "title": null,
            "year": "1925",
            "position": 2,
            "@id": "b0836217463"
        }
    ],
    "@xsi:schemaLocation": "http://example.com/ns/collection collection.xsd"
}
```

From version 1.0 there are two module level API for simplify the JSON serialization and deserialization task. See the *xmlschema.to_json()* and *xmlschema.from_json()* in the *Document level API* section.

2.8 XSD validation modes

Starting from the version 0.9.10 the library uses XSD validation modes *strict*/*lax*/*skip*, both for schemas and for XML instances. Each validation mode defines a specific behaviour:

strict Schemas are validated against the meta-schema. The processor stops when an error is found in a schema or during the validation/decode of XML data.

lax Schemas are validated against the meta-schema. The processor collects the errors and continues, eventually replacing missing parts with wildcards. Undecodable XML data are replaced with *None*.

skip Schemas are not validated against the meta-schema. The processor doesn't collect any error. Undecodable XML data are replaced with the original text.

The default mode is *strict*, both for schemas and for XML data. The mode is set with the *validation* argument, provided when creating the schema instance or when you want to validate/decode XML data. For example you can build a schema using a *strict* mode and then decode XML data using the *validation* argument setted to 'lax'.

2.9 Lazy validation

From release v1.0.12 the document validation and decoding API has an optional argument *lazy=False*, that can be changed to True for operating with a lazy `XMLResource`. The lazy mode can be useful for validating and decoding big XML data files. This is still an experimental feature that will be refined and integrated in future versions.

2.10 XSD 1.0 and 1.1 support

From release v1.0.14 XSD 1.1 support has been added to the library through the class `XMLSchema11`. You have to use this class for XSD 1.1 schemas instead the default class `XMLSchema` that is still linked to XSD 1.0 validator `XMLSchema10`. From next minor release (v1.1) the default class will become `XMLSchema11`.

2.11 XML entity-based attacks protection

The XML data resource loading is protected using the *SafeXMLParser* class, a subclass of the pure Python version of XMLParser that forbids the use of entities. The protection is applied both to XSD schemas and to XML data. The usage of this feature is regulated by the XMLSchema's argument *defuse*. For default this argument has value *'remote'* that means the protection on XML data is applied only to data loaded from remote. Other values for this argument can be *'always'* and *'never'*.

2.12 Processing limits

From release v1.0.16 a module has been added in order to group constants that define processing limits, generally to protect against attacks prepared to exhaust system resources. These limits usually don't need to be changed, but this possibility has been left at the module level for situations where a different setting is needed.

2.12.1 Limit on XSD model groups checking

Model groups of the schemas are checked against restriction violations and *Unique Particle Attribution* violations. To avoids XSD model recursion attacks a depth limit of 15 levels is set. If this limit is exceeded an `XMLSchemaModelDepthError` is raised, the error is caught and a warning is generated. If you need to set an higher limit for checking all your groups you can import the library and change the value of `MAX_MODEL_DEPTH` in the limits module:

```
>>> import xmlschema
>>> xmlschema.limits.MAX_MODEL_DEPTH = 20
```

2.12.2 Limit on XML data depth

A limit of 9999 on maximum depth is set for XML validation/decoding/encoding to avoid attacks based on extremely deep XML data. To increase or decrease this limit change the value of `MAX_XML_DEPTH` in the module *limits* after the import of the package:

```
>>> import xmlschema
>>> xmlschema.limits.MAX_XML_DEPTH = 1000
```

CHAPTER 3

API Documentation

3.1 Document level API

xmlschema.**validate**(*xml_document*, *schema=None*, *cls=None*, *path=None*, *schema_path=None*, *use_defaults=True*, *namespaces=None*, *locations=None*, *base_url=None*, *defuse='remote'*, *timeout=300*, *lazy=False*)

Validates an XML document against a schema instance. This function builds an *XMLSchema* object for validating the XML document. Raises an *XMLSchemaValidationError* if the XML document is not validated against the schema.

Parameters

- **xml_document** – can be an *XMLResource* instance, a file-like object a path to a file or an URI of a resource or an Element instance or an ElementTree instance or a string containing the XML data. If the passed argument is not an *XMLResource* instance a new one is built using this and *defuse*, *timeout* and *lazy* arguments.
- **schema** – can be a schema instance or a file-like object or a file path or a URL of a resource or a string containing the schema.
- **cls** – class to use for building the schema instance (for default *XMLSchema* is used).
- **path** – is an optional XPath expression that matches the elements of the XML data that have to be decoded. If not provided the XML root element is used.
- **schema_path** – an XPath expression to select the XSD element to use for decoding. If not provided the *path* argument or the *source* root tag are used.
- **use_defaults** – defines when to use element and attribute defaults for filling missing required values.
- **namespaces** – is an optional mapping from namespace prefix to URI.
- **locations** – additional schema location hints, in case a schema instance has to be built.
- **base_url** – is an optional custom base URL for remapping relative locations, for default uses the directory where the XSD or alternatively the XML document is located.

- **defuse** – optional argument to pass for construct schema and *XMLResource* instances.
- **timeout** – optional argument to pass for construct schema and *XMLResource* instances.
- **lazy** – optional argument for construct the *XMLResource* instance.

xmlschema.**is_valid**(*xml_document*, *schema=None*, *cls=None*, *path=None*, *schema_path=None*, *use_defaults=True*, *namespaces=None*, *locations=None*, *base_url=None*, *defuse='remote'*, *timeout=300*, *lazy=False*)

Like *validate()* except that do not raises an exception but returns `True` if the XML document is valid, `False` if it's invalid.

xmlschema.**iter_errors**(*xml_document*, *schema=None*, *cls=None*, *path=None*, *schema_path=None*, *use_defaults=True*, *namespaces=None*, *locations=None*, *base_url=None*, *defuse='remote'*, *timeout=300*, *lazy=False*)

Creates an iterator for the errors generated by the validation of an XML document. Takes the same arguments of the function *validate()*.

xmlschema.**to_dict**(*xml_document*, *schema=None*, *cls=None*, *path=None*, *process_namespaces=True*, *locations=None*, *base_url=None*, *defuse='remote'*, *timeout=300*, *lazy=False*, ***kwargs*)

Decodes an XML document to a Python's nested dictionary. The decoding is based on an XML Schema class instance. For default the document is validated during the decoding phase. Raises an *XMLSchemaValidationError* if the XML document is not validated against the schema.

Parameters

- **xml_document** – can be an *XMLResource* instance, a file-like object a path to a file or an URI of a resource or an Element instance or an ElementTree instance or a string containing the XML data. If the passed argument is not an *XMLResource* instance a new one is built using this and *defuse*, *timeout* and *lazy* arguments.
- **schema** – can be a schema instance or a file-like object or a file path or a URL of a resource or a string containing the schema.
- **cls** – class to use for building the schema instance (for default uses *XMLSchema*).
- **path** – is an optional XPath expression that matches the elements of the XML data that have to be decoded. If not provided the XML root element is used.
- **process_namespaces** – indicates whether to use namespace information in the decoding process.
- **locations** – additional schema location hints, in case a schema instance has to be built.
- **base_url** – is an optional custom base URL for remapping relative locations, for default uses the directory where the XSD or alternatively the XML document is located.
- **defuse** – optional argument to pass for construct schema and *XMLResource* instances.
- **timeout** – optional argument to pass for construct schema and *XMLResource* instances.
- **lazy** – optional argument for construct the *XMLResource* instance.
- **kwargs** – other optional arguments of `XMLSchema.iter_decode()` as keyword arguments.

Returns an object containing the decoded data. If `validation='lax'` keyword argument is provided the validation errors are collected and returned coupled in a tuple with the decoded data.

Raises *XMLSchemaValidationError* if the object is not decodable by the XSD component, or also if it's invalid when `validation='strict'` is provided.

xmlschema.**to_json**(*xml_document*, *fp=None*, *schema=None*, *cls=None*, *path=None*, *converter=None*, *process_namespaces=True*, *locations=None*, *base_url=None*, *defuse='remote'*, *timeout=300*, *lazy=False*, *json_options=None*, ***kwargs*)

Serialize an XML document to JSON. For default the XML data is validated during the decoding phase. Raises an *XMLSchemaValidationError* if the XML document is not validated against the schema.

Parameters

- **xml_document** – can be an *XMLResource* instance, a file-like object a path to a file or an URI of a resource or an Element instance or an ElementTree instance or a string containing the XML data. If the passed argument is not an *XMLResource* instance a new one is built using this and *defuse*, *timeout* and *lazy* arguments.
- **fp** – can be a `write()` supporting file-like object.
- **schema** – can be a schema instance or a file-like object or a file path or an URL of a resource or a string containing the schema.
- **cls** – schema class to use for building the instance (for default uses *XMLSchema*).
- **path** – is an optional XPath expression that matches the elements of the XML data that have to be decoded. If not provided the XML root element is used.
- **converter** – an *XMLSchemaConverter* subclass or instance to use for the decoding.
- **process_namespaces** – indicates whether to use namespace information in the decoding process.
- **locations** – additional schema location hints, in case a schema instance has to be built.
- **base_url** – is an optional custom base URL for remapping relative locations, for default uses the directory where the XSD or alternatively the XML document is located.
- **defuse** – optional argument to pass for construct schema and *XMLResource* instances.
- **timeout** – optional argument to pass for construct schema and *XMLResource* instances.
- **lazy** – optional argument for construct the *XMLResource* instance.
- **json_options** – a dictionary with options for the JSON serializer.
- **kwargs** – optional arguments of `XMLSchema.iter_decode()` as keyword arguments to variate the decoding process.

Returns a string containing the JSON data if *fp* is *None*, otherwise doesn't return anything. If `validation='lax'` keyword argument is provided the validation errors are collected and returned, eventually coupled in a tuple with the JSON data.

Raises *XMLSchemaValidationError* if the object is not decodable by the XSD component, or also if it's invalid when `validation='strict'` is provided.

xmlschema.**from_json**(*source*, *schema*, *path=None*, *converter=None*, *json_options=None*, ***kwargs*)

Deserialize JSON data to an XML Element.

Parameters

- **source** – can be a string or a `read()` supporting file-like object containing the JSON document.
- **schema** – an *XMLSchema* or an *XMLSchema11* instance.
- **path** – is an optional XPath expression for selecting the element of the schema that matches the data that has to be encoded. For default the first global element of the schema is used.
- **converter** – an *XMLSchemaConverter* subclass or instance to use for the encoding.

- **json_options** – a dictionary with options for the JSON deserializer.
- **kwargs** – Keyword arguments containing options for converter and encoding.

Returns An element tree's Element instance. If `validation='lax'` keyword argument is provided the validation errors are collected and returned coupled in a tuple with the Element instance.

Raises *XMLSchemaValidationError* if the object is not encodable by the schema, or also if it's invalid when `validation='strict'` is provided.

3.2 Schema level API

class xmlschema.**XMLSchema10**

class xmlschema.**XMLSchema11**

The classes for XSD v1.0 and v1.1 schema instances. They are both generated by the meta-class `XMLSchemaMeta` and take the same API of `XMLSchemaBase`.

xmlschema.**XMLSchema**

The default class for schema instances.

alias of `xmlschema.validators.schema.XMLSchema10`

class xmlschema.**XMLSchemaBase**(*source*, *namespace=None*, *validation='strict'*, *global_maps=None*, *converter=None*, *locations=None*, *base_url=None*, *defuse='remote'*, *timeout=300*, *build=True*, *use_meta=True*, *loglevel=None*)

Base class for an XML Schema instance.

Parameters

- **source** (*Element or ElementTree or str or file-like object*) – an URI that reference to a resource or a file path or a file-like object or a string containing the schema or an Element or an ElementTree document.
- **namespace** (*str or None*) – is an optional argument that contains the URI of the namespace. When specified it must be equal to the *targetNamespace* declared in the schema.
- **validation** (*str*) – defines the XSD validation mode to use for build the schema, it's value can be 'strict', 'lax' or 'skip'.
- **global_maps** (*XsdGlobals or None*) – is an optional argument containing an *XsdGlobals* instance, a mediator object for sharing declaration data between dependents schema instances.
- **converter** (*XMLSchemaConverter or None*) – is an optional argument that can be an *XMLSchemaConverter* subclass or instance, used for defining the default XML data converter for XML Schema instance.
- **locations** (*dict or list or None*) – schema location hints, that can include additional namespaces to import after processing schema's import statements. Usually filled with the couples (namespace, url) extracted from xsi:schemaLocations. Can be a dictionary or a sequence of couples (namespace URI, resource URL).
- **base_url** (*str or None*) – is an optional base URL, used for the normalization of relative paths when the URL of the schema resource can't be obtained from the source argument.
- **defuse** (*str or None*) – defines when to defuse XML data. Can be 'always', 'remote' or 'never'. For default defuse only remote XML data.

- **timeout** (*int*) – the timeout in seconds for fetching resources. Default is *300*.
- **build** (*bool*) – defines whether build the schema maps. Default is *True*.
- **use_meta** (*bool*) – if *True* the schema processor uses the package meta-schema, otherwise a new meta-schema is added at the end. In the latter case the meta-schema is rebuilt if any base namespace has been overridden by an import. Ignored if the argument *global_maps* is provided.
- **loglevel** (*int*) – for setting a different logging level for schema initialization and building. For default is WARNING (30). For INFO level set it with 20, for DEBUG level with 10. The default loglevel is restored after schema building, when exiting the initialization method.

Variables

- **XSD_VERSION** (*str*) – store the XSD version (1.0 or 1.1).
- **BUILDERS** (*namedtuple*) – a namedtuple with attributes related to schema components classes. Used for build local components within parsing methods.
- **BUILDERS_MAP** (*dict*) – a dictionary that maps from tag to class for XSD global components. Used for build global components within lookup functions.
- **BASE_SCHEMAS** (*dict*) – a dictionary from namespace to schema resource for meta-schema bases.
- **FALLBACK_LOCATIONS** (*dict*) – fallback schema location hints for other standard namespaces.
- **meta_schema** (*XMLSchema*) – the XSD meta-schema instance.
- **attribute_form_default** (*str*) – the schema's *attributeFormDefault* attribute, defaults to 'unqualified'.
- **element_form_default** (*str*) – the schema's *elementFormDefault* attribute, defaults to 'unqualified'
- **block_default** (*str*) – the schema's *blockDefault* attribute, defaults to ''.
- **final_default** (*str*) – the schema's *finalDefault* attribute, defaults to ''.
- **default_attributes** (XsdAttributeGroup) – the XSD 1.1 schema's *defaultAttributes* attribute, defaults to None.
- **xpath_tokens** (*dict*) – symbol table for schema bound XPath 2.0 parsers. Initially set to None it's redefined at instance level with a dictionary at first use of the XPath selector. The parser symbol table is extended with schema types constructors.
- **target_namespace** (*str*) – is the *targetNamespace* of the schema, the namespace to which belong the declarations/definitions of the schema. If it's empty no namespace is associated with the schema. In this case the schema declarations can be reused from other namespaces as *chameleon* definitions.
- **validation** (*str*) – validation mode, can be 'strict', 'lax' or 'skip'.
- **maps** (XsdGlobals) – XSD global declarations/definitions maps. This is an instance of XsdGlobal, that store the global_maps argument or a new object when this argument is not provided.
- **converter** (XMLSchemaConverter) – the default converter used for XML data decoding/encoding.
- **locations** (*NamespaceResourcesMap*) – schemas location hints.

- **namespaces** (*dict*) – a dictionary that maps from the prefixes used by the schema into namespace URI.
- **imports** (*dict*) – a dictionary of namespace imports of the schema, that maps namespace URI to imported schema object, or *None* in case of unsuccessful import.
- **includes** – a dictionary of included schemas, that maps a schema location to an included schema. It also comprehend schemas included by "xs:redefine" or "xs:override" statements.
- **warnings** (*list*) – warning messages about failure of import and include elements.
- **notations** (*NamespaceView*) – *xsd:notation* declarations.
- **types** (*NamespaceView*) – *xsd:simpleType* and *xsd:complexType* global declarations.
- **attributes** (*NamespaceView*) – *xsd:attribute* global declarations.
- **attribute_groups** (*NamespaceView*) – *xsd:attributeGroup* definitions.
- **groups** (*NamespaceView*) – *xsd:group* global definitions.
- **elements** (*NamespaceView*) – *xsd:element* global declarations.

root
Root element of the schema.

get_text()
Gets the XSD text of the schema. If the source text is not available creates an encoded string representation of the XSD tree.

url
Schema resource URL, is *None* if the schema is built from a string.

tag
Schema root tag. For compatibility with the ElementTree API.

id
The schema's *id* attribute, defaults to `None`.

version
The schema's *version* attribute, defaults to `None`.

schema_location
A list of location hints extracted from the *xsi:schemaLocation* attribute of the schema.

no_namespace_schema_location
A location hint extracted from the *xsi:noNamespaceSchemaLocation* attribute of the schema.

target_prefix
The prefix associated to the *targetNamespace*.

default_namespace
The namespace associated to the empty prefix ''.

base_url
The base URL of the source of the schema.

builtin_types = <bound method XMLSchemaBase.builtin_types of <class 'xmlschema.validat

root_elements
The list of global elements that are not used by reference in any model of the schema. This is implemented as lazy property because it's computationally expensive to build when the schema model is complex.

classmethod create_meta_schema(*source=None*, *base_schemas=None*, *global_maps=None*)
Creates a new meta-schema instance.

Parameters

- **source** – an optional argument referencing to or containing the XSD meta-schema resource. Required if the schema class doesn't already have a meta-schema.
- **base_schemas** – an optional dictionary that contains namespace URIs and schema locations. If provided it's used as substitute for class 's BASE_SCHEMAS. Also a sequence of (namespace, location) items can be provided if there are more schema documents for one or more namespaces.
- **global_maps** – is an optional argument containing an *XsdGlobals* instance for the new meta schema. If not provided a new map is created.

classmethod create_schema(**args*, ***kwargs*)
Creates a new schema instance of the same class of the caller.

create_any_content_group(*parent*, *any_element=None*)
Creates a model group related to schema instance that accepts any content.

Parameters

- **parent** – the parent component to set for the any content group.
- **any_element** – an optional any element to use for the content group. When provided it's copied, linked to the group and the minOccurs/maxOccurs are set to 0 and 'unbounded'.

create_any_attribute_group(*parent*)
Creates an attribute group related to schema instance that accepts any attribute.

Parameters **parent** – the parent component to set for the any attribute group.

create_any_type()
Creates an xs:anyType instance related to schema instance.

get_locations(*namespace*)
Get a list of location hints for a namespace.

include_schema(*location*, *base_url=None*)
Includes a schema for the same namespace, from a specific URL.

Parameters

- **location** – is the URL of the schema.
- **base_url** – is an optional base URL for fetching the schema resource.

Returns the included *XMLSchema* instance.

import_schema(*namespace*, *location*, *base_url=None*, *force=False*, *build=False*)
Imports a schema for an external namespace, from a specific URL.

Parameters

- **namespace** – is the URI of the external namespace.
- **location** – is the URL of the schema.
- **base_url** – is an optional base URL for fetching the schema resource.
- **force** – if set to *True* imports the schema also if the namespace is already imported.
- **build** – defines when to build the imported schema, the default is to not build.

Returns the imported *XMLSchema* instance.

resolve_qname(*qname*, *namespace_imported=True*)
QName resolution for a schema instance.

Parameters

- **qname** – a string in xs:QName format.
- **namespace_imported** – if this argument is *True* raises an *XMLSchemaNamespaceError* if the namespace of the QName is not the *targetNamespace* and the namespace is not imported by the schema.

Returns an expanded QName in the format "{*namespace-URI*}*local-name*".

Raises *XMLSchemaValueError* for an invalid xs:QName is found, *XMLSchemaKeyError* if the namespace prefix is not declared in the schema instance.

iter_globals(*schema=None*)
Creates an iterator for XSD global definitions/declarations related to schema namespace.

Parameters **schema** – Optional argument for filtering only globals related to a schema instance.

iter_components(*xsd_classes=None*)
Creates an iterator for traversing all XSD components of the validator.

Parameters **xsd_classes** – returns only a specific class/classes of components, otherwise returns all components.

classmethod check_schema(*schema*, *namespaces=None*)
Validates the given schema against the XSD meta-schema (`meta_schema`).

Parameters

- **schema** – the schema instance that has to be validated.
- **namespaces** – is an optional mapping from namespace prefix to URI.

Raises *XMLSchemaValidationError* if the schema is invalid.

build()
Builds the schema's XSD global maps.

clear()
Clears the schema's XSD global maps.

built
Property that is `True` if XSD validator has been fully parsed and built, `False` otherwise. For schemas the property is checked on all global components. For XSD components check only the building of local subcomponents.

validation_attempted
Property that returns the *validation status* of the XSD validator. It can be 'full', 'partial' or 'none'.

https://www.w3.org/TR/xmlschema-1/#e-validation_attempted
https://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/#e-validation_attempted

validity
Property that returns the XSD validator's validity. It can be 'valid', 'invalid' or 'notKnown'.

https://www.w3.org/TR/xmlschema-1/#e-validity

https://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/#e-validity

all_errors
A list with all the building errors of the XSD validator and its components.

get_converter(*converter=None*, *namespaces=None*, ***kwargs*)
Returns a new converter instance.

Parameters

- **converter** – can be a converter class or instance. If it's an instance the new instance is copied from it and configured with the provided arguments.
- **namespaces** – is an optional mapping from namespace prefix to URI.
- **kwargs** – optional arguments for initialize the converter instance.

Returns a converter instance.

validate(*source*, *path=None*, *schema_path=None*, *use_defaults=True*, *namespaces=None*)
Validates an XML data against the XSD schema/component instance.

Raises *XMLSchemaValidationError* if XML *data* instance is not a valid.

is_valid(*source*, *path=None*, *schema_path=None*, *use_defaults=True*, *namespaces=None*)
Like *validate()* except that do not raises an exception but returns `True` if the XML data is valid, `False` if it's invalid.

iter_errors(*source*, *path=None*, *schema_path=None*, *use_defaults=True*, *namespaces=None*)
Creates an iterator for the errors generated by the validation of an XML data against the XSD schema/component instance.

Parameters

- **source** – the source of XML data. Can be an *XMLResource* instance, a path to a file or an URI of a resource or an opened file-like object or an Element instance or an ElementTree instance or a string containing the XML data.
- **path** – is an optional XPath expression that matches the elements of the XML data that have to be decoded. If not provided the XML root element is selected.
- **schema_path** – an alternative XPath expression to select the XSD element to use for decoding. Useful if the root of the XML data doesn't match an XSD global element of the schema.
- **use_defaults** – Use schema's default values for filling missing data.
- **namespaces** – is an optional mapping from namespace prefix to URI.

decode(*source*, *path=None*, *schema_path=None*, *validation='strict'*, **args*, ***kwargs*)
Decodes XML data. Takes the same arguments of the method `XMLSchema.iter_decode()`.

iter_decode(*source*, *path=None*, *schema_path=None*, *validation='lax'*, *process_namespaces=True*, *namespaces=None*, *use_defaults=True*, *decimal_type=None*, *datetime_types=False*, *converter=None*, *filler=None*, *fill_missing=False*, *max_depth=None*, ***kwargs*)
Creates an iterator for decoding an XML source to a data structure.

Parameters

- **source** – the source of XML data. Can be an *XMLResource* instance, a path to a file or an URI of a resource or an opened file-like object or an Element instance or an ElementTree instance or a string containing the XML data.

- **path** – is an optional XPath expression that matches the elements of the XML data that have to be decoded. If not provided the XML root element is selected.
- **schema_path** – an alternative XPath expression to select the XSD element to use for decoding. Useful if the root of the XML data doesn't match an XSD global element of the schema.
- **validation** – defines the XSD validation mode to use for decode, can be 'strict', 'lax' or 'skip'.
- **process_namespaces** – indicates whether to use namespace information in the decoding process, using the map provided with the argument *namespaces* and the map extracted from the XML document.
- **namespaces** – is an optional mapping from namespace prefix to URI.
- **use_defaults** – indicates whether to use default values for filling missing data.
- **decimal_type** – conversion type for *Decimal* objects (generated by XSD *decimal* built-in and derived types), useful if you want to generate a JSON-compatible data structure.
- **datetime_types** – if set to *True* the datetime and duration XSD types are decoded, otherwise their origin XML string is returned.
- **converter** – an *XMLSchemaConverter* subclass or instance to use for the decoding.
- **filler** – an optional callback function to fill undecodable data with a typed value. The callback function must accepts one positional argument, that can be an XSD Element or an attribute declaration. If not provided undecodable data is replaced by *None*.
- **fill_missing** – if set to *True* the decoder fills also missing attributes. The filling value is *None* or a typed value if the *filler* callback is provided.
- **max_depth** – maximum level of decoding, for default there is no limit.
- **kwargs** – keyword arguments with other options for converter and decoder.

Returns yields a decoded data object, eventually preceded by a sequence of validation or decoding errors.

encode(*obj*, *path=None*, *validation='strict'*, **args*, ***kwargs*)

Encodes to XML data. Takes the same arguments of the method `XMLSchema.iter_encode()`.

Returns An ElementTree's Element or a list containing a sequence of ElementTree's elements if the argument *path* matches multiple XML data chunks. If *validation* argument is 'lax' a 2-items tuple is returned, where the first item is the encoded object and the second item is a list containing the errors.

iter_encode(*obj*, *path=None*, *validation='lax'*, *namespaces=None*, *converter=None*, *unordered=False*, ***kwargs*)

Creates an iterator for encoding a data structure to an ElementTree's Element.

Parameters

- **obj** – the data that has to be encoded to XML data.
- **path** – is an optional XPath expression for selecting the element of the schema that matches the data that has to be encoded. For default the first global element of the schema is used.
- **validation** – the XSD validation mode. Can be 'strict', 'lax' or 'skip'.
- **namespaces** – is an optional mapping from namespace prefix to URI.
- **converter** – an *XMLSchemaConverter* subclass or instance to use for the encoding.

- **unordered** – a flag for explicitly activating unordered encoding mode for content model data. This mode uses content models for a reordered-by-model iteration of the child elements.
- **kwargs** – Keyword arguments containing options for converter and encoding.

Returns yields an Element instance/s or validation/encoding errors.

3.3 XSD global maps API

class xmlschema.**XsdGlobals**(*validator*, *validation='strict'*)

Mediator class for related XML schema instances. It stores the global declarations defined in the registered schemas. Register a schema to add it's declarations to the global maps.

Parameters

- **validator** – the origin schema class/instance used for creating the global maps.
- **validation** – the XSD validation mode to use, can be 'strict', 'lax' or 'skip'.

build()

Build the maps of XSD global definitions/declarations. The global maps are updated adding and building the globals of not built registered schemas.

check(*schemas=None*, *validation='strict'*)

Checks the global maps. For default checks all schemas and raises an exception at first error.

Parameters

- **schemas** – optional argument with the set of the schemas to check.
- **validation** – overrides the default validation mode of the validator.

Raise XMLSchemaParseError

clear(*remove_schemas=False*, *only_unbuilt=False*)

Clears the instance maps and schemas.

Parameters

- **remove_schemas** – removes also the schema instances.
- **only_unbuilt** – removes only not built objects/schemas.

copy(*validator=None*, *validation=None*)

Makes a copy of the object.

iter_globals()

Creates an iterator for XSD global definitions/declarations.

iter_schemas()

Creates an iterator for the schemas registered in the instance.

lookup(*tag*, *qname*)

General lookup method for XSD global components.

Parameters

- **tag** – the expanded QName of the XSD the global declaration/definition (eg. '{http://www.w3.org/2001/XMLSchema}element'), that is used to select the global map for lookup.
- **qname** – the expanded QName of the component to be looked-up.

Returns an XSD global component.

Raises an XMLSchemaValueError if the *tag* argument is not appropriate for a global component, an XMLSchemaKeyError if the *qname* argument is not found in the global map.

register(*schema*)
Registers an XMLSchema instance.

unbuilt
Property that returns a list with unbuilt components.

3.4 XML Schema converters

The base class *XMLSchemaConverter* is used for defining generic converters. The subclasses implement some of the most used conventions for converting XML to JSON data.

class xmlschema.converters.**ElementData**(*tag*, *text*, *content*, *attributes*)
Namedtuple for Element data interchange between decoders and converters. The field *tag* is a string containing the Element's tag, *text* can be *None* or a string representing the Element's text, *content* can be *None*, a list containing the Element's children or a dictionary containing element name to list of element contents for the Element's children (used for unordered input data), *attributes* can be *None* or a dictionary containing the Element's attributes.

class xmlschema.**XMLSchemaConverter**(*namespaces=None*, *dict_class=None*, *list_class=None*, *etree_element_class=None*, *text_key='$'*, *attr_prefix='@'*, *cdata_prefix=None*, *indent=4*, *strip_namespaces=False*, *preserve_root=False*, *force_dict=False*, *force_list=False*, ***kwargs*)
Generic XML Schema based converter class. A converter is used to compose decoded XML data for an Element into a data structure and to build an Element from encoded data structure. There are two methods for interfacing the converter with the decoding/encoding process. The method *element_decode* accepts ElementData instance, containing the element parts, and returns a data structure. The method *element_encode* accepts a data structure and returns an ElementData that can be

Parameters

- **namespaces** – map from namespace prefixes to URI.
- **dict_class** – dictionary class to use for decoded data. Default is *dict*.
- **list_class** – list class to use for decoded data. Default is *list*.
- **etree_element_class** – the class that has to be used to create new XML elements, if not provided uses the ElementTree's Element class.
- **text_key** – is the key to apply to element's decoded text data.
- **attr_prefix** – controls the mapping of XML attributes, to the same name or with a prefix. If *None* the converter ignores attributes.
- **cdata_prefix** – is used for including and prefixing the character data parts of a mixed content, that are labeled with an integer instead of a string. Character data parts are ignored if this argument is *None*.
- **indent** – number of spaces for XML indentation (default is 4).
- **strip_namespaces** – if set to *True* removes namespace declarations from data and namespace information from names, during decoding or encoding. Defaults to *False*.

- **preserve_root** – if set to *True* the root element is preserved, wrapped into a single-item dictionary. Applicable only to default converter and to `ParkerConverter`.
- **force_dict** – if set to *True* complex elements with simple content are decoded with a dictionary also if there are no decoded attributes. Applicable to default converter only. Defaults to *False*.
- **force_list** – if set to *True* child elements are decoded within a list in any case. Applicable to default converter only. Defaults to *False*.

Variables

- **dict** – dictionary class to use for decoded data.
- **list** – list class to use for decoded data.
- **etree_element_class** – Element class to use
- **text_key** – key for decoded Element text
- **attr_prefix** – prefix for attribute names
- **cdata_prefix** – prefix for character data parts
- **indent** – indentation to use for rebuilding XML trees
- **strip_namespaces** – remove namespace information
- **preserve_root** – preserve the root element on decoding
- **force_dict** – force dictionary for complex elements with simple content
- **force_list** – force list for child elements

lossy
The converter ignores some kind of XML data during decoding/encoding.

lossless
The negation of *lossy* property, preserved for backward compatibility.

losslessly
The XML data is decoded without loss of quality, neither on data nor on data model shape. Only losslessly converters can be always used to encode to an XML data that is strictly conformant to the schema.

copy(***kwargs*)

map_attributes(*attributes*)
Creates an iterator for converting decoded attributes to a data structure with appropriate prefixes. If the instance has a not-empty map of namespaces registers the mapped URIs and prefixes.

Parameters **attributes** – A sequence or an iterator of couples with the name of the attribute and the decoded value. Default is *None* (for *simpleType* elements, that don't have attributes).

map_content(*content*)
A generator function for converting decoded content to a data structure. If the instance has a not-empty map of namespaces registers the mapped URIs and prefixes.

Parameters **content** – A sequence or an iterator of tuples with the name of the element, the decoded value and the *XsdElement* instance associated.

etree_element(*tag*, *text=None*, *children=None*, *attrib=None*, *level=0*)
Builds an ElementTree's Element using arguments and the element class and the indent spacing stored in the converter instance.

Parameters

- **tag** – the Element tag string.
- **text** – the Element text.
- **children** – the list of Element children/subelements.
- **attrib** – a dictionary with Element attributes.
- **level** – the level related to the encoding process (0 means the root).

Returns an instance of the Element class setted for the converter instance.

element_decode(*data*, *xsd_element*, *level=0*)
Converts a decoded element data to a data structure.

Parameters

- **data** – ElementData instance decoded from an Element node.
- **xsd_element** – the *XsdElement* associated to decoded the data.
- **level** – the level related to the decoding process (0 means the root).

Returns a data structure containing the decoded data.

element_encode(*obj*, *xsd_element*, *level=0*)
Extracts XML decoded data from a data structure for encoding into an ElementTree.

Parameters

- **obj** – the decoded object.
- **xsd_element** – the *XsdElement* associated to the decoded data structure.
- **level** – the level related to the encoding process (0 means the root).

Returns an ElementData instance.

map_qname(*qname*)
Converts an extended QName to the prefixed format. Only registered namespaces are mapped.

Parameters **qname** – a QName in extended format or a local name.

Returns a QName in prefixed format or a local name.

unmap_qname(*qname*, *name_table=None*)
Converts a QName in prefixed format or a local name to the extended QName format. Local names are converted only if a default namespace is included in the instance. If a *name_table* is provided a local name is mapped to the default namespace only if not found in the name table.

Parameters

- **qname** – a QName in prefixed format or a local name
- **name_table** – an optional lookup table for checking local names.

Returns a QName in extended format or a local name.

class xmlschema.**UnorderedConverter**(*namespaces=None*, *dict_class=None*, *list_class=None*, *etree_element_class=None*, *text_key='$'*, *attr_prefix='@'*, *cdata_prefix=None*, *indent=4*, *strip_namespaces=False*, *preserve_root=False*, *force_dict=False*, *force_list=False*, ***kwargs*)
Same as *XMLSchemaConverter* but `element_encode()` returns a dictionary for the content of the element, that can be used directly for unordered encoding mode. In this mode the order of the elements in the encoded output is based on the model visitor pattern rather than the order in which the elements were added

to the input dictionary. As the order of the input dictionary is not preserved, character data between sibling elements are interleaved between tags.

class xmlschema.**ParkerConverter**(*namespaces=None*, *dict_class=None*, *list_class=None*, *preserve_root=False*, ***kwargs*)

XML Schema based converter class for Parker convention.

ref: http://wiki.open311.org/JSON_and_XML_Conversion/#the-parker-convention ref: https://developer.mozilla.org/en-US/docs/Archive/JXON#The_Parker_Convention

Parameters

- **namespaces** – Map from namespace prefixes to URI.
- **dict_class** – Dictionary class to use for decoded data. Default is *dict* for Python 3.6+ or *OrderedDict* for previous versions.
- **list_class** – List class to use for decoded data. Default is *list*.
- **preserve_root** – If *True* the root element will be preserved. For default the Parker convention remove the document root element, returning only the value.

class xmlschema.**BadgerFishConverter**(*namespaces=None*, *dict_class=None*, *list_class=None*, ***kwargs*)

XML Schema based converter class for Badgerfish convention.

ref: http://www.sklar.com/badgerfish/ ref: http://badgerfish.ning.com/

Parameters

- **namespaces** – Map from namespace prefixes to URI.
- **dict_class** – Dictionary class to use for decoded data. Default is *dict* for Python 3.6+ or *OrderedDict* for previous versions.
- **list_class** – List class to use for decoded data. Default is *list*.

class xmlschema.**AbderaConverter**(*namespaces=None*, *dict_class=None*, *list_class=None*, ***kwargs*)

XML Schema based converter class for Abdera convention.

ref: http://wiki.open311.org/JSON_and_XML_Conversion/#the-abdera-convention ref: https://cwiki.apache.org/confluence/display/ABDERA/JSON+Serialization

Parameters

- **namespaces** – Map from namespace prefixes to URI.
- **dict_class** – Dictionary class to use for decoded data. Default is *dict* for Python 3.6+ or *OrderedDict* for previous versions.
- **list_class** – List class to use for decoded data. Default is *list*.

class xmlschema.**JsonMLConverter**(*namespaces=None*, *dict_class=None*, *list_class=None*, ***kwargs*)

XML Schema based converter class for JsonML (JSON Mark-up Language) convention.

ref: http://www.jsonml.org/ ref: https://www.ibm.com/developerworks/library/x-jsonml/

Parameters

- **namespaces** – Map from namespace prefixes to URI.
- **dict_class** – Dictionary class to use for decoded data. Default is *dict* for Python 3.6+ or *OrderedDict* for previous versions.
- **list_class** – List class to use for decoded data. Default is *list*.

3.5 Resource access API

class xmlschema.**XMLResource**(*source*, *base_url=None*, *defuse='remote'*, *timeout=300*, *lazy=True*)

XML resource reader based on ElementTree and urllib.

Parameters

- **source** – a string containing the XML document or file path or an URL or a file like object or an ElementTree or an Element.
- **base_url** – is an optional base URL, used for the normalization of relative paths when the URL of the resource can't be obtained from the source argument.
- **defuse** – set the usage of SafeXMLParser for XML data. Can be 'always', 'remote' or 'never'. Default is 'remote' that uses the defusedxml only when loading remote data.
- **timeout** – the timeout in seconds for the connection attempt in case of remote data.
- **lazy** – if a value *False* is provided the XML data is fully loaded into and processed from memory. For default only the root element of the source is loaded, except in case the *source* argument is an Element or an ElementTree instance.

root
The XML tree root Element.

document
The resource as ElementTree XML document. It's *None* if the instance is lazy or if it's an lxml Element.

text
The XML text source, *None* if it's not available.

url
The source URL, *None* if the instance is created from an Element tree or from a string.

base_url
The effective base URL used for completing relative locations.

namespace
The namespace of the XML resource.

copy(***kwargs*)
Resource copy method. Change init parameters with keyword arguments.

tostring(*indent=''*, *max_lines=None*, *spaces_for_tab=4*, *xml_declaration=False*)
Generates a string representation of the XML resource.

open()
Returns a opened resource reader object for the instance URL. If the source attribute is a seekable file-like object rewind the source and return it.

load()
Loads the XML text from the data source. If the data source is an Element the source XML text can't be retrieved.

is_lazy()
Returns *True* if the XML resource is lazy.

is_loaded()
Returns *True* if the XML text of the data source is loaded.

iter(*tag=None*)
XML resource tree iterator.

iter_location_hints(*root_only=False*)
Yields schema location hints from the XML resource.

Parameters **root_only** – if *True* yields only the location hints declared in the root element.

get_namespaces(*namespaces=None*, *root_only=False*)
Extracts namespaces with related prefixes from the XML resource. If a duplicate prefix declaration is encountered and the prefix maps a different namespace, adds the namespace using a different generated prefix. The empty prefix '' is used only if it's declared at root level to avoid erroneous mapping of local names. In other cases uses 'default' prefix as substitute.

Parameters

- **namespaces** – builds the namespace map starting over the dictionary provided.
- **root_only** – if *True* extracts only the namespaces declared in the root element.

Returns a dictionary for mapping namespace prefixes to full URI.

get_locations(*locations=None*, *root_only=False*)
Extracts a list of normalized schema location hints from the XML resource. The locations are normalized using the base URL of the instance.

Parameters

- **locations** – a dictionary or a list of namespace resources that is inserted before the schema location hints extracted from the XML resource.
- **root_only** – if *True* extracts only the location hints declared in the root element.

Returns a list of couples containing namespace location hints.

static defusing(*source*)
Defuse an XML source, raising an *ElementTree.ParseError* if the source contains entity definitions or remote entity loading.

Parameters **source** – a filename or file object containing XML data.

parse(*source*)
An equivalent of *ElementTree.parse()* that can protect from XML entities attacks. When protection is applied XML data are loaded and defused before building the ElementTree instance. The protection applied is based on value of *defuse* attribute and *base_url* property.

Parameters **source** – a filename or file object containing XML data.

Returns an ElementTree instance.

iterparse(*source*, *events=None*)
An equivalent of *ElementTree.iterparse()* that can protect from XML entities attacks. When protection is applied the iterator yields pure-Python Element instances. The protection applied is based on resource *defuse* attribute and *base_url* property.

Parameters

- **source** – a filename or file object containing XML data.
- **events** – a list of events to report back. If omitted, only "end" events are reported.

fromstring(*text*)
An equivalent of *ElementTree.fromstring()* that can protect from XML entities attacks. The protection applied is based on resource *defuse* attribute and *base_url* property.

Parameters **text** – a string containing XML data.

Returns the root Element instance.

xmlschema.**fetch_resource**(*location*, *base_url=None*, *timeout=30*)
Fetch a resource trying to accessing it. If the resource is accessible returns the URL, otherwise raises an error (XMLSchemaURLError).

Parameters

- **location** – an URL or a file path.
- **base_url** – reference base URL for normalizing local and relative URLs.
- **timeout** – the timeout in seconds for the connection attempt in case of remote data.

Returns a normalized URL.

xmlschema.**fetch_schema**(*source*, *locations=None*, *base_url=None*, *defuse='remote'*, *timeout=30*)
Like *fetch_schema_locations()* but returns only a reachable location hint for a schema related to the source's namespace.

xmlschema.**fetch_schema_locations**(*source*, *locations=None*, *base_url=None*, *defuse='remote'*, *timeout=30*)
Fetches schema location hints from an XML data source and a list of location hints. If an accessible schema location is not found raises a ValueError.

Parameters

- **source** – can be an *XMLResource* instance, a file-like object a path to a file or an URI of a resource or an Element instance or an ElementTree instance or a string containing the XML data. If the passed argument is not an *XMLResource* instance a new one is built using this and *defuse*, *timeout* and *lazy* arguments.
- **locations** – a dictionary or dictionary items with additional schema location hints.
- **base_url** – the same argument of the *XMLResource*.
- **defuse** – the same argument of the *XMLResource*.
- **timeout** – the same argument of the *XMLResource* but with a reduced default.

Returns A 2-tuple with the URL referring to the first reachable schema resource and a list of dictionary items with normalized location hints.

xmlschema.**load_xml_resource**(*source*, *element_only=True*, ***resource_options*)
Load XML data source into an Element tree, returning the root Element, the XML text and an url, if available. Usable for XML data files of small or medium sizes, as XSD schemas. This helper function is deprecated from v1.0.17, use *XMLResource* instead.

Parameters

- **source** – an URL, a filename path or a file-like object.
- **element_only** – if True the function returns only the root Element of the tree.
- **resource_options** – keyword arguments for providing *XMLResource* init options.

Returns a tuple with three items (root Element, XML text and XML URL) or only the root Element if 'element_only' argument is True.

xmlschema.**normalize_url**(*url*, *base_url=None*, *keep_relative=False*)
Returns a normalized URL doing a join with a base URL. URL scheme defaults to 'file' and backslashes are replaced with slashes. For file paths the os.path.join is used instead of urljoin.

Parameters

- **url** – a relative or absolute URL.

- **base_url** – the reference base URL for construct the normalized URL from the argument. For compatibility between "os.path.join" and "urljoin" a trailing '/' is added to not empty paths.
- **keep_relative** – if set to *True* keeps relative file paths, which would not strictly conformant to URL format specification.

Returns A normalized URL.

3.6 XSD components API

Note: For XSD components only methods included in the following documentation are considered part of the stable API, the others are considered internals that can be changed without forewarning.

3.6.1 XSD elements

class xmlschema.validators.**Xsd11Element**(*elem*, *schema*, *parent*)
Class for XSD 1.1 *element* declarations.

class xmlschema.validators.**XsdElement**(*elem*, *schema*, *parent*)
Class for XSD 1.0 *element* declarations.

Variables

- **type** – the XSD simpleType or complexType of the element.
- **attributes** – the group of the attributes associated with the element.

3.6.2 XSD attributes

class xmlschema.validators.**Xsd11Attribute**(*elem*, *schema*, *parent*)
Class for XSD 1.1 *attribute* declarations.

class xmlschema.validators.**XsdAttribute**(*elem*, *schema*, *parent*)
Class for XSD 1.0 *attribute* declarations.

Variables **type** – the XSD simpleType of the attribute.

3.6.3 XSD types

class xmlschema.validators.**XsdType**(*elem*, *schema*, *parent=None*, *name=None*)
Common base class for XSD types.

has_mixed_content()
Returns *True* if the instance is a complexType with mixed content, *False* otherwise.

has_simple_content()
Returns *True* if the instance is a simpleType or a complexType with simple content, *False* otherwise.

static is_atomic()
Returns *True* if the instance is an atomic simpleType, *False* otherwise.

static is_complex()
Returns *True* if the instance is a complexType, *False* otherwise.

is_element_only()
Returns *True* if the instance is a complexType with element-only content, *False* otherwise.

is_emptiable()
Returns *True* if the instance has an emptiable value or content, *False* otherwise.

is_empty()
Returns *True* if the instance has an empty value or content, *False* otherwise.

static is_simple()
Returns *True* if the instance is a simpleType, *False* otherwise.

class xmlschema.validators.**Xsd11ComplexType**(*elem*, *schema*, *parent*, *name=None*, ***kwargs*)
Class for XSD 1.1 *complexType* definitions.

class xmlschema.validators.**XsdComplexType**(*elem*, *schema*, *parent*, *name=None*, ***kwargs*)
Class for XSD 1.0 *complexType* definitions.

Variables

- **attributes** – the attribute group related with the type.
- **content_type** – the content type, that can be a model group or a simple type.
- **mixed** – if *True* the complex type has mixed content.

class xmlschema.validators.**XsdSimpleType**(*elem*, *schema*, *parent*, *name=None*, *facets=None*)
Base class for simpleTypes definitions. Generally used only for instances of xs:anySimpleType.

class xmlschema.validators.**XsdAtomicBuiltin**(*elem*, *schema*, *name*, *python_type*, *base_type=None*, *admitted_facets=None*, *facets=None*, *to_python=None*, *from_python=None*)
Class for defining XML Schema built-in simpleType atomic datatypes. An instance contains a Python's type transformation and a list of validator functions. The 'base_type' is not used for validation, but only for reference to the XML Schema restriction hierarchy.

Type conversion methods:

- to_python(value): Decoding from XML
- from_python(value): Encoding to XML

class xmlschema.validators.**XsdList**(*elem*, *schema*, *parent*, *name=None*)
Class for 'list' definitions. A list definition has an item_type attribute that refers to an atomic or union simpleType definition.

class xmlschema.validators.**Xsd11Union**(*elem*, *schema*, *parent*, *name=None*)

class xmlschema.validators.**XsdUnion**(*elem*, *schema*, *parent*, *name=None*)
Class for 'union' definitions. A union definition has a member_types attribute that refers to a 'simpleType' definition.

class xmlschema.validators.**Xsd11AtomicRestriction**(*elem*, *schema*, *parent*, *name=None*, *facets=None*, *base_type=None*)
Class for XSD 1.1 atomic simpleType and complexType's simpleContent restrictions.

class xmlschema.validators.**XsdAtomicRestriction**(*elem*, *schema*, *parent*, *name=None*, *facets=None*, *base_type=None*)
Class for XSD 1.0 atomic simpleType and complexType's simpleContent restrictions.

3.6.4 Attribute and model groups

class xmlschema.validators.**XsdAttributeGroup**(*elem*, *schema*, *parent*, *derivation=None*, *base_attributes=None*)
Class for XSD *attributeGroup* definitions.

class xmlschema.validators.**Xsd11Group**(*elem*, *schema*, *parent*)
Class for XSD 1.1 *model group* definitions.

class xmlschema.validators.**XsdGroup**(*elem*, *schema*, *parent*)
Class for XSD 1.0 *model group* definitions.

3.6.5 Wildcards

class xmlschema.validators.**Xsd11AnyElement**(*elem*, *schema*, *parent*)
Class for XSD 1.1 *any* declarations.

class xmlschema.validators.**XsdAnyElement**(*elem*, *schema*, *parent*)
Class for XSD 1.0 *any* wildcards.

class xmlschema.validators.**Xsd11AnyAttribute**(*elem*, *schema*, *parent=None*, *name=None*)
Class for XSD 1.1 *anyAttribute* declarations.

class xmlschema.validators.**XsdAnyAttribute**(*elem*, *schema*, *parent=None*, *name=None*)
Class for XSD 1.0 *anyAttribute* wildcards.

class xmlschema.validators.**XsdOpenContent**(*elem*, *schema*, *parent*)
Class for XSD 1.1 *openContent* model definitions.

class xmlschema.validators.**XsdDefaultOpenContent**(*elem*, *schema*)
Class for XSD 1.1 *defaultOpenContent* model definitions.

3.6.6 Identity constraints

class xmlschema.validators.**XsdIdentity**(*elem*, *schema*, *parent*)
Common class for XSD identity constraints.

Variables

- **selector** – the XPath selector of the identity constraint.
- **fields** – a list containing the XPath field selectors of the identity constraint.

class xmlschema.validators.**XsdSelector**(*elem*, *schema*, *parent*)
Class for defining an XPath selector for an XSD identity constraint.

class xmlschema.validators.**XsdFieldSelector**(*elem*, *schema*, *parent*)
Class for defining an XPath field selector for an XSD identity constraint.

class xmlschema.validators.**Xsd11Unique**(*elem*, *schema*, *parent*)

class xmlschema.validators.**XsdUnique**(*elem*, *schema*, *parent*)

class xmlschema.validators.**Xsd11Key**(*elem*, *schema*, *parent*)

class xmlschema.validators.**XsdKey**(*elem*, *schema*, *parent*)

class xmlschema.validators.**Xsd11Keyref**(*elem*, *schema*, *parent*)

class xmlschema.validators.**XsdKeyref**(*elem*, *schema*, *parent*)
Implementation of xs:keyref.

Variables **refer** – reference to a *xs:key* declaration that must be in the same element or in a descendant element.

3.6.7 Facets

class xmlschema.validators.**XsdFacet**(*elem*, *schema*, *parent*, *base_type*)
XML Schema constraining facets base class.

class xmlschema.validators.**XsdWhiteSpaceFacet**(*elem*, *schema*, *parent*, *base_type*)
XSD *whiteSpace* facet.

class xmlschema.validators.**XsdLengthFacet**(*elem*, *schema*, *parent*, *base_type*)
XSD *length* facet.

class xmlschema.validators.**XsdMinLengthFacet**(*elem*, *schema*, *parent*, *base_type*)
XSD *minLength* facet.

class xmlschema.validators.**XsdMaxLengthFacet**(*elem*, *schema*, *parent*, *base_type*)
XSD *maxLength* facet.

class xmlschema.validators.**XsdMinInclusiveFacet**(*elem*, *schema*, *parent*, *base_type*)
XSD *minInclusive* facet.

class xmlschema.validators.**XsdMinExclusiveFacet**(*elem*, *schema*, *parent*, *base_type*)
XSD *minExclusive* facet.

class xmlschema.validators.**XsdMaxInclusiveFacet**(*elem*, *schema*, *parent*, *base_type*)
XSD *maxInclusive* facet.

class xmlschema.validators.**XsdMaxExclusiveFacet**(*elem*, *schema*, *parent*, *base_type*)
XSD *maxExclusive* facet.

class xmlschema.validators.**XsdTotalDigitsFacet**(*elem*, *schema*, *parent*, *base_type*)
XSD *totalDigits* facet.

class xmlschema.validators.**XsdFractionDigitsFacet**(*elem*, *schema*, *parent*, *base_type*)
XSD *fractionDigits* facet.

class xmlschema.validators.**XsdExplicitTimezoneFacet**(*elem*, *schema*, *parent*, *base_type*)
XSD 1.1 *explicitTimezone* facet.

class xmlschema.validators.**XsdAssertionFacet**(*elem*, *schema*, *parent*, *base_type*)
XSD 1.1 *assertion* facet for simpleType definitions.

class xmlschema.validators.**XsdEnumerationFacets**(*elem*, *schema*, *parent*, *base_type*)
Sequence of XSD *enumeration* facets. Values are validates if match any of enumeration values.

class xmlschema.validators.**XsdPatternFacets**(*elem*, *schema*, *parent*, *base_type*)
Sequence of XSD *pattern* facets. Values are validates if match any of patterns.

3.6.8 Other XSD components

class xmlschema.validators.**XsdAssert**(*elem*, *schema*, *parent*, *base_type*)
Class for XSD *assert* constraint definitions.

class xmlschema.validators.**XsdAlternative**(*elem*, *schema*, *parent*)
XSD 1.1 type *alternative* definitions.

class xmlschema.validators.**XsdNotation**(*elem*, *schema*, *parent=None*, *name=None*)
Class for XSD *notation* declarations.

class xmlschema.validators.**XsdAnnotation**(*elem*, *schema*, *parent=None*, *name=None*)
Class for XSD *annotation* definitions.

Variables

- **appinfo** – a list containing the xs:appinfo children.
- **documentation** – a list containing the xs:documentation children.

3.6.9 XSD Validation API

This API is implemented for XSD schemas, elements, attributes, types, attribute groups and model groups.

class xmlschema.validators.**ValidationMixin**
Mixin for implementing XML data validators/decoders. A derived class must implement the methods *iter_decode* and *iter_encode*.

is_valid(*source*, *use_defaults=True*, *namespaces=None*)
Like *validate()* except that do not raises an exception but returns `True` if the XML document is valid, `False` if it's invalid.

Parameters

- **source** – the source of XML data. For a schema can be a path to a file or an URI of a resource or an opened file-like object or an Element Tree instance or a string containing XML data. For other XSD components can be a string for an attribute or a simple type validators, or an ElementTree's Element otherwise.
- **use_defaults** – indicates whether to use default values for filling missing data.
- **namespaces** – is an optional mapping from namespace prefix to URI.

validate(*source*, *use_defaults=True*, *namespaces=None*)
Validates an XML data against the XSD schema/component instance.

Parameters

- **source** – the source of XML data. For a schema can be a path to a file or an URI of a resource or an opened file-like object or an Element Tree instance or a string containing XML data. For other XSD components can be a string for an attribute or a simple type validators, or an ElementTree's Element otherwise.
- **use_defaults** – indicates whether to use default values for filling missing data.
- **namespaces** – is an optional mapping from namespace prefix to URI.

Raises `XMLSchemaValidationError` if XML *data* instance is not a valid.

decode(*source*, *validation='strict'*, ***kwargs*)
Decodes XML data.

Parameters

- **source** – the XML data. Can be a string for an attribute or for a simple type components or a dictionary for an attribute group or an ElementTree's Element for other components.
- **validation** – the validation mode. Can be 'lax', 'strict' or 'skip.
- **kwargs** – optional keyword arguments for the method *iter_decode()*.

Returns a dictionary like object if the XSD component is an element, a group or a complex type; a list if the XSD component is an attribute group; a simple data type object otherwise. If *validation* argument is 'lax' a 2-items tuple is returned, where the first item is the decoded object and the second item is a list containing the errors.

Raises `XMLSchemaValidationError` if the object is not decodable by the XSD component, or also if it's invalid when `validation='strict'` is provided.

iter_decode(*source*, *validation='lax'*, ***kwargs*)
Creates an iterator for decoding an XML source to a Python object.

Parameters

- **source** – the XML data source.
- **validation** – the validation mode. Can be 'lax', 'strict' or 'skip.
- **kwargs** – keyword arguments for the decoder API.

Returns Yields a decoded object, eventually preceded by a sequence of validation or decoding errors.

iter_encode(*obj*, *validation='lax'*, ***kwargs*)
Creates an iterator for Encode data to an Element.

Parameters

- **obj** – The data that has to be encoded.
- **validation** – The validation mode. Can be 'lax', 'strict' or 'skip'.
- **kwargs** – keyword arguments for the encoder API.

Returns Yields an Element, eventually preceded by a sequence of validation or encoding errors.

iter_errors(*source*, *use_defaults=True*, *namespaces=None*)
Creates an iterator for the errors generated by the validation of an XML data against the XSD schema/component instance.

Parameters

- **source** – the source of XML data. For a schema can be a path to a file or an URI of a resource or an opened file-like object or an Element Tree instance or a string containing XML data. For other XSD components can be a string for an attribute or a simple type validators, or an ElementTree's Element otherwise.
- **use_defaults** – Use schema's default values for filling missing data.
- **namespaces** – is an optional mapping from namespace prefix to URI.

encode(*obj*, *validation='strict'*, ***kwargs*)
Encodes data to XML.

Parameters

- **obj** – the data to be encoded to XML.
- **validation** – the validation mode. Can be 'lax', 'strict' or 'skip.
- **kwargs** – optional keyword arguments for the method *iter_encode()*.

Returns An element tree's Element if the original data is a structured data or a string if it's simple type datum. If *validation* argument is 'lax' a 2-items tuple is returned, where the first item is the encoded object and the second item is a list containing the errors.

Raises `XMLSchemaValidationError` if the object is not encodable by the XSD component, or also if it's invalid when `validation='strict'` is provided.

iter_encode(*obj*, *validation='lax'*, ***kwargs*)
Creates an iterator for Encode data to an Element.

Parameters

- **obj** – The data that has to be encoded.
- **validation** – The validation mode. Can be 'lax', 'strict' or 'skip'.
- **kwargs** – keyword arguments for the encoder API.

Returns Yields an Element, eventually preceded by a sequence of validation or encoding errors.

3.6.10 ElementTree and XPath API

This API is implemented for XSD schemas, elements and complexType's assertions.

class xmlschema.**ElementPathMixin**
Mixin abstract class for enabling ElementTree and XPath API on XSD components.

Variables

- ***text*** – The Element text. Its value is always *None*. For compatibility with the ElementTree API.
- **tail** – The Element tail. Its value is always *None*. For compatibility with the ElementTree API.

tag
Alias of the *name* attribute. For compatibility with the ElementTree API.

attrib
Returns the Element attributes. For compatibility with the ElementTree API.

get(*key*, *default=None*)
Gets an Element attribute. For compatibility with the ElementTree API.

iter(*tag=None*)
Creates an iterator for the XSD element and its subelements. If tag is not *None* or '*', only XSD elements whose matches tag are returned from the iterator. Local elements are expanded without repetitions. Element references are not expanded because the global elements are not descendants of other elements.

iterchildren(*tag=None*)
Creates an iterator for the child elements of the XSD component. If *tag* is not *None* or '*', only XSD elements whose name matches tag are returned from the iterator.

find(*path*, *namespaces=None*)
Finds the first XSD subelement matching the path.

Parameters

- **path** – an XPath expression that considers the XSD component as the root element.
- **namespaces** – an optional mapping from namespace prefix to namespace URI.

Returns The first matching XSD subelement or `None` if there is not match.

findall(*path*, *namespaces=None*)
Finds all XSD subelements matching the path.

Parameters

- **path** – an XPath expression that considers the XSD component as the root element.
- **namespaces** – an optional mapping from namespace prefix to full name.

Returns a list containing all matching XSD subelements in document order, an empty list is returned if there is no match.

iterfind(*path*, *namespaces=None*)
Creates and iterator for all XSD subelements matching the path.

Parameters

- **path** – an XPath expression that considers the XSD component as the root element.
- **namespaces** – is an optional mapping from namespace prefix to full name.

Returns an iterable yielding all matching XSD subelements in document order.

3.7 Errors and exceptions

exception xmlschema.**XMLSchemaException**
The base exception that let you catch all the errors generated by the library.

exception xmlschema.**XMLSchemaRegexError**
Raised when an error is found when parsing an XML Schema regular expression.

exception xmlschema.**XMLSchemaValidatorError**(*validator*, *message*, *elem=None*, *source=None*, *namespaces=None*)
Base class for XSD validator errors.

Parameters

- **validator** (*XsdValidator or function*) – the XSD validator.
- **message** (*str or unicode*) – the error message.
- **elem** (*Element*) – the element that contains the error.
- **source** (XMLResource) – the XML resource that contains the error.
- **namespaces** (*dict*) – is an optional mapping from namespace prefix to URI.

Variables **path** – the XPath of the element, calculated when the element is set or the XML resource is set.

exception xmlschema.**XMLSchemaNotBuiltError**(*validator*, *message*)
Raised when there is an improper usage attempt of a not built XSD validator.

Parameters

- **validator** (*XsdValidator*) – the XSD validator.
- **message** (*str or unicode*) – the error message.

exception xmlschema.**XMLSchemaParseError**(*validator*, *message*, *elem=None*)
Raised when an error is found during the building of an XSD validator.

Parameters

- **validator** (*XsdValidator or function*) – the XSD validator.
- **message** (*str or unicode*) – the error message.
- **elem** (*Element*) – the element that contains the error.

exception xmlschema.**XMLSchemaModelError**(*group*, *message*)
Raised when a model error is found during the checking of a model group.

Parameters

- **group** (XsdGroup) – the XSD model group.
- **message** (*str or unicode*) – the error message.

exception xmlschema.**XMLSchemaModelDepthError**(*group*)
Raised when recursion depth is exceeded while iterating a model group.

exception xmlschema.**XMLSchemaValidationError**(*validator*, *obj*, *reason=None*, *source=None*, *namespaces=None*)
Raised when the XML data is not validated with the XSD component or schema. It's used by decoding and encoding methods. Encoding validation errors do not include XML data element and source, so the error is limited to a message containing object representation and a reason.

Parameters

- **validator** (*XsdValidator or function*) – the XSD validator.
- **obj** (*Element or tuple or str or list or int or float or bool*) – the not validated XML data.
- **reason** (*str or unicode*) – the detailed reason of failed validation.
- **source** (XMLResource) – the XML resource that contains the error.
- **namespaces** (*dict*) – is an optional mapping from namespace prefix to URI.

exception xmlschema.**XMLSchemaDecodeError**(*validator*, *obj*, *decoder*, *reason=None*, *source=None*, *namespaces=None*)
Raised when an XML data string is not decodable to a Python object.

Parameters

- **validator** (*XsdValidator or function*) – the XSD validator.
- **obj** (*Element or tuple or str or list or int or float or bool*) – the not validated XML data.
- **decoder** (*type or function*) – the XML data decoder.
- **reason** (*str or unicode*) – the detailed reason of failed validation.
- **source** (XMLResource) – the XML resource that contains the error.
- **namespaces** (*dict*) – is an optional mapping from namespace prefix to URI.

exception xmlschema.**XMLSchemaEncodeError**(*validator*, *obj*, *encoder*, *reason=None*, *source=None*, *namespaces=None*)
Raised when an object is not encodable to an XML data string.

Parameters

- **validator** (*XsdValidator or function*) – the XSD validator.
- **obj** (*Element or tuple or str or list or int or float or bool*) – the not validated XML data.
- **encoder** (*type or function*) – the XML encoder.
- **reason** (*str or unicode*) – the detailed reason of failed validation.
- **source** (XMLResource) – the XML resource that contains the error.
- **namespaces** (*dict*) – is an optional mapping from namespace prefix to URI.

exception xmlschema.**XMLSchemaChildrenValidationError**(*validator*, *elem*, *index*, *particle*, *occurs=0*, *expected=None*, *source=None*, *namespaces=None*)

Raised when a child element is not validated.

Parameters

- **validator** (*XsdValidator or function*) – the XSD validator.
- **elem** (*Element or ElementData*) – the not validated XML element.
- **index** (*int*) – the child index.
- **particle** (*ParticleMixin*) – the validator particle that generated the error. Maybe the validator itself.
- **occurs** (*int*) – the particle occurrences.
- **expected** (*str or list or tuple*) – the expected element tags/object names.
- **source** (*XMLResource*) – the XML resource that contains the error.
- **namespaces** (*dict*) – is an optional mapping from namespace prefix to URI.

exception xmlschema.**XMLSchemaIncludeWarning**

A schema include fails.

exception xmlschema.**XMLSchemaImportWarning**

A schema namespace import fails.

exception xmlschema.**XMLSchemaTypeTableWarning**

Not equivalent type table found in model.

CHAPTER 4

Customizing output data with converters

XML data decoding and encoding is handled using an intermediate converter class instance that takes charge of composing inner data and mapping of namespaces and prefixes.

Because XML is a structured format that includes data and metadata information, as attributes and namespace declarations, is necessary to define conventions for naming the different data objects in a distiguishable way. For example a wide-used convention is to prefixing attribute names with an '@' character. With this convention the attribute *name='John'* is decoded to *'@name': 'John'*, or *'level='10'* is decoded to *'@level': 10*.

A related topic is the mapping of namespaces. The expanded namespace representation is used within XML objects of the ElementTree library. For example *{http://www.w3.org/2001/XMLSchema}string* is the fully qualified name of the XSD string type, usually referred as *xs:string* or *xsd:string* with a namespace declaration. With string serialization of XML data the names are remapped to prefixed format. This mapping is generally useful also if you serialize XML data to another format like JSON, because prefixed name is more manageable and readable than expanded format.

4.1 Available converters

The library includes some converters. The default converter `XMLSchemaConverter` is the base class of other converter types. Each derived converter type implements a well know convention, related to the conversion from XML to JSON data format:

- `ParkerConverter`: Parker convention
- `BadgerFishConverter`: BadgerFish convention
- `AbderaConverter`: Apache Abdera project convention
- `JsonMLConverter`: JsonML (JSON Mark-up Language) convention

A summary of these and other conventions can be found on the wiki page JSON and XML Conversion.

The base class, that not implements any particular convention, has several options that can be used to variate the converting process. Some of these options are not used by other predefined converter types (eg. *force_list* and *force_dict*) or are used with a fixed value (eg. *text_key* or *attr_prefix*). See *XML Schema converters* for details about base class options and attributes.

4.2 Create a custom converter

To create a new customized converter you have to subclass the `XMLSchemaConverter` and redefine the two methods *element_decode* and *element_encode*. These methods are based on the namedtuple *ElementData*, an Element-like data structure that stores the decoded Element parts. This namedtuple is used by decoding and encoding methods as an intermediate data structure.

The namedtuple *ElementData* has four attributes:

- **tag**: the element's tag string;
- **text**: the element's text, that can be a string or *None* for empty elements;
- **content**: the element's children, can be a list or *None*;
- **attributes**: the element's attributes, can be a dictionary or *None*.

The method *element_decode* receives as first argument an *ElementData* instance with decoded data. The other arguments are the XSD element to use for decoding and the level of the XML decoding process, used to add indent spaces for a readable string serialization. This method uses the input data element to compose a decoded data, typically a dictionary or a list or a value for simple type elements.

On the opposite the method *element_encode* receives the decoded object and decompose it in order to get and returns an *ElementData* instance. This instance has to contain the parts of the element that will be then encoded an used to build an XML Element instance.

These two methods have also the responsibility to map and unmap object names, but don't have to decode or encode data, a task that is delegated to the methods of the XSD components.

Depending on the format defined by your new converter class you may provide a different value for properties *lossless* and *losslessly*. The *lossless* has to be *True* if your new converter class preserves all XML data information (eg. as the *BadgerFish* convention). Your new converter can be also *losslessly* if it's lossless and the element model structure and order is maintained (like the JsonML convention).

Furthermore your new converter class can has a more specific *__init__* method in order to avoid the usage of unused options or to set the value of some other options. Finally refer also to the code of predefined derived converters to see how you can build your own one.

CHAPTER 5

Testing

5.1 Test scripts

The tests of the *xmlschema* library are implemented using the Python's *unitest* library. The test scripts are located under the installation base into `tests/` subdirectory. There are several test scripts, each one for a different topic:

test_helpers.py Tests for helper functions and classes

test_meta.py Tests for the XSD meta-schema and XSD builtins

test_models.py Tests concerning model groups validation

test_package.py Tests regarding ElementTree import and code packaging

test_regex.py Tests about XSD regular expressions

test_resources.py Tests about XML/XSD resources access

test_schemas.py Tests about parsing of XSD schemas and components

test_validators.py Tests regarding XML data validation/decoding/encoding

test_xpath.py Tests for XPath parsing and selectors

You can run all above tests with the script *test_all.py*. From the project source base, if you have the *tox automation tool* installed, you can run all tests with all supported Python's versions using the command `tox`.

5.2 Test cases based on files

Two scripts (*test_schemas.py*, *test_validators.py*) create the most tests dinamically, loading a set of XSD or XML files. Only a small set of test files is published in the repository for copyright reasons. You can found the published test files into `xmlschema/tests/test_cases/` subdirectory.

You can locally extend the test with your set of files. For doing this create a `test_cases/` directory at repository level and then copy your XSD/XML files into it. Finally you have to create a file called *testfiles* in your `test_cases/` directory:

```
cd test_cases/
touch testfiles
```

Fill this file with the list of paths of files you want to be tested, one per line, as in the following example:

```
# XHTML
XHTML/xhtml11-mod.xsd
XHTML/xhtml-datatypes-1.xsd

# Quantum Espresso
qe/qes.xsd
qe/qes_neb.xsd
qe/qes_with_choice_no_nesting.xsd
qe/silicon.xml
qe/silicon-1_error.xml --errors 1
qe/silicon-3_errors.xml --errors=3
qe/SrTiO_3.xml
qe/SrTiO_3-2_errors.xml --errors 2
```

The test scripts create a test for each listed file, dependant from the context. For example the script that test the schemas uses only *.xsd* files, where instead the script that tests the validation uses both types, validating each XML file against its schema and each XSD against the meta-schema.

If a file has errors insert an integer number after the path. This is the number of errors that the XML Schema validator have to found to pass the test.

From version 1.0.0 each test-case line is parsed for those additional arguments:

-L URI URL Schema location hint overrides.

–version=VERSION XSD schema version to use for the test case (default is 1.0).

–errors=NUM Number of errors expected (default=0).

–warnings=NUM Number of warnings expected (default=0).

–inspect Inspect using an observed custom schema class.

–defuse=(always, remote, never) Define when to use the defused XML data loaders.

–timeout=SEC Timeout for fetching resources (default=300).

–skip Skip strict encoding checks (for cases where test data uses default or fixed values or some test data are skipped by wildcards processContents).

–debug Activate the debug mode (only the cases with *–debug* are executed).

If you put a `--help` on the first case line the argument parser show you all the options available.

Note: Test case line options are changed from version 1.0.0, with the choice of using almost only double dash prefixed options, in order to simplify text search in long *testfiles*, and add or remove options without the risk to change also parts of filepaths.

To run tests with also your personal set of files you have to add a `-x/--extra` option to the command, for example:

```
python xmlschema/tests/test_all.py -x
```

or:

```
tox -- -x
```

5.3 Testing with the W3C XML Schema 1.1 test suite

From release v1.0.11, using the script *test_w3c_suite.py*, you can run also tests based on the W3C XML Schema 1.1 test suite. To run these tests clone the W3C repo on the project's parent directory and than run the script:

```
git clone https://github.com/w3c/xsdtests.git
python xmlschema/xmlschema/tests/test_w3c_suite.py
```

You can also provides additional options for select a different set of tests:

–xml Add tests for instances, skipped for default.

–xsd10 Run only XSD 1.0 tests.

–xsd11 Run only XSD 1.1 tests.

–valid Run only tests signed as *valid*.

–invalid Run only tests signed as *invalid*.

[NUM [NUM . . .]] Run only the cases that match a list of progressive numbers, associated to the test classes by the script.

5.4 Testing other schemas and instances

From release v1.0.12, using the script *test_files.py*, you can test schemas or XML instances passing them as arguments:

```
$ cd xmlschema/tests/
$ python test_files.py test_cases/examples/vehicles/*.xsd
Add test 'TestSchema001' for file 'test_cases/examples/vehicles/bikes.xsd' ...
Add test 'TestSchema002' for file 'test_cases/examples/vehicles/cars.xsd' ...
Add test 'TestSchema003' for file 'test_cases/examples/vehicles/types.xsd' ...
Add test 'TestSchema004' for file 'test_cases/examples/vehicles/vehicles-max.xsd' ...
Add test 'TestSchema005' for file 'test_cases/examples/vehicles/vehicles.xsd' ...
.....
----------------------------------------------------------------------
Ran 5 tests in 0.147s

OK
```

CHAPTER 6

Release notes

6.1 License

The *xmlschema* library is distributed under the terms of the MIT License.

6.2 Support

The project is hosted on GitHub, refer to the xmlschema's project page for source code and for an issue tracker.

Index

S

T

U

V

X