CONTENTS

1 Introduction  ........................................ 1
  1.1 Features ............................................. 1
  1.2 Installation .......................................... 2
  1.3 License ................................................ 2
  1.4 Support ................................................ 2

2 Usage .................................................. 3
  2.1 Create a schema instance ............................... 3
  2.2 Validation ............................................ 5
  2.3 Data decoding and encoding ............................ 6
  2.4 Decoding to JSON ..................................... 12
  2.5 XML resources and documents ........................ 13
  2.6 Meta-schemas and XSD sources ......................... 13

3 Other features ........................................ 17
  3.1 XSD 1.0 and 1.1 support .............................. 17
  3.2 CLI interface .......................................... 17
  3.3 XSD validation modes ................................... 17
  3.4 Namespaces mapping options .......................... 18
  3.5 Lazy validation ........................................ 19
  3.6 XML entity-based attacks protection ................... 19
  3.7 Access control on accessing resources ............... 19
  3.8 Processing limits ...................................... 20
  3.9 Translations of parsing/validation error messages ... 20

4 Converters for XML data ............................... 21
  4.1 Available converters ................................... 21
  4.2 Create a custom converter ............................. 22

5 Schema components .................................... 23
  5.1 Accessing schema components ........................ 23
  5.2 Component structure ................................... 24
  5.3 XSD types ............................................. 26

6 Testing ................................................ 29
  6.1 Test scripts ........................................... 29
  6.2 Test cases based on files .............................. 29
  6.3 Testing with the W3C XML Schema 1.1 test suite ..... 31
  6.4 Direct testing of schemas and instances .............. 31

7 Extra features ......................................... 33
7.1 Code generation with Jinja2 templates .................................................. 33
7.2 WSDL 1.1 documents .............................................................................. 35

A Package API .......................................................................................... 37
A.1 Errors and exceptions ................................................................. 37
A.2 Document level API ................................................................. 39
A.3 Schema level API ................................................................. 44
A.4 Global maps API ................................................................. 53
A.5 Converters API ................................................................. 54
A.6 Data objects API ................................................................. 59
A.7 URL normalization API .......................................................... 60
A.8 XML resources API .............................................................. 60
A.9 Translation API ................................................................. 65
A.10 Namespaces API ................................................................. 66
A.11 XPath API .............................................................................. 66
A.12 Validation API ................................................................. 67
A.13 Particles API ........................................................................... 70
A.14 Main XSD components .......................................................... 70
A.15 Other XSD components .......................................................... 72
A.16 Extra features API .............................................................. 77

Index ........................................................................................................ 79
The `xmlschema` library is an implementation of XML Schema for Python (supports Python 3.7+).

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`
- XML attacks protection using an XMLParser that forbids entities
- Access control on resources addressed by an URL or filesystem path
- Downloading XSD files from a remote URL and storing them for offline use
- XML data bindings based on DataElement class
- Static code generation with Jinja2 templates
1.2 Installation

You can install the library with pip in a Python 3.7+ environment:

```bash
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.

1.3 License

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

1.4 Support

This software is hosted on GitHub, refer to the `xmlschema`'s project page for source code and for an issue tracker.
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('tests/test_cases/examples/vehicles/vehicles.xsd')
```

The argument can be also a file-like object or a string containing the schema definition:

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

>>> 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:

```python
>>> schema_xsd = open('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" />
```

In these cases you can provide an appropriate `base_url` optional argument to define the reference directory path for other includes and imports:
2.1.1 Non standard options for schema instance creation

Other options for schema instance creation are available using non-standard methods. Most cases require to use the `build` option to delay the schema build after the loading of all schema resources. For example:

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

```python
>>> schema.include_schema('tests/test_cases/examples/vehicles/cars.xsd')
>>> schema.include_schema('tests/test_cases/examples/vehicles/bikes.xsd')
>>> schema.build()
```

Another option, available since release v1.6.1, is to provide a list of schema sources, particularly useful when sources have no locations associated:

```python
>>> sources = [open('tests/test_cases/examples/vehicles/vehicles.xsd'),
              open('tests/test_cases/examples/vehicles/cars.xsd'),
              open('tests/test_cases/examples/vehicles/bikes.xsd'),
              open('tests/test_cases/examples/vehicles/types.xsd')]
>>> schema = xmlschema.XMLSchema(sources)
```

or similarly to the previous example one can use the method `xmlschema.XMLSchemaBase.add_schema()`:

```python
>>> schema_file = open('tests/test_cases/examples/vehicles/vehicles.xsd')
>>> schema = xmlschema.XMLSchema(schema_file, build=False)
>>> _ = schema.add_schema(open('tests/test_cases/examples/vehicles/cars.xsd'))
>>> _ = schema.add_schema(open('tests/test_cases/examples/vehicles/bikes.xsd'))
>>> _ = schema.add_schema(open('tests/test_cases/examples/vehicles/types.xsd'))
>>> schema.build()
```

**Note:** Anyway, the advice is to build intermediate XSD schemas instead for loading all the schemas needed in a standard way, because XSD mechanisms of imports, includes, redefines, and overrides are usually supported when you submit your schemas to other XSD validators.

2.1.2 Creating a local copy of a remote XSD schema for offline use

Sometimes, it is advantageous to validate XML files using an XSD schema located at a remote location while also having the option to store the same schema locally for offline use.

```python
import xmlschema

schema = xmlschema.XMLSchema("https://www.omg.org/spec/ReqIF/20110401/reqif.xsd")

schema.export(target='my_schemas', save_remote=True)

schema = xmlschema.XMLSchema("my_schemas/reqif.xsd")  # works without internet
```

With these commands, a folder `my_schemas` is created and contains the XSD files that can be used without access to the internet.
The resulting XSD files are identical to their remote source files, with the only difference being that xmlschema transforms the remote URLs into local URLs. The export command bundles a set of a target XSD file and all its dependencies by changing the schemaLocation attributes into xs:import/xs:include statements as follows:

```xml
```

becomes

```xml
```

**Note:** Since release v2.5.0 the schemaLocation attributes are rewritten with local paths that don’t start with the target directory path, in order to be reusable from any working directory. Furthermore for default the residual redundant imports from different location hints, are cleaned stripping schemaLocation attributes from them.

### 2.2 Validation

A schema instance has methods to validate an XML document against the schema.

The first method is `xmlschema.XMLSchemaBase.is_valid()`, that returns `True` if the XML argument is validated by the schema loaded in the instance, and returns `False` if the document is invalid.

```python
>>> import xmlschema
>>> schema = xmlschema.XMLSchema('tests/test_cases/examples/vehicles/vehicles.xsd')
>>> schema.is_valid('tests/test_cases/examples/vehicles/vehicles.xml')
True
>>> schema.is_valid('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.XMLSchemaBase.validate()`, that raises an error when the XML doesn’t conform to the schema:

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

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

Schema:

```xml
<x:s: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:

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

```python
>>> xml_file = 'tests/test_cases/examples/vehicles/vehicles.xml'
>>> xsd_file = 'tests/test_cases/examples/vehicles/vehicles.xsd'
>>> xmlschema.validate(xml_file, schema=xsd_file)
```

## 2.3 Data decoding and encoding

A schema instance can be also used for decoding an XML document to a nested dictionary:

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

>>> xs = xmlschema.XMLSchema('tests/test_cases/examples/vehicles/vehicles.xsd')
>>> pprint(xs.to_dict('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 match the datatypes declared in the XSD schema:

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

>>> xs = xmlschema.XMLSchema('tests/test_cases/examples/collection/collection.xsd')
>>> pprint(xs.to_dict('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'},
               'title': 'Object-1'}]
```
Decoded data can be encoded back to XML:

```python
>>> obj = schema.decode('tests/test_cases/examples/collection/collection.xml')
>>> collection = schema.encode(obj)
>>> collection
<Element {http://example.com/ns/collection}collection at ...>
>>> print(xmlschema.etree_tostring(collection, {'col': 'http://example.com/ns/collection ...'}))
  <object id="b0836217462" available="true">
    <position>1</position>
    <title>The Umbrellas</title>
    <year>1886</year>
    <author id="PAR">
      <name>Pierre-Auguste Renoir</name>
      <born>1841-02-25</born>
      <dead>1919-12-03</dead>
      <qualification>painter</qualification>
    </author>
    <estimation>10000.00</estimation>
  </object>
  <object id="b0836217463" available="true">
    <position>2</position>
    <title />
    <year>1925</year>
    <author id="JM">
      <name>Joan Miró</name>
      <born>1893-04-20</born>
      <dead>1983-12-25</dead>
      <qualification>painter, sculptor and ceramicist</qualification>
    </author>
  </object>
</col:collection>
```

(continues on next page)
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.

### 2.3.1 Decoding a part using XPath

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

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

**Note:** An XPath expression for the schema **considers the schema as the root element with global elements as its children.**

### 2.3.2 Validating and decoding ElementTree's data

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

```python
>>> from xml.etree import ElementTree
>>> xt = ElementTree.parse('tests/test_cases/examples/vehicles/vehicles.xml')
>>> xs.is_valid(xt)
True
>>> pprint(xs.to_dict(xt, process_namespaces=False, depth=2))
{'@xmlns:vh': 'http://example.com/vehicles',
 'vt:schemaLocation': 'http://example.com/vehicles',
 'vh:bikes': ['{vh:bike': [{'@make': 'Harley-Davidson', '@model': 'WL'},
 wmake': 'Yamaha', wmodel': 'XS650'}],
 'vh:cars': ['{vh:car': [{'@make': 'Porsche', '@model': '911'},
 wmake': 'Porsche', wmodel': '911'}]}]
```

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

```python
>>> import xmlschema
>>> from pprint import pprint
>>> import lxml.etree as ElementTree
>>> xs = xmlschema.XMLSchema('tests/test_cases/examples/vehicles/vehicles.xsd')
>>> xt = ElementTree.parse('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'}]}
```

### 2.3.3 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. These 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 = 'tests/test_cases/examples/vehicles/vehicles.xsd'
>>> xml_document = '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',
 '@xmlns:xsi': 'http://www.w3.org/2001/XMLSchema-instance',
 '@xsi:schemaLocation': 'http://example.com/vehicles vehicles.xsd',
 'vh:vehicles': {'@xmlns': {'vh': 'http://example.com/vehicles',
 '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'}]}
```

(continues on next page)
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 `Converters for XML data` section for more information about converters.

### 2.3.4 Control the decoding of XSD atomic datatypes

XSD datatypes are decoded to Python basic datatypes. Python strings are used for all string-based XSD types and others, like `xs:hexBinary` or `xs:QName`. Python integers are used for `xs:integer` and derived types, `bool` for `xs:boolean` values and `decimal.Decimal` for `xs:decimal` values.

Currently there are three options for variate the decoding of XSD atomic datatypes:

- **decimal_type**
  - decoding type for `xs:decimal` (is `decimal.Decimal` for default)

- **datetime_types**
  - if set to `True` decodes datetime and duration types to their respective XSD atomic types instead of keeping the XML string value

- **binary_types**
  - if set to `True` decodes `xs:hexBinary` and `xs:base64Binary` types to their respective XSD atomic types instead of keeping the XML string value

### 2.3.5 Filling missing values

Incompatible values are decoded with `None` when the validation mode is `lax`. For these situations there are two options for changing the behavior of the decoder:

- **filler**
  - a callback function to fill undecodable data with a typed value. The callback function must accept 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.
2.3.6 Control the decoding of elements

These options concern the decoding of XSD elements:

**value_hook**

A function that will be called with any decoded atomic value and the XSD type used for decoding. The return value will be used instead of the original value.

**keep_empty**

If set to `True` empty elements that are valid are decoded with an empty string value instead of `None`.

**element_hook**

A function that is called with decoded element data before calling the converter decode method. Takes an `ElementData` instance plus optionally the XSD element and the XSD type, and returns a new `ElementData` instance.

2.3.7 Control the decoding of wildcards

These two options are specific for the content processed with an XSD wildcard:

**keep_unknown**

If set to `True` unknown tags are kept and are decoded with `xs:anyType`. For default unknown tags not decoded by a wildcard are discarded.

**process_skipped**

Process XML data that match a wildcard with `processContents='skip'`.

2.3.8 Control the decoding depth

**max_depth**

Maximum level of decoding, for default there is no limit. With lazy resources is automatically set to `source.lazy_depth` for managing lazy decoding. Available also for validation methods.

**depth_filler**

A callback function for replacing data over the `max_depth` level. The callback function must accept one positional argument, that can be an XSD Element. For default deeper data is replaced with `None` values when `max_depth` is provided.

2.3.9 Control the validation

**extra_validator**

An optional function for performing non-standard validations on XML data. The provided function is called for each traversed element, with the XML element as 1st argument and the corresponding XSD element as 2nd argument. It can be also a generator function and has to raise/yield `XMLSchemaValidationError` exceptions.

**validation_hook**

An optional function for stopping or changing validation/decoding at element level. The provided function must accept two arguments, the XML element and the matching XSD element. If the value returned by this function is evaluated to false then the validation/decoding process continues without changes, otherwise it’s stopped or changed. If the value returned is a validation mode the validation/decoding process continues changing the current validation mode to the returned value, otherwise the element and its content are not processed. For validation only this function can also stop validation suddenly raising a `XMLSchemaStopValidation` exception.
2.4 Decoding to JSON

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

```python
>>> import xmlschema
>>> import json
>>> xml_document = '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 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`:

```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"
        },
    ]
}
```

(continues on next page)
{  
    "@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.

### 2.5 XML resources and documents

Schemas and XML instances processing are based on the class xmlschema.XMLResource, that handles the loading and the iteration of XSD/XML data. Starting from v1.3.0 xmlschema.XMLResource has been empowered with ElementTree-like XPath API. From the same release a new class xmlschema.XmlDocument is available for representing XML resources with a related schema:

```python
>>> import xmlschema
d>>> xml_document = xmlschema.XmlDocument('tests/test_cases/examples/vehicles/vehicles.xml')
>>> xml_document.schema
XMLSchema10(name='vehicles.xsd', namespace='http://example.com/vehicles')
```

This class can be used to derive specialized schema-related classes. See WSDL 1.1 documents section for an application example.

### 2.6 Meta-schemas and XSD sources

Schema classes xmlschema.XMLSchema10 and xmlschema.XMLSchema11 have built-in meta-schema instances, related to the XSD namespace, that can be used directly to validate XSD sources without build a new schema:

```python
>>> from xmlschema import XMLSchema
>>> XMLSchema.meta_schema.validate('tests/test_cases/examples/vehicles/vehicles.xsd')
>>> XMLSchema.meta_schema.validate('tests/test_cases/examples/vehicles/invalid.xsd')
Traceback (most recent call last):
  ...
  ...
xmlschema.validators.exceptions.XMLSchemaValidationError: failed validating ...
Reason: use of attribute 'name' is prohibited

Schema:

```xml
<x:schema xmlns:x="http://www.w3.org/2001/XMLSchema" base="xs:complexType">
  <xs:restriction>
    <xs:sequence>
      <xs:element ref="xs:annotation" minOccurs="0" />
      <xs:group ref="xs:complexTypeModel" />
    </xs:sequence>
    <xs:attribute name="name" use="prohibited" />
    <xs:attribute name="abstract" use="prohibited" />
    <xs:attribute name="final" use="prohibited" />
    <xs:attribute name="block" use="prohibited" />
    <xs:anyAttribute namespace="##other" processContents="lax" />
  </xs:restriction>
</xs:schema>
```

Instance:

```xml
<x:schema xmlns:x="http://www.w3.org/2001/XMLSchema" name="vehiclesType">
  <xs:sequence>
    <xs:element ref="vh:cars" />
    <xs:element ref="vh:bikes" />
  </xs:sequence>
</xs:schema>
```

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

Furthermore also decode and encode methods can be applied on XSD files or sources:

```python
from xmlschema import XMLSchema
obj = XMLSchema.meta_schema.decode('tests/test_cases/examples/vehicles/vehicles.xsd')
from pprint import pprint
pprint(obj)
```

{ "attributeFormDefault": 'unqualified',
  'blockDefault': [],
  'elementFormDefault': 'qualified',
  'finalDefault': [],
  'targetNamespace': 'http://example.com/vehicles',
  'xmlns:vh': 'http://example.com/vehicles',
  'xmlns:xs': 'http://www.w3.org/2001/XMLSchema',
  'attribute': { '@name': 'step', '@type': 'xs:positiveInteger' },
  'element': { '@abstract': False,
    '@name': 'vehicles',
    '@nillable': False,
    'complexType': { '@mixed': False,
      'sequence': { '@maxOccurs': 1,
        '@minOccurs': 1,
        'element': [ { '@maxOccurs': 1,
          '@minOccurs': 1,
          '@nillable': False,
          'ref': 'vh:cars' },
          { '@maxOccurs': 1,
            '@minOccurs': 1,
            '@nillable': False,
            'ref': 'vh:bikes' } ] } } } }
Note: Building a new schema for XSD namespace could be not trivial because other schemas are required for base namespaces (e.g. XML namespace ‘http://www.w3.org/XML/1998/namespace’). This is particularly true for XSD 1.1 because the XSD meta-schema lacks of built-in list types definitions, so a patch schema is required.
OTHER FEATURES

Schema objects and package APIs include a set of other features that have been added since a specific release. These features are regulated by arguments, alternative classes or module parameters.

3.1 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 `xmlschema.XMLSchema11`. You have to use this class for XSD 1.1 schemas instead the default class `xmlschema.XMLSchema`, that is linked to XSD 1.0 validator `xmlschema.XMLSchema10`.

The XSD 1.1 validator can be used also for validating XSD 1.0 schemas, except for a restricted set of cases related to content extension in a complexType (the extension of a complex content with simple base is allowed in XSD 1.0 and forbidden in XSD 1.1).

3.2 CLI interface

Starting from the version v1.2.0 the package has a CLI interface with three console scripts:

- `xmlschema-validate` Validate a set of XML files.
- `xmlschema-xml2json` Decode a set of XML files to JSON.
- `xmlschema-json2xml` Encode a set of JSON files to XML.

3.3 XSD validation modes

Since the version v0.9.10 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’. 

Note: From release v1.1.1 the iter_decode() and iter_encode() methods propagate errors also for skip validation mode. The errors generated in skip mode are discarded by the top-level methods decode() and encode().

3.4 Namespaces mapping options

Since the earlier releases the validation/decoding/encoding methods include the namespaces optional argument that can be used to provide a custom namespace mapping. In versions prior to 3 of the library the XML declarations are loaded and merged over the custom mapping during the XML document traversing, using alternative prefixes in case of collision.

With version 3.0 the processing of namespace information of the XML document has been improved, with the default of maintaining an exact namespace mapping between the XML source and the decoded data.

The feature is available both with the decoding and encoding API with the new converter option xmlns_processing, that permits to change the processing mode of the namespace declarations of the XML document.

The preferred mode is ‘stacked’, the mode that maintains a stack of namespace mapping contexts, with the active context that always match the namespace declarations defined in the XML document. In this case the namespace map is updated dynamically, adding and removing the XML declarations found in internal elements. This choice provide the most accurate mapping of the namespace information of the XML document.

Use the option value ‘collapsed’ for loading all namespace declarations in a single map. In this case the declarations are merged into the namespace map of the converter, using alternative prefixes in case of collision. This is the legacy behaviour of versions prior to 3 of the library.

With ‘root-only’ only the namespace declarations of the XML document root are loaded. In this case you are expected to provide the internal namespace information with namespaces argument.

Use ‘none’ to not load any namespace declaration of the XML document. Use this option if you don’t want to map namespaces to prefixes or you want to provide a fully custom namespace mapping.

For default xmlns_processing option is set automatically depending by the converter class capability and the XML data source. The option is available also for encoding with updated converter classes that can retrieve xmlns declarations from decoded data (e.g. xmlschema.JsonMLConverter or the default converter). For decoding the default is set to ‘stacked’ or ‘collapsed’, for encoding the default can be also ‘none’ if no namespace declaration can be retrieved from XML data (e.g. xmlschema.ParkerConverter).
3.5 Lazy validation

From release v1.0.12 the document validation and the decoding API have an optional argument `lazy=False`, that can be changed to `True` for operating with a lazy `xmlschema.XMLResource`. The lazy mode can be useful for validating and decoding big XML data files, consuming less memory.

From release v1.1.0 the lazy mode can be also set with a non negative integer. A zero is equivalent to `False`, a positive value means that lazy mode is activated and defines also the lazy depth to use for traversing the XML data tree.

Lazy mode works better with validation because is not needed to use converters for shaping decoded data.

3.6 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. Providing ‘nonlocal’ all XML data are defused except local files. Other values for this argument can be ‘always’ and ‘never’, with obvious meaning.

3.7 Access control on accessing resources

From release v1.2.0 the schema class includes an argument named `allow` for protecting the access to XML resources identified by an URL or filesystem path. For default all types of URLs are allowed. Provide a different value to restrict the set of URLs that the schema instance can access:

- **all**
  - All types of URL and file paths are allowed.
- **remote**
  - Only remote resource URLs are allowed.
- **local**
  - Only file paths and file-related URLs are allowed.
- **sandbox**
  - Allows only the file paths and URLs that are under the directory path identified by `source` argument or `base_url` argument.
- **none**
  - No URL based or file path access is allowed.

**Warning:** For protecting services that are freely accessible for validation (eg. a web on-line validator that has a form for loading schema and/or XML instance) the recommendation is to provide ‘always’ for the `defuse` argument and ‘none’ for the `allow` argument. These settings prevent attacks to your local filesystem, through direct paths or injection in XSD schema imports or includes.

For XSD schemas, if you want to permit imports of namespaces located on other web services you can provide ‘remote’ for the `allow` argument and provide an `XMLResource` instance, initialized providing `allow='none'`, as the `source` argument for the main schema.
3.8 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.

3.8.1 Limit on XSD model groups checking

Model groups of the schemas are checked against restriction violations and Unique Particle Attribution violations. To avoid 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 a higher limit for checking all your groups you can import the library and change the value of MAX_MODEL_DEPTH in the limits module:

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

3.8.2 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:

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

3.9 Translations of parsing/validation error messages

From release v1.11.0 translation of parsing/validation error messages can be activated:

```python
>>> import xmlschema
>>> xmlschema.translation.activate()
```

Note: Activation depends by the default language in your environment and if it matches translations provided with the library. You can build your custom translation from the template included in the repository (xmlschema/locale/xmlschema.pot) and then use it in your runs providing localedir and languages arguments to activation call. See Translation API for information.

Translations for default do not interfere with other translations installed at runtime and can be deactivated after:

```python
>>> xmlschema.translation.deactivate()
```
CHAPTER
FOUR

CONVERTERS FOR XML DATA

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

Because XML is a structured format that includes data and metadata information, as attributes and namespace decla-
rations, is necessary to define conventions for naming the different data objects in a distinguishable way. For example
a wide-used convention is to prefixing attribute names with an ‘@’ character. With this convention the attribute
\texttt{name='John'} is decoded to \texttt{@name: 'John'}, or \texttt{level='10'} is decoded to \texttt{@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 \texttt{http://www.w3.org/2001/XMLSchema\_string} is the fully qualified name of the
XSD string type, usually referred as \texttt{xs:string} or \texttt{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 \texttt{xmlschema.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:

- \texttt{xmlschema.ParkerConverter}: Parker convention
- \texttt{xmlschema.BadgerFishConverter}: BadgerFish convention
- \texttt{xmlschema.AbderaConverter}: Apache Abdera project convention
- \texttt{xmlschema.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 con-
verting process. Some of these options are not used by other predefined converter types (eg. \texttt{force\_list} and \texttt{force\_dict})
or are used with a fixed value (eg. \texttt{text\_key} or \texttt{attr\_prefix}). See Converters API for details about base class options and
attributes.

Moreover there are also other two converters useful for specific cases:

- \texttt{xmlschema.UnorderedConverter}: like default converter but with unordered decoding and encoding.
- \texttt{xmlschema.ColumnarConverter}: a converter that remaps attributes as child elements in a columnar shape
  (available since release v1.2.0).
- \texttt{xmlschema.DataElementConverter}: a converter that converts XML to a tree of \texttt{xmlschema.DataElement}
  instances, Element-like objects with decoded values and schema bindings (available since release v1.5.0).
4.2 Create a custom converter

To create a new customized converter you have to subclass the `xmlschema.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 (e.g. 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
FIVE

SCHEMA COMPONENTS

After the building a schema object contains a set of components that represent the definitions/declarations defined in loaded schema files. These components, sometimes referred as Post Schema Validation Infoset or PSVI, constitute an augmentation of the original information contained into schema files.

5.1 Accessing schema components

Taking the `collection.xsd` as sample schema to illustrate the access to components, we can iterate the entire set of components, globals an locals, using the `iter_components()` generator function:

```python
>>> import xmlschema

>>> schema = xmlschema.XMLSchema('tests/test_cases/examples/collection/collection.xsd')

>>> for xsd_component in schema.iter_components():
    ...     xsd_component

XMLSchema10(name='collection.xsd', namespace='http://example.com/ns/collection')
XsdComplexType(name='personType')
XsdAttributeGroup(['id'])
XsdAttribute(name='id')
XsdGroup(model='sequence', occurs=[1, 1])
XsdElement(name='name', occurs=[1, 1])
...
XsdElement(name='object', occurs=[1, None])
XsdElement(name='person', occurs=[1, 1])
```

For taking only global components use `iterGlobals()` instead:

```python
>>> for xsd_component in schema.iter_globals():
    ...     xsd_component

... XsdComplexType(name='personType')
XsdComplexType(name='objType')
XsdElement(name='collection', occurs=[1, 1])
XsdElement(name='person', occurs=[1, 1])
```
5.1.1 Access with XPath API

Another method for retrieving XSD elements and attributes of a schema is to use XPath expressions with `find` or `findall` methods:

```python
>>> from pprint import pprint
>>> namespaces = {'': 'http://example.com/ns/collection'}
>>> schema.find('collection/object', namespaces)
XsdElement(name='object', occurs=[1, None])
>>> pprint(schema.findall('collection/object/*', namespaces))
[XsdElement(name='position', occurs=[1, 1]),
 XsdElement(name='title', occurs=[1, 1]),
 XsdElement(name='year', occurs=[1, 1]),
 XsdElement(name='author', occurs=[1, 1]),
 XsdElement(name='estimation', occurs=[0, 1]),
 XsdElement(name='characters', occurs=[0, 1])]
```

5.1.2 Access to global components

Accessing a specific type of global component a dictionary access may be preferred:

```python
>>> schema.elements['person']
XsdElement(name='person', occurs=[1, 1])
>>> schema.types['personType']
XsdComplexType(name='personType')
```

The schema object has a dictionary attribute for each type of XSD declarations (`elements`, `attributes` and `notations`) and for each type of XSD definitions (`types`, `model groups`, `attribute groups`, `identity constraints` and `substitution groups`). These dictionaries are only views of common dictionaries, shared by all the loaded schemas in a structure called `maps`:

```python
>>> schema.maps
XsdGlobals(validator=XMLSchema10(name='collection.xsd', ...)
```

```python
>>> person = schema.elements['person']
>>> person
XsdElement(name='person', occurs=[1, 1])
>>> schema.maps.elements[person.qualified_name]
XsdElement(name='person', occurs=[1, 1])
```

5.2 Component structure

Only the main component classes are available at package level:

XsdComponent
   The base class of every XSD component.

XsdType
   The base class of every XSD type, both complex and simple types.

XsdElement
   The XSD 1.0 element class, base also of XSD 1.1 element class.
XsdAttribute

The XSD 1.0 attribute class, base also of XSD 1.1 attribute class.

The full schema components are provided only by accessing the xmlschema.validators subpackage, for example:

```python
>>> import xmlschema
>>> xmlschema.validators.Xsd11Element
<class 'xmlschema.validators.elements.Xsd11Element'>
```

5.2.1 Connection with the schema

Every component is linked to its container schema and a reference node of its XSD schema document:

```python
>>> person = schema.elements['person']
>>> person.schema
XMLSchema10(name='collection.xsd', namespace='http://example.com/ns/collection')
>>> person.elem
<Element '{http://www.w3.org/2001/XMLSchema}element' at ...>
>>> person.tostring()
'<xs:element xmlns:xs="http://www.w3.org/2001/XMLSchema" name="person" type="personType"/>
```

5.2.2 Naming options

A component that has a name (eg. elements or global types) can be referenced with a different name format, so there are some properties for getting these formats:

```python
>>> vh_schema = xmlschema.XMLSchema('tests/test_cases/examples/vehicles/vehicles.xsd')
>>> car = vh_schema.find('vh:vehicles/vh:cars/vh:car')
>>> car.name
'car'
>>> car.local_name
'car'
>>> car.prefixed_name
'vh:car'
>>> car.qualified_name
'{http://example.com/vehicles}car'
>>> car.attributes['model'].name
'model'
>>> car.attributes['model'].qualified_name
'{http://example.com/vehicles}model'
```
5.2.3 Decoding and encoding

Every schema component includes methods for data conversion:

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

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

>>> xs = xmlschema.XMLSchema('tests/test_cases/examples/vehicles/vehicles.xsd')
>>> xt = ElementTree.parse('tests/test_cases/examples/vehicles/vehicles.xml')
>>> root = xt.getroot()

>>> pprint(xs.elements['cars'].decode(root[0]))
{
'@xmlns:vh': 'http://example.com/vehicles',
'vh: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'}]
}
```

5.3 XSD types

Every element or attribute declaration has a `type` attribute for accessing its XSD type:

```python
>>> person = schema.elements['person']
>>> person.type
XsdComplexType(name='personType')
```

5.3.1 Simple types

Simple types are used on attributes and elements that contains a text value:

```python
>>> schema = xmlschema.XMLSchema('tests/test_cases/examples/vehicles/vehicles.xsd')
>>> schema.attributes['step']
XsdAttribute(name='vh:step')
>>> schema.attributes['step'].type
XsdAtomicBuiltin(name='xs:positiveInteger')
```

A simple type doesn’t have attributes but can have facets-related validators or properties:
5.3.2 Complex types

Complex types are used only for elements with attributes or with child elements.

For accessing the attributes there is always defined and attribute group, also when the complex type has no attributes:

```python
type = schema.types['objType']
type.attributes
```

For accessing the content model there use the attribute `content`. In most cases the element’s type is a complexType with a complex content and in these cases `content` is a not-empty XsdGroup:

```python
person = schema.elements['person']
person.type.has_complex_content()
for item in person.type.content:
    print(item)
```

**Note:** The attribute `content_type` has been renamed to `content` in v1.2.1 in order to avoid confusions between the complex type and its content. A property with the old name will be maintained until v2.0.

Model groups can be nested with very complex structures, so there is an generator function `iter_elements()` to traverse a model group:

```python
for e in person.type.content.iter_elements():
    print(e)
```
Sometimes a complex type can have a simple content, in these cases content is a simple type.

### 5.3.3 Content types

An element can have four different content types:

- **empty**: deny child elements, deny text content
- **simple**: deny child elements, allow text content
- **element-only**: allow child elements, deny intermingled text content
- **mixed**: allow child elements and intermingled text content

For attributes only empty or simple content types are possible, because they can have only a simpleType value.

The reference methods for checking the content type are respectively `is_empty()`, `has_simple_content()`, `is_element_only()` and `has_mixed_content()`.

### 5.3.4 Access to content validator

The content type checking can be complicated if you want to know which is the content validator without use a type checking. To making this simpler there are two properties defined for XSD types:

- **simple_type**: a simple type in case of simple content or when an empty content is based on an empty simple type, None otherwise.
- **model_group**: a model group in case of mixed or element-only content or when an empty content is based on an empty model group, None otherwise.
The tests of the `xmlschema` library are implemented using the Python’s `unittest` library. From version v1.1.0 the test scripts have been moved into the directory `tests/` of the source distribution. Only a small subpackage `extras/testing/`, containing a specialized `UnitTest` subclass, a factory and builders for creating test classes for XSD and XML file, has been left into the package’s code.

### 6.1 Test scripts

There are several test scripts, each one for a different target. These scripts can be run individually or by the `unittest` module. For example to run XPath tests through the `unittest` module use the command:

```
$ python -m unittest -k tests.test_xpath
...........
Ran 10 tests in 0.133s
OK
```

The same run can be launched with the command `$ python tests/test_xpath.py` but an additional header, containing info about the package location, the Python version and the machine platform, is displayed before running the tests.

Under the base directory `tests/` there are the test scripts for the base modules of the package. The subdirectory `tests/validators` includes tests for XSD validators building (schemas and their components) and the subdirectory `tests/validation` contains tests validation of XSD/XML and decoding/encoding of XML files.

To run all tests use the command `python -m unittest`. Also, the script `*test_all.py*` can launched during development to run all the tests except memory and packaging tests. 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*`.

### 6.2 Test cases based on files

Three scripts (`test_all.py`, `test_schemas.py`, `test_validation.py`) create many tests dinamically, building test classes from a set of XSD/XML files. Only a small set of test files is published in the repository for copyright reasons. You can find the repository test files into `tests/test_cases/` subdirectory.

You can locally extend the test with your set of files. For doing this create a submodule or a directory outside the repository directory and then copy your XSD/XML files into it. Create an index file called `testfiles` into the base directory were you put your cases and fill it with the list of paths of files you want to be tested, one per line, as in the following example:
The test scripts create a test for each listed file, dependant from the context. For example the script `test_schemas.py` uses only `.xsd` files, where instead the script `tests_validation.py` uses only `.xml` files.

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).

- `--lax-encode`
  - Use lax mode on encode checks (for cases where test data uses default or fixed values or some test data are skipped by wildcards processContents). Ignored on schema tests.

- `--debug`
  - Activate the debug mode (only the cases with `--debug` are executed).

- `--codegen`
  - Test code generation with XML data bindings module.

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

To run tests with also your personal set of files you have provide the path to your custom `testfile` index, for example:

```
python xmlschema/tests/test_all.py ../extra-schemas/testfiles
```
6.3 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 subset of W3C tests, run `test_w3_suite.py --help` to show available options.

6.4 Direct testing of 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 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
SEVEN

EXTRA FEATURES

The subpackage `xmlschema.extras` acts as a container of a set of extra modules or subpackages that can be useful for specific needs.

These codes are not imported during normal library usage and may require additional dependencies to be installed. This choice should be facilitate the implementation of other optional functionalities without having an impact on the base configuration.

7.1 Code generation with Jinja2 templates

The module `xmlschema.extras.codegen` provides an abstract base class `xmlschema.extras.codegen.AbstractGenerator` for generate source code from parsed XSD schemas. The Jinja2 engine is embedded in that class and is empowered with a set of custom filters and tests for accessing to defined XSD schema components.

7.1.1 Schema based filters

Within templates you can use a set of additional filters, available for all generator subclasses:

- **name**
  Get the unqualified name of the object. Invalid chars for identifiers are replaced by an underscore.

- **qname**
  Get the QName of the object in prefixed form. Invalid chars for identifiers are replaced by an underscore.

- **namespace**
  Get the namespace URI of the XSD component.

- **type_name**
  Get the unqualified name of an XSD type. For default ‘Type’ or ‘_type’ suffixes are removed. Invalid chars for identifiers are replaced by an underscore.

- **type_qname**
  Get the QName of an XSD type in prefixed form. For default ‘Type’ or ‘_type’ suffixes are removed. Invalid chars for identifiers are replaced by an underscore.

- **sort_types**
  Sort a sequence or a map of XSD types, in reverse dependency order, detecting circularities.
7.1.2 Schema based tests

Within templates you can also use a set of tests, available for all generator classes:

**derivation**
Test if an XSD type instance is a derivation of any of a list of other types. Other types are provided by qualified names.

**extension**
Test if an XSD type instance is an extension of any of a list of other types. Other types are provided by qualified names.

**restriction**
Test if an XSD type instance is a restriction of any of a list of other types. Other types are provided by qualified names.

**multi_sequence**
Test if an XSD type is a complex type with complex content that at least one child can have multiple occurrences.

7.1.3 Type mapping

Each implementation of a generator class has an additional filter for translating types using the types map of the instance. For example *xmlschema.extras.codegen.PythonGenerator* has the filter *python_type*.

These filters are based on a common method *map_type* that uses an instance dictionary built at initialization time from a class maps for builtin types and an optional initialization argument for the types defined in the schema.

7.1.4 Defining additional Jinja2 filters and tests

Defining a generator class you can add filters and tests using *filter_method* and *test_method* decorators:

```python
>>> from xmlschema.extras.codegen import AbstractGenerator, filter_method, test_method

>>> class DemoGenerator(AbstractGenerator):
...     formal_language = 'Demo'

...     @filter_method
...     def my_filter_method(self, obj):
...         """A method that filters an object using the schema."""

...     @staticmethod
...     @test_method
...     def my_test_method(obj):
...         """A static method that test an object."""
```
7.2 WSDL 1.1 documents

The module `xmlschema.extras.wsdl` provides a specialized schema-related XML document for WSDL 1.1.

An example of specialization is the class `xmlschema.extras.wsdl.Wsdl11Document`, usable for validating and parsing WSDL 1.1 documents, that can be imported from `wsdl` module of the `extra` subpackage:

```python
>>> from xmlschema.extras.wsdl import Wsdl11Document
>>> wsdl_document = Wsdl11Document('tests/test_cases/examples/stockquote/
stockquoteservice.wsdl')
>>> wsdl_document.schema
<xmlschema.XMLSchema10(name='wsdl.xsd', namespace='http://schemas.xmlsoap.org/wsdl/')>
```

A parsed WSDL 1.1 document can aggregate a set of WSDL/XSD files for building interrelated set of definitions in multiple namespaces. The XMLResource base class and schema validation assure a fully checked WSDL document with protections against XML attacks. See `xmlschema.extras.wsdl.Wsdl11Document` API for details.
A.1 Errors and exceptions

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

exception XMLResourceError
   Raised when an error is found accessing an XML resource.

exception XMLSchemaNamespaceError
   Raised when a wrong runtime condition is found with a namespace.

exception XMLSchemaValidatorError (validator: XsdValidator | Callable[[Any], None], message: str, elem: T | None = None, source: Any | None = None, namespaces: T | None = None)
   Base class for XSD validator errors.
   Parameters
   • validator – the XSD validator.
   • message – the error message.
   • elem – the element that contains the error.
   • source – the XML resource or the decoded data that contains the error.
   • namespaces – is an optional mapping from namespace prefix to URI.

exception XMLSchemaNotBuiltError (validator: XsdValidator, message: str)
   Raised when there is an improper usage attempt of a not built XSD validator.
   Parameters
   • validator – the XSD validator.
   • message – the error message.

exception XMLSchemaParseError (validator: XsdValidator, message: str, elem: T | None = None)
   Raised when an error is found during the building of an XSD validator.
   Parameters
   • validator – the XSD validator.
   • message – the error message.
   • elem – the element that contains the error.
exception XMLSchemaModelError(
group: XsdGroup, message: str
)
Raised when a model error is found during the checking of a model group.

Parameters

• group – the XSD model group.

• message – the error message.

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

exception XMLSchemaValidationError(
validator: XsdValidator | Callable[[Any], None],
obj: Any, reason: str | None = None,
source: Any | None = None,
namespaces: T | None = 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 – the XSD validator.

• obj – the not validated XML data.

• reason – the detailed reason of failed validation.

• source – the XML resource that contains the error.

• namespaces – is an optional mapping from namespace prefix to URI.

exception XMLSchemaDecodeError(
validator: XsdValidator | Callable[[Any], None],
obj: Any, decoder: Any,
reason: str | None = None,
source: Any | None = None,
namespaces: T | None = None
)
Raised when an XML data string is not decodable to a Python object.

Parameters

• validator – the XSD validator.

• obj – the not validated XML data.

• decoder – the XML data decoder.

• reason – the detailed reason of failed validation.

• source – the XML resource that contains the error.

• namespaces – is an optional mapping from namespace prefix to URI.

exception XMLSchemaEncodeError(
validator: XsdValidator | Callable[[Any], None],
obj: Any, encoder: Any,
reason: str | None = None,
source: Any | None = None,
namespaces: T | None = None
)
Raised when an object is not encodable to an XML data string.

Parameters

• validator – the XSD validator.

• obj – the not validated XML data.

• encoder – the XML encoder.

• reason – the detailed reason of failed validation.

• source – the XML resource that contains the error.
namespaces – is an optional mapping from namespace prefix to URI.

exception XMLSchemaChildrenValidationException

validator: XsdValidator, elem: T | None, index: int, particle: T | None, occurs: int = 0, expected: Iterable[T | None],
None = None, source: Any | None = None, namespaces: T | None

Raised when a child element is not validated.

Parameters

- **validator** – the XSD validator.
- **elem** – the not validated XML element.
- **index** – the child index.
- **particle** – the model particle that generated the error. Maybe the validator itself.
- **occurs** – the particle occurrences.
- **expected** – the expected element tags/object names.
- **source** – the XML resource that contains the error.
- **namespaces** – is an optional mapping from namespace prefix to URI.

invalid_tag: str | None

The tag of the invalid child element, None in case of an incomplete content.

invalid_child

The invalid child element, if any, None otherwise. It’s None in case of incomplete content or if the parent
has been cleared during lazy validation.

exception XMLSchemaStopValidation

Stops the validation process.

exception XMLSchemaIncludeWarning

A schema include fails.

exception XMLSchemaImportWarning

A schema namespace import fails.

exception XMLSchemaTypeTableWarning

Not equivalent type table found in model.

### A.2 Document level API

**validate**

```
validate(xml_document: T | None | XMLResource, schema: XMLSchemaBase | None = None, cls:
Type[XMLSchemaBase] | None = None, path: str | None = None, schema_path: str | None = None,
use_defaults: bool = True, namespaces: T | None = None, locations: T | None = None, base_url: str |
None = None, defuse: str = 'remote', timeout: int = 300, lazy: T | None = False, thin_lazy: bool = True,
uri_mapper: T | None = None, use_location_hints: bool = True) → None
```

Validates an XML document against a schema instance. This function builds an XMLSchema object for validating
the XML document. Raises an XMLSchemaValidationException 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 a 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 `XMLSchema10` 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, used if 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** – an optional argument for building the schema and the `XMLResource` instance.

• **timeout** – an optional argument for building the schema and the `XMLResource` instance.

• **lazy** – an optional argument for building the `XMLResource` instance.

• **thin_lazy** – an optional argument for building the `XMLResource` instance.

• **uri_mapper** – an optional argument for building the schema from location hints.

• **use_location_hints** – for default, in case a schema instance has to be built, uses also schema locations hints provided within XML data. Set this option to `False` to ignore these schema location hints.

```python
is_valid(xml_document: T | None | XMLResource, schema: XMLSchemaBase | None = None, cls:
Type[XMLSchemaBase] | None = None, path: str | None = None, schema_path: str | None = None,
use_defaults: bool = True, namespaces: T | None = None, locations: T | None = None, base_url: str |
None = None, defuse: str = 'remote', timeout: int = 300, lazy: T | None = False, thin_lazy: bool = True,
uri_mapper: T | None = None, use_location_hints: bool = bool) → bool
```

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

```python
iter_errors(xml_document: T | None | XMLResource, schema: XMLSchemaBase | None = None, cls:
Type[XMLSchemaBase] | None = None, path: str | None = None, schema_path: str | None = None,
use_defaults: bool = True, namespaces: T | None = None, locations: T | None = None, base_url: str |
None = None, defuse: str = 'remote', timeout: int = 300, lazy: T | None = False, thin_lazy: bool = True,
uri_mapper: T | None = None, use_location_hints: bool = bool) → Iterator[XMLSchemaValidationError]
```

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

```python
iter_decode(xml_document: T | None | XMLResource, schema: XMLSchemaBase | None = None, cls:
Type[XMLSchemaBase] | None = None, path: str | None = None, validation: str = 'lax', locations: T |
None = None, base_url: str | None = None, defuse: str = 'remote', timeout: int = 300, lazy: T | None
= False, thin_lazy: bool = True, uri_mapper: T | None = None, use_location_hints: bool = True,
**kwargs: Any) → Iterator[Any | XMLSchemaValidationError]
```

40 Appendix A. Package API
Creates an iterator for decoding an XML source to a data structure. For default the document is validated during the decoding phase and if it's invalid then one or more `XMLSchemaValidationError` instances are yielded before the decoded data.

**Parameters**

- `xml_document` – can be an `XMLResource` instance, a file-like object a path to a file or a 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 `XMLSchema10`).

- `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.

- `validation` – defines the XSD validation mode to use for decode, can be ‘strict’, ‘lax’ or ‘skip’.

- `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` – an optional argument for building the schema and the `XMLResource` instance.

- `timeout` – an optional argument for building the schema and the `XMLResource` instance.

- `lazy` – an optional argument for building the `XMLResource` instance.

- `thin_lazy` – an optional argument for building the `XMLResource` instance.

- `uri_mapper` – an optional argument for building the schema from location hints.

- `use_location_hints` – for default, in case a schema instance has to be built, uses also schema locations hints provided within XML data. Set this option to `False` to ignore these schema location hints.

- `kwargs` – other optional arguments of `XMLSchemaBase.iter_decode()` as keyword arguments.

**Raises**

- `XMLSchemaValidationError` if the XML document is invalid and `validation='strict'` is provided.

**to_dict**

```
xmlschema.Documentation.Release 3.0.1

Decodes an XML document to a Python’s nested dictionary. Takes the same arguments of the function `iter_decode()`, but `validation` mode defaults to ‘strict’.

**Returns**

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

A.2. Document level API 41
```
Raises `XMLSchemaValidationError` if the XML document is invalid and validation='strict' is provided.

`to_json(xml_document: T | None | XMLResource, fp: IO[str] | None = None, schema: XMLSchemaBase | None = None, cls: Type[XMLSchemaBase] | None = None, path: str | None = None, validation: str = 'strict', locations: T | None = None, base_url: str | None = None, defuse: str = 'remote', timeout: int = 300, lazy: T | None = False, thin_lazy: bool = True, uri_mapper: T | None = None, use_location_hints: bool = True, json_options: Dict[str, Any] | None = None, **kwargs: Any) → T | None`  

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 a 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 a URL of a resource or a string containing the schema.
- `cls` – schema class to use for building the instance (for default uses `XMLSchema10`).
- `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.
- `validation` – defines the XSD validation mode to use for decode, can be ‘strict’, ‘lax’ or ‘skip’.
- `locations` – additional schema location hints, in case the 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` – an optional argument for building the schema and the `XMLResource` instance.
- `timeout` – an optional argument for building the schema and the `XMLResource` instance.
- `uri_mapper` – an optional argument for building the schema from location hints.
- `lazy` – an optional argument for building the `XMLResource` instance.
- `thin_lazy` – an optional argument for building the `XMLResource` instance.
- `use_location_hints` – for default, in case a schema instance has to be built, uses also schema locations hints provided within XML data. Set this option to `False` to ignore these schema location hints.
- `json_options` – a dictionary with options for the JSON serializer.
- `kwargs` – optional arguments of `XMLSchemaBase.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.
to_etree(obj: Any, schema: XMLSchemaBase | T | None = None, cls: Type[XMLSchemaBase] | None = None, path: str | None = None, validation: str = 'strict', namespaces: T | None = None, use_defaults: bool = True, converter: T | None = None, unordered: bool = False, **kwargs: Any) → T | None

Encodes a data structure/object to an ElementTree’s Element.

Parameters

- **obj** – the Python object that has to be encoded to XML data.
- **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. If not provided a dummy schema is used.
- **cls** – class to use for building the schema instance (for default uses `XMLSchema10`).
- **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.
- **use_defaults** – whether to use default values for filling missing data.
- **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** – other optional arguments of `XMLSchemaBase.iter_encode()` and options for the converter.

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.

from_json(source: str | bytes | IO[str], schema: XMLSchemaBase | T | None = None, cls: Type[XMLSchemaBase] | None = None, path: str | None = None, validation: str = 'strict', namespaces: T | None = None, use_defaults: bool = True, converter: T | None = None, unordered: bool = False, json_options: Dict[str, Any] | None = None, **kwargs: Any) → T | None

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 `XMLSchema10` or an `XMLSchema11` instance.
- **cls** – class to use for building the schema instance (for default uses `XMLSchema10`).
- **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.
- **use_defaults** – whether to use default values for filling missing data.
- **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.

• **json_options** – a dictionary with options for the JSON deserializer.

• **kwargs** – other optional arguments of `XMLSchemaBase.iter_encode()` and options for converter.

**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.

### A.3 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 `xmlschema.XMLSchemaBase`.

**XMLSchema**
alias of `XMLSchema10`

**class `XMLSchemaBase`**

Base class for an XML Schema instance.

**Parameters**

• **source** – a 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 or an `XMLResource` instance. A multi source initialization is supported providing a not empty list of XSD sources.

- **namespace** – is an optional argument that contains the URI of the namespace that has to used in case the schema has no namespace (chameleon schema). For other cases, when specified, it must be equal to the `targetNamespace` of the schema.

- **validation** – the XSD validation mode to use for build the schema, that can be 'strict' (default), 'lax' or 'skip'.

- **global_maps** – is an optional argument containing an `XsdGlobals` instance, a mediator object for sharing declaration data between dependents schema instances.

- **converter** – 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** – schema extra location hints, that can include custom resource locations (e.g. local XSD file instead of remote resource) or additional namespaces to import after processing schema’s import statements. Can be a dictionary or a sequence of couples (namespace URI, resource URL). Extra locations passed using a tuple container are not normalized.
• **base_url** – 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.

• **allow** – the security mode for accessing resource locations. Can be ‘all’, ‘remote’, ‘local’ or ‘sandbox’. Default is ‘all’ that means all types of URLs are allowed. With ‘remote’ only remote resource URLs are allowed. With ‘local’ only file paths and URLs are allowed. With ‘sandbox’ only file paths and URLs that are under the directory path identified by source or by the base_url argument are allowed.

• **defuse** – defines when to defuse XML data using a SafeXMLParser. Can be ‘always’, ‘remote’ or ‘never’. For default defuses only remote XML data.

• **timeout** – the timeout in seconds for fetching resources. Default is 300.

• **urimapper** – an optional URI mapper for using relocated or URN-addressed resources. Can be a dictionary or a function that takes the URI string and returns a URL, or the argument if there is no mapping for it.

• **build** – defines whether build the schema maps. Default is True.

• **use_meta** – if True the schema processor uses the validator 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.

• **usefallback** – if True the schema processor uses the validator fallback location hints to load well-known namespaces (e.g. xhtml).

• **xpath3** – if True an XSD 1.1 schema instance uses the XPath 3 processor for assertions. For default a full XPath 2.0 processor is used for XSD 1.1 assertions.

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

• **BASE_SCHEMAS** – a dictionary from namespace to schema resource for meta-schema bases.

• **fallback_locations** – fallback schema location hints for other standard namespaces.

• **meta_schema** – the XSD meta-schema instance.

• **attribute_form_default** – the schema’s attributeFormDefault attribute. Default is ‘unqualified’.

• **element_form_default** – the schema’s elementFormDefault attribute. Default is ‘unqualified’.

• **block_default** – the schema’s blockDefault attribute. Default is ‘‘.

• **final_default** – the schema’s finalDefault attribute. Default is ‘‘.

• **default_attributes** – the XSD 1.1 schema’s defaultAttributes attribute. Default is None.

• **xpath_tokens** – 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** – 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.

- **maps** – XSD global declarations/definitions maps. This is an instance of _XsdGlobals_, that stores the global_maps argument or a new object when this argument is not provided.

- **converter** – the default converter used for XML data decoding/encoding.

- **locations** – schema location hints.

- **namespaces** – a dictionary that maps from the prefixes used by the schema into namespace URI.

- **imports** – 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 comprehends schemas included by “xs:redefine” or “xs:override” statements.

- **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.

```
meta_schema:  XMLSchemaBase | None = None

root
    Root element of the schema.

get_text() → str
    Returns the source text of the XSD schema.

name:  str | None = None

url
    Schema resource URL, is _None_ if the schema is built from an Element or a string.

base_url
    The base URL of the source of the schema.

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 ‘‘.

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.

simple_types
Returns a list containing the global simple types.

complex_types
Returns a list containing the global complex types.

classmethod builtin_types() → NamespaceView[T | None]
Returns the XSD built-in types of the meta-schema.

classmethod create_meta_schema(source: str | None = None, base_schemas: None | Dict[str, str] | List[Tuple[str, str]] = None, global_maps: XsdGlobals | None = None) → T | 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 is used as substitute for class 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.

create_any_content_group(parent: XsdComplexType | XsdGroup, any_element: XsdAnyElement | None = None) → XsdGroup
Creates a model group related to schema instance that accepts any content.

Parameters

• parent – the parent component to set for the 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: XsdComplexType | XsdElement) → XsdAttributeGroup
Creates an attribute group related to schema instance that accepts any attribute.

Parameters

parent – the parent component to set for the attribute group.

create_any_type() → XsdComplexType
Creates a xs:anyType equivalent type related with the wildcards connected to global maps of the schema instance in order to do a correct namespace lookup during wildcards validation.

get_locations(namespace: str) → List[str]
Get a list of location hints for a namespace.
include_schema(location: str, base_url: str | None = None, build: bool = False) \rightarrow T | 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.
- **build** – defines when to build the imported schema, the default is to not build.

Returns
the included `XMLSchema` instance.

import_schema(namespace: str, location: str, base_url: str | None = None, force: bool = False, build: bool = False) \rightarrow T | None

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.

add_schema(source: T | None, namespace: str | None = None, build: bool = False) \rightarrow T | None

Add another schema source to the maps of the instance.

Parameters
- **source** – a 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** – is an optional argument that contains the URI of the namespace that has to used in case the schema has no namespace (chameleon schema). For other cases, when specified, it must be equal to the `targetNamespace` of the schema.
- **build** – defines when to build the imported schema, the default is to not build.

Returns
the added `XMLSchema` instance.

export(target: str, save_remote: bool = False, remove_residuals: bool = True, exclude_locations: List[str] | None = None) \rightarrow None

Exports a schema instance. The schema instance is exported to a directory with also the hierarchy of imported/included schemas.

Parameters
- **target** – a path to a local empty directory.
- **save_remote** – if `True` is provided saves also remote schemas.
- **remove_residuals** – for default removes residual schema locations from redundant import statements.
- **exclude_locations** – explicitly exclude schema locations from substitution or removal.
resolve_qname\( (qname: str, namespace_imported: bool = True) \rightarrow str \)

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\( (schema: T | None = None) \rightarrow Iterator[T | None | Tuple[Any, ...]] \)

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\( (xsd_classes: T | None = None) \rightarrow Iterator[XsdComponent | T | None] \)

Iterates yielding the schema and its components. For default includes all the relevant components of the schema, excluding only facets and empty attribute groups. The first returned component is the schema itself.

**Parameters**

- **xsd_classes** – provide a class or a tuple of classes to restrict the range of component types yielded.

build\( () \rightarrow None \)

Builds the schema’s XSD global maps.

clear\( () \rightarrow None \)

Clears the schema’s XSD global maps.

built

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\( (converter: T | None = None, **kwargs: Any) \rightarrow XMLSchemaConverter \)

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.

• **kwargs** — optional arguments for initialize the converter instance.

**Returns**
a converter instance.

```python
validate(source: T | None | XMLResource, path: str | None = None, schema_path: str | None = None, use_defaults: bool = True, namespaces: T | None = None, max_depth: int | None = None, extra_validator: T | None = None, validation_hook: T | None = None, allow_empty: bool = True, use_location_hints: bool = False) → None
```

Validates 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 a 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.

• **max_depth** — maximum level of validation, for default there is no limit. With lazy resources is set to `source.lazy_depth` for managing lazy validation.

• **extra_validator** — an optional function for performing non-standard validations on XML data. The provided function is called for each traversed element, with the XML element as 1st argument and the corresponding XSD element as 2nd argument. It can be also a generator function and has to raise/yield `XMLSchemaValidationError` exceptions.

• **validation_hook** — an optional function for stopping or changing validation at element level. The provided function must accept two arguments, the XML element and the matching XSD element. If the value returned by this function is evaluated to false then the validation process continues without changes, otherwise the validation process is stopped or changed. If the value returned is a validation mode the validation process continues changing the current validation mode to the returned value, otherwise the element and its content are not processed. The function can also stop validation suddenly raising a `XmlSchemaStopValidation` exception.

• **allow_empty** — for default providing a path argument empty selections of XML data are allowed. Provide `False` to generate a validation error.

• **use_location_hints** — for default schema locations hints provided within XML data are ignored in order to avoid the change of schema instance. Set this option to `True` to activate dynamic schema loading using schema location hints.

**Raises**

`XMLSchemaValidationError` if the XML data instance is invalid.

```python
is_valid(source: T | None | XMLResource, path: str | None = None, schema_path: str | None = None, use_defaults: bool = True, namespaces: T | None = None, max_depth: int | None = None, extra_validator: T | None = None, validation_hook: T | None = None, allow_empty: bool = True, use_location_hints: bool = False) → bool
```
Like `validate()` except that does not raise an exception but returns `True` if the XML data instance is valid, `False` if it is invalid.

```python
iter_errors(source: T | None | XMLResource, path: str | None = None, schema_path: str | None = None,
            use_defaults: bool = True, namespaces: T | None = None,
            max_depth: int | None = None, extra_validator: T | None = None, validation_hook: T | None = None, allow_empty: bool =
            True, use_location_hints: bool = False) → Iterator[XMLSchemaValidationError]
```

Creates an iterator for the errors generated by the validation of an XML data against the XSD schema/component instance. Accepts the same arguments of `validate()`.

```python
decode(source: T | None | XMLResource, path: str | None = None, schema_path: str | None = None,
        validation: str = 'strict', *args: Any, **kwargs: Any) → Any | None
```

Decodes XML data. Takes the same arguments of the method `iter_decode()`.

```python
iter_decode(source: T | None | XMLResource, path: str | None = None, schema_path: str | None = None,
            validation: str = 'lax', process_namespaces: bool = True, namespaces: T | None = None,
            use_defaults: bool = True, use_location_hints: bool = False, decimal_type: Type[Union[Decim
            alType | Decimal]] | None = None, datetime_types: bool = False, binary_types: bool = False,
            converter: T | None = None, filler: T | None = None, fill_missing: bool = False, keep_empty: bool = False,
            keep_unknown: bool = False, process_skipped: bool = False, max_depth: int | None = None,
            depth_filler: T | None = None, extra_validator: T | None = None, validation_hook: T | None =
            None, element_hook: T | None = None, errors: List[XMLSchemaValidationError] | None = None,
            **kwargs: Any) → Iterator[Union[DataObject, XMLSchemaValidationError]]
```

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 a
  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 de-
  coding. 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** – whether to use namespace information in the decoding process, using
  the map provided with the argument `namespaces` and the namespace declarations
  extracted from the XML document.

- **namespaces** – is an optional mapping from namespace prefix to URI that inte-
  grate/override the root namespace declarations of the XML source. In case of prefix collision
  an alternate prefix is used for the root XML namespace declaration.

- **use_defaults** – whether to use default values for filling missing data.

- **use_location_hints** – for default schema locations hints provided within XML data are
  ignored in order to avoid the change of schema instance. Set this option to `True` to activate
  dynamic schema loading using schema location hints.

- **decimal_type** – conversion type for Decimal objects (generated by `xs: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 kept decoded, otherwise their origin XML string is returned.

• **binary_types** – if set to `True` `xs:hexBinary` and `xs:base64Binary` types are kept decoded, otherwise their origin XML string is returned.

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

• **filler** – an optional callback function to fill undecodable data with a typed value. The callback function must accept 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.

• **keep_empty** – if set to `True` empty elements that are valid are decoded with an empty string value instead of a `None`.

• **keep_unknown** – if set to `True` unknown tags are kept and are decoded with `xs:anyType`. For default unknown tags not decoded by a wildcard are discarded.

• **process_skipped** – process XML data that match a wildcard with `processContents='skip'`.

• **max_depth** – maximum level of decoding, for default there is no limit. With lazy resources is set to `source.lazy_depth` for managing lazy decoding.

• **depth_filler** – an optional callback function to replace data over the `max_depth` level. The callback function must accept one positional argument, that can be an XSD Element. If not provided deeper data are replaced with `None` values.

• **extra_validator** – an optional function for performing non-standard validations on XML data. The provided function is called for each traversed element, with the XML element as 1st argument and the corresponding XSD element as 2nd argument. It can be also a generator function and has to raise/yield `XMLSchemaValidationError` exceptions.

• **validation_hook** – an optional function for stopping or changing validated decoding at element level. The provided function must accept two arguments, the XML element and the matching XSD element. If the value returned by this function is evaluated to `false` then the decoding process continues without changes, otherwise the decoding process is stopped or changed. If the value returned is a validation mode the decoding process continues changing the current validation mode to the returned value, otherwise the element and its content are not decoded.

• **value_hook** – an optional function that will be called with any decoded atomic value and the XSD type used for decoding. The return value will be used instead of the original value.

• **element_hook** – an optional function that is called with decoded element data before calling the converter decode method. Takes an `ElementData` instance plus optionally the XSD element and the XSD type, and returns a new `ElementData` instance.

• **errors** – optional internal collector for validation errors.

• **kwargs** – keyword arguments with other options for converters.

**Returns**

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

```python
def encode(obj: Any, path: str | None = None, validation: str = 'strict', *args: Any, **kwargs: Any) -> Any | None
```

Encodes to XML data. Takes the same arguments of the method `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: Any, path: str | None = None, validation: str = 'lax', namespaces: T | None = None, use_defaults: bool = True, converter: T | None = None, unordered: bool = False, process_skipped: bool = False, max_depth: int | None = None, **kwargs: Any) → Iterator[T | None | XMLSchemaValidationError]

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.
• use_defaults – whether to use default values for filling missing data.
• 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.
• process_skipped – process XML decoded data that match a wildcard with processContents='skip'.
• max_depth – maximum level of encoding, for default there is no limit.
• kwargs – keyword arguments with other options for building the converter instance.

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

A.4 Global maps API

class XsdGlobals(validator: T | None, validation: str = 'strict')

Mediator class for related XML schema instances. It stores the global declarations defined in the registered schemas. Register a schema to add its 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() → None
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: Iterable[T | None] | None = None, validation: str = 'strict') → None
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: bool = False, only_unbuilt: bool = False) → None`

Clears the instance maps and schemas.

**Parameters**

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

`copy(validation: str | None = None, validator: T | None = None) → XsdGlobals`

Creates a shallow copy of the object. The associated schemas do not change the original global maps. This is useful for sharing the same meta-schema without copying the full tree objects, saving time and memory.

`iter_globals() → Iterator[T | None]`

Creates an iterator for the XSD global components of built schemas.

`iter_schemas() → Iterator[T | None]`

Creates an iterator for the registered schemas.

`lookup(tag: str, qname: str) → T | None`

General lookup method for XSD global components.

**Parameters**

• **tag** – the expanded QName of the XSD the global declaration/definition (e.g. `'{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: T | None) → None`

Registers an XMLSchema instance.

`property unbuilt: List[XsdComponent | T | None]`

Property that returns a list with unbuilt components.

### A.5 Converters API

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 ElementData**(tag, text, content, attributes, xmlns)

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 El-
ment’s children (used for unordered input data). attributes can be None or a dictionary containing the Element’s
attributes, xmlns can be None or a list of couples containing namespace declarations.

```python
class XMLSchemaConverter(namespaces: T | None = None, dict_class: Type[Dict[str, Any]] | None = None,
list_class: Type[List[Any]] | None = None, etree_element_class: Type[Element] |
None = None, text_key: str | None = '$', attr_prefix: str | None = '@', cdata_prefix:
str | None = None, indent: int = 4, process_namespaces: bool = True,
strip_namespaces: bool = False, xmlns_processing: str | None = None, source:
XMLResource | None = None, preserve_root: bool = False, force_dict: bool =
False, force_list: bool = False, **kwargs: Any)
```

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 interfac-
ing the converter with the decoding/encoding process. The method `element_decode` accepts an ElementData
tuple, containing the element parts, and returns a data structure. The method `element_encode` accepts a data
structure and returns an ElementData tuple. For default character data parts are ignored. Prefixes and text key
can be changed also using alphanumeric values but ambiguities with schema elements could affect XML data
re-encoding.

**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).
- `process_namespaces` – whether to use namespace information in name mapping methods.
  If set to `False` then the name mapping methods simply return the provided name.
- `strip_namespaces` – if set to `True` removes namespace declarations from data and name-
  space information from names, during decoding or encoding. Defaults to `False`.
- `xmlns_processing` – defines the processing mode of XML namespace declarations. Can
  be ‘stacked’, ‘collapsed’, ‘root-only’ or ‘none’, with the meaning defined for the Namespace-
  Mapper base class. For default the xmlns processing mode is chosen between ‘stacked’,
  ‘collapsed’ and ‘none’, depending on the provided XML source and the capabilities and the
  settings of the converter instance.
- `source` – the origin of XML data. Can be an `XMLResource` instance or `None`.
- `preserve_root` – if set to `True` the root element is preserved, wrapped into a single-
  item dictionary. Applicable only to default converter, to `UnorderedConverter` and to
  `ParkerConverter`.
- `force_dict` – if set to `True` complex elements with simple content are decoded with a dic-
  tionary also if there are no decoded attributes. Applicable only to default converter and to
  `UnorderedConverter`. Defaults to `False`.

A.5. Converters API 55
• **force_list** – if set to True child elements are decoded within a list in any case. Applicable only to default converter and to *UnorderedConverter*. 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
- **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.

**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**(*keep_namespaces*: bool = True, **kwargs: Any*) → *XMLSchemaConverter*

Creates a new converter instance from the existing, replacing options provided with keyword arguments.

**Parameters**

- **keep_namespaces** – whether to keep the namespaces of the converter if they are not replaced by a keyword argument.

**map_attributes**(*attributes*: Iterable[Tuple[str, Any]]) → Iterator[Tuple[str, Any]]

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*: Iterable[Tuple[str, Any, Any]]) → Iterator[Tuple[str, Any, Any]]

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.


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 is set for the converter instance.

[element_decode](#)

```python
def element_decode(data: ElementData, xsd_element: XsdElement, xsd_type: T | None = None, level: int = 0) -> Any
```

Converts a decoded element data to a data structure.

Parameters
- **data** – ElementData instance decoded from an Element node.
- **xsd_element** – the XsdElement associated to decode the data.
- **xsd_type** – optional XSD type for supporting dynamic type through xsi:type or xs:alternative.
- **level** – the level related to the decoding process (0 means the root).

Returns
a data structure containing the decoded data.

[element_encode](#)

```python
def element_encode(obj: Any, xsd_element: XsdElement, level: int = 0) -> ElementData
```

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](#)

```python
def map_qname(qname: str) -> str
```

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](#)

```python
def unmap_qname(qname: str, name_table: Container[str | None] | None = None, xmlns: List[Tuple[str, str]] | None = None) -> str
```

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.
• **xmlns** – an optional list of namespace declarations that integrate or override the namespace map.

**Returns**
- a QName in extended format or a local name.

```python
class UnorderedConverter(namespaces: T | None = None, dict_class: Type[Dict[str, Any]] | None = None, list_class: Type[List[Any]] | None = None, etree_element_class: Type[Element] | None = None, text_key: str | None = '$', attr_prefix: str | None = '@', cdata_prefix: str | None = None, process_namespaces: bool = True, strip_namespaces: bool = False, xmlns_processing: str | None = None, source: XMLResource | None = None, preserve_root: bool = False, force_dict: bool = False, force_list: bool = False, **kwargs: Any)
```

Same as `XMLSchemaConverter` but `XMLSchemaConverter.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.

```python
class ParkerConverter(namespaces: T | None = None, dict_class: Type[Dict[str, Any]] | None = None, list_class: Type[List[Any]] | None = None, preserve_root: bool = False, **kwargs: Any)
```

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`.
- **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 BadgerFishConverter(namespaces: T | None = None, dict_class: Type[Dict[str, Any]] | None = None, list_class: Type[List[Any]] | None = None, **kwargs: Any)
```

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`.
- **list_class** – List class to use for decoded data. Default is `list`.

```python
class AbderaConverter(namespaces: T | None = None, dict_class: Type[Dict[str, Any]] | None = None, list_class: Type[List[Any]] | None = None, **kwargs: Any)
```

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`.

• **list_class** – List class to use for decoded data. Default is `list`.

class JsonMLConverter:
```python
definition:
    namespaces: T | None = None,
    dict_class: Type[Dict[str, Any]] | None = None,
    list_class: Type[List[Any]] | None = None,
    **kwargs: Any
```
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`.

• **list_class** – List class to use for decoded data. Default is `list`.

class ColumnarConverter:
```python
definition:
    namespaces: T | None = None,
    dict_class: Type[Dict[str, Any]] | None = None,
    list_class: Type[List[Any]] | None = None,
    attr_prefix: str | None = '',
    **kwargs: Any
```
XML Schema based converter class for columnar formats.

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`.

• **attr_prefix** – used as separator string for renaming the decoded attributes. Can be the empty string (the default) or a single/double underscore.

### A.6 Data objects API

class DataElement:
```python
definition:
    tag: str
    value: Any | None = None,
    attrib: Dict[str, Any] | None = None,
    nsmap: MutableMapping[str, str] | None = None,
    xmlns: List[Tuple[str, str]] | None = None,
    xsd_element: XsdElement | None = None,
    xsd_type: T | None = None
```
Data Element, an Element like object with decoded data and schema bindings.

Parameters

• **tag** – a string containing a QName in extended format.

• **value** – the simple typed value of the element.

• **attrib** – the typed attributes of the element.

• **nsmap** – an optional map from prefixes to namespaces.

• **xsd_element** – an optional XSD element association.

• **xsd_type** – an optional XSD type association. Can be provided also if the instance is not bound with an XSD element.

class DataElementConverter:
```python
definition:
    namespaces: T | None = None,
    data_element_class: Type[DataElement] | None = None,
    map_attribute_names: bool = True,
    **kwargs: Any
```
XML Schema based converter class for DataElement objects.

Parameters
• **namespaces** – a dictionary map from namespace prefixes to URI.

• **data_element_class** – MutableSequence subclass to use for decoded data. Default is `DataElement`.

• **map_attribute_names** – define if map the names of attributes to prefixed form. Defaults to `True`. If `False` the names are kept to extended format.

```python
class DataBindingConverter:
    def __init__(self, namespaces: T | None = None, data_element_class: Type[DataElement] | None = None, map_attribute_names: bool = True, **kwargs: Any):
        pass
```

A `DataElementConverter` that uses XML data binding classes for decoding. Takes the same arguments of its parent class but the argument `data_element_class` is used for define the base for creating the missing XML binding classes.

### A.7 URL normalization API

```python
normalize_url(url: str, base_url: str | None = None, keep_relative: bool = False) -> str
```

Returns a normalized URL eventually joining it to a base URL if it’s a relative path. Path names are converted to ‘file’ scheme URLs.

- **Parameters**
  - `url` – a relative or absolute URL.
  - `base_url` – a reference base URL.
  - `keep_relative` – if set to `True` keeps relative file paths, which would not strictly conformant to specification (RFC 8089), because `urlopen()` doesn’t accept a simple pathname.

- **Returns**
  - a normalized URL string.

```python
normalize_locations(locations: T | None, base_url: str | None = None, keep_relative: bool = False) -> T | None
```

Returns a list of normalized locations. The locations are normalized using the base URL of the instance.

- **Parameters**
  - `locations` – a dictionary or a list of couples containing namespace location hints.
  - `base_url` – the reference base URL for construct the normalized URL from the argument.
  - `keep_relative` – if set to `True` keeps relative file paths, which would not strictly conformant to URL format specification.

- **Returns**
  - a list of couples containing normalized namespace location hints.

### A.8 XML resources API

```python
fetch_resource(location: str, base_url: str | None = None, timeout: int = 30) -> str
```

Fetches a resource by trying to access it. If the resource is accessible returns its normalized URL, otherwise raises an `urllib.error.URLError`.

- **Parameters**
  - `location` – a 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.

fetch_schema_locations(source: XMLResource | T | None, locations: T | None = None, base_url: str | None = None, allow: str = 'all', defuse: str = 'remote', timeout: int = 30, uri_mapper: T | None = None, root_only: bool = True) → Tuple[str, T | None]

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 a 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.
• allow – the same argument of the XMLResource, applied to location hints only.
• defuse – the same argument of the XMLResource.
• timeout – the same argument of the XMLResource but with a reduced default.
• uri_mapper – an optional argument for building the schema from location hints.
• root_only – if True extracts from the XML source only the location hints of the root element.

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

fetch_schema(source: XMLResource | T | None, locations: T | None = None, base_url: str | None = None, allow: str = 'all', defuse: str = 'remote', timeout: int = 30, uri_mapper: T | None = None, root_only: bool = True) → str

Like fetch_schema_locations() but returns only the URL of a loadable XSD schema from location hints fetched from the source or provided by argument.

class XMLResource(source: T | None, base_url: None | str | Path | bytes = None, allow: str = 'all', defuse: str = 'remote', timeout: int = 300, lazy: bool = False, thin_lazy: bool = True, uri_mapper: T | None = None)

XML resource reader based on ElementTree and urllib.

Parameters
• source – a string containing the XML document or file path or a 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. For security the access to a local file resource is always denied if the base_url is a remote URL.
• allow – defines the security mode for accessing resource locations. Can be ‘all’, ‘remote’, ‘local’, ‘sandbox’ or ‘none’. Default is ‘all’, which means all types of URLs are allowed. With ‘remote’ only remote resource URLs are allowed. With ‘local’ only file paths and URLs are allowed. With ‘sandbox’ only file paths and URLs that are under the directory
path identified by the `base_url` argument are allowed. If you provide ‘none’, no resources will be allowed from any location.

- **defuse** – defines when to defuse XML data using a `SafeXMLParser`. Can be ‘always’, ‘remote’, ‘nonlocal’ or ‘never’. For default defuses only remote XML data. With ‘always’ all the XML data that is not already parsed is defused. With ‘nonlocal’ it defuses unparsed data except local files. With ‘never’ no XML data source is defused.

- **timeout** – the timeout in seconds for the connection attempt in case of remote data.

- **lazy** – if a value `False` or 0 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. A positive integer also defines the depth at which the lazy resource can be better iterated (`True` means 1).

- **thin_lazy** – for default, in order to reduce the memory usage, during the iteration of a lazy resource at `lazy_depth` level, deletes also the preceding elements after the use.

- **uri_mapper** – an optional URI mapper for using relocated or URN-addressed resources. Can be a dictionary or a function that takes the URI string and returns a URL, or the argument if there is no mapping for it.

**root**

The XML tree root Element.

**text**

The XML text source, `None` if it’s not available.

**name**

The source name, is `None` if the instance is created from an Element or a string.

**url**

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

**base_url**

The effective base URL used for completing relative locations.

**filepath**

The resource filepath if the instance is created from a local file, `None` otherwise.

**namespace**

The namespace of the XML resource.

```python
def parse(source: T | None, lazy: bool | int = False) -> None
```

```python
def tostring(namespaces: MutableMapping[str, str] | None = None, indent: str = '', max_lines: int | None = None, spaces_for_tab: int = 4, xml_declaration: bool = False, encoding: str = 'unicode', method: str = 'xml') -> str
```

Serialize an XML resource to a string.

**Parameters**

- **namespaces** – is an optional mapping from namespace prefix to URI. Provided namespaces are registered before serialization. Ignored if the provided `elem` argument is a lxml Element instance.

- **indent** – the baseline indentation.

- **max_lines** – if truncate serialization after a number of lines (default: do not truncate).
• **spaces_for_tab** – number of spaces for replacing tab characters. For default tabs are replaced with 4 spaces, provide None to keep tab characters.

• **xml_declaration** – if set to True inserts the XML declaration at the head.

• **encoding** – if “unicode” (the default) the output is a string, otherwise it’s binary.

• **method** – is either “xml” (the default), “html” or “text”.

  Returns a Unicode string.

  **open() → IO**

  Returns an 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() → None**

  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() → bool**

  Returns True if the XML resource is lazy.

  **lazy_depth**

  The depth at which the XML tree of the resource is fully loaded during iterations methods. Is a positive integer for lazy resources and 0 for fully loaded XML trees.

  **is_remote() → bool**

  Returns True if the resource is related with remote XML data.

  **is_local() → bool**

  Returns True if the resource is related with local XML data.

  **is_loaded() → bool**

  Returns True if the XML text of the data source is loaded.

  **iter**(tag: str | None = None) → Iterator[T | None]

  XML resource tree iterator. If tag is not None or ‘*’, only elements whose tag equals tag are returned from the iterator. In a lazy resource the yielded elements are full over or at lazy_depth level, otherwise are incomplete and thin for default.

  **iter_depth**(mode: int = 1, ancestors: List[T | None] | None = None) → Iterator[T | None]

  Iterates XML subtrees. For fully loaded resources yields the root element. On lazy resources the argument mode can change the sequence and the completeness of yielded elements. There are four possible modes, that generate different sequences of elements:

  1. Only the elements at depth_level level of the tree
  2. Only the elements at depth_level level of the tree removing the preceding elements of ancestors (thin lazy tree)
  3. Only a root element pruned at depth_level
  4. The elements at depth_level and then a pruned root
  5. An incomplete root at start, the elements at depth_level and a pruned root

  **Parameters**

  • **mode** – an integer in range [1..5] that defines the iteration mode.
• **ancestors** – provide a list for tracking the ancestors of yielded elements.

```python
iterfind(path: str, namespaces: T | None = None, ancestors: List[T | None] | None = None) → Iterator[T | None]
```

Apply XPath selection to XML resource that yields full subtrees.

**Parameters**

- **path** – an XPath 2.0 expression that selects element nodes. Selecting other values or nodes raise an error.
- **namespaces** – an optional mapping from namespace prefixes to URIs used for parsing the XPath expression.
- **ancestors** – provide a list for tracking the ancestors of yielded elements.

```python
find(path: str, namespaces: T | None = None, ancestors: List[T | None] | None = None) → T | None
```

```python
findall(path: str, namespaces: T | None = None) → List[T | None]
```

```python
iter_location_hints(tag: str | None = None) → Iterator[Tuple[str, str]]
```

Yields all schema location hints of the XML resource. If tag is not None or ‘*’, only location hints of elements whose tag equals tag are returned from the iterator.

```python
get_namespaces(namespaces: T | None = None, root_only: bool = True) → T | None
```

Extracts namespaces with related prefixes from the XML resource. If a duplicate prefix is encountered in a xmlns declaration, and this is mapped to 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 it uses the prefix ‘default’ as substitute.

**Parameters**

- **namespaces** – is an optional mapping from namespace prefix to URI that integrate/override the namespace declarations of the root element.
- **root_only** – if True extracts only the namespaces declared in the root element, otherwise scan the whole tree for further namespace declarations. A full namespace map can be useful for cases where the element context is not available.

**Returns**

a dictionary for mapping namespace prefixes to full URI.

```python
get_locations(locations: T | None = None, root_only: bool = True) → T | None
```

Extracts a list of schema location hints from the XML resource. The locations are normalized using the base URL of the instance.

**Parameters**

- **locations** – a sequence of schema location hints inserted before the ones extracted from the XML resource. Locations passed within a tuple container are not normalized.
- **root_only** – if True extracts only the location hints of the root element.

**Returns**

a list of couples containing normalized location hints.

class XmlDocument(source: T | None, schema: XMLSchemaBase | T | None = None, cls: Type[XMLSchemaBase] | None = None, validation: str = 'strict', namespaces: T | None = None, locations: T | None = None, base_url: str | None = None, allow: str = 'all', defuse: str = 'remote', timeout: int = 300, lazy: T | None = False, thin_lazy: bool = True, uri_mapper: T | None = None, use_location_hints: bool = True)
An XML document bound with its schema. If no schema is get from the provided context and validation argument is `skip` the XML document is associated with a generic schema, otherwise a ValueError is raised.

**Parameters**

- **source** – a string containing XML data or a file path or a URL or a file like object or an ElementTree or an Element.
- **schema** – can be a `xmlschema.XMLSchema` instance or a file-like object or a file path or a URL of a resource or a string containing the XSD schema.
- **cls** – class to use for building the schema instance (for default `XMLSchema10` is used).
- **validation** – the XSD validation mode to use for validating the XML document, that can be 'strict' (default), 'lax' or 'skip'.
- **namespaces** – is an optional mapping from namespace prefix to URI.
- **locations** – resource location hints, that can be a dictionary or a sequence of couples (namespace URI, resource URL).
- **base_url** – the base URL for base `xmlschema.XMLResource` initialization.
- **allow** – the security mode for base `xmlschema.XMLResource` initialization.
- **defuse** – the defuse mode for base `xmlschema.XMLResource` initialization.
- **timeout** – the timeout for base `xmlschema.XMLResource` initialization.
- **lazy** – the lazy mode for base `xmlschema.XMLResource` initialization.
- **thin_lazy** – the thin_lazy option for base `xmlschema.XMLResource` initialization.
- **uri_mapper** – an optional argument for building the schema from location hints.
- **use_location_hints** – for default, in case a schema instance has to be built, uses also schema locations hints provided within XML data. Set this option to `False` to ignore these schema location hints.

### A.9 Translation API

**activate**

```python
activate(localedir: None | str | Path = None, languages: Iterable[str] | None = None, fallback: bool = True, install: bool = False) → None
```

Activate translation of xmlschema parsing/validation error messages.

**Parameters**

- **localedir** – a string or Path-like object to locale directory
- **languages** – list of language codes
- **fallback** – for default fallback mode is activated
- **install** – if `True` installs function `_()` in Python’s builtins namespace

**deactivate**

```python
deactivate() → None
```

Deactivate translation of xmlschema parsing/validation error messages.
A.10 Namespaces API

Classes for converting namespace representation or for accessing namespace objects:

```python
class NamespaceResourcesMap(*args: Any, **kwargs: Any)
    Dictionary for storing information about namespace resources. The values are lists of objects. Setting an existing value appends the object to the value. Setting a value with a list sets/replaces the value.

class NamespaceMapper(namespaces: T | None = None, process_namespaces: bool = True, strip_namespaces: bool = False, xmlns_processing: str | None = None, source: Any | None = None)
    A class to map/unmap namespace prefixes to URIs. An internal reverse mapping from URI to prefix is also maintained for keep name mapping consistent within updates.

    Parameters

    • namespaces – initial data with mapping of namespace prefixes to URIs.
    • process_namespaces – whether to use namespace information in name mapping methods. If set to False then the name mapping methods simply return the provided name.
    • strip_namespaces – if set to True then the name mapping methods return the local part of the provided name.
    • xmlns_processing – defines the processing mode of XML namespace declarations. The preferred mode is ‘stacked’, the mode that processes the namespace declarations using a stack of contexts related with elements and levels. This is the processing mode that always matches the XML namespace declarations defined in the XML document. Provide ‘collapsed’ for loading all namespace declarations of the XML source in a single map, renaming colliding prefixes. Provide ‘root-only’ to use only the namespace declarations of the XML document root. Provide ‘none’ to not use any namespace declaration of the XML document. For default the xmlns processing mode is ‘stacked’ if the XML source is an XMLResource instance, otherwise is ‘none’.
    • source – the origin of XML data. Can be an XMLResource instance, an XML decoded data or None.

class NamespaceView(qname_dict: Dict[str, T], namespace_uri: str)
    A read-only map for filtered access to a dictionary that stores objects mapped from QNames in extended format.
```

A.11 XPath API

Implemented through a mixin class on XSD schemas and elements.

```python
class ElementPathMixin
    Mixin abstract class for enabling ElementTree and XPath 2.0 API on XSD components.

    Variables

    • text – the Element text, for compatibility with the ElementTree API.
    • tail – the Element tail, 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: str, default: Any = None) → Any

Gets an Element attribute. For compatibility with the ElementTree API.

iter(tag: str | None = None) → Iterator[E]

Creates an iterator for the XSD element and its subelements. If tag is not None or ‘*’, only XSD elements whose names match 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: str | None = None) → Iterator[E]

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: str, namespaces: T | None = None) → E | 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 no match.

findall(path: str, namespaces: T | None = None) → List[E]

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: str, namespaces: T | None = None) → Iterator[E]

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 – an optional mapping from namespace prefix to full name.

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

A.12 Validation API

Implemented for XSD schemas, elements, attributes, types, attribute groups and model groups.

class ValidationMixin

Mixin for implementing XML data validators/decoders on XSD components. A derived class must implement the methods iter_decode and iter_encode.
is_valid(obj: ST, use_defaults: bool = True, namespaces: T | None = None, max_depth: int | None = None, extra_validator: T | None = None) → bool

Like validate() except that does not raise an exception but returns True if the XML data instance is valid, False if it is invalid.

validate(obj: ST, use_defaults: bool = True, namespaces: T | None = None, max_depth: int | None = None, extra_validator: T | None = None) → None

Validates XML data against the XSD schema/component instance.

Parameters

- **obj** – the XML data. 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.
- **max_depth** – maximum level of validation, for default there is no limit.
- **extra_validator** – an optional function for performing non-standard validations on XML data. The provided function is called for each traversed element, with the XML element as 1st argument and the corresponding XSD element as 2nd argument. It can be also a generator function and has to raise/yield xmlschema.XMLSchemaValidationError exceptions.

Raises

xmlschema.XMLSchemaValidationError if the XML data instance is invalid.

decode(obj: ST, validation: str = 'strict', **kwarg: Any) → DT | None

Decodes XML data.

Parameters

- **obj** – the XML data. Can be a string for an attribute or for 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.
- **kwarg** – 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

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

iter_decode(obj: ST, validation: str = 'lax', **kwarg: Any) → DT | None

Creates an iterator for decoding an XML source to a Python object.

Parameters

- **obj** – the XML data.
- **validation** – the validation mode. Can be ‘lax’, ‘strict’ or ‘skip.
- **kwarg** – keyword arguments for the decoder API.
Returns
Yields a decoded object, eventually preceded by a sequence of validation or decoding errors.

iter_encode(obj: Any, validation: str = 'lax', **kwargs: Any) → Any | None
Creates an iterator for encoding data to an Element tree.

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(obj: ST, use_defaults: bool = True, namespaces: T | None = None, max_depth: int | None = None, extra_validator: T | None = None) → Iterator[XMLSchemaValidationError]
Creates an iterator for the errors generated by the validation of an XML data against the XSD schema/component instance. Accepts the same arguments of validate().

encode(obj: Any, validation: str = 'strict', **kwargs: Any) → Any | None
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
xmlschema.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: Any, validation: str = 'lax', **kwargs: Any) → Any | None
Creates an iterator for encoding data to an Element tree.

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.
A.13 Particles API

Implemented for XSD model groups, elements and element wildcards.

class ParticleMixin(min_occurs: int = 1, max_occurs: int | None = 1)

Mixin for objects related to XSD Particle Schema Components:

https://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/structures.html#p
https://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/structures.html#t

Variables

• min_occurs – the minOccurs property of the XSD particle. Defaults to 1.
• max_occurs – the maxOccurs property of the XSD particle. Defaults to 1, a None value means ‘unbounded’.

is_empty() → bool
Tests if max_occurs == 0. A zero-length model group is considered empty.

is_emptiable() → bool
Tests if min_occurs == 0. A model group that can have zero-length is considered emptiable. For model groups the test outcome depends also on nested particles.

is_single() → bool
Tests if the particle has max_occurs == 1. For elements the test outcome depends also on parent group. For model groups the test outcome depends also on nested model groups.

is_multiple() → bool
Tests the particle can have multiple occurrences.

is_ambiguous() → bool
Tests if min_occurs != max_occurs.

is_univocal() → bool
Tests if min_occurs == max_occurs.

is_missing(occurs: int) → bool
Tests if provided occurrences are under the minimum.

is_over(occurs: int) → bool
Tests if provided occurrences are over the maximum.

A.14 Main XSD components

class XsdComponent(elem: Element, schema: T | None, parent: XsdComponent | None = None, name: str | None = None)

Class for XSD components. See: https://www.w3.org/TR/xmlschema-ref/

Parameters

• elem – ElementTree’s node containing the definition.
• schema – the XMLSchema object that owns the definition.
• parent – the XSD parent, None means that is a global component that has the schema as parent.
• **name** – name of the component, maybe overwritten by the parse of the `elem` argument.

**Variables**

- **qualified** (`bool`) – for name matching, unqualified matching may be admitted only for elements and attributes.

- **target_namespace**
  Property that references to schema's targetNamespace.

- **local_name**
  The local part of the name of the component, or `None` if the name is `None`.

- **qualified_name**
  The name of the component in extended format, or `None` if the name is `None`.

- **prefixed_name**
  The name of the component in prefixed format, or `None` if the name is `None`.

**is_global** () → `bool`

Returns `True` if the instance is a global component, `False` if it’s local.

**is_matching** (`name: str | None`, `default_namespace: str | None = None`, **kwargs: Any) → `bool`

Returns `True` if the component name is matching the name provided as argument, `False` otherwise. For XSD elements the matching is extended to substitutes.

**Parameters**

- **name** – a local or fully-qualified name.
- **default_namespace** – used by the XPath processor for completing the name argument in case it’s a local name.
- **kwargs** – additional options that can be used by certain components.

**tostring** (`indent: str = ''`, `max_lines: int | None = None`, `spaces_for_tab: int = 4`) → `str | bytes`

Serializes the XML elements that declare or define the component to a string.

**class XsdType** (`elem: Element`, `schema: T | None`, `parent: XsdComponent | None = None`, `name: str | None = None`)

Common base class for XSD types.

- **simple_type**
  Property that is the instance itself for a `simpleType`. For a `complexType` is the instance’s `content` if this is a `simpleType` or `None` if the instance’s `content` is a model group.

- **model_group**
  Property that is `None` for a `simpleType`. For a `complexType` is the instance’s `content` if this is a model group or `None` if the instance’s `content` is a `simpleType`.

- **has_complex_content** () → `bool`
  Returns `True` if the instance is a `complexType` with mixed or element-only content, `False` otherwise.

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

- **has_simple_content** () → `bool`
  Returns `True` if the instance has a simple content, `False` otherwise.

- **is_atomic** () → `bool`
  Returns `True` if the instance is an atomic `simpleType`, `False` otherwise.
static is_complex() → bool
Returns True if the instance is a complexType, False otherwise.

is_datetime() → bool
Returns True if the instance is a datetime/duration XSD builtin-type, False otherwise.

is_derived(other: T | None | Tuple[T | None, T | None], derivation: str | None = None) → bool
Returns True if the instance is derived from other, False otherwise. The optional argument derivation can be a string containing the words ‘extension’ or ‘restriction’ or both.

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

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

is_empty() → bool
Returns True if the instance has an empty content, False otherwise.

is_list() → bool
Returns True if the instance is a list simpleType, False otherwise.

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

class XsdElement(elem: Element, schema: T | None, parent: XsdComponent | None = None, build: bool = True)
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.

class XsdAttribute(elem: Element, schema: T | None, parent: XsdComponent | None = None, name: str | None = None)
Class for XSD 1.0 attribute declarations.

Variables

• type – the XSD simpleType of the attribute.

A.15 Other XSD components

A.15.1 Elements and attributes

class Xsd11Element(elem: Element, schema: T | None, parent: XsdComponent | None = None, build: bool = True)
Class for XSD 1.1 element declarations.

class Xsd11Attribute(elem: Element, schema: T | None, parent: XsdComponent | None = None, name: str | None = None)
Class for XSD 1.1 attribute declarations.
A.15.2 Types

class Xsd11ComplexType(elem: T | None, schema: T | None, parent: XsdComponent | None = None, name: str | None = None, **kwargs: Any)

Class for XSD 1.1 complexType definitions.

class XsdComplexType(elem: T | None, schema: T | None, parent: XsdComponent | None = None, name: str | None = None, **kwargs: Any)

Class for XSD 1.0 complexType definitions.

Variables

• attributes – the attribute group related with the complexType.
• content – the content of the complexType can be a model group or a simple type.
• mixed – if True the complex type has mixed content.

content: XsdGroup | XsdSimpleType = None

class XsdSimpleType(elem: T | None, schema: T | None, parent: XsdComponent | None = None, name: str | None = None, facets: Dict[str | None, XsdFacet | Callable[[Any], None] | List[XsdAssertionFacet]] | None = None)

Base class for simpleTypes definitions. Generally used only for instances of xs:anySimpleType.

enumeration

max_value

min_value

class XsdAtomicBuiltin(elem: T | None, schema: T | None, name: str, python_type: Type[Any], base_type: XsdAtomicBuiltin | None = None, admitted_facets: Set[str] | None = None, facets: Dict[str | None, XsdFacet | Callable[[Any], None] | List[XsdAssertionFacet]] | None = None, to_python: Any = None, from_python: Any = 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 XsdList(elem: T | None, schema: T | None, parent: XsdComponent | None = None, name: str | None = None)

Class for ‘list’ definitions. A list definition has an item_type attribute that refers to an atomic or union simpleType definition.

class Xsd11Union(elem: T | None, schema: T | None, parent: XsdComponent | None = None, name: str | None = None)

class XsdUnion(elem: T | None, schema: T | None, parent: XsdComponent | None = None, name: str | None = None)

Class for ‘union’ definitions. A union definition has a member_types attribute that refers to a ‘simpleType’ definition.

class Xsd11AtomicRestriction(elem: T | None, schema: T | None, parent: XsdComponent | None = None, name: str | None = None, facets: Dict[str | None, XsdFacet | Callable[[Any], None] | List[XsdAssertionFacet]] | None = None, base_type: T | None = None)

Class for XSD 1.1 atomic simpleType and complexType’s simpleContent restrictions.
class XsdAtomicRestriction(elem: T | None, schema: T | None, parent: XsdComponent | None = None, name: str | None = None, facets: Dict[str | None, XsdFacet | Callable[[Any], None] | List[XsdAssertionFacet]] | None = None, base_type: T | None = None)

Class for XSD 1.0 atomic simpleType and complexType’s simpleContent restrictions.

A.15.3 Attribute and model groups

class XsdAttributeGroup(elem: T | None, schema: T | None, parent: XsdComponent | None = None, derivation: str | None = None, base_attributes: XsdAttributeGroup | None = None)

Class for XSD attributeGroup definitions.

class Xsd11Group(elem: T | None, schema: T | None, parent: XsdComplexType | XsdGroup | None = None)

Class for XSD 1.1 model group definitions.

class XsdGroup(elem: T | None, schema: T | None, parent: XsdComplexType | XsdGroup | None = None)

Class for XSD 1.0 model group definitions.

A.15.4 Wildcards

class Xsd11AnyElement(elem: T | None, schema: T | None, parent: XsdComponent)

Class for XSD 1.1 any declarations.

class XsdAnyElement(elem: T | None, schema: T | None, parent: XsdComponent)

Class for XSD 1.0 any wildcards.

class Xsd11AnyAttribute(elem: Element, schema: T | None, parent: XsdComponent | None = None, name: str | None = None)

Class for XSD 1.1 anyAttribute declarations.

class XsdAnyAttribute(elem: Element, schema: T | None, parent: XsdComponent | None = None, name: str | None = None)

Class for XSD 1.0 anyAttribute wildcards.

class XsdOpenContent(elem: T | None, schema: T | None, parent: XsdComponent)

Class for XSD 1.1 openContent model definitions.

class XsdDefaultOpenContent(elem: T | None, schema: T | None)

Class for XSD 1.1 defaultOpenContent model definitions.

A.15.5 Identity constraints

class XsdIdentity(elem: T | None, schema: T | None, parent: XsdElement | None)

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 XsdSelector(elem: T | None, schema: T | None, parent: XsdIdentity | None)

Class for defining an XPath selector for an XSD identity constraint.
class **XsdFieldSelector**(elem: T | None, schema: T | None, parent: XsdIdentity | None)

Class for defining an XPath field selector for an XSD identity constraint.

class **Xsd11Unique**(elem: T | None, schema: T | None, parent: XsdElement | None)

class **XsdUnique**(elem: T | None, schema: T | None, parent: XsdElement | None)

class **Xsd11Key**(elem: T | None, schema: T | None, parent: XsdElement | None)

class **XsdKey**(elem: T | None, schema: T | None, parent: XsdElement | None)

class **Xsd11Keyref**(elem: T | None, schema: T | None, parent: XsdElement | None)

class **XsdKeyref**(elem: T | None, schema: T | None, parent: XsdElement | None)

Implementation of xs:keyref.

**Variables**

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

### A.15.6 Facets

class **XsdFacet**(elem: T | None, schema: T | None, parent: XsdList | XsdAtomicRestriction, base_type: T | None)

XML Schema constraining facets base class.

class **XsdWhiteSpaceFacet**(elem: T | None, schema: T | None, parent: XsdList | XsdAtomicRestriction, base_type: T | None)

XSD whiteSpace facet.

class **XsdLengthFacet**(elem: T | None, schema: T | None, parent: XsdList | XsdAtomicRestriction, base_type: T | None)

XSD length facet.

class **XsdMinLengthFacet**(elem: T | None, schema: T | None, parent: XsdList | XsdAtomicRestriction, base_type: T | None)

XSD minLength facet.

class **XsdMaxLengthFacet**(elem: T | None, schema: T | None, parent: XsdList | XsdAtomicRestriction, base_type: T | None)

XSD maxLength facet.

class **XsdMinInclusiveFacet**(elem: T | None, schema: T | None, parent: XsdList | XsdAtomicRestriction, base_type: T | None)

XSD minInclusive facet.

class **XsdMinExclusiveFacet**(elem: T | None, schema: T | None, parent: XsdList | XsdAtomicRestriction, base_type: T | None)

XSD minExclusive facet.

class **XsdMaxInclusiveFacet**(elem: T | None, schema: T | None, parent: XsdList | XsdAtomicRestriction, base_type: T | None)

XSD maxInclusive facet.

class **XsdMaxExclusiveFacet**(elem: T | None, schema: T | None, parent: XsdList | XsdAtomicRestriction, base_type: T | None)

XSD maxExclusive facet.
class XsdTotalDigitsFacet(elem: T | None, schema: T | None, parent: XsdList | XsdAtomicRestriction, base_type: T | None)

XSD totalDigits facet.

class XsdFractionDigitsFacet(elem: T | None, schema: T | None, parent: XsdAtomicRestriction, base_type: T | None)

XSD fractionDigits facet.

class XsdExplicitTimezoneFacet(elem: T | None, schema: T | None, parent: XsdList | XsdAtomicRestriction, base_type: T | None)

XSD 1.1 explicitTimezone facet.

class XsdAssertionFacet(elem: T | None, schema: T | None, parent: XsdList | XsdAtomicRestriction, base_type: T | None)

XSD 1.1 assertion facet for simpleType definitions.

class XsdEnumerationFacets(elem: T | None, schema: T | None, parent: XsdAtomicRestriction, base_type: T | None)

Sequence of XSD enumeration facets. Values are validates if match any of enumeration values.

class XsdPatternFacets(elem: T | None, schema: T | None, parent: XsdAtomicRestriction, base_type: T | None)

Sequence of XSD pattern facets. Values are validates if match any of patterns.

A.15.7 Others

class XsdAssert(elem: T | None, schema: T | None, parent: XsdComplexType, base_type: XsdComplexType)

Class for XSD assert constraint definitions.

class XsdAlternative(elem: T | None, schema: T | None, parent: XsdElement)

XSD 1.1 type alternative definitions.

class XsdNotation(elem: Element, schema: T | None, parent: XsdComponent | None = None, name: str | None = None)

Class for XSD notation declarations.

class XsdAnnotation(elem: Element, schema: T | None, parent: XsdComponent | None = None, name: str | None = None)

Class for XSD annotation definitions.

Variables

• appinfo – a list containing the xs:appinfo children.
• documentation – a list containing the xs:documentation children.
A.16 Extra features API

A.16.1 Code generators

```python
class AbstractGenerator(schema, searchpath=None, types_map=None)
```
Abstract base class for code generators based on Jinja2 template engine.

**Parameters**
- `schema` – the source or the instance of the XSD schema.
- `searchpath` – additional search path for custom templates. If provided the search path has priority over searchpaths defined in generator class.
- `types_map` – a dictionary with custom mapping for XSD types.

```python
map_type(obj)
```
Maps an XSD type to a type declaration of the target language. This method is registered as filter with a name dependant from the language name (eg. c_type).

**Parameters**
- `obj` – an XSD type or another type-related declaration as an attribute or an element.

**Returns**
an empty string for non-XSD objects.

```python
list_templates(extensions=None, filter_func=None)
matching_templates(name)
get_template(name, parent=None, global_vars=None)
select_template(names, parent=None, global_vars=None)
render(names, parent=None, global_vars=None)
render_to_files(names, parent=None, global_vars=None, output_dir='.', force=False)
```

```python
class PythonGenerator(schema, searchpath=None, types_map=None)
```
A Python code generator for XSD schemas.

A.16.2 WSDL 1.1 documents

```python
class Wsdl11Document(source, schema=None, cls=None, validation='strict', namespaces=None, maps=None, locations=None, base_url=None, allow='all', defuse='remote', timeout=300)
```
Class for WSDL 1.1 documents.

**Parameters**
- `source` – a string containing XML data or a file path or an URL or a file like object or an ElementTree or an Element.
- `schema` – additional schema for providing XSD types and elements to the WSDL document. Can be a `xmlschema.XMLSchema` instance or a file-like object or a file path or a URL of a resource or a string containing the XSD schema.
- `cls` – class to use for building the schema instance (for default `xmlschema.XMLSchema10` is used).
• **validation** – the XSD validation mode to use for validating the XML document, that can be 'strict' (default), 'lax' or 'skip'.

• **maps** – WSDL definitions shared maps.

• **namespaces** – is an optional mapping from namespace prefix to URI.

• **locations** – resource location hints, that can be a dictionary or a sequence of couples (namespace URI, resource URL).

• **base_url** – the base URL for base `xmlschema.XMLResource` initialization.

• **allow** – the security mode for base `xmlschema.XMLResource` initialization.

• **defuse** – the defuse mode for base `xmlschema.XMLResource` initialization.

• **timeout** – the timeout for base `xmlschema.XMLResource` initialization.

**messages**
- WSDL 1.1 messages.

**port_types**
- WSDL 1.1 port types.

**bindings**
- WSDL 1.1 bindings.

**services**
- WSDL 1.1 services.
<table>
<thead>
<tr>
<th>A</th>
<th>AbderaConverter (class in xmlschema), 58</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AbstractGenerator (class in xmlschema.extras.codegen), 77</td>
</tr>
<tr>
<td></td>
<td>activate() (in module xmlschema.translation), 65</td>
</tr>
<tr>
<td></td>
<td>add_schema() (XMLSchemaBase method), 48</td>
</tr>
<tr>
<td></td>
<td>all_errors (XMLSchemaBase attribute), 49</td>
</tr>
<tr>
<td></td>
<td>attrib (ElementPathMixin attribute), 66</td>
</tr>
<tr>
<td>B</td>
<td>BadgerFishConverter (class in xmlschema), 58</td>
</tr>
<tr>
<td></td>
<td>base_url (XMLResource attribute), 62</td>
</tr>
<tr>
<td></td>
<td>base_url (XMLSchemaBase attribute), 46</td>
</tr>
<tr>
<td></td>
<td>bindings (Wsdl11Document attribute), 78</td>
</tr>
<tr>
<td></td>
<td>build() (XMLSchemaBase method), 49</td>
</tr>
<tr>
<td></td>
<td>build() (XsdGlobals method), 53</td>
</tr>
<tr>
<td></td>
<td>built (XMLSchemaBase attribute), 49</td>
</tr>
<tr>
<td></td>
<td>builtin_types() (XMLSchemaBase class method), 47</td>
</tr>
<tr>
<td>C</td>
<td>check() (XsdGlobals method), 53</td>
</tr>
<tr>
<td></td>
<td>clear() (XMLSchemaBase method), 49</td>
</tr>
<tr>
<td></td>
<td>clear() (XsdGlobals method), 54</td>
</tr>
<tr>
<td></td>
<td>ColumnarConverter (class in xmlschema), 59</td>
</tr>
<tr>
<td></td>
<td>complex_types (XMLSchemaBase attribute), 47</td>
</tr>
<tr>
<td></td>
<td>content (XsdComplexType attribute), 73</td>
</tr>
<tr>
<td></td>
<td>copy() (XMLSchemaConverter method), 56</td>
</tr>
<tr>
<td></td>
<td>copy() (XsdGlobals method), 54</td>
</tr>
<tr>
<td></td>
<td>create_any_attribute_group() (XMLSchemaBase method), 47</td>
</tr>
<tr>
<td></td>
<td>create_any_content_group() (XMLSchemaBase method), 47</td>
</tr>
<tr>
<td></td>
<td>create_any_type() (XMLSchemaBase method), 47</td>
</tr>
<tr>
<td></td>
<td>create_meta_schema() (XMLSchemaBase class method), 47</td>
</tr>
<tr>
<td>D</td>
<td>DataBindingConverter (class in xmlschema), 60</td>
</tr>
<tr>
<td></td>
<td>DataElement (class in xmlschema), 59</td>
</tr>
<tr>
<td></td>
<td>DataElementConverter (class in xmlschema), 59</td>
</tr>
<tr>
<td></td>
<td>deactivate() (in module xmlschema.translation), 65</td>
</tr>
<tr>
<td></td>
<td>decode() (ValidationMixint method), 68</td>
</tr>
<tr>
<td></td>
<td>default_namespace (XMLSchemaBase attribute), 47</td>
</tr>
<tr>
<td>E</td>
<td>element_decode() (XMLSchemaConverter method), 57</td>
</tr>
<tr>
<td></td>
<td>element_encode() (XMLSchemaConverter method), 57</td>
</tr>
<tr>
<td></td>
<td>ElementData (class in xmlschema), 54</td>
</tr>
<tr>
<td></td>
<td>ElementPathMixin (class in xmlschema), 66</td>
</tr>
<tr>
<td></td>
<td>encode() (ValidationMixin method), 69</td>
</tr>
<tr>
<td></td>
<td>encode() (XMLSchemaBase method), 52</td>
</tr>
<tr>
<td></td>
<td>enumeration (XsdSimpleType attribute), 73</td>
</tr>
<tr>
<td></td>
<td>etree_element() (XMLSchemaConverter method), 56</td>
</tr>
<tr>
<td></td>
<td>export() (XMLSchemaBase method), 48</td>
</tr>
<tr>
<td>F</td>
<td>fetch_resource() (in module xmlschema), 60</td>
</tr>
<tr>
<td></td>
<td>fetch_schema() (in module xmlschema), 61</td>
</tr>
<tr>
<td></td>
<td>fetch_schema_locations() (in module xmlschema), 61</td>
</tr>
<tr>
<td></td>
<td>filepath (XMLResource attribute), 61</td>
</tr>
<tr>
<td></td>
<td>find() (ElementPathMixin method), 67</td>
</tr>
<tr>
<td></td>
<td>find() (XMLResource method), 64</td>
</tr>
<tr>
<td></td>
<td>findall() (ElementPathMixin method), 67</td>
</tr>
<tr>
<td></td>
<td>findall() (XMLResource method), 64</td>
</tr>
<tr>
<td></td>
<td>from_json() (in module xmlschema), 43</td>
</tr>
<tr>
<td>G</td>
<td>get() (ElementPathMixin method), 66</td>
</tr>
<tr>
<td></td>
<td>get_converter() (XMLSchemaBase method), 49</td>
</tr>
<tr>
<td></td>
<td>get_locations() (XMLResource method), 64</td>
</tr>
<tr>
<td></td>
<td>get_locations() (XMLSchemaBase method), 47</td>
</tr>
<tr>
<td></td>
<td>get_namespaces() (XMLResource method), 64</td>
</tr>
<tr>
<td></td>
<td>get_template() (AbstractGenerator method), 77</td>
</tr>
<tr>
<td></td>
<td>get_text() (XMLSchemaBase method), 46</td>
</tr>
<tr>
<td>H</td>
<td>has_complex_content() (XsdType method), 71</td>
</tr>
<tr>
<td></td>
<td>has_mixed_content() (XsdType method), 71</td>
</tr>
<tr>
<td></td>
<td>has_simple_content() (XsdType method), 71</td>
</tr>
<tr>
<td>I</td>
<td>id (XMLSchemaBase attribute), 46</td>
</tr>
</tbody>
</table>

79
import_schema() (XMLSchemaBase method), 48
include_schema() (XMLSchemaBase method), 47
invalid_child (XMLSchemaChildrenValidationException attribute), 39
invalid_tag (XMLSchemaChildrenValidationException attribute), 39
is_ambiguous() (ParticleMixin method), 70
is_atomic() (XsdType method), 71
is_complex() (XsdType static method), 71
is_datetime() (XsdType method), 72
is_derived() (XsdType method), 72
is_element_only() (XsdType method), 72
is_emptiable() (ParticleMixin method), 70
is_empty() (XsdType method), 72
is_global() (XsdComponent method), 71
is_lazy() (XMLResource method), 63
is_list() (XsdType method), 72
is_loaded() (XMLResource method), 63
is_local() (XMLResource method), 63
is_matching() (XsdComponent method), 71
is_missing() (ParticleMixin method), 70
is_multiple() (ParticleMixin method), 70
is_over() (ParticleMixin method), 70
is_remote() (XMLResource method), 63
is_simple() (XsdType static method), 72
is_single() (ParticleMixin method), 70
is_univocal() (ParticleMixin method), 70
is_valid() (in module xmlschema), 40
is_valid() (ValidationMixin method), 67
is_valid() (XMLSchemaBase method), 50
iter() (ElementPathMixin method), 67
iter() (XMLResource method), 63
iter_components() (XMLSchemaBase method), 49
iter_decode() (in module xmlschema), 40
iter_decode() (ValidationMixin method), 68
iter_decode() (XMLSchemaBase method), 51
iter_depth() (XMLResource method), 63
iter_encode() (ValidationMixin method), 69
iter_encode() (XMLSchemaBase method), 53
iter_errors() (in module xmlschema), 40
iter_errors() (ValidationMixin method), 69
iter_errors() (XMLSchemaBase method), 51
iter_globals() (XMLSchemaBase method), 49
iter_globals() (XsdGlobals method), 54
iter_location_hints() (XMLResource method), 64
iter_schemas() (XsdGlobals method), 54
iterchildren() (ElementPathMixin method), 67
iterfind() (ElementPathMixin method), 67
iterfind() (XMLResource method), 64

J
JsonMLConverter (class in xmlschema), 59

L
lazy_depth (XMLResource attribute), 63
list_templates() (AbstractGenerator method), 77
load() (XMLResource method), 63
local_name (XsdComponent attribute), 71
lookup() (XsdGlobals method), 54
losslessly (XMLSchemaConverter attribute), 56
lossy (XMLSchemaConverter attribute), 56

M
map_attributes() (XMLSchemaConverter method), 56
map_content() (XMLSchemaConverter method), 56
map_qname() (XMLSchemaConverter method), 57
map_type() (AbstractGenerator method), 77
matching_templates() (AbstractGenerator method), 77
max_value (XsdSimpleType attribute), 73
messages (Wsdl11Document attribute), 78
meta_schema (XMLSchemaBase attribute), 46
min_value (XsdSimpleType attribute), 73
model_group (XsdType attribute), 71

N
name (XMLResource attribute), 62
name (XMLSchemaBase attribute), 46
namespace (XMLResource attribute), 62
NamespaceMapper (class in xmlschema.namespaces), 66
NamespaceResourcesMap (class in xmlschema.namespaces), 66
NamespaceView (class in xmlschema.namespaces), 66
no_namespace_schema_location (XMLSchemaBase attribute), 46
normalize_locations() (in module xmlschema), 60
normalize_url() (in module xmlschema), 60

O
open() (XMLResource method), 63

P
ParkerConverter (class in xmlschema), 58
parse() (XMLResource method), 62
ParticleMixin (class in xmlschema.validators), 70
port_types (Wsdl11Document attribute), 78
prefixed_name (XsdComponent attribute), 71
PythonGenerator (class in xmlschema.extras.codegen), 77

Q
qualified_name (XsdComponent attribute), 71

R
register() (XsdGlobals method), 54
render() (AbstractGenerator method), 77
render_to_files() (AbstractGenerator method), 77
resolve_qname() (XMLSchemaBase method), 48
root (XMLResource attribute), 62
root (XMLSchemaBase attribute), 46
root_elements (XMLSchemaBase attribute), 47

S
schema_location (XMLSchemaBase attribute), 46
select_template() (AbstractGenerator method), 77
services (Wsdl11Document attribute), 78
simple_type (XsdComponent attribute), 71
simple_types (XMLSchemaBase attribute), 47

T
tag (ElementPathMixin attribute), 66
tag (XMLSchemaBase attribute), 46
target_namespace (XsdComponent attribute), 71
target_prefix (XMLSchemaBase attribute), 46
text (XMLResource attribute), 62
to_dict() (in module xmlschema), 41
to_etree() (in module xmlschema), 43
to_json() (in module xmlschema), 42
tostring() (XMLResource method), 62
tostring() (XsdComponent method), 71

U
unbuilt (XsdGlobals property), 54
unmap_qname() (XMLSchemaConverter method), 57
UnorderedConverter (class in xmlschema), 58
url (XMLResource attribute), 62
url (XMLSchemaBase attribute), 46

V
validate() (in module xmlschema), 39
validate() (ValidationMixin method), 68
validate() (XMLSchemaBase method), 50
validation_attempted (XMLSchemaBase attribute), 49
ValidationMixin (class in xmlschema.validators), 67
validity (XMLSchemaBase attribute), 49
version (XMLSchemaBase attribute), 46

W
Wsdl11Document (class in xmlschema.extras.wsdl), 77

X
XmlDocument (class in xmlschema), 64
XMLResource (class in xmlschema), 61
XMLResourceError, 37
XMLSchema (in module xmlschema), 44
xmlschema.XMLSchema10 (built-in class), 44
xmlschema.XMLSchema11 (built-in class), 44
XMLSchemaBase (class in xmlschema), 44

Y
XMLSchemaChildrenValidationWarning, 39
XMLSchemaConverter (class in xmlschema), 55
XMLSchemaDecodeError, 38
XMLSchemaEncodeError, 38
XMLSchemaException, 37
XMLSchemaImportWarning, 39
XMLSchemaIncludeWarning, 39
XMLSchemaModelDepthError, 38
XMLSchemaModelError, 37
XMLSchemaNamespaceError, 37
XMLSchemaNotBuiltError, 37
XMLSchemaParseError, 37
XMLSchemaStopValidation, 39
XMLSchemaTypeError, 39
XMLSchemaValidationError, 38
XMLSchemaValidatorError, 37
Xsd11AnyAttribute (class in xmlschema.validators), 74
Xsd11AnyElement (class in xmlschema.validators), 74
Xsd11AtomicRestriction (class in xmlschema.validators), 73
Xsd11Attribute (class in xmlschema.validators), 72
Xsd11ComplexType (class in xmlschema.validators), 73
Xsd11Element (class in xmlschema.validators), 72
Xsd11Group (class in xmlschema.validators), 74
Xsd11Key (class in xmlschema.validators), 75
Xsd11Keyref (class in xmlschema.validators), 75
Xsd11Union (class in xmlschema.validators), 73
Xsd11Unique (class in xmlschema.validators), 75
XsdAlternative (class in xmlschema.validators), 76
XsdAnnotation (class in xmlschema.validators), 76
XsdAnyAttribute (class in xmlschema.validators), 74
XsdAnyElement (class in xmlschema.validators), 74
XsdAssert (class in xmlschema.validators), 76
XsdAssertionFacet (class in xmlschema.validators), 74
XsdAtomicBuiltin (class in xmlschema.validators), 76
XsdAtomicRestriction (class in xmlschema.validators), 73
XsdAttribute (class in xmlschema), 72
XsdAttributeGroup (class in xmlschema.validators), 74
XsdComplexType (class in xmlschema.validators), 73
XsdComponent (class in xmlschema), 70
XsdDefaultOpenContent (class in xmlschema.validators), 74
XsdElement (class in xmlschema), 72
XsdEnumerationFacets (class in xmlschema.validators), 76
XsdExplicitTimezoneFacet (class in xmlschema.validators), 76
XsdFacet (class in xmlschema.validators), 75
XsdFieldSelector (class in xmlschema.validators), 74
XsdFractionDigitsFacet (class in xmlschema.validators), 76
XsdGlobals (class in xmlschema), 53
XsdGroup (class in xmlschema.validators), 74