osa Documentation

Release 0.1.5

Sergey Bozhenkov

June 11, 2013

CONTENTS

osa is a fast/slim library to consume WSDL 1.1/SOAP 1.1 services. It is created with the following three requirements in mind: fast calls, small memory footprint and convenience of use. I was not able to find a library that meets all my requirements, especially for large messages (millions of elements). Therefore I created this library by combining ideas found in suds (nice printing), soaplib (serialization/deserialization) and Scio (WSDL 1.1 parsing).

At the moment the library is limited to wrapped document/literal SOAP 1.1 convention. To include other call conventions one has to extend the `to_xml()` and `from_xml()` methods of the `Message` *class*. The structure of the library is briefly explained *here*. The *XML* processing is performed with the help of `cElementTree` module.

To install the library please do the usual *Python* magic:

```
>>> python setup.py install
```

Online help is available for all classes, please see also section *Using* for examples.

Contents:

CHAPTER

ONE

USING

To use the library do the import:

```
>>> import osa
```

This exposes the top level class *Client*. It the only one class used to consume a service by a normal user. The client is initialized by full address of a WSDL 1.1 document:

```
>>> cl = osa.Client("http://.../HelloWorldService?wsdl")
```

Convenience print functions are available at several levels, e.g. to find information about the client one can enter:

```
>>> cl
```

which returns names of all found services in the WSDL 1.1 document and location of the service:

```
service HelloWorldService from:
    http://.../HelloWorldService?wsdl
```

The top level client is a container for class definitions constructed from *XML* types in the supplied WSDL 1.1 focument and for remote method wrappers. All types are contained in `cl.types` and all methods are available through `cl.service`. It is again possible to inspect them by printing:

```
>>> cl.types
```

which lists all known types and help if available:

```
Person
    no documentation
Name
    no documentation
...
```

Similarly:

```
>>> cl.service
```

prints all found methods and there short description if available:

```
sayHello
    str[] result | None = sayHello(sayHello msg)
    str[] result | None = sayHello(Person person , int time...
echoString
    str result  = echoString(echoString msg)
    str result  = echoString(str msg )
    str result  = echoString...
    ...
```

It is worth noting once more that if any documentation is available in the initial WSDL 1.1 document it is propagated to types and methods.

To create an instance of a type in `cl.types` is easy (note that tab completion works both for types and methods):

```
>>> person = cl.types.Person()
```

To inspect the new instance simply print it:

```
>>> person
(Person){
name = None (Name)
weight = None (int)
age = None (int)
height = None (int)
}
```

As can be seen all attributes of the new instance are empty, i.e. they are `None`. Expected types of attributes are given after `None` in the brackets. Sometimes it useful to initialize immediately all obligatory (non-nillable) attributes. To do this one can use `deep` keyword to class constructors:

```
>>> person = cl.types.Person(deep = True)
```

which initializes the whole hierarchy:

```
(Person){
name = (Name){
            firstName =
            lastName =
       }
weight = 0
age = 0
height = 0
}
```

The attributes can be set with the usual dot-convention:

```
>>> person.name.firstName = "Osa"
>>> person.name.lastName = "Wasp"
```

To call a method one can access it directly from :py:attr‘cl.service‘. Help to a method can be viewed by simply printing its doc (*ipython* style):

```
>>> cl.service.sayHello ?
```

This shows possible call signatures and gives help from the WSDL 1.1 document:

```
Type:             Method
Base Class:       <class 'osa.methods.Method'>
String Form:   str[] result | None = sayHello(sayHello msg)
Namespace:        Interactive
File:             /usr/local/lib/python2.6/site-packages/osa-0.1-py2.6.
egg/osa/methods.py
Docstring:
    str[] result | None = sayHello(sayHello msg)
    str[] result | None = sayHello(Person person , int times )
    str[] result | None = sayHello(person=Person , times=int )

               says hello to person.name.firstName given number of times
               illustrates usage of compex types

...
```

It is possible to call any method in four different formats:

- single input parameter with proper wrapper message for this functions
- expanded positional parameters: children of the wrapper message
- expanded keyword parameters
- mixture of positional and keyword parameters.

The help page shows all possible signatures with explained types. On return the message is expanded so that a real output is returned instead of the wrapper. The return type is also shown in the help. Please note, that lists are used in place of arrays for any types, this is shown by brackets `[]`. Finally, let's make the call:

```
>>> cl.service.sayHello(person, 5)
['Hello, Osa', 'Hello, Osa', 'Hello, Osa', 'Hello, Osa', 'Hello, Osa']
```

The library can also handle *XML* `anyType` properly in most of the cases: *any* variable chooses the suitable type from the service and uses it to do the conversion from *XML* to *Python*.

The library can be used with large messages, e.g about 8 millions of double elements are processed in few tens of seconds only. The transient peak memory consumption for such a message is of the order of 1 GB.

CHAPTER

TWO

STRUCTURE

This section briefly explains the library structure. It is useful for those who wants to improve it.

The top level *Client* class is simply a container. On construction it creates an instance of *WSDLParser* and processes the service description by calling its methods `parse()`. Afterwards the parser is deleted. As a result of the initial processing two dictionaries are available: containing newly created types and methods.

Types and methods are generated by the *WSDLParser*, for types it internally uses *XMLSchemaParser*. The types are constructed by using meta-class *ComplexTypeMeta*. This meta-class has a special convention to pass children names and types. The methods are wrapped as instances of *Method* class. The latter class has a suitable `__call__()` method and contains information about input and output arguments as instances of the *Message* class in attributes `input` and `output` correspondingly.

The top level *Client* class creates sub-containers for types and methods: `types` and `service`. This containers have special print function to display help. Types and methods are set as direct attributes of the corresponding containers, so that the usual dot-access and tab-completion are possible. The attributes of the `types` container are class definitions, so that to create a new instance one has to add the brackets *()*. The attributes of the `service` container are callable method wrappers.

To allow a correct `anyType` processing the *WSDLParser* updates special dictionary of *XMLAny* class by all discovered classes.

Every function call is processed by `__call__()` method of a *Method* instance. The call method uses the input message `input` to convert its arguments to *XML* string (`to_xml()`). Afterwards *urllib2* is used to send the request to service. The service response is deserialized by using the output message `output` (`from_xml()`). The deserialized result is returned to the user.

The input points for serialization is a *Message* instance. The message first analyzes the input arguments and if required wraps them into a top level message. Afterwards `to_xml()` methods of al children are called with a proper *XML* element. The children create *XML* elements for them and propagate the call to their children and so on. The process is continued until the bottom of the hierarchy is reached. Only the primitive *types* set the real text tag. The deserialization process is similar: in this case `from_xml()` is propagated and all children classes are constructed. In

addition the output message parser expands the response wrapper, so that the user sees the result without the shell.

At the moment only wrapped document/literal convention is realized. The format of the message is determined by `to_xml()` and `from_xml()`. Therefore, to introduce other conventions (rpc, encoded) one has to modify these two methods only.

The library uses `cElementTree` module for *XML* processing. This module has about 2 times lower memory footprint as the usual `lxml` library.

CHAPTER

THREE

API INDEX

3.1 Client

Top level access to SOAP service.

class osa.client.**Client**(*wsdl_url*)

Bases: object

Top level class to talk to soap services.

This is an access point to service functionality. The client accepts WSDL address and uses osa.wsdl.WSDLParser to get all defined types and operations. The types are set to client.types and operations are set to self.service.

To examine present types or operations simply print (or touch repr):

```
>>> client.types
```

or:

```
>>> client.service
```

correspondingly.

To create type simply call:

```
>>> client.types.MyTypeName().
```

Class constructor will also create all obligatory (non-nillable) children. To call an operation:

```
>>> client.service.MyOperationName(arg1, arg2, arg3, ...),
```

where arguments are of required types. Arguments can also be passed as keywords or a ready wrapped message.

If any help is available in the WSDL document it is propagated to the types and operations, see e.g. help client.types.MyTypeName. In addition the help page on an operation displays its call signature.

Nice printing is also available for all types defined in client.types:

```
>>> print(client.types.MyTypeName())
```

Warning: Only document/literal wrapped convention is implemented at the moment.

In reality client.types and client.service are simply containers. The content of these containers is set from results of parsing the wsdl document by `osa.wsdl.WSDLParser.get_types` and `osa.wsdl.WSDLParser.get_services` correspondingly. See also `osa.wsdl.WSDLParser.parse`.

The client.types container consists of auto generated (by `osa.xmlschema.XMLSchemaParser`) class definitions. So that a call to a member returns and instance of the new type. New types are auto-generated according to a special convention by metaclass `osa.xmltypes.ComplexTypeMeta`.

The client.service container consists of methods wrapers methods.Method. The method wrapper is callable with free number of parameters. The input and output requirements of a method are contained in methods.Message instances `osa.methods.Method.input` and `osa.methods.Method.output` correspondingly. On a call a method converts the input to XML by using Method.input, sends request to the service and finally decodes the response from XML by Method.output.

Parameters **wsdl_url** : str

Address of wsdl document to consume.

Methods

create_services_containers()
Create methods containers for easy access.

As a result of this method, self.service with available operations is created. If there are several services in the supplied wsdl, than self.service_1, self.service_2 are created.

create_types_container()
Create types container class for easy access.

As a result of this method, self.types contains all the defined classes with their short names, i.e. without namespace prefix. If a name collision is detected, the second and all consecutive classes are appended with a counter.

osa.client.**str_for_containers**(*self*)
Nice printing for types and method containers.

Containers must have _container attribute containing all elements to be printed.

3.2 WSDL parser

Conversion of WSDL documents into Python.

class osa.wsdl.**WSDLParser**(*wsdl_url*)
Bases: object

Parser to get types and methods defined in the document.

Methods

get_bindings(*operations*)
Check binding document/literal and http transport.

If any of the conditions is not satisfied the binding is dropped, i.e. not present in the return value. This also sets soapAction and use_parts of the messages.

Parameters **operations** : dict as returned by get_operations

Returns **out** : dict

Map similar to that from get_operations but with binding names instead of portType names.

get_messages(*types*)
Construct messages from message section.

Parameters **types** : dictionary of types

Types as returned by get_types().

Returns **out** : dict

Map message name -> Message instance

get_operations(*messages*)

Get list of operations with messages from portType section.

Parameters **messages** : dict

Dictionary of message from get_messages.

Returns **out** : dict

{portType -> {operation name -> Method instance}} The method here does not have location.

get_services(*bindings*)
Find all services an make final list of operations.

This also sets location to all operations.

Parameters **bindings** : dic from get_bindings.

Returns **out** : dict

Dictionary {service -> {operation name -> method}.

get_types()
Constructs a map of all types defined in the document.

Returns **out** : dict

A map of found types {type_name : complex class}

parse()
Do parsing, return types, services.

Returns **out** : (types, services)

Conversion of XML Schema types into Python classes.

class osa.xmlschema.**XMLSchemaParser**(*root*)
Bases: object

Parser to get types from an XML Schema.

Methods

static convert_xmltypes_to_python(*xtypes*)
Convert xml types definitions in the dictionary into Python classes.

Parameters **xtypes** : dictionary name -> xml element

A dictionary as returned by get_list_of_defined_types.

Returns **out** : dictionary name -> Python class

static create_alias(*name*, *alias_type*, *xtypes*, *types*)
Create a copy of known class with proper namespace.

Parameters **name** : str

Name of the new class.

alias_type : str

The target alias

xtypes : dictionary class name -> xml node

types : dictionary of classes

The new aliases is appended here.

static create_complex_class(*name*, *element*, *xtypes*, *types*)
Create complex class.

Parameters **name** : str

Class name

element : xml element

Class node.

xtypes : dictionary class name -> xml node

types : dictionary class name -> Python class

The result is appended here.

static **create_empty_class**(*name*, *types*)

Create empty class, i.e. no children.

Parameters **name** : str

Name of the new class.

alias_type : str

The target alias

xtypes : dictionary class name -> xml node

types : dictionary of classes

The new aliases is appended here.

static **create_string_enumeration**(*name*, *element*, *types*)

Creates a copy of XMLStringEnumertion with properly set allowed values.

The created class is attached to types.

Parameters **name** : str

Name of the new class.

element : `etree.Element`

XML description of the enumeration

types : dictionary of classes

static **create_type**(*name*, *element*, *xtypes*, *types*)

Creates proper type for the element.

The created types is appended to the types.

Parameters **name** : str

Class name

element : xml element

Class node.

xtypes : dictionary class name -> xml node

types : dictionary class name -> Python class

The result is appended here.

generate_classes()
Generate Python classes from this schema.

Returns **out** : dictionary

Dictionary of types {ns}name -> Python class

static get_doc(*x*)
Extract documentation from element.

Parameters **x** : xml element

Returns **out** : str

Documentation from whatever found <documentation> out </documentation>

get_list_of_defined_types()
Construct a dictionary: type name -> xml node

Types are given by complexType, simpleType or element. Types from imported schemas are included as well. Type names include namespaces.

Returns **out** : dict

A dictionary of defined types.

static get_type_by_name(*name*, *xtypes*, *types*)
Return requested class from primmap or as created from xml.

Parameters **name** : str

Type name.

xtypes : dict

List of xml elements to look in.

types : dict

List of already created classes to look in.

Returns **out** : class

3.3 Methods wrapper

Python class for input/output messages.

class osa.message.**Message**(*name*, *parts*, *use_parts=None*)

Bases: object

Message for input and output of service operations.

Messages perform conversion of Python to xml and backwards of the calls and returns.

At the moment only document/literal wrapped is implemented.

Parameters **name** : str

Namespace qualified name of the message.

parts : list

List of message parts in the form (part name, part type class). This description is usually found in message part of a WSDL document. Note, that due to binding section not all message parts are used for encoding. The parts that are used are given be use_parts.

use_parts : list

List of parts to be really used for encoding/decoding. This comes from wsdl binding section. Yes, they are not quite from this planet. In any case, in the present implementation I assume doc/literal wrapped and use only the very first part from this member for encoding.

Methods

from_xml(*body*, *header=None*)

Convert from xml message to Python.

to_xml(**arg*, ***kw*)

Convert from Python into xml message.

This function accepts parameters as they are supplied to the method call and tries to convert it to a message. Arguments can be in one of four forms:

- 1 argument of proper message type for this operation
- positional arguments - members of the proper message type
- keyword arguments - members of the message type.
- a mixture of positional and keyword arguments.

Keyword arguments must have at least one member: _body which contains etree.Element to append the conversion result to.

SOAP operation class.

class osa.method.**Method**(*name*, *input*, *output*, *doc=None*, *action=None*, *location=None*)

Bases: object

Definition of a single SOAP method, including location, action, name and input and output classes.

Parameters **name** : str

Name of operation

input : osa.message.Message instance

Input message.

output : osa.message.Message instance

Output message.

doc : str, optional - default to None

Documentation of the method as found in portType section of WSDL.

action : str

Soap action string.

location : str

Location as found in service part of WSDL.

3.4 XML types

class osa.xmltypes.**XMLType**

Bases: object

Base xml schema type.

It defines basic functions to_xml and from_xml.

Methods

check_constraints(*n*, *min_occurs*, *max_occurs*)

Performs constraints checking.

Parameters **n** : int

Actual number of occurrences.

min_occurs [int] Minimal allowed number of occurrences.

max_occurs [int or 'unbounded'] Maximal allowed number of occurrences.

Raises **ValueError** :

If constraints are not satisfied.

classmethod **from_file**(*fname*)
Create an instance from file.

Parameters **fname** : str

Filename to parse.

Returns **out** : new instance

from_xml(*element*)
Function to convert from xml to python representation.

This is basic function and it is suitable for complex types. Primitive types must overload it.

Parameters **element** : etree.Element

Element to recover from.

to_file(*fname*)
Save to file as an xml string.

Parameters **fname** : str

Filename to use.

to_xml(*parent*, *name*)
Function to convert to xml from python representation.

This is basic function and it is suitable for complex types. Primitive types must overload it.

Parameters **parent** : etree.Element

Parent xml element to append this child to.

name : str

Full qualified (with namespace) name of this element.

class osa.xmltypes.**XMLString**
Bases: osa.xmltypes.XMLType, str

Methods

class osa.xmltypes.**XMLInteger**
Bases: osa.xmltypes.XMLType, int

Methods

class osa.xmltypes.**XMLDouble**
Bases: osa.xmltypes.XMLType, float

Methods

class osa.xmltypes.**XMLBoolean**
Bases: osa.xmltypes.XMLType, str

Methods

class osa.xmltypes.**XMLAny**
Bases: osa.xmltypes.XMLType, str

Methods

class osa.xmltypes.**XMLDecimal**
Bases: osa.xmltypes.XMLType, decimal.Decimal

Methods

class osa.xmltypes.**XMLDate**(**arg*)
Bases: osa.xmltypes.XMLType

Methods

from_xml(*element*)
expect ISO formatted dates

class osa.xmltypes.**XMLDateTime**(**arg*)
Bases: osa.xmltypes.XMLType

Methods

class osa.xmltypes.**ComplexTypeMeta**
Metaclass to create complex types on the fly.

__new__(*name*, *bases*, *attributes*)
Method to create new types.

_children attribute must be present in attributes. It describes the arguments to be present in the new type. The he _children argument must be a list of the form: [{'name':'arg1', 'min':1, 'max':1, 'type':ClassType}, ...]

Parameters **cls** : this class

name [str] Name of the new type.

bases [tuple] List of bases classes.

attributes [dict] Attributes of the new type.

3.5 SOAP constants

CHAPTER

FOUR

LICENSE

I release the library under terms of GPL. I borrowed some ideas from suds, soaplib and Scio libraries. The first two are released under LGPL. The last one has its own license, the text of which and the copyright are given below.

If someone knows better about compatibility of all these licenses, please let me know.

4.1 Scio license

Please note, that almost no original Scio code is preserved. Only algorithm of WSDL 1.1 parsing is partially preserved.

```
# Copyright (c) 2011, Leapfrog Online, LLC
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions are met:
#     * Redistributions of source code must retain the above copyright
#       notice, this list of conditions and the following disclaimer.
#     * Redistributions in binary form must reproduce the above copyright
#       notice, this list of conditions and the following disclaimer in the
#       documentation and/or other materials provided with the distribution.
#     * Neither the name of the Leapfrog Online, LLC nor the
#       names of its contributors may be used to endorse or promote products
#       derived from this software without specific prior written permission.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
# ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
# WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
# DISCLAIMED. IN NO EVENT SHALL <COPYRIGHT HOLDER> BE LIABLE FOR ANY
# DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
# (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
# LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
```

```
# ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
# (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
# SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

CHAPTER

FIVE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

o

PYTHON MODULE INDEX

o

INDEX

Symbols

C

F

G

M

O

P

S

T

W

X