xmlschema Documentation
Release 1.0.7

Davide Brunato

Nov 15, 2018
The `xmlschema` library is an implementation of XML Schema for Python (supports Python 2.7 and Python 3.4+). 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 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
- XML-based attacks prevention using the external package `defusedxml`
1.2 Installation

You can install the library with `pip` in a Python 2.7 or Python 3.4+ environment:

```
pip install xmlschema
```

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

Import the library in your code with:

```python
import xmlschema
```

The module initialization builds the XSD meta-schemas and of 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:

```python
>>> import xmlschema

>>> schema = xmlschema.XMLSchema('xmlschema/tests/cases/examples/vehicles/vehicles.xsd')
```

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

```python
>>> import xmlschema

>>> schema_file = open('xmlschema/tests/cases/examples/vehicles/vehicles.xsd')

>>> schema = xmlschema.XMLSchema(schema_file)
```

Alternatively you can pass a string containing the schema definition:

```python
>>> import xmlschema

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

This option might not work when the schema includes other local subschemas, because the package cannot know anything about the schema’s source location:
>>> import xmlschema
>>> schema_xsd = open('xmlschema/tests/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" />

2.2 XSD declarations

The schema object includes XSD declarations (notations, types, elements, attributes, groups, attribute_groups, substitution_groups). The global XSD declarations are available as attributes of the schema instance:

>>> import xmlschema
>>> from pprint import pprint

>>> schema = xmlschema.XMLSchema('xmlschema/tests/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')})

Those declarations 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/1999/xlink}actuateType',
 '{http://www.w3.org/1999/xlink}arcType',
 '{http://www.w3.org/1999/xlink}arcroleType',
 '{http://www.w3.org/1999/xlink}extended']

>>> pprint(sorted(schema.maps.elements.keys())[:10])
['{http://example.com/vehicles}bikes',
 '{http://example.com/vehicles}cars',
 '{http://example.com/vehicles}vehicles',
 '{http://www.w3.org/1999/xlink}arc',
 '{http://www.w3.org/1999/xlink}locator',
 '{http://www.w3.org/1999/xlink}resource',
 '{http://www.w3.org/1999/xlink}title',
 '{http://www.w3.org/2001/XMLSchema-hasFacetAndProperty}hasFacet',
 '{http://www.w3.org/2001/XMLSchema-hasFacetAndProperty}hasProperty',
 '{http://www.w3.org/2001/XMLSchema}all']
Schema objects include methods for finding XSD elements and attributes in the schema. Those are methods of the ElementTree’s API, so you can use an XPath expression for defining the search criteria:

```python
>>> 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.

```python
>>> import xmlschema
>>> schema = xmlschema.XMLSchema('xmlschema/tests/cases/examples/vehicles/vehicles.xsd')
>>> schema.is_valid('xmlschema/tests/cases/examples/vehicles/vehicles.xml')
True
>>> schema.is_valid('xmlschema/tests/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:

```python
>>> import xmlschema
>>> schema = xmlschema.XMLSchema('xmlschema/tests/cases/examples/vehicles/vehicles.xsd')
>>> schema.is_valid('xmlschema/tests/cases/examples/vehicles/vehicles.xml')
月底 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 ...
Reason: character data between child elements not allowed!
```

Schema:

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

Instance:

```xml
<ns0:cars xmlns:ns0="http://example.com/vehicles">
  NOT ALLOWED CHARACTER DATA
  <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:

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

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

## 2.4 Data decoding and encoding

Each schema component includes methods for data conversion:

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

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

```python
>>> import xmlschema
>>> from pprint import pprint
>>> from xml.etree import ElementTree

>>> xs = xmlschema.XMLSchema('xmlschema/tests/cases/examples/vehicles/vehicles.xsd')
>>> xt = ElementTree.parse('xmlschema/tests/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'}))

{'@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:

```python
>>> import xmlschema
>>> from pprint import pprint

>>> import xmlschema
>>> from pprint import pprint
>>> from xml.etree import ElementTree

>>> xs = xmlschema.XMLSchema('xmlschema/tests/cases/examples/vehicles/vehicles.xsd')

>>> pprint(xs.to_dict('xmlschema/tests/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'}]}}
```

(continues on next page)
The decoded values coincide with the datatypes declared in the XSD schema:

```python
>>> import xmlschema
>>> from pprint import pprint
>>> xs = xmlschema.XMLSchema('xmlschema/tests/cases/examples/collection/collection.xsd')
>>> pprint(xs.to_dict('xmlschema/tests/cases/examples/collection/collection.xml'))
{'@xmlns:col': 'http://example.com/ns/collection', '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'},
{'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.

```python
>>> xs = xmlschema.XMLSchema('xmlschema/tests/cases/examples/vehicles/vehicles.xsd')
>>> pprint(xs.to_dict('xmlschema/tests/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:

```python
>>> import xmlschema
>>> from pprint import pprint
>>> from xml.etree import ElementTree

>>> xs = xmlschema.XMLSchema('xmlschema/tests/cases/examples/vehicles/vehicles.xsd')
>>> xt = ElementTree.parse('xmlschema/tests/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://...'}

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

```python
>>> namespaces = {'xsi': 'http://www.w3.org/2001/XMLSchema-instance', 'vh': 'http://example.com/vehicles'}
>>> pprint(xs.to_dict(xt, namespaces=namespaces))
```

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

```python
>>> import xmlschema
>>> from pprint import pprint
>>> import lxml.etree as ElementTree

>>> xs = xmlschema.XMLSchema('xmlschema/tests/cases/examples/vehicles/vehicles.xsd')
>>> xt = ElementTree.parse('xmlschema/tests/cases/examples/vehicles/vehicles.xml')
>>> xs.is_valid(xt)
True
>>> pprint(xs.to_dict(xt))

```
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:

```python
>>> import xmlschema
>>> from pprint import pprint

>>> xml_schema = 'xmlschema/tests/cases/examples/vehicles/vehicles.xsd'
>>> xml_document = 'xmlschema/tests/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:

```python
>>> 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 XML Schema 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:

```python
>>> import xmlschema
>>> import json

>>> xml_document = 'xmlschema/tests/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)
 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
```

(continues on next page)
chunks = list(chunks)

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

```python
>>> 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",
            "position": 1,
            "estimation": "10000.00",
            "@id": "b0836217462"
        },
        {
            "@available": true,
            "author": {
                "qualification": "painter, sculptor and ceramicist",
                "born": "1893-04-20",
                "@id": "JM",
                "name": "Joan Miró",
                "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 XML attacks prevention

Starting from the release 0.9.27 the XML data loading is protected using the defusedxml package. 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’.
CHAPTER 3

API Documentation

3.1 Document level API

xmlschema.validate(xml_document, schema=None, cls=None, use_defaults=True, namespaces=None, locations=None, base_url=None)

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 a file-like object or a string containing the XML data or a file path or a URL of a resource or an ElementTree/Element instance.
• 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 – schema class to use for building the instance (for default uses XMLSchema).
• use_defaults – defines when to use elements 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.

xmlschema.to_dict(xml_document, schema=None, cls=None, path=None, process_namespaces=True, locations=None, base_url=None, **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 a file-like object or a string containing the XML data or a file path or a URL of a resource or an ElementTree/Element instance.

• **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** – schema class to use for building the instance (for default uses `XMLSchema`).

• **path** – is an optional XPath expression that matches the subelement of the document that have to be decoded. The XPath expression considers the schema as the root element with global elements as its children.

• **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.

• **kwargs** – optional arguments of `XMLSchema.iter_decode()` as keyword arguments to variate the decoding process.

**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.

```python
xmlschema.to_json(xml_document, fp=None, schema=None, cls=None, path=None, converter=None, process_namespaces=True, locations=None, base_url=None, 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 a file-like object or a string containing the XML data or a file path or an URI of a resource or an ElementTree/Element instance.

• **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 subelement of the document that have to be decoded. The XPath expression considers the schema as the root element with global elements as its children.

• **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.

• **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.

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

Deserializes 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` 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**

The class for XSD v1.0 schema instances. It’s generated by the meta-class `XMLSchemaMeta` and takes the same API of `XMLSchemaBase`.

**xmlschema.XMLSchema**

The default class for schema instances.

```python
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)

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 for namespace imports. 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.

**Variables**

• **XSD_VERSION**(str) – store the XSD version (1.0 or 1.1).

• **meta_schema**(XMLSchema) – the XSD meta-schema instance.

• **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) – schema location hints.

• **namespaces**(list) – a dictionary that maps from the prefixes used by the schema into namespace URI.

• **warnings** – 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.

attribute_form_default
    The schema’s attributeFormDefault attribute, defaults to 'unqualified'

element_form_default
    The schema’s elementFormDefault attribute, defaults to 'unqualified'.

block_default
    The schema’s blockDefault attribute, defaults to None.

final_default
    The schema’s finalDefault 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.

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.

builtin_types = <bound method XMLSchemaBase.builtin_types of <class 'xmlschema.validators.schema.XMLSchemaBase'>

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)
    Imports a schema for an external namespace, from a specific URL.

3.2. Schema level API
xmlschema Documentation, Release 1.0.7

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** – is set to `True` imports the schema also if the namespace is already imported.

Returns the imported `XMLSchema` instance.

```python
classmethod create_schema(*args, **kwargs)
```

Creates a new schema instance of the same class of the caller.

```python
classmethod create_any_content_group(parent, name=None)
```

Creates a model group related to schema instance that accepts any content.

```python
classmethod create_any_attribute_group(parent, name=None)
```

Creates an attribute group related to schema instance that accepts any attribute.

```python
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.

```python
build()
```

Builds the schema XSD global maps.

```python
built
```

Property that is `True` if schema validator has been fully parsed and built, `False` otherwise.

```python
validation_attempted
```

Property that returns the XSD component validation status. 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

```python
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

```python
all_errors
```

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

```python
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.
iter_globals (schema=None)
Creates an iterator for XSD global definitions/declarations.

Parameters
- schema – Optional schema instance.

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, 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.

is_valid (source, use_defaults=True)
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.

iter_errors (source, 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. 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.
- path – is an optional XPath expression that defines the parts of the document that have to
be validated. The XPath expression considers the schema as the root element with global
elements as its children.
- use_defaults – Use schema’s default values for filling missing data.
• namespaces – is an optional mapping from namespace prefix to URI.

**decode**(source, *args, **kwargs)
Decodes XML data using the XSD schema/component.

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.

• **args** – arguments that maybe passed to XMLSchema.iter_decode().

• **kwargs** – keyword arguments from the ones included in the optional arguments of the XMLSchema.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, path=None, validation='lax', process_namespaces=True, namespaces=None, use_defaults=True, decimal_type=None, converter=None, defuse=None, timeout=None, **kwargs)
Creates an iterator for decoding an XML source to a data structure.

Parameters

• **source** – the XML data source. 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.

• **path** – is an optional XPath expression that matches the parts of the document that have to be decoded. The XPath expression considers the schema as the root element with global elements as its children.

• **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.

• **converter** – an XMLSchemaConverter subclass or instance to use for the decoding.

• **defuse** – Overrides when to defuse XML data. Can be ‘always’, ‘remote’ or ‘never’.

• **timeout** – Overrides the timeout setted for the schema.

• **kwargs** – Keyword arguments containing options for converter and decoding.

Returns Yields a decoded data object, eventually preceded by a sequence of validation or decoding errors.
encode \( (obj, *args, **kwargs) \)

Encodes data to XML using the XSD schema/component.

**Parameters**
- \( obj \) – the data to be encoded to XML.
- \( args \) – arguments that maybe passed to XMLSchema.iter_encode().
- \( kwargs \) – keyword arguments from the ones included in the optional arguments of the XMLSchema.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 \( (obj, path=None, validation='lax', namespaces=None, converter=None, **kwargs) \)

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

**Parameters**
- \( obj \) – the data that has to be encoded.
- \( path \) – 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.
- \( kwargs \) – Keyword arguments containing options for converter and encoding.

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

### 3.3 ElementTree and XPath API

**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 full name.
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.

3.4 XSD globals 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 XMLSchema class to use for global maps.
- validation – the XSD validation mode to use, can be ‘strict’, ‘lax’ or ‘skip’.

build ()
Update the global maps adding the global not built registered schemas.

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.

```python
copy(validation=None)
```

Makes a copy of the object.

```python
iterGlobals()
```

Creates an iterator for XSD global definitions/declarations.

```python
iterSchemas()
```

Creates an iterator for the schemas registered in the instance.

```python
register(schema)
```

Registers an XMLSchema instance.

### 3.5 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.

```python
class xmlschema.converters.ElementData(tag, text, content, attributes)
```

Namedtuple for Element data interchange between decoders and converters.

```python
class xmlschema.XMLSchemaConverter(namespaces=None, dict_class=None, list_class=None, text_key='$', attr_prefix='@', cdata_prefix=None, etree_element_class=None, indent=4, **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.

**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`.
- `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 CDATA parts of a mixed content, that are labeled with an integer instead of a string. CDATA parts are ignored if this argument is `None`.
- `etree_element_class` – the class that has to be used to create new XML elements, if not provided uses the ElementTree’s Element class.
- `indent` – number of spaces for XML indentation (default is 4).

**Variables**

- `dict` – dictionary class to use for decoded data.
- `list` – list class to use for decoded data.
- `text_key` – key for decoded Element text
- `attr_prefix` – prefix for attribute names
- `cdata_prefix` – prefix for character data parts
- `etree_element_class` – Element class to use
• **indent** – indentation to use for rebuilding XML trees

**lossless**
The converter can ignore some kind of XML data during decoding.

**losslessly**
The format of decoded data is without loss of quality. Only losslessly formats can be always used to encode to an XML data that is strictly conformant to the schema.

```python
copy(**kwargs)
```

```python
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).

```python
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.

```python
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.

```python
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.

```python
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.

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

XML Schema based converter class for 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.

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

XML Schema based converter class for Badgerfish 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`.

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

XML Schema based converter class for Abdera 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`.

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

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

3.5. XML Schema converters
3.6 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** – 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 defusedxml library for parsing 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 set to False the source is fully loaded into and processed from memory. Default is True.

root
The XML tree root Element.

document
The ElementTree document, None if the instance is lazy or is not created from another document or from an URL.

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 base URL for completing relative locations.

namespace
The namespace of the XML document.

parse
The ElementTree parse method, depends from ‘defuse’ and ‘url’ attributes.

iterparse
The ElementTree iterparse method, depends from ‘defuse’ and ‘url’ attributes.

fromstring
The ElementTree fromstring method, depends from ‘defuse’ and ‘url’ attributes.

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.

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()  
Gets True the XML resource is lazy.

is_loaded()  
Gets True the XML text of the data source is loaded.

iter(tag=None)  
XML resource tree elements lazy iterator.

iter_location_hints()  
Yields schema location hints from the XML tree.

get_namespaces()  
Extracts namespaces with related prefixes from the XML resource. If a duplicate prefix declaration is encountered then adds the namespace using a different prefix, but only in the case if the namespace URI is not already mapped by another prefix.

Returns A dictionary for mapping namespace prefixes to full URI.

get_locations(locations=None)  
Returns a list of schema location hints. The locations are normalized using the base URL of the instance. The locations argument can be a dictionary or a list of namespace resources, that are inserted before the schema location hints extracted from the XML resource.

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, **resource_options)  
Fetches the schema URL for the source’s root of an XML data source. If an accessible schema location is not found raises a ValueError.

Parameters

• source – An an Element or an Element Tree with XML data or an URL or a file-like object.
• locations – A dictionary or dictionary items with schema location hints.
• resource_options – keyword arguments for providing XMLResource class init options.

Returns An URL referring to a reachable schema resource.

xmlschema.fetch_schema_locations(source, locations=None, **resource_options)  
Fetches the schema URL for the source’s root of an XML data source and a list of location hints. If an accessible schema location is not found raises a ValueError.

Parameters

• source – an Element or an Element Tree with XML data or an URL or a file-like object.
• locations – a dictionary or dictionary items with Schema location hints.
• **resource_options** – keyword arguments for providing `XMLResource` class init options.

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

```python
def 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.

**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.

```python
def 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.

### 3.7 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`

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.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.XMLSchemaChildrenValidationWarning`

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.
4.1 Test scripts

The tests of the xmlschema library are implemented using the Python’s unittest 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 ElementTree functionalities
- 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 packaging and forgotten development code
- test_resources.py Tests about XML/XSD resources access
- test_resources.py Tests about XSD regular expressions
- test_schemas.py Tests about parsing of XSD Schemas
- test_validators.py Tests regarding XML data validation/decoding/encoding
- test_xpath.py Tests for XPath parsing and selectors

You can run all 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.

4.2 Test 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/examples/ subdirectory.
You can locally extend the test with your set of files. For make this create the base subdirectory `xmlschema/tests/extra-schemas/` and then copy your XSD/XML files into it. After the files are copied create a new file called `testfiles` into the `extra-schemas/` subdirectory:

```
cd tests/extra-schemas/
touch testfiles
```

Fill the file `testfiles` 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 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 filepath.

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

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

```
tox -- -x
```
CHAPTER 5

Release notes

5.1 License

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

5.2 Support

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

5.3 Roadmap

• XSD 1.1
Index

| A | AbderaConverter (class in xmlschema), 25 |
|   | all_errors (xmlschema.XMLSchemaBase attribute), 18 |
|   | attrib (xmlschema.ElementPathMixin attribute), 21 |
|   | attribute_form_default (xmlschema.XMLSchemaBase attribute), 17 |
| B | BadgerFishConverter (class in xmlschema), 25 |
|   | base_url (xmlschema.XMLResource attribute), 26 |
|   | base_url (xmlschema.XMLSchemaBase attribute), 17 |
|   | block_default (xmlschema.XMLSchemaBase attribute), 17 |
|   | build() (xmlschema.XMLSchemaBase method), 18 |
|   | build() (xmlschema.XsdGlobals method), 22 |
|   | built (xmlschema.XMLSchemaBase attribute), 18 |
|   | builtin_types (xmlschema/XMLSchemaBase attribute), 17 |
| C | check_schema() (xmlschema.XMLSchemaBase class method), 18 |
|   | clear() (xmlschema.XsdGlobals method), 22 |
|   | copy() (xmlschema.XMLResource method), 26 |
|   | copy() (xmlschema.XMLSchemaConverter method), 24 |
|   | copy() (xmlschema.XsdGlobals method), 23 |
|   | create_any_attribute_group() (xmlschema.XMLSchemaBase method), 18 |
|   | create_any_content_group() (xmlschema.XMLSchemaBase method), 18 |
|   | create_schema() (xmlschema.XMLSchemaBase class method), 18 |
| D | decode() (xmlschema.XMLSchemaBase method), 20 |
|   | default_namespace (xmlschema.XMLSchemaBase attribute), 17 |
|   | document (xmlschema.XMLResource attribute), 26 |
| E | element_decode() (xmlschema.XMLSchemaConverter method), 24 |
|   | element_encode() (xmlschema.XMLSchemaConverter method), 24 |
|   | element_form_default (xmlschema.XMLSchemaBase attribute), 17 |
|   | ElementData (class in xmlschema.converters), 23 |
|   | ElementPathMixin (class in xmlschema), 21 |
|   | encode() (xmlschema.XMLSchemaBase method), 20 |
|   | etree_element() (xmlschema.XMLSchemaConverter method), 24 |
| F | fetch_resource() (in module xmlschema), 27 |
|   | fetch_schema() (in module xmlschema), 27 |
|   | fetch_schema_locations() (in module xmlschema), 27 |
|   | final_default (xmlschema.XMLSchemaBase attribute), 17 |
|   | find() (xmlschema.ElementPathMixin method), 22 |
|   | findall() (xmlschema.ElementPathMixin method), 22 |
|   | from_json() (in module xmlschema), 15 |
|   | fromstring (xmlschema.XMLResource attribute), 26 |
| G | get() (xmlschema.ElementPathMixin method), 21 |
|   | get_converter() (xmlschema.XMLSchemaBase method), 19 |
|   | get_locations() (xmlschema.XMLResource method), 27 |
|   | get_locations() (xmlschema.XMLSchemaBase method), 17 |
|   | get_namespaces() (xmlschema.XMLResource method), 27 |
|   | get_text() (xmlschema.XMLSchemaBase method), 16 |
| I | id (xmlschema.XMLSchemaBase attribute), 17 |
import_schema() (xmlschema.XMLSchemaBase method), 17
include_schema() (xmlschema.XMLSchemaBase method), 17
is_lazy() (xmlschema.XMLResource method), 26
is_valid() (xmlschema.XMLSchemaBase method), 19
iter() (xmlschema.ElementPathMixin method), 21
iter() (xmlschema.XMLResource method), 27
iter_components() (xmlschema.XMLSchemaBase method), 18
iter_decode() (xmlschema.XMLSchemaBase method), 20
iter_encode() (xmlschema.XMLSchemaBase method), 21
iter_globals() (xmlschema.XMLSchemaBase method), 19
iter Globals() (xmlschema.XsdGlobals method), 23
iter_location_hints() (xmlschema.XMLResource method), 27
iter_schemas() (xmlschema.XsdGlobals method), 23
iterchildren() (xmlschema.ElementPathMixin method), 22
iterfind() (xmlschema.ElementPathMixin method), 22
iterparse() (xmlschema.XMLResource attribute), 26

J
JsonMLConverter (class in xmlschema), 25

L
load() (xmlschema.XMLResource method), 26
load_xml_resource() (in module xmlschema), 28
lossless (xmlschema.XMLSchemaConverter attribute), 24
losslessly (xmlschema.XMLSchemaConverter attribute), 24

M
map_attributes() (xmlschema.XMLSchemaConverter method), 24
map_content() (xmlschema.XMLSchemaConverter method), 24

N
namespace (xmlschema.XMLResource attribute), 26
no_namespace_schema_location
(xmlschema.XMLSchemaBase attribute), 17
normalize_url() (in module xmlschema), 28

O
open() (xmlschema.XMLResource method), 26

P
ParkerConverter (class in xmlschema), 25
parse (xmlschema.XMLResource attribute), 26

R
register() (xmlschema.XsdGlobals method), 23
root (xmlschema.XMLResource attribute), 26
root (xmlschema.XMLSchemaBase attribute), 16
root_elements (xmlschema.XMLSchemaBase attribute), 17

S
schema_location (xmlschema.XMLSchemaBase attribute), 17

T
tag (xmlschema.ElementPathMixin attribute), 21
tag (xmlschema.XMLSchemaBase attribute), 17
target_prefix (xmlschema.XMLSchemaBase attribute), 17
text (xmlschema.XMLResource attribute), 26
to_dict() (in module xmlschema), 13
to_json() (in module xmlschema), 14
tostring() (xmlschema.XMLResource method), 26

U
url (xmlschema.XMLResource attribute), 26
url (xmlschema.XMLSchemaBase attribute), 17

V
validate() (in module xmlschema), 13
validate() (xmlschema.XMLSchemaBase method), 19
validation_attempted (xmlschema.XMLSchemaBase attribute), 18
validity (xmlschema.XMLSchemaBase attribute), 18
version (xmlschema.XMLSchemaBase attribute), 17

X
XMLResource (class in xmlschema), 26
XMLSchema (in module xmlschema), 15
xmlschema.XMLSchema10 (built-in class), 15
XMLSchemaBase (class in xmlschema), 15
XMLSchemaChildrenValidationError, 30
XMLSchemaConverter (class in xmlschema), 23
XMLSchemaDecodeError, 29
XMLSchemaEncodeError, 29
XMLSchemaException, 28
XMLSchemaImportWarning, 30
XMLSchemaIncludeWarning, 30
XMLSchemaNotBuiltError, 29
XMLSchemaParseError, 29
XMLSchemaRegexError, 28
XMLSchemaValidationWarning, 29
XMLSchemaValidatorError, 28
XsdGlobals (class in xmlschema), 22