elementpath Manual

Release 4.4.0

Davide Brunato

Mar 13, 2024

CONTENTS

CHAPTER

ONE

INTRODUCTION

The proposal of this package is to provide XPath 1.0, 2.0, 3.0 and 3.1 selectors for ElementTree XML data structures, both for the standard ElementTree library and for the lxml.etree library.

For lxml.etree this package can be useful for providing XPath 2.0/3.0/3.1 selectors, because lxml.etree already has it's own implementation of XPath 1.0.

1.1 Installation and usage

You can install the package with *pip* in a Python 3.8+ environment:

```
pip install elementpath
```

For using it import the package and apply the selectors on ElementTree nodes:

```
>>> import elementpath
>>> from xml.etree import ElementTree
>>> root = ElementTree.XML('<A><B1/><B2><C1/><C2/><C3/></B2></A>')
>>> elementpath.select(root, '/A/B2/*')
[<Element 'C1' at ...>, <Element 'C2' at ...>, <Element 'C3' at ...>]
```

The *select* API provides the standard XPath result format that is a list or an elementary datatype's value. If you want only to iterate over results you can use the generator function *iter_select* that accepts the same arguments of *select*.

The selectors API works also using XML data trees based on the lxml.etree library:

```
>>> import elementpath
>>> import lxml.etree as etree
>>> root = etree.XML('<A><B1/><B2><C1/><C2/><C3/></B2></A>')
>>> elementpath.select(root, '/A/B2/*')
[<Element C1 at ...>, <Element C2 at ...>, <Element C3 at ...>]
```

When you need to apply the same XPath expression to several XML data you can also use the *Selector* class, creating an instance and then using it to apply the path on distinct XML data:

```
>>> import elementpath
>>> import lxml.etree as etree
>>> selector = elementpath.Selector('/A/*/*')
>>> root = etree.XML('<A><B1/><B2><C1/><C2/><C3/></B2></A>')
>>> selector.select(root)
[<Element C1 at ...>, <Element C2 at ...>, <Element C3 at ...>]
```

(continues on next page)

(continued from previous page)

```
>>> root = etree.XML('<A><B1><C0/></B1><B2><C1/><C2/><C3/></B2></A>')
>>> selector.select(root)
[<Element C0 at ...>, <Element C1 at ...>, <Element C2 at ...>, <Element C3 at ...>]
```

Public API classes and functions are described into the elementpath manual on the "Read the Docs" site.

For default the XPath 2.0 is used. If you need XPath 1.0 parser provide the *parser* argument:

```
>>> from elementpath import select, XPath1Parser
>>> from xml.etree import ElementTree
>>> root = ElementTree.XML('<A><B1/><B2><C1/><C2/><C3/></B2></A>')
>>> select(root, '/A/B2/*', parser=XPath1Parser)
[<Element 'C1' at ...>, <Element 'C2' at ...>, <Element 'C3' at ...>]
```

For XPath 3.0/3.1 import the parser from *elementpath.xpath3* subpackage, that is not loaded for default:

```
>>> from elementpath.xpath3 import XPath3Parser
>>> select(root, 'math:atan(1.0e0)', parser=XPath3Parser)
0.7853981633974483
```

Note: *XPath3Parser* is an alias of *XPath31Parser*.

If you need only XPath 3.0 you can also use a more specific subpackage, avoiding the loading of XPath 3.1 implementation:

```
>>> from elementpath.xpath30 import XPath30Parser
>>> select(root, 'math:atan(1.0e0)', parser=XPath30Parser)
0.7853981633974483
```

1.2 Contributing

You can contribute to this package reporting bugs, using the issue tracker or by a pull request. In case you open an issue please try to provide a test or test data for reproducing the wrong behaviour. The provided testing code shall be added to the tests of the package.

The XPath parsers are based on an implementation of the Pratt's Top Down Operator Precedence parser. The implemented parser includes some lookup-ahead features, helpers for registering tokens and for extending language implementations. Also the token class has been generalized using a *MutableSequence* as base class. See *tdop.py* for the basic internal classes and *xpath1_parser.py* for extensions and for a basic usage of the parser.

If you like you can use the basic parser and tokens provided by the *tdop.py* module to implement other types of parsers (I think it could be also a funny exercise!).

1.3 License

This software is distributed under the terms of the MIT License. See the file 'LICENSE' in the root directory of the present distribution, or http://opensource.org/licenses/MIT.

CHAPTER

TWO

ADVANCED TOPICS

2.1 Parsing expressions

An XPath expression (the *path*) is analyzed using a parser instance, having as result a tree of tokens:

```
>>> from elementpath import XPath2Parser, XPathToken
>>>
>>> parser = XPath2Parser()
>>> token = parser.parse('/root/(: comment :) child[@attr]')
>>> isinstance(token, XPathToken)
True
>>> token
<_SolidusOperator object at 0x...
>>> str(token)
"'/' operator"
>>> token.tree
'(/ (/ (root)) ([ (child) (@ (attr))))'
>>> token.source
'/root/child[@attr]'
```

Providing a wrong expression an error is raised:

```
>>> token = parser.parse('/root/#child2/@attr')
Traceback (most recent call last):
  .........
elementpath.exceptions.ElementPathSyntaxError: '#' unknown at line 1, column 7:␣
↪[err:XPST0003] unknown symbol '#'
```

The result tree is also checked with a static evaluation, that uses only the information provided by the parser instance (e.g. statically known namespaces). In *elementpath* a parser instance represents the XPath static context. Static evaluation is not based on any XML input data but permits to found many errors related with operators and function arguments:

```
>>> token = parser.parse('1 + "1"')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../elementpath/xpath2/xpath2_parser.py", ..., in parse
    root_token.evaluate()  # Static context evaluation
  .........
elementpath.exceptions.ElementPathTypeError: '+' operator at line 1, column 3:␣
↪[err:XPTY0004] ...
```

2.2 Dynamic evaluation

Evaluation on XML data is performed using the XPath dynamic context, represented by *XPathContext* objects.

```
>>> from xml.etree import ElementTree
>>> from elementpath import XPathContext
>>>
>>> root = ElementTree.XML('<root><child/><child attr="10"/></root>')
>>> context = XPathContext(root)
>>> token.evaluate(context)
[ElementNode(elem=<Element 'child' at ...)]
```

In this case an error is raised if you don't provide a context:

```
>>> token.evaluate()
Traceback (most recent call last):
  .........
elementpath.exceptions.MissingContextError: '/' operator at line 1, column 6:␣
↪[err:XPDY0002] Dynamic context required for evaluate
```

Expressions that not depend on XML data can be evaluated also without a context:

```
>>> token = parser.parse('concat("foo", " ", "bar")')
>>> token.evaluate()
'foo bar'
```

For more details on parsing and evaluation of XPath expressions see the XPath processing model.

2.3 Node trees

In the XPath Data Model there are seven kinds of nodes: document, element, attribute, text, namespace, processing instruction, and comment.

For a fully compliant XPath processing all the seven node kinds have to be represented and processed, considering theirs properties (called accessors) and their position in the belonging document.

But the ElementTree components don't implement all the necessary characteristics, forcing to use workaround tricks, that make the code more complex. So since version v3.0 the data processing is based on XPath node types, that act as wrappers of elements of the input ElementTree structures. Node trees building requires more time and memory for handling dynamic context and for iterating the trees, but is overall fast because simplify the rest of the code.

Node trees are automatically created at dynamic context initialization:

```
>>> from xml.etree import ElementTree
>>> from elementpath import XPathContext, get_node_tree
>>>
>>> root = ElementTree.XML('<root><child/><child attr="10"/></root>')
>>> context = XPathContext(root)
>>> context.root
ElementNode(elem=<Element 'root' at ...>)
>>> context.root.children
[ElementNode(elem=<Element 'child' at ...>), ElementNode(elem=<Element 'child' at ...>)]
```

If the same XML data is applied several times for dynamic evaluation it maybe convenient to build the node tree before, in the way to create it only once:

```
>>> root_node = get_node_tree(root)
>>> context = XPathContext(root_node)
>>> context.root is root_node
True
```

2.4 The context root and the context item

Selector functions and class simplify the XML data processing. Often you only have to provide the root element and the path expression.

But other keyword arguments, related to parser or context initialization, can be provided. Of these arguments the item has a particular relevance, because it defines the initial context item for performing dynamic evaluation.

If you have this XML data:

```
>>> from xml.etree import ElementTree
>>> from elementpath import select
>>>
>>> root = ElementTree.XML('<root><child1/><child2/><child3/></root>')
```

using a select on it with the self-shortcut expression, gives back the root element:

```
>>> select(root, '.')
[<Element 'root' at ...>]
```

But if you want to use a specific child as the initial context item you have to provide the extra argument *item*:

```
>>> select(root, '.', item=root[1])
[<Element 'child2' at ...>]
```

The same result can be obtained providing the same child element as argument *root*:

```
>>> select(root[1], '.')
[<Element 'child2' at ...>]
```

But this is not always true, because in the latter case the evaluation is done using a subtree of nodes:

```
>>> select(root, 'root()', item=root[1])
[<Element 'root' at ...>]
>>> select(root[1], 'root()')
[<Element 'child2' at ...>]
```

Both choices can be useful, depends if you need to keep the whole tree or to restrict the scope to a subtree.

The context *item* can be set with an XPath node, an atomic value or an XPath function.

Note: Since release v4.2.0 the *root* is optional. If the argument *root* is absent the argument *item* is mandatory and the dynamic context remain without a root.

2.5 The root document and the root element

Warning: The initialization of context root and item is changed in release v4.2.0.

Since then the provided XML is still considered a document for default, but the item is set with the root instead of *None* and the new attribute *document* is set with a dummy document for handling the document position. The dummy document is not referred by the root element and is discarded from results.

Canonically the dynamic evaluation is performed on an XML document, created from an ElementTree instance:

```
>>> from xml.etree import ElementTree
>>> from io import StringIO
>>> from elementpath import select, XPathContext
>>>
>>> doc = ElementTree.parse(StringIO('<root><child1/><child2/><child3/></root>'))
>>> doc
<xml.etree.ElementTree.ElementTree object at ...>
```

In this case a document node is created at context initialization and the context item is set to context root:

```
>>> context = XPathContext(doc)
>>> context.root
DocumentNode(document=<xml.etree.ElementTree.ElementTree object at ...>)
>>> context.item is context.root
True
>>> context.document is context.root
True
```

Providing a root element the document is not created and the context item is set to root element node. In this case the context document is a dummy document:

```
>>> root = ElementTree.XML('<root><child1/><child2/><child3/></root>')
>>> context = XPathContext(root)
>>> context.root
ElementNode(elem=<Element 'root' at ...>)
>>> context.item is context.root
True
>>> context.document
DocumentNode(document=<xml.etree.ElementTree.ElementTree object at ...>)
>>> context.root.parent is None
True
```

Exception to this is if XML data root has siblings and if you process the data with lxml:

```
>>> import lxml.etree as etree
>>> root = etree.XML('<!-- comment --><root><child/></root>')
>>> context = XPathContext(root)
>>> context.root
DocumentNode(document=<lxml.etree._ElementTree object at ...>)
>>> context.item is context.root
True
>>> context.document is context.root
```

(continues on next page)

(continued from previous page)

```
True
```

Provide the option *fragment* with value *True* for processing an XML root element as a fragment. In this case a dummy document is not created and the context document is set to *None*:

```
>>> root = ElementTree.XML('<root><child1/><child2/><child3/></root>')
>>> context = XPathContext(root, fragment=True)
>>> context.root
ElementNode(elem=<Element 'root' at ...>)
>>> context.item is context.root
True
>>> context.document is None
True
```

CHAPTER

THREE

PUBLIC XPATH API

The package includes some classes and functions that implement XPath selectors, parsers, tokens, contexts and schema proxy.

3.1 XPath selectors

select(*root*, *path*, *namespaces=None*, *parser=None*, ***kwargs*)

XPath selector function that apply a *path* expression on *root* Element.

Parameters

- **root** – the root of the XML document, usually an ElementTree instance or an Element. A schema or a schema element can also be provided, or an already built node tree. You can also provide *None*, in which case no XML root node is set in the dynamic context, and you have to provide the keyword argument *item*.
- **path** – the XPath expression.
- **namespaces** – a dictionary with mapping from namespace prefixes into URIs.
- **parser** – the parser class to use, that is *XPath2Parser* for default.
- **kwargs** – other optional parameters for the parser instance or the dynamic context.

Returns

a list with XPath nodes or a basic type for expressions based on a function or literal.

iter_select(*root*, *path*, *namespaces=None*, *parser=None*, ***kwargs*)

A function that creates an XPath selector generator for apply a *path* expression on *root* Element.

Parameters

- **root** – the root of the XML document, usually an ElementTree instance or an Element. A schema or a schema element can also be provided, or an already built node tree. You can also provide *None*, in which case no XML root node is set in the dynamic context, and you have to provide the keyword argument *item*.
- **path** – the XPath expression.
- **namespaces** – a dictionary with mapping from namespace prefixes into URIs.
- **parser** – the parser class to use, that is *XPath2Parser* for default.
- **kwargs** – other optional parameters for the parser instance or the dynamic context.

Returns

a generator of the XPath expression results.

class Selector(*path*, *namespaces=None*, *parser=None*, ***kwargs*)

XPath selector class. Create an instance of this class if you want to apply an XPath selector to several target data.

Parameters

- **path** – the XPath expression.
- **namespaces** – a dictionary with mapping from namespace prefixes into URIs.
- **parser** – the parser class to use, that is *XPath2Parser* for default.
- **kwargs** – other optional parameters for the XPath parser instance.

Variables

- **path** (*str*) – the XPath expression.
- **parser** (*XPath1Parser or XPath2Parser*) – the parser instance.
- **root_token** (*XPathToken*) – the root of tokens tree compiled from path.

namespaces

A dictionary with mapping from namespace prefixes into URIs.

select(*root*, ***kwargs*)

Applies the instance's XPath expression on *root* Element.

Parameters

- **root** – the root of the XML document, usually an ElementTree instance or an Element.
- **kwargs** – other optional parameters for the XPath dynamic context.

Returns

a list with XPath nodes or a basic type for expressions based on a function or literal.

iter_select(*root*, ***kwargs*)

Creates an XPath selector generator for apply the instance's XPath expression on *root* Element.

Parameters

- **root** – the root of the XML document, usually an ElementTree instance or an Element.
- **kwargs** – other optional parameters for the XPath dynamic context.

Returns

a generator of the XPath expression results.

3.2 XPath parsers

class XPath1Parser(*namespaces=None*, *strict=True*)

XPath 1.0 expression parser class. Provide a *namespaces* dictionary argument for mapping namespace prefixes to URI inside expressions. If *strict* is set to *False* the parser enables also the parsing of QNames, like the ElementPath library.

Parameters

- **namespaces** – a dictionary with mapping from namespace prefixes into URIs.
- **strict** – a strict mode is *False* the parser enables parsing of QNames in extended format, like the Python's ElementPath library. Default is *True*.

DEFAULT_NAMESPACES = {'xml': 'http://www.w3.org/XML/1998/namespace'}

Namespaces known statically by default.

version = '1.0'

The XPath version string.

Helper methods for defining token classes:

classmethod axis(*symbol*, *reverse_axis=False*, *bp=80*)

Register a token for a symbol that represents an XPath *axis*.

classmethod function(*symbol*, *prefix=None*, *label='function'*, *nargs=None*, *sequence_types=()*, *bp=90*)

Registers a token class for a symbol that represents an XPath function.

class XPath2Parser(*namespaces=None*, *strict=True*, *compatibility_mode=False*, *default_collation=None*, *default_namespace=None*, *function_namespace=None*, *xsd_version=None*, *schema=None*, *base_uri=None*, *variable_types=None*, *document_types=None*, *collection_types=None*, *default_collection_type='node()*'*)

XPath 2.0 expression parser class. This is the default parser used by XPath selectors. A parser instance represents also the XPath static context. With *variable_types* you can pass a dictionary with the types of the in-scope variables. Provide a *namespaces* dictionary argument for mapping namespace prefixes to URI inside expressions. If *strict* is set to *False* the parser enables also the parsing of QNames, like the ElementPath library. There are some additional XPath 2.0 related arguments.

Parameters

- **namespaces** – a dictionary with mapping from namespace prefixes into URIs.
- **variable_types** – a dictionary with the static context's in-scope variable types. It defines the associations between variables and static types.
- **strict** – if strict mode is *False* the parser enables parsing of QNames, like the ElementPath library. Default is *True*.
- **compatibility_mode** – if set to *True* the parser instance works with XPath 1.0 compatibility rules.
- **default_namespace** – the default namespace to apply to unprefixed names. For default no namespace is applied (empty namespace '').
- **function_namespace** – the default namespace to apply to unprefixed function names. For default the namespace "http://www.w3.org/2005/xpath-functions" is used.
- **schema** – the schema proxy class or instance to use for types, attributes and elements lookups. If an *AbstractSchemaProxy* subclass is provided then a schema proxy instance is built without the optional argument, that involves a mapping of only XSD builtin types. If it's not provided the XPath 2.0 schema's related expressions cannot be used.
- **base_uri** – an absolute URI maybe provided, used when necessary in the resolution of relative URIs.
- **default_collation** – the default string collation to use. If not set the environment's default locale setting is used.
- **document_types** – statically known documents, that is a dictionary from absolute URIs onto types. Used for type check when calling the *fn:doc* function with a sequence of URIs. The default type of a document is 'document-node()'.
- **collection_types** – statically known collections, that is a dictionary from absolute URIs onto types. Used for type check when calling the *fn:collection* function with a sequence of URIs. The default type of a collection is 'node()*'.

- **default_collection_type** – this is the type of the sequence of nodes that would result from calling the *fn:collection* function with no arguments. Default is 'node()*'.

class **XPath30Parser**(**args*, *decimal_formats=None*, *defuse_xml=True*, ***kwargs*)

XPath 3.0 expression parser class. Accepts all XPath 2.0 options as keyword arguments, but the *strict* option is ignored because XPath 3.0+ has braced URI literals and the expanded name syntax is not compatible.

Parameters

- **args** – the same positional arguments of class *elementpath.XPath2Parser*.
- **decimal_formats** – a mapping with statically known decimal formats.
- **defuse_xml** – if *True* defuse XML data before parsing, that is the default.
- **kwargs** – the same keyword arguments of class *elementpath.XPath2Parser*.

class **XPath31Parser**(**args*, *decimal_formats=None*, *defuse_xml=True*, ***kwargs*)

XPath 3.1 expression parser class.

3.3 XPath tokens

class **XPathToken**(*parser*, *value=None*)

Base class for XPath tokens.

evaluate(*context=None*)

Evaluate default method for XPath tokens.

Parameters
context – The XPath dynamic context.

select(*context=None*)

Select operator that generates XPath results.

Parameters
context – The XPath dynamic context.

Context manipulation helpers:

get_argument(*context*, *index=0*, *required=False*, *default_to_context=False*, *default=None*, *cls=None*, *promote=None*)

Get the argument value of a function of constructor token. A zero length sequence is converted to a *None* value. If the function has no argument returns the context's item if the dynamic context is not *None*.

Parameters

- **context** – the dynamic context.
- **index** – an index for select the argument to be got, the first for default.
- **required** – if set to *True* missing or empty sequence arguments are not allowed.
- **default_to_context** – if set to *True* then the item of the dynamic context is returned when the argument is missing.
- **default** – the default value returned in case the argument is an empty sequence. If not provided returns *None*.
- **cls** – if a type is provided performs a type checking on item.
- **promote** – a class or a tuple of classes that are promoted to *cls* class.

`atomization`(*context=None*)

Helper method for value atomization of a sequence.

Ref: https://www.w3.org/TR/xpath31/#id-atomization

Parameters
context – the XPath dynamic context.

`get_atomized_operand`(*context=None*)

Get the atomized value for an XPath operator.

Parameters
context – the XPath dynamic context.

Returns
the atomized value of a single length sequence or *None* if the sequence is empty.

`iter_comparison_data`(*context*)

Generates comparison data couples for the general comparison of sequences. Different sequences maybe generated with an XPath 2.0 parser, depending on compatibility mode setting.

Ref: https://www.w3.org/TR/xpath20/#id-general-comparisons

Parameters
context – the XPath dynamic context.

`get_operands`(*context*, *cls=None*)

Returns the operands for a binary operator. Float arguments are converted to decimal if the other argument is a *Decimal* instance.

Parameters

- **context** – the XPath dynamic context.
- **cls** – if a type is provided performs a type checking on item.

Returns
a couple of values representing the operands. If any operand is not available returns a *(None, None)* couple.

`get_results`(*context*)

Returns results formatted according to XPath specifications.

Parameters
context – the XPath dynamic context.

Returns
a list or a simple datatype when the result is a single simple type generated by a literal or function token.

`select_results`(*context*)

Generates formatted XPath results.

Parameters
context – the XPath dynamic context.

`adjust_datetime`(*context*, *cls*)

XSD datetime adjust function helper.

Parameters

- **context** – the XPath dynamic context.

- **cls** – the XSD datetime subclass to use.

Returns

an empty list if there is only one argument that is the empty sequence or the adjusted XSD datetime instance.

Schema context methods .. automethod:: select_xsd_nodes .. automethod:: add_xsd_type .. automethod:: get_xsd_type .. automethod:: get_typed_node

Data accessor helpers .. automethod:: data_value .. automethod:: boolean_value .. automethod:: string_value .. automethod:: number_value .. automethod:: schema_node_value

Error management helper:

error(*code*, *message_or_error=None*)

3.4 XPath contexts

class XPathContext(*root=None*, *namespaces=None*, *uri=None*, *fragment=False*, *item=None*, *position=1*, *size=1*, *axis=None*, *variables=None*, *current_dt=None*, *timezone=None*, *documents=None*, *collections=None*, *default_collection=None*, *text_resources=None*, *resource_collections=None*, *default_resource_collection=None*, *allow_environment=False*, *default_language=None*, *default_calendar=None*, *default_place=None*)

The XPath dynamic context. The static context is provided by the parser.

Usually the dynamic context instances are created providing only the root element. Variable values argument is needed if the XPath expression refers to in-scope variables. The other optional arguments are needed only if a specific position on the context is required, but have to be used with the knowledge of what is their meaning.

Parameters

- **root** – the root of the XML document, usually an ElementTree instance or an Element. A schema or a schema element can also be provided, or an already built node tree. For default is *None*, in which case no XML root is set, and you have to provide an *item* argument.
- **namespaces** – a dictionary with mapping from namespace prefixes into URIs, used when namespace information is not available within document and element nodes. This can be useful when the dynamic context has additional namespaces and root is an Element or an ElementTree instance of the standard library.
- **uri** – an optional URI associated with the root element or the document.
- **fragment** – if *True* a root element is considered a fragment, otherwise a root element is considered the root of an XML document, and a dummy document is created for selection. In this case the dummy document value is not included in the results.
- **item** – the context item. A *None* value means that the context is positioned on the document node.
- **position** – the current position of the node within the input sequence.
- **size** – the number of items in the input sequence.
- **axis** – the active axis. Used to choose when apply the default axis ('child' axis).
- **variables** – dictionary of context variables that maps a QName to a value.
- **current_dt** – current dateTime of the implementation, including explicit timezone.
- **timezone** – implicit timezone to be used when a date, time, or dateTime value does not have a timezone.

- **documents** – available documents. This is a mapping of absolute URI strings into document nodes. Used by the function fn:doc.
- **collections** – available collections. This is a mapping of absolute URI strings onto sequences of nodes. Used by the XPath 2.0+ function fn:collection.
- **default_collection** – this is the sequence of nodes used when fn:collection is called with no arguments.
- **text_resources** – available text resources. This is a mapping of absolute URI strings onto text resources. Used by XPath 3.0+ function fn:unparsed-text/fn:unparsed-text-lines.
- **resource_collections** – available URI collections. This is a mapping of absolute URI strings to sequence of URIs. Used by the XPath 3.0+ function fn:uri-collection.
- **default_resource_collection** – this is the sequence of URIs used when fn:uri-collection is called with no arguments.
- **allow_environment** – defines if the access to system environment is allowed, for default is *False*. Used by the XPath 3.0+ functions fn:environment-variable and fn:available-environment-variables.

class XPathSchemaContext(*root=None*, *namespaces=None*, *uri=None*, *fragment=False*, *item=None*, *position=1*, *size=1*, *axis=None*, *variables=None*, *current_dt=None*, *timezone=None*, *documents=None*, *collections=None*, *default_collection=None*, *text_resources=None*, *resource_collections=None*, *default_resource_collection=None*, *allow_environment=False*, *default_language=None*, *default_calendar=None*, *default_place=None*)

The XPath dynamic context base class for schema bounded parsers. Use this class as dynamic context for schema instances in order to perform a schema-based type checking during the static analysis phase. Don't use this as dynamic context on XML instances.

3.5 XML Schema proxy

The XPath 2.0 parser can be interfaced with an XML Schema processor through a schema proxy. An `XMLSchemaProxy` class is defined for interfacing schemas created with the *xmlschema* package. This class is based on an abstract class *elementpath.AbstractSchemaProxy*, that can be used for implementing concrete interfaces to other types of XML Schema processors.

class AbstractSchemaProxy(*schema*, *base_element=None*)

Abstract base class for defining schema proxies. An implementation can override initialization type annotations

Parameters

- **schema** – a schema instance compatible with the XsdSchemaProtocol.
- **base_element** – the schema element used as base item for static analysis.

bind_parser(*parser*)

Binds a parser instance with schema proxy adding the schema's atomic types constructors. This method can be redefined in a concrete proxy to optimize schema bindings.

Parameters

parser – a parser instance.

get_context()

Get a context instance for static analysis phase.

Returns
an *XPathSchemaContext* instance.

`find`(*path*, *namespaces=None*)

Find a schema element or attribute using an XPath expression.

Parameters

- **path** – an XPath expression that selects an element or an attribute node.
- **namespaces** – an optional mapping from namespace prefix to namespace URI.

Returns
The first matching schema component, or `None` if there is no match.

`get_type`(*qname*)

Get the XSD global type from the schema's scope. A concrete implementation must return an object that supports the protocols *XsdTypeProtocol*, or *None* if the global type is not found.

Parameters
qname – the fully qualified name of the type to retrieve.

Returns
an object that represents an XSD type or *None*.

`get_attribute`(*qname*)

Get the XSD global attribute from the schema's scope. A concrete implementation must return an object that supports the protocol *XsdAttributeProtocol*, or *None* if the global attribute is not found.

Parameters
qname – the fully qualified name of the attribute to retrieve.

Returns
an object that represents an XSD attribute or *None*.

`get_element`(*qname*)

Get the XSD global element from the schema's scope. A concrete implementation must return an object that supports the protocol *XsdElementProtocol* interface, or *None* if the global element is not found.

Parameters
qname – the fully qualified name of the element to retrieve.

Returns
an object that represents an XSD element or *None*.

`abstract is_instance`(*obj*, *type_qname*)

Returns *True* if *obj* is an instance of the XSD global type, *False* if not.

Parameters

- **obj** – the instance to be tested.
- **type_qname** – the fully qualified name of the type used to test the instance.

`abstract cast_as`(*obj*, *type_qname*)

Converts *obj* to the Python type associated with an XSD global type. A concrete implementation must raises a *ValueError* or *TypeError* in case of a decoding error or a *KeyError* if the type is not bound to the schema's scope.

Parameters

- **obj** – the instance to be cast.
- **type_qname** – the fully qualified name of the type used to convert the instance.

abstract iter_atomic_types()

Returns an iterator for not builtin atomic types defined in the schema's scope. A concrete implementation must yield objects that implement the protocol *XsdTypeProtocol*.

3.6 XPath nodes

XPath nodes are processed using a set of classes derived from *elementpath.XPathNode*. This class hierarchy is as simple as possible, with a focus on speed a low memory consumption.

class XPathNode

The base class of all XPath nodes. Used only for type checking.

The seven XPath node types:

class AttributeNode(*name*, *value*, *parent=None*, *position=1*, *xsd_type=None*)

A class for processing XPath attribute nodes.

Parameters

- **name** – the attribute name.
- **value** – a string value or an XSD attribute when XPath is applied on a schema.
- **parent** – the parent element node.
- **position** – the position of the node in the document.
- **xsd_type** – an optional XSD type associated with the attribute node.

class NamespaceNode(*prefix*, *uri*, *parent=None*, *position=1*)

A class for processing XPath namespace nodes.

Parameters

- **prefix** – the namespace prefix.
- **uri** – the namespace URI.
- **parent** – the parent element node.
- **position** – the position of the node in the document.

class TextNode(*value*, *parent=None*, *position=1*)

A class for processing XPath text nodes. An Element's property (elem.text or elem.tail) with a *None* value is not a text node.

Parameters

- **value** – a string value.
- **parent** – the parent element node.
- **position** – the position of the node in the document.

class CommentNode(*elem*, *parent=None*, *position=1*)

A class for processing XPath comment nodes.

Parameters

- **elem** – the wrapped Comment Element.
- **parent** – the parent element node.

- **position** – the position of the node in the document.

class ProcessingInstructionNode(*elem*, *parent=None*, *position=1*)

A class for XPath processing instructions nodes.

Parameters

- **elem** – the wrapped Processing Instruction Element.
- **parent** – the parent element node.
- **position** – the position of the node in the document.

class ElementNode(*elem*, *parent=None*, *position=1*, *nsmap=None*, *xsd_type=None*)

A class for processing XPath element nodes that uses lazy properties to diminish the average load for a tree processing.

Parameters

- **elem** – the wrapped Element or XSD schema/element.
- **parent** – the parent document node or element node.
- **position** – the position of the node in the document.
- **nsmap** – an optional mapping from prefix to namespace URI.
- **xsd_type** – an optional XSD type associated with the element node.

class DocumentNode(*document*, *uri=None*, *position=1*)

A class for XPath document nodes.

Parameters

- **document** – the wrapped ElementTree instance.
- **position** – the position of the node in the document, usually 1, or 0 for lxml standalone root elements with siblings.

There are also other two specialized versions of ElementNode usable on specific cases:

class LazyElementNode(*elem*, *parent=None*, *position=1*, *nsmap=None*, *xsd_type=None*)

A fully lazy element node, slower but better if the node does not to be used in a document context. The node extends descendants but does not record positions and a map of elements.

class SchemaElementNode(*elem*, *parent=None*, *position=1*, *nsmap=None*, *xsd_type=None*)

An element node class for wrapping the XSD schema and its elements. The resulting structure can be a tree or a set of disjoint trees. With more roots only one of them is the schema node.

3.7 Node tree builders

Node trees are automatically created during the initialization of an *elementpath.XPathContext*. But if you need to process the same XML data more times there is an helper API for creating document or element based node trees:

get_node_tree(*root*, *namespaces=None*, *uri=None*, *fragment=False*)

Returns a tree of XPath nodes that wrap the provided root tree.

Parameters

- **root** – an Element or an ElementTree or a schema or a schema element.

- **namespaces** – an optional mapping from prefixes to namespace URIs, Ignored if root is a lxml etree or a schema structure.
- **uri** – an optional URI associated with the root element or the document.
- **fragment** – if *True* a root element is considered a fragment, otherwise a root element is considered the root of an XML document. If the root is a document node or an ElementTree instance, and fragment is *True* then use the root element and returns an element node.

build_node_tree(*root*, *namespaces=None*, *uri=None*)

Returns a tree of XPath nodes that wrap the provided root tree.

Parameters

- **root** – an Element or an ElementTree.
- **namespaces** – an optional mapping from prefixes to namespace URIs.
- **uri** – an optional URI associated with the document or the root element.

build_lxml_node_tree(*root*, *uri=None*, *fragment=False*)

Returns a tree of XPath nodes that wrap the provided lxml root tree.

Parameters

- **root** – a lxml Element or a lxml ElementTree.
- **uri** – an optional URI associated with the document or the root element.
- **fragment** – if *True* a root element is considered a fragment, otherwise a root element is considered the root of an XML document.

build_schema_node_tree(*root*, *uri=None*, *elements=None*, *global_elements=None*)

Returns a tree of XPath nodes that wrap the provided XSD schema structure.

Parameters

- **root** – a schema or a schema element.
- **uri** – an optional URI associated with the root element.
- **elements** – a shared map from XSD elements to tree nodes. Provided for linking together parts of the same schema or other schemas.
- **global_elements** – a list for schema global elements, used for linking the elements declared by reference.

3.8 XPath regular expressions

translate_pattern(*pattern*, *flags=0*, *xsd_version='1.0'*, *back_references=True*, *lazy_quantifiers=True*, *anchors=True*)

Translates a pattern regex expression to a Python regex pattern. With default options the translator processes XPath 2.0/XQuery 1.0 regex patterns. For XML Schema patterns set all boolean options to *False*.

Parameters

- **pattern** – the source XML Schema regular expression.
- **flags** – regex flags as represented by Python's re module.
- **xsd_version** – apply regex rules of a specific XSD version, '1.0' for default.
- **back_references** – if *True* supports back-references and capturing groups.

- **lazy_quantifiers** – if *True* supports lazy quantifiers (*?, +?).
- **anchors** – if *True* supports ^ and $ anchors, otherwise the translated pattern is anchored to its boundaries and anchors are treated as normal characters.

3.9 Exception classes

exception ElementPathError(*message*, *code=None*, *token=None*)

Base exception class for elementpath package.

Parameters

- **message** – the message related to the error.
- **code** – an optional error code.
- **token** – an optional token instance related with the error.

exception MissingContextError(*message*, *code=None*, *token=None*)

Raised when the dynamic context is required for evaluate the XPath expression.

exception RegexError

Error in a regular expression or in a character class specification. This exception is derived from *Exception* base class and is raised only by the regex subpackage.

exception ElementPathLocaleError(*message*, *code=None*, *token=None*)

There are also other exceptions, multiple derived from the base exception *elementpath.ElementPathError* and Python built-in exceptions:

exception ElementPathKeyError(*message*, *code=None*, *token=None*)

exception ElementPathNameError(*message*, *code=None*, *token=None*)

exception ElementPathOverflowError(*message*, *code=None*, *token=None*)

exception ElementPathRuntimeError(*message*, *code=None*, *token=None*)

exception ElementPathSyntaxError(*message*, *code=None*, *token=None*)

exception ElementPathTypeError(*message*, *code=None*, *token=None*)

exception ElementPathValueError(*message*, *code=None*, *token=None*)

exception ElementPathZeroDivisionError(*message*, *code=None*, *token=None*)

CHAPTER

FOUR

PRATT'S PARSER API

The TDOP (Top Down Operator Precedence) parser implemented within this library is a variant of the original Pratt's parser based on a class for the parser and meta-classes for tokens.

The parser base class includes helper functions for registering token classes, the Pratt's methods and a regexp-based tokenizer builder. There are also additional methods and attributes to help the developing of new parsers. Parsers can be defined by class derivation and following a tokens registration procedure. These classes are not available at package level but only within module *elementpath.tdop*.

4.1 Token base class

class Token(*parser*, *value=None*)

Token base class for defining a parser based on Pratt's method.

Each token instance is a list-like object. The number of token's items is the arity of the represented operator, where token's items are the operands. Nullary operators are used for symbols, names and literals. Tokens with items represent the other operators (unary, binary and so on).

Each token class has a *symbol*, a lbp (left binding power) value and a rbp (right binding power) value, that are used in the sense described by the Pratt's method. This implementation of Pratt tokens includes two extra attributes, *pattern* and *label*, that can be used to simplify the parsing of symbols in a concrete parser.

Parameters

- **parser** – The parser instance that creates the token instance.
- **value** – The token value. If not provided defaults to token symbol.

Variables

- **symbol** – the symbol of the token class.
- **lbp** – Pratt's left binding power, defaults to 0.
- **rbp** – Pratt's right binding power, defaults to 0.
- **pattern** – the regex pattern used for the token class. Defaults to the escaped symbol. Can be customized to match more detailed conditions (e.g. a function with its left round bracket), in order to simplify the related code.
- **label** – defines the typology of the token class. Its value is used in representations of the token instance and can be used to restrict code choices without more complicated analysis. The label value can be set as needed by the parser implementation (eg. 'function', 'axis', 'constructor function' are used by the XPath parsers). In the base parser class defaults to 'symbol' with 'literal' and 'operator' as possible alternatives. If set by a tuple of values the token class label is transformed to a multi-value label, that means the token class can covers

multiple roles (e.g. as XPath function or axis). In those cases the definitive role is defined at parse time (nud and/or led methods) after the token instance creation.

arity

tree

Returns a tree representation string.

source

Returns the source representation string.

nud()

Pratt's null denotation method

led(*left*)

Pratt's left denotation method

evaluate()

Evaluation method

iter(**symbols*)

Returns a generator for iterating the token's tree.

Helper methods for checking symbols and for error raising:

expected(**symbols*, *message=None*)

unexpected(**symbols*, *message=None*)

wrong_syntax(*message=None*)

wrong_value(*message='invalid value'*)

wrong_type(*message='invalid type'*)

4.2 Parser base class

class Parser

Parser class for implementing a Top-Down Operator Precedence parser.

Variables

- **symbol_table** – a dictionary that stores the token classes defined for the language.
- **token_base_class** – the base class for creating language's token classes.
- **tokenizer** – the language tokenizer compiled regexp.

position

Property that returns the current line and column indexes.

Parsing methods:

parse(*source*)

Parses a source code of the formal language. This is the main method that has to be called for a parser's instance.

Parameters

source – The source string.

Returns
The root of the token's tree that parse the source.

`advance`(**symbols*, *message=None*)

The Pratt's function for advancing to next token.

Parameters

- **symbols** – Optional arguments tuple. If not empty one of the provided symbols is expected. If the next token's symbol differs the parser raises a parse error.
- **message** – Optional custom message for unexpected symbols.

Returns
The current token instance.

`advance_until`(**stop_symbols*)

Advances until one of the symbols is found or the end of source is reached, returning the raw source string placed before. Useful for raw parsing of comments and references enclosed between specific symbols.

Parameters
stop_symbols – The symbols that have to be found for stopping advance.

Returns
The source string chunk enclosed between the initial position and the first stop symbol.

`expression`(*rbp=0*)

Pratt's function for parsing an expression. It calls token.nud() and then advances until the right binding power is less the left binding power of the next token, invoking the led() method on the following token.

Parameters
rbp – right binding power for the expression.

Returns
left token.

Helper methods for checking parser status:

`is_source_start`()

Returns *True* if the parser is positioned at the start of the source, ignoring the spaces.

`is_line_start`()

Returns *True* if the parser is positioned at the start of a source line, ignoring the spaces.

`is_spaced`(*before=True*, *after=True*)

Returns *True* if the source has an extra space (whitespace, tab or newline) immediately before or after the current position of the parser.

Parameters

- **before** – if *True* considers also the extra spaces before the current token symbol.
- **after** – if *True* considers also the extra spaces after the current token symbol.

Helper methods for building new parsers:

`classmethod register`(*symbol*, ***kwargs*)

Register/update a token class in the symbol table.

Parameters

- **symbol** – The identifier symbol for a new class or an existent token class.
- **kwargs** – Optional attributes/methods for the token class.

Returns
A token class.

classmethod unregister(*symbol*)

Unregister a token class from the symbol table.

classmethod duplicate(*symbol*, *new_symbol*, ***kwargs*)

Duplicate a token class with a new symbol.

classmethod literal(*symbol*, *bp=0*)

Register a token for a symbol that represents a *literal*.

classmethod nullary(*symbol*, *bp=0*)

Register a token for a symbol that represents a *nullary* operator.

classmethod prefix(*symbol*, *bp=0*)

Register a token for a symbol that represents a *prefix* unary operator.

classmethod postfix(*symbol*, *bp=0*)

Register a token for a symbol that represents a *postfix* unary operator.

classmethod infix(*symbol*, *bp=0*)

Register a token for a symbol that represents an *infix* binary operator.

classmethod infixr(*symbol*, *bp=0*)

Register a token for a symbol that represents an *infixr* binary operator.

classmethod method(*symbol*, *bp=0*)

Register a token for a symbol that represents a custom operator or redefine a method for an existing token.

classmethod build()

Builds the parser class. Checks if all declared symbols are defined and builds the regex tokenizer using the symbol related patterns.

classmethod create_tokenizer(*symbol_table*)

Returns a regex based tokenizer built from a symbol table of token classes. The returned tokenizer skips extra spaces between symbols.

A regular expression is created from the symbol table of the parser using a template. The symbols are inserted in the template putting the longer symbols first. Symbols and their patterns can't contain spaces.

Parameters
symbol_table – a dictionary containing the token classes of the formal language.

INDEX

N

P

R

S

T

U

V

W

X