Easy XML Documentation

Release 1.2.0

dp_wiz <aenor.realm@gmail.com>

Jun 12, 2017

Contents

XML Element container and builder.

CHAPTER 1

Trivial stuff

```
>>> exml.Element("tag")
Element(<tag/>)
```

```
>>> exml.Element("node", attr="value", intval=1)
Element(<node attr="value" intval="1"/>)
```

“call” element to set children

```
>>> Element("text")("body")
Element(<text>body</text>)
```

```
>>> print Element("nobody", expects=True)(
...     "Spanish Inquisition",
...     " ",
...     Element("not", at="all")
... )
<nobody expects="True">Spanish Inquisition <not at="all"/></nobody>
```

List-like append for adding children nodes

```
>>> list = Element("list")
>>> list.append(Element("item")("lorem"))
>>> list.append(Element("item")("ipsum"))
>>> print list
<list><item>lorem</item><item>ipsum</item></list>
```

Alternatively, start with an empty Element

```
>>> e = Element()
>>> e.node = "test"
>>> e['from'] = "scratch"
>>> e.append("ok")
>>> print e
<test from="scratch">ok</test>
```

Some NameSpace examples

From http://www.w3.org/TR/REC-xml-names/#ns-decl

```
>>> print Element("x", xmlns__edi="http://ecommerce.example.org/schema")('''
...   <!-- the "edi" prefix is bound to http://ecommerce.example.org/schema
...        for the "x" element and contents -->
... ''')
<x xmlns:edi="http://ecommerce.example.org/schema">
  <!-- the "edi" prefix is bound to http://ecommerce.example.org/schema
       for the "x" element and contents -->
</x>
```

```
>>> print Element("book", xmlns="urn:loc.gov:books", xmlns__isbn="urn:ISBN:0-395-
↪36341-6")(
...     Element("title")("Cheaper by the Dozen"),
...     Element("isbn:number")("1568491379"))
<book xmlns="urn:loc.gov:books" xmlns:isbn="urn:ISBN:0-395-36341-6"><title>Cheaper by␣
↪the Dozen</title><isbn:number>1568491379</isbn:number></book>
```

CHAPTER 2

Using “E the Builder” shortcut

Node name goes into attribute name and then there are only clean attributes in parens’.

```
>>> E = Builder()
>>> E.use_getattr
<function Element("use_getattr", ...) at 0x...>
```

A little more complex example:

```
>>> print E.tree(x="mas")(
...     E.stump(),
...     E.trunk()(
...         E.branch()(
...             E.ball(color="red"),
...             E.ball(color="gold"),
...         ),
...         E.branch()(
...             E.ball(color="green"),
...         ),
...     ),
...     E.tip()(
...         E.star(points="5")
...     )
... )
<tree x="mas"><stump/><trunk><branch><ball color="red"/><ball color="gold"/></branch>
↪<branch><ball color="green"/></branch></trunk><tip><star points="5"/></tip></tree>
```

Double-underscores-to-namespace trick works here too:

```
>>> print E.ns__are(fine="too")
<ns:are fine="too"/>
```

CHAPTER 3

SOAP Requests

As an example (and my primary use case) soap.py contains rather simplistic WSDL-ignorant request generator.

Service assumes it is you, who knows schema definitions and correct types, so just set it to endpoint and pass a request namespace.

```
>>> service = BasicSOAP("https://bathroom.hostname.tld/Cleaning.aspx", 'Cleaning-
↪Service')
```

BasicSOAP is used to simply generate a SOAP 1.1 request and does not process server response.

```
>>> service._request_xml('SomeMethod', Some="parameters", EverythingGoesRaw=1,␣
↪TakeCareOf={'your': 'types'})
Element(<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"␣
↪xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/
↪XMLSchema-instance"><soap:Header/><soap:Body><SomeMethod xmlns="Cleaning-Service">
↪<request><EverythingGoesRaw>1</EverythingGoesRaw><Some>parameters</Some><TakeCareOf>
↪{'your': 'types'}</TakeCareOf></request></SomeMethod></soap:Body></soap:Envelope>)
```

You can subclass SOAP/BasicSOAP and override header and/or body methods (envelope too, but..).

```
>>> class AuthSOAP(SOAP):
...     def _header(self):
...         return E.soap__Header()(
...             E.AuthenticationHeader(xmlns=self.request_NS)(
...                 E.User()("Me"),
...                 E.Pwd()("Cat")))
>>> AuthSOAP("http://knock-knock.com", "SecureService")._request_xml('Hello', lets=
↪'party')
Element(<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"␣
↪xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/
↪XMLSchema-instance"><soap:Header><AuthenticationHeader xmlns="SecureService"><User>
↪Me</User><Pwd>Cat</Pwd></AuthenticationHeader></soap:Header><soap:Body><Hello xmlns=
↪"SecureService"><request><lets>party</lets></request></Hello></soap:Body></
↪soap:Envelope>)
```

If lxml is installed, there is a SOAP class, which utilizes etree parser to process SOAP response.

LxmlProcessor mixin lifts value from trivial result nodes. You can override its behavior in _result_map/_result_reduce methods.

```
>>> service = SOAP("http://www.kirupafx.com/WebService/TopMovies.asmx", "http://www.
↪kirupafx.com")
>>> service.GetMovieAtNumber(input=1)
'The Godfather (1972)'
```

Result nodes that have same name are wrapped into lists. Check your classes!

```
>>> service = SOAP("http://www.kirupafx.com/WebService/TopMovies.asmx", "http://www.
↪kirupafx.com")
>>> service.GetTop10()
{'string': ['The Godfather (1972)', 'The Shawshank Redemption (1994)', 'The␣
↪Godfather: Part II (1974)', 'The Lord of the Rings: The Return of the King (2003)',
↪'Casablanca', "Schindler's List", 'Shichinin no samurai (1954)', 'Buono, il brutto,␣
↪il cattivo, Il (1966)', 'Pulp Fiction (1994)', 'Star Wars: Episode V - The Empire␣
↪Strikes Back (1980)']}
```