declxml Documentation

Release 1.1.0

Greg Atkin

Sep 16, 2018

Contents

The declxml library provides a simple, declarative API for parsing and serializing XML documents. For the most common and straightforward processing tasks, declxml aims to replace the need for writing and maintaining dozens or hundreds of lines of imperative serialization and parsing logic required when using lower-level APIs such as ElementTree directly. The declxml library was inspired by the simplicity and declarative nature of Golang's XML processing library.

declxml works with *processors* which declaratively define the structure of an XML document. Processors are used to both serialize and parse XML data as well as to perform a basic level of validation.

CHAPTER 1

Installation

Install using either pip

```
pip install declxml
```

or pipenv

```
pipenv install declxml
```

1.1 Guide

The basic building blocks used in the declxml library are *processor* objects. Processors are used to define the structure of an XML document. There are two types of processors:

- **Primitive processors - Used for processing simple, primitive values like** booleans, integers, floats, and strings.
- **Aggregate processors - Used for processing aggregate values such as** dictionaries, arrays, and objects. Aggregate processors are themselves composed of other processors.

Primitive processors are created using the processor factory function that corresponds to the type of value to process. The factory functions offer several options for configuring a processor. The following creates a basic processor for integer values contained within an "id" element:

```
>>> import declxml as xml

>>> xml.integer('id')
<declxml._PrimitiveValue object at ...>
```

Aggregate processors are created by specifying a list of child processors that compose the aggregate. The following creates a processor for dictionary values contained within a "user" element that itself contains a "user-name" and an "id" sub-element:

```
>>> import declxml as xml

>>> xml.dictionary('user', [
...     xml.integer('id'),
...     xml.string('user-name')
... ])
<declxml._Dictionary object at ...>
```

1.1.1 Parsing and Serialization

Processors define the structure of an XML document and are used to both parse and serialize data to and from XML.

```
>>> import declxml as xml
>>> from pprint import pprint

>>> author_xml = """
... <author>
...     <name>Robert A. Heinlein</name>
...     <birth-year>1907</birth-year>
... </author>
... """

>>> author_processor = xml.dictionary('author', [
...     xml.string('name'),
...     xml.integer('birth-year')
... ])

>>> pprint(xml.parse_from_string(author_processor, author_xml))
{'birth-year': 1907, 'name': 'Robert A. Heinlein'}


>>> author = {'name': 'Isaac Asimov', 'birth-year': 1920}
>>> print(xml.serialize_to_string(author_processor, author, indent='  '))
<?xml version="1.0" encoding="utf-8"?>
<author>
  <name>Isaac Asimov</name>
  <birth-year>1920</birth-year>
</author>
```

1.1.2 Attributes

Processors may be configured to read and write values from attributes.

```
>>> import declxml as xml
>>> from pprint import pprint

>>> author_processor = xml.dictionary('author', [
...     xml.string('name'),
...     xml.integer('birth-year'),
...     xml.string('birth-year', attribute='birth-month')
... ])

>>> author_xml = """
... <author>
```

(continues on next page)

(continued from previous page)

```
...     <name>Robert A. Heinlein</name>
...     <birth-year birth-month="July">1907</birth-year>
... </author>
... """

>>> pprint(xml.parse_from_string(author_processor, author_xml))
{'birth-month': 'July', 'birth-year': 1907, 'name': 'Robert A. Heinlein'}


>>> author = {
...     'name': 'Isaac Asimov',
...     'birth-year': 1920,
...     'birth-month': 'January'
... }
>>> print(xml.serialize_to_string(author_processor, author, indent='    '))
<?xml version="1.0" encoding="utf-8"?>
<author>
    <name>Isaac Asimov</name>
    <birth-year birth-month="January">1920</birth-year>
</author>
```

1.1.3 Validation

Processors can perform basic validation such as ensuring required elements are present.

```
>>> import declxml as xml

>>> author_processor = xml.dictionary('author', [
...     xml.string('name'),
...     xml.integer('birth-year')
... ])

>>> author_xml = """
... <author>
...     <name>Robert A. Heinlein</name>
... </author>
... """

>>> xml.parse_from_string(author_processor, author_xml)
Traceback (most recent call last):
...
MissingValue: Missing required element "birth-year" at author/birth-year
```

Processors also ensure values are of the correct type.

```
>>> import declxml as xml

>>> author_processor = xml.dictionary('author', [
...     xml.string('name'),
...     xml.integer('birth-year')
... ])

>>> author_xml = """
... <author>
...     <name>Robert A. Heinlein</name>
```

(continues on next page)

(continued from previous page)

```
...     <birth-year>Hello</birth-year>
... </author>
... """

>>> xml.parse_from_string(author_processor, author_xml)
Traceback (most recent call last):
...
InvalidPrimitiveValue: Invalid numeric value "Hello" at author/birth-year
```

1.1.4 Optional and Default Values

Processors may specify optional and default values.

```
>>> import declxml as xml
>>> from pprint import pprint

>>> author_processor = xml.dictionary('author', [
...     xml.string('name'),
...     xml.integer('birth-year'),
...     xml.string('genre', required=False, default='Sci-Fi')
... ])

>>> author_xml = """
... <author>
...     <name>Robert A. Heinlein</name>
...     <birth-year>1907</birth-year>
... </author>
... """

>>> pprint(xml.parse_from_string(author_processor, author_xml))
{'birth-year': 1907, 'genre': 'Sci-Fi', 'name': 'Robert A. Heinlein'}


>>> author_xml = """
... <author>
...     <name>J. K. Rowling</name>
...     <birth-year>1965</birth-year>
...     <genre>Fantasy</genre>
... </author>
... """

>>> pprint(xml.parse_from_string(author_processor, author_xml))
{'birth-year': 1965, 'genre': 'Fantasy', 'name': 'J. K. Rowling'}
```

1.1.5 Aliases

By default, processors use the element name as the name of the value in Python. An alias can be provided to use a different name for the value in Python.

```
>>> import declxml as xml
>>> from pprint import pprint

>>> author_xml = """
```

(continues on next page)

(continued from previous page)

```
... <author>
...     <name>Robert A. Heinlein</name>
...     <birth-year>1907</birth-year>
... </author>
... """

>>> author_processor = xml.dictionary('author', [
...     xml.string('name', alias='author_name'),
...     xml.integer('birth-year', alias='year_born')
... ])

>>> pprint(xml.parse_from_string(author_processor, author_xml))
{'author_name': 'Robert A. Heinlein', 'year_born': 1907}


>>> author = {'author_name': 'Isaac Asimov', 'year_born': 1920}
>>> print(xml.serialize_to_string(author_processor, author, indent='   '))
<?xml version="1.0" encoding="utf-8"?>
<author>
   <name>Isaac Asimov</name>
   <birth-year>1920</birth-year>
</author>
```

1.1.6 Omitting Empty Values

Processors can be configured to omit missing or falsey values when serializing. Only optional values may be omitted.

```
>>> import declxml as xml

>>> author_processor = xml.dictionary('author', [
...     xml.string('name'),
...     xml.integer('birth-year'),
...     xml.string('nationality', required=False, omit_empty=True)
... ])

>>> author = {'name': 'Isaac Asimov', 'birth-year': 1920, 'nationality': ''}
>>> print(xml.serialize_to_string(author_processor, author, indent='    '))
<?xml version="1.0" encoding="utf-8"?>
<author>
    <name>Isaac Asimov</name>
    <birth-year>1920</birth-year>
</author>

>>> author = {'name': 'Robert A. Heinlein', 'birth-year': 1907, 'nationality':
↪'American'}
>>> print(xml.serialize_to_string(author_processor, author, indent='    '))
<?xml version="1.0" encoding="utf-8"?>
<author>
    <name>Robert A. Heinlein</name>
    <birth-year>1907</birth-year>
    <nationality>American</nationality>
</author>
```

1.1.7 Arrays

Processors can be defined for array values. When creating an array processor, a processor must be specified for processing the array's items. An array is treated as optional if its item processor is configured as optional.

An array can be either *embedded* or *nested*. An embedded array is embedded directly within its parent as in the following:

```
>>> import declxml as xml
>>> from pprint import pprint

>>> author_processor = xml.dictionary('author', [
...     xml.string('name'),
...     xml.array(xml.string('book'), alias='books')
... ])

>>> author_xml = """
... <author>
...     <name>Robert A. Heinlein</name>
...     <book>Starship Troopers</book>
...     <book>Stranger in a Strange Land</book>
... </author>
... """

>>> pprint(xml.parse_from_string(author_processor, author_xml))
{'books': ['Starship Troopers', 'Stranger in a Strange Land'],
 'name': 'Robert A. Heinlein'}
```

A nested array is nested within a separate array element

```
>>> import declxml as xml
>>> from pprint import pprint

>>> author_processor = xml.dictionary('author', [
...     xml.string('name'),
...     xml.array(xml.string('book'), nested='books')
... ])

>>> author_xml = """
... <author>
...     <name>Robert A. Heinlein</name>
...     <books>
...         <book>Starship Troopers</book>
...         <book>Stranger in a Strange Land</book>
...     </books>
... </author>
... """

>>> pprint(xml.parse_from_string(author_processor, author_xml))
{'books': ['Starship Troopers', 'Stranger in a Strange Land'],
 'name': 'Robert A. Heinlein'}
```

1.1.8 Composing Processors

Processors can be composed to define more complex document structures.

```
>>> import declxml as xml
>>> from pprint import pprint

>>> genre_xml = """
... <genre-authors>
...     <genre>Science Fiction</genre>
...     <author>
...         <name>Robert A. Heinlein</name>
...         <birth-year>1907</birth-year>
...         <book>
...             <title>Starship Troopers</title>
...             <year-published>1959</year-published>
...         </book>
...         <book>
...             <title>Stranger in a Strange Land</title>
...             <year-published>1961</year-published>
...         </book>
...     </author>
...     <author>
...         <name>Isaac Asimov</name>
...         <birth-year>1920</birth-year>
...         <book>
...             <title>I, Robot</title>
...             <year-published>1950</year-published>
...         </book>
...         <book>
...             <title>Foundation</title>
...             <year-published>1951</year-published>
...         </book>
...     </author>
... </genre-authors>
... """

>>> book_processor = xml.dictionary('book', [
...     xml.string('title'),
...     xml.integer('year-published')
... ])

>>> author_processor = xml.dictionary('author', [
...     xml.string('name'),
...     xml.integer('birth-year'),
...     xml.array(book_processor, alias='books')
... ])

>>> genre_processor = xml.dictionary('genre-authors', [
...     xml.string('genre'),
...     xml.array(author_processor, alias='authors')
... ])


>>> pprint(xml.parse_from_string(genre_processor, genre_xml))
{'authors': [{'birth-year': 1907,
              'books': [{'title': 'Starship Troopers',
                         'year-published': 1959},
                        {'title': 'Stranger in a Strange Land',
                         'year-published': 1961}],
              'name': 'Robert A. Heinlein'},
```

(continues on next page)

(continued from previous page)

```
             {'birth-year': 1920,
              'books': [{'title': 'I, Robot', 'year-published': 1950},
                        {'title': 'Foundation', 'year-published': 1951}],
              'name': 'Isaac Asimov'}],
 'genre': 'Science Fiction'}
```

1.1.9 User-Defined Classes

Processors can also be created for parsing and serializing XML data to and from user-defined classes. Simply provide the class to the processor factory function.

```
>>> import declxml as xml

>>> class Author:
...
...     def __init__(self):
...         self.name = None
...         self.birth_year = None
...
...     def __repr__(self):
...         return 'Author(name=\'{}\', birth_year={})'.format(
...             self.name, self.birth_year)


>>> author_processor = xml.user_object('author', Author, [
...     xml.string('name'),
...     xml.integer('birth-year', alias='birth_year')
... ])

>>> author_xml = """
... <author>
...     <name>Robert A. Heinlein</name>
...     <birth-year>1907</birth-year>
... </author>
... """

>>> xml.parse_from_string(author_processor, author_xml)
Author(name='Robert A. Heinlein', birth_year=1907)

>>> author = Author()
>>> author.name = 'Isaac Asimov'
>>> author.birth_year = 1920

>>> print(xml.serialize_to_string(author_processor, author, indent='    '))
<?xml version="1.0" encoding="utf-8"?>
<author>
  <name>Isaac Asimov</name>
  <birth-year>1920</birth-year>
</author>
```

Note that the class provided to the *user_object* factory function must have a zero-argument constructor. It is also possible to pass any other callable object that takes zero parameters and returns an object instance to which parsed values will be read into.

1.1.10 Named Tuples

Processors may also be created for named tuple values.

```
>>> from collections import namedtuple
>>> import declxml as xml


>>> Author = namedtuple('Author', ['name', 'birth_year'])


>>> author_processor = xml.named_tuple('author', Author, [
...     xml.string('name'),
...     xml.integer('birth-year', alias='birth_year')
... ])

>>> author_xml = """
... <author>
...     <name>Robert A. Heinlein</name>
...     <birth-year>1907</birth-year>
... </author>
... """

>>> xml.parse_from_string(author_processor, author_xml)
Author(name='Robert A. Heinlein', birth_year=1907)

>>> author = Author(name='Isaac Asimov', birth_year=1920)
>>> print(xml.serialize_to_string(author_processor, author, indent='    '))
<?xml version="1.0" encoding="utf-8"?>
<author>
  <name>Isaac Asimov</name>
  <birth-year>1920</birth-year>
</author>
```

1.2 Advanced Guide

This guide presents some of the more advanced features supported by declxml that can be useful when they are needed.

1.2.1 XPath Syntax

declxml supports a very small subset of XPath syntax that enables greater expressiveness when defining processors.

The Dot (.) Selector

The dot (.) selector can be used in a processor to refer to the parent processor's element. For instance, the dot operator can be used to refer to attributes on a childless element.

```
>>> import declxml as xml
>>> from pprint import pprint

>>> books_xml = """
... <books>
...     <book title="I, Robot" author="Isaac Asimov" />
```

(continues on next page)

(continued from previous page)

```
...     <book title="Foundation" author="Isaac Asimov" />
...     <book title="Nemesis" author="Isaac Asimov" />
... </books>
... """

>>> books_processor = xml.array(xml.dictionary('book', [
...     xml.string('.', attribute='title'),  # Select the attribute "title" on the␣
↪element "book"
...     xml.string('.', attribute='author'),
... ]), nested='books')


>>> pprint(xml.parse_from_string(books_processor, books_xml))
[{'author': 'Isaac Asimov', 'title': 'I, Robot'},
 {'author': 'Isaac Asimov', 'title': 'Foundation'},
 {'author': 'Isaac Asimov', 'title': 'Nemesis'}]
```

The dot operator can also be used to group an element's attribute values with the values of the element's children.

```
>>> import declxml as xml
>>> from pprint import pprint

>>> author_xml = """
... <author name="Liu Cixin">
...     <book>The Three Body Problem</book>
...     <book>The Dark Forest</book>
...     <book>Deaths End</book>
... </author>
... """

>>> author_processor = xml.dictionary('author', [
...     xml.string('.', attribute='name'),
...     xml.array(xml.string('book'), alias='books')
... ])

>>> pprint(xml.parse_from_string(author_processor, author_xml))
{'books': ['The Three Body Problem', 'The Dark Forest', 'Deaths End'],
 'name': 'Liu Cixin'}
```

The Path (/) Selector

The path selector (/) can be used to select descendant elements which can be useful for flattening out deeply nested XML data.

```
>>> import declxml as xml
>>> from pprint import pprint

>>> hugo_xml = """
... <awards>
...     <hugo>
...         <winners>
...             <winner>
...                 <year>2017</year>
...                 <book>
...                     <title>The Obelisk Gate</title>
```

(continues on next page)

(continued from previous page)

```
...                     <author>N. K. Jemisin</author>
...                 </book>
...             </winner>
...             <winner>
...                 <year>2016</year>
...                 <book>
...                     <title>The Fifth Season</title>
...                     <author> N.K. Jemisin</author>
...                 </book>
...             </winner>
...             <winner>
...                 <year>2015</year>
...                 <book>
...                     <title>The Three Body Problem</title>
...                     <author>Liu Cixin</author>
...                 </book>
...             </winner>
...         </winners>
...     </hugo>
... </awards>
... """

>>> hugo_processor = xml.array(xml.dictionary('winner', [
...     xml.integer('year'),
...     xml.string('book/title', alias='title'),
...     xml.string('book/author', alias='author'),
... ]), nested='awards/hugo/winners')

>>> pprint(xml.parse_from_string(hugo_processor, hugo_xml))
[{'author': 'N. K. Jemisin', 'title': 'The Obelisk Gate', 'year': 2017},
 {'author': 'N.K. Jemisin', 'title': 'The Fifth Season', 'year': 2016},
 {'author': 'Liu Cixin', 'title': 'The Three Body Problem', 'year': 2015}]
```

The data will be serialized back into the deeply nested XML structure if the processor is used to perform serialization.

It is *highly* recommended to provide aliases when using XPath syntax to ensure that when a value is parsed and assigned a name (e.g. a field of a dictionary, object, or namedtuple), the name of the value is a valid Python identifier without any '.' or '/' characters.

1.2.2 Hooks

Hooks are an advanced feature that allow arbitrary code to be executed during the parsing and serialization process. A *Hooks* object is associated with a processor and contains two functions: *after_parse* and *before_serialize*.

Both of these functions (which can be any callable object) are provided two parameters and should return a single value. The first parameter provided to both functions is a *ProcessorStateView* object which contains information about the current state of the processor when the function is invoked.

The *after_parse* function is invoked after a processor has parsed a value from the XML data. The second parameter provided to the *after_parse* function is the value parsed by the processor from the XML data. The *after_parse* function must return a single value which will be used by the processor as its parse result. The value returned by *after_parse* replaces the value parsed from the XML data as the processor's parse result.

The *before_serialize* function is invoked before a processor serializes a value to XML. The second parameter provided to the *before_serialize* function is the value to be serialized by the processor to XML. The *before_serialize* function must return a single value which the processor will serialize to XML. The value returned by *before_serialize* replaces

the value provided to the processor to serialize to XML.

There are three intended use cases for hooks (though since hooks can be any arbitrary callable objects, there should be flexibility for other use cases):

- Value transformations
- Validation
- Debugging

Value Transformations

Sometimes it is useful to be able to transform values read from XML during parsing into a shape more convenient for the application to use and transform values during serialization back into shapes that better fit the XML structure.

Hooks can be used to achieve this by simply returning the transformed value from the *after_parse* and *before_serialize* functions. This works because whatever value a processor was going to use for parsing or serialization is replaced by the value returned by *after_parse* or *before_serialize*.

As a basic example, if we want to make sure all strings read from an XML document are uppercase when used in our application and lowercase when written to XML, we could use hooks to perform value transformations.

```
>>> import declxml as xml

>>> hooks = xml.Hooks(
...     after_parse=lambda _, x: x.upper(),
...     before_serialize=lambda _, x: x.lower()
... )

>>> processor = xml.dictionary('data', [
...     xml.string('message', hooks=hooks),
... ])

>>> xml_string = """
... <data>
...    <message>hello</message>
... </data>
... """

>>> xml.parse_from_string(processor, xml_string)
{'message': 'HELLO'}

>>> data = {'message': 'GOODBYE'}
>>> print(xml.serialize_to_string(processor, data, indent='    '))
<?xml version="1.0" encoding="utf-8"?>
<data>
    <message>goodbye</message>
</data>
```

When using hooks to perform value transformations, it is a good idea to ensure that the transformations performed by *after_parse* and *before_serialize* are inverse operations of each other so that parsing and serialization work correctly when using transformed values. This is particularly important when values are transformed into different types.

Validation

By default, declxml only performs a very basic level of validation by ensuring that required values are present and that they are of the correct type. Hooks provide the ability to perform additional, application-specific validation.

When performing validation, we can use the *ProcessorStateView* object provided as the first parameter to the *after_parse* and *before_serialize* functions. The *ProcessorStateView* object provides a useful method, *raise_error*, for reporting errors. This method will raise an application-provided exception with a custom error message and will include information about the current state of the processor in the error message.

For example, if we want to ensure that integer values are in a specific range, we could use hooks to perform the validation.

```
>>> import declxml as xml

>>> def validate(state, value):
...     if value not in range(1, 4):
...         state.raise_error(
...             RuntimeError,
...             'Invalid value {}'.format(value)
...         )
...
...     # Important! Don't forget to return the value
...     return value

>>> hooks = xml.Hooks(
...     after_parse=validate,
...     before_serialize=validate
... )

>>> processor = xml.dictionary('data', [
...     xml.integer('value', hooks=hooks),
... ])

>>> xml_string = """
... <data>
...     <value>567</value>
... </data>
... """

>>> xml.parse_from_string(processor, xml_string)
Traceback (most recent call last):
...
RuntimeError: Invalid value 567 at data/value

>>> data = {'value': -90}
>>> xml.serialize_to_string(processor, data)
Traceback (most recent call last):
...
RuntimeError: Invalid value -90 at data/value
```

When using hooks for validation, it is important to remember to return the value from the *before_parse* and *after_serialize* functions since the processor will used the value returned by those functions as its parsing result and the value to serialize to XML, respectively.

Debugging

Hooks can also be used to debug processors. We can use the *ProcessorStateView* object provided to the *before_parse* and *after_serialize* functions to include information about which values are received in which locations in the XML document.

```
>>> import declxml as xml

>>> def trace(state, value):
...     print('Got {} at {}'.format(value, state))
...
...     # Important! Don't forget to return the value
...     return value

>>> hooks = xml.Hooks(
...     after_parse=trace,
...     before_serialize=trace
... )

>>> processor = xml.dictionary('data', [
...     xml.integer('value', hooks=hooks),
... ], hooks=hooks)

>>> xml_string = """
... <data>
...     <value>42</value>
... </data>
... """

>>> xml.parse_from_string(processor, xml_string)
Got 42 at data/value
Got {'value': 42} at data
{'value': 42}

>>> data = {'value': 17}
>>> print(xml.serialize_to_string(processor, data, indent='    '))
Got {'value': 17} at data
Got 17 at data/value
<?xml version="1.0" encoding="utf-8"?>
<data>
    <value>17</value>
</data>
```

1.3 API Reference

declxml is a library for declaratively processing XML documents.

1.3.1 Processors

The declxml library uses processors to define the structure of XML documents. Processors can be divided into two categories: *primitive processors* and *aggregate processors*.

1.3.2 Primitive Processors

Primitive processors are used to parse and serialize simple, primitive values. The following module functions are used to construct primitive processors:

declxml.**boolean**(*element_name*, *attribute=None*, *required=True*, *alias=None*, *default=False*, *omit_empty=False*, *hooks=None*)

Create a processor for boolean values.

Parameters

- **element_name** – Name of the XML element containing the value. Can also be specified using supported XPath syntax.
- **attribute** – If specified, then the value is searched for under the attribute within the element specified by element_name. If not specified, then the value is searched for as the contents of the element specified by element_name.
- **required** – Indicates whether the value is required when parsing and serializing.
- **alias** – If specified, then this is used as the name of the value when read from XML. If not specified, then the element_name is used as the name of the value.
- **default** – Default value to use if the element is not present. This option is only valid if required is specified as False.
- **omit_empty** – If True, then Falsey values will be omitted when serializing to XML. Note that Falsey values are never omitted when they are elements of an array. Falsey values can be omitted only when they are standalone elements.
- **hooks** – A Hooks object.

Returns A declxml processor object.

declxml.**floating_point**(*element_name*, *attribute=None*, *required=True*, *alias=None*, *default=0.0*, *omit_empty=False*, *hooks=None*)

Create a processor for floating point values.

See also *declxml.boolean()*

declxml.**integer**(*element_name*, *attribute=None*, *required=True*, *alias=None*, *default=0*, *omit_empty=False*, *hooks=None*)

Create a processor for integer values.

See also *declxml.boolean()*

declxml.**string**(*element_name*, *attribute=None*, *required=True*, *alias=None*, *default=''*, *omit_empty=False*, *strip_whitespace=True*, *hooks=None*)

Create a processor for string values.

Parameters **strip_whitespace** – Indicates whether leading and trailing whitespace should be stripped from parsed string values.

See also *declxml.boolean()*

1.3.3 Aggregate Processors

Aggregate processors are composed of other processors. These processors are used for values such as dictionaries, arrays, and user objects.

declxml.**array**(*item_processor*, *alias=None*, *nested=None*, *omit_empty=False*, *hooks=None*)

Create an array processor that can be used to parse and serialize array data.

XML arrays may be nested within an array element, or they may be embedded within their parent. A nested array would look like:

```
<root-element>
    <some-element>ABC</some-element>
    <nested-array>
        <array-item>0</array-item>
        <array-item>1</array-item>
    </nested-array>
</root-element>
```

The corresponding embedded array would look like:

```
<root-element>
    <some-element>ABC</some-element>
    <array-item>0</array-item>
    <array-item>1</array-item>
</root-element>
```

An array is considered required when its item processor is configured as being required.

Parameters

- **item_processor** – A declxml processor object for the items of the array.
- **alias** – If specified, the name given to the array when read from XML. If not specified, then the name of the item processor is used instead.
- **nested** – If the array is a nested array, then this should be the name of the element under which all array items are located. If not specified, then the array is treated as an embedded array. Can also be specified using supported XPath syntax.
- **omit_empty** – If True, then nested arrays will be omitted when serializing if they are empty. Only valid when nested is specified. Note that an empty array may only be omitted if it is not itself contained within an array. That is, for an array of arrays, any empty arrays in the outer array will always be serialized to prevent information about the original array from being lost when serializing.
- **hooks** – A Hooks object.

Returns A declxml processor object.

declxml.**dictionary**(*element_name*, *children*, *required=True*, *alias=None*, *hooks=None*)

Create a processor for dictionary values.

Parameters

- **element_name** – Name of the XML element containing the dictionary value. Can also be specified using supported XPath syntax.
- **children** – List of declxml processor objects for processing the children contained within the dictionary.
- **required** – Indicates whether the value is required when parsing and serializing.
- **alias** – If specified, then this is used as the name of the value when read from XML. If not specified, then the element_name is used as the name of the value.
- **hooks** – A Hooks object.

Returns A declxml processor object.

declxml.**named_tuple**(*element_name*, *tuple_type*, *child_processors*, *required=True*, *alias=None*, *hooks=None*)

Create a processor for namedtuple values.

Parameters **tuple_type** – The namedtuple type.

See also *declxml.dictionary()*

declxml.**user_object**(*element_name*, *cls*, *child_processors*, *required=True*, *alias=None*, *hooks=None*)
Create a processor for user objects.

Parameters **cls** – Class object with a no-argument constructor or other callable no-argument object.

See also *declxml.dictionary()*

1.3.4 Hooks

class declxml.**Hooks**(*after_parse=None*, *before_serialize=None*)
Contains functions to be invoked during parsing and serialization.

A Hooks object contains two functions: after_parse and before_serialize. Both of these functions receive two arguments and should return one value.

As their first argument, both functions will receive a ProcessorStateView object representing the current state of the processor when the function is invoked.

The after_parse function will receive as its second argument the value parsed by the processor from the XML data during parsing. This function must return a value that will be used by the processor as its parse result.

Similarly, the before_serialize function will receive as its second argument the value to be serialized to XML by the processor during serialization. This function must return a value that the processor will serialize to XML.

Both the after_parse and before_serialize functions are optional.

class declxml.**ProcessorStateView**(*processor_state*)
Provides an immutable view of the processor state.

locations
Get iterator of the ProcessorLocations visited by the processor.

Represents the current location of the processor in the XML document.

A ProcessorLocation represents a single location of a processor in an XML document. It is a namedtuple with two fields: element_path and array_index. The element_path field contains the path to the element in the XML document relative to the previous location in the list of locations. The array_index field contains the index of the element if it is in an array, otherwise it is None if the element is not in an array.

Returns Iterator of ProcessorLocation objects.

raise_error(*exception_type*, *message=''*)
Raise an error with the processor state included in the error message.

Parameters

- **exception_type** – Type of exception to raise
- **message** – Error message

1.3.5 Parsing

declxml.**parse_from_file**(*root_processor*, *xml_file_path*, *encoding='utf-8'*)
Parse the XML file using the processor starting from the root of the document.

Parameters

- **root_processor** – Root processor of the XML document.
- **xml_file_path** – Path to XML file to parse.
- **encoding** – Encoding of the file.

Returns Parsed value.

declxml.**parse_from_string**(*root_processor*, *xml_string*)
Parse the XML string using the processor starting from the root of the document.

Parameters **xml_string** – XML string to parse.

See also *declxml.parse_from_file()*

1.3.6 Serialization

declxml.**serialize_to_file**(*root_processor*, *value*, *xml_file_path*, *encoding='utf-8'*, *indent=None*)
Serialize the value to an XML file using the root processor.

Parameters

- **root_processor** – Root processor of the XML document.
- **value** – Value to serialize.
- **xml_file_path** – Path to the XML file to which the serialized value will be written.
- **encoding** – Encoding of the file.
- **indent** – If specified, then the XML will be formatted with the specified indentation.

declxml.**serialize_to_string**(*root_processor*, *value*, *indent=None*)
Serialize the value to an XML string using the root processor.

Returns The serialized XML string.

See also *declxml.serialize_to_file()*

1.3.7 Exceptions

exception declxml.**XmlError**
Base error class representing errors processing XML data.

exception declxml.**InvalidPrimitiveValue**
Represents errors due to invalid primitive values.

exception declxml.**InvalidRootProcessor**
Represents errors due to invalid root processors.

exception declxml.**MissingValue**
Represents errors due to missing required values.

CHAPTER 2

Usage

With declxml, you declaratively define the structure of your XML document using processors which can be used for both parsing and serialization.

```
>>> import declxml as xml
>>> from pprint import pprint

>>> author_processor = xml.dictionary('author', [
...     xml.string('name'),
...     xml.integer('birth-year'),
...     xml.array(xml.dictionary('book', [
...         xml.string('title'),
...         xml.integer('published')
...     ]), alias='books')
... ])

>>> author_xml = """
... <author>
...     <name>Robert A. Heinlein</name>
...     <birth-year>1907</birth-year>
...     <book>
...         <title>Starship Troopers</title>
...         <published>1959</published>
...     </book>
...     <book>
...         <title>Stranger in a Strange Land</title>
...         <published>1961</published>
...     </book>
... </author>
... """

>>> pprint(xml.parse_from_string(author_processor, author_xml))
{'birth-year': 1907,
 'books': [{'published': 1959, 'title': 'Starship Troopers'},
           {'published': 1961, 'title': 'Stranger in a Strange Land'}],
```

(continues on next page)

(continued from previous page)

```
 'name': 'Robert A. Heinlein'}

# The same processor is used for serializing as well.
>>> author = {
...     'birth-year': 1920,
...     'name': 'Issac Asimov',
...     'books': [
...         {
...             'title': 'I, Robot',
...             'published': 1950
...         },
...         {
...             'title': 'Foundation',
...             'published': 1951
...         }
...     ]
... }

>>> print(xml.serialize_to_string(author_processor, author, indent='    '))
<?xml version="1.0" encoding="utf-8"?>
<author>
   <name>Issac Asimov</name>
   <birth-year>1920</birth-year>
   <book>
      <title>I, Robot</title>
      <published>1950</published>
   </book>
   <book>
      <title>Foundation</title>
      <published>1951</published>
   </book>
</author>
```

Python Module Index

d

Index