XML Support: Pickling Tools 1.6.0

XML Support: New in PicklingTools 1.3.0

The XML format and Python dictionary are fairly equivalent formats: they both allow recursive, heterogeneous structures for storing data. In many ways, XML is yet another serialization format and the PicklingTools embraces XML as yet another serialization (with some limitations: DTD is not supported (1.3.0 has no support at all, 1.3.1 reads but ignores DTD), although support for namespaces is coming).

> New in PicklingTools 1.4.1 is support for a Python C-Extension module that speeds up conversion from dict to XML by 6-10x and from XML to dict by 60-100x.

If you are using XML as a key-value format, then PicklingTools and XML are essentially equivalent. Consider the tags and content of following simple XML document:

```
<doc>
  <chapter>1</chapter>
  <chapter>2</chapter>
  <appendix>
    <A>3.0</A>
    <B>4.0</B>
  <appendix>
<doc>
```

These are equivalent to the keys and values of the following Python dictionary:

```
>>> d = {
...  'chapter': [1,2],
...  'appendix': {
...      'A': 3.0,
...      'B': 4.0
...   }
... }
```

Python dictionaries tend to be easier to manipulate in Python and C++ (which is why they are the currency of the PicklingTools). but XML does have some advantages over Python Dictionaries:

1. XML is intrinsically ordered, whereas Python dictionaries aren't (but can be with the OrderedDict, see below)
2. XML can represent true documents: this is XML's intrinsic advantage

XML, Dictionaries and Ordering

Consider the ordering issue: in XML, the order of tags and content is preserved in an XML document as the tags and content always are processed in the order they appear. A Python dictionary, however, doesn't necessarily preserve the order of keys-values. Consider:

```
>>> d = { 'chapter':[1,2], 'appendix':{'A':3.0, 'B':4.0} }
>>> for key,value in d.iteritems() :
...   print key,
# Output:  chapter appendix        OR      appendix chapter
```

In the example above, "chapter appendix" is JUST AS LIKELY as "appendix chapter" as output because the Python dictionary is a *hash table only* and doesn't preserve order.

If insertion order is really a desired feature, The OrderedDict in a new data structure Python 2.7 that captures this (The equivalent in C++ is the OTab and was introduced in PicklingTools 1.2.0). The OrderedDict is just like a Python dictionary (in fact, it inherits from it), but it preserves insertion order just like XML:

```
>>> from collections import OrderedDict # Available as of Python 2.7
>>> od = OrderedDict([
...      ('chapter': [1,2]),            # Long form of the OrderedDict
...      ('appendix', OrderedDict([
...            ('A', 3.0),
...            ('B', 4.0),
...      ])),
... ])
>>> print od['chapter']  # Just like Python dict otherwise
```

Unfortunately, the default output form of OrderedDict is not as clean as the equivalent dictionary (as the OrderedDict is currently represented as a list of tuples), but it is still just as easy to manipulate in Python or C++.

To be clear: *ordered dictionary* means that keys are ordered by insertion order *NOT by sorting order*. You can still look up values via key (i.e., a['key']), but if you ITERATE through items, you iterate through them in the order they were inserted (or in the case of literals, the order they were listed).

Python uses the above long form to represent OrderedDicts, but we are hopeful to upgrade Python to make OrderedDicts a more first-class object. The notation that C++ uses is to use 'o{ }' to represent Ordered Dictionaries. Consider, C++ can use a little letter o to indicate it's an ordered dictionary:

```
o{
  'chapter': [1,2],
  'appendix': o{        // C++ output can choose between the short
      'A': 4.0          // form (this example) or the long form
      'B': 3.0,         // (above) of OrderedDict.  Unfortunately,
   }                    // Python DOES NOT understand this short form.
}
```

From Python, the OrderedDict behaves JUST like the Python dictionary, except for the fact that the insertion order of the keys is preserved. Thus, when you iterate (or print), you see the original insertion order.:

```
# Python:  2.7 and above
>>> from collections import OrderedDict
>>> od = OrderedDict([('chapter',[1,2]),\
...                  ('appendix',OrderedDict([('A',3.0),('B',4.0)]))])
>>> for (key, value) in od.iteritems() :
...    print key                    # For OrderedDict, preserves
...                                 # insertion order
chapter
appendix

// C++:  PicklingTools 1.2.0 and above
OTab oo = OTab("o{'chapter':[1,2], 'appendix':o{'A':4.0,'B':3.0} }");
for (It ii(od); ii(); ) {
    cout << ii.key() << endl;
}
// chapter
// appendix
```

By using the OrderedDict, the insertion order can be preserved. Many times, however, the insertion order is not relevant: a user may simply care for the absence/presence of keys in the table, in which case, a Python dictionary is fine to use.

In the case ordering is an issue, XML and Python dictionaries can be equivalent: you just have to use the Python OrderedDict instead of the dict: Simply choose the XML_LOAD_USE_OTABS options in the XMLLoader when translating from XML to Python dictionaries.

XML, Dictionaries, and Documents

When it comes to representing documents, XML is the medium to use; This is XML's raison d'etre: tags interspersed with content and data and attributes, Consider the very simple XML document below:

```
<text>It was the <it color='green'>best</it> of times,
      it was the <it color='red'>worst</it> of times.
</text>
```

There is no real "easy" equivalent of the above in Python dictionaries: you can make up a format which captures all the information, but in the end, it is just a hack over XML:

```
>>> { 'text': o{ 0:'It was the ', 'it':{ 'color='green, '_':'best',
...              2:'of times\n',
...              3:'     it was the ', 'it':'worst', 4:' of times.\n'  } }
```

The above *kind of* works as a translation between XML and Python dictionaries, but breaks down quickly with more complex documents with attributes, nested tags or content interspersed.

If you are manipulating documents like above, don't use Python Dictionaries! Use some other format that is made for documents: XML, LaTeX, and REStructed Text are some alternatives for expressing documents. (In fact, REStructured Text permeates the PicklingTools documentation because it's a simple way to produce documentation in text, PDF and HTML)

For key-value pairs, we can translate directly between XML and Python dictionaries. For documents, Python dictionaries are the wrong choice.

> Sidebar: It can make sense to have a document embedded with the text of a Python dictionary, if you want to keep meta-information around it:

```
>>> book = {
...   'book': 'A Tale of Two Cities',
...   'REStructuedText': 'It was the *best* of times, it was the *worst* of times',
...   'XML': '<top>It was the <it>best</it> of time, it was the <it>worst</it> of ti
... }
```

Translating between XML and Python Dictionaries

The PicklingTools offers tools to translate between XML and Python dictionaries (both directions) from both C++ and Python. The tools assume one major maxim:

```
Assumption: We are using XML to represent recursive, heterogeneous
key-values data structures.  In this case, we can translate back and forth
between XML and Python dictionaries and not lose information.
```

The interfaces are essentially the same in both Python and C++: there is an XMLDumper which converts from plain dictionaries to XML, and an XMLLoader which converts from XML to plain dictionaries. The

Python tools are easier to use than the C++ tools, so we'll discuss those first, but all the interfaces for both are basically the same.

> There was quite a bit of work when doing the XML tools in Python and C++ to make sure the interfaces were the same and the outputs were the same as well. There are two tests called xmldump_test.[py/cc] and xmlload_test.[py/cc] that use exactly the same output to compare against. Although there are tools in Python and C++ to deal with XML separately, the C++ and Python XML tools here have been written completely from scratch so both the Python and C++ can be maintained in parallel. The Python and C++ code in the XMLDumper/XMLLoader is remarkably similar for maintenance purposes: any changes in the Python can easily be propagated to the C++ and vice-versa.

Python Tools: XMLDumper

To convert from Python dictionaries to XML, use the XMLdumper. The online documentation is quite good:

```
>>> import xmldumper
>>> help(xmldumper)
```

Let's start with a simple example and convert a simple dict to XML:

```
>>> example = { 'a':1, 'b':2.2, 'c':'three' }
>>> from xmldumper import *
>>> import sys    # for sys.stdout
>>> xd = XMLDumper(sys.stdout)      # dump XML to stdout
>>> xd.XMLDumpKeyValue('top', example)
```

The output:

```
<top><a>1</a><b>2.2</b><c>three</c></top>
```

This is a tad unreadable, but sometimes you may want to compress your XML output all together. Most of the time, though, you will probably want to use the *pretty print* version, which indents to show nesting:

```
>>> xd = XMLDumper(sys.stdout, XML_DUMP_PRETTY)
>>> xd.XMLDumpKeyValue('top', example)
```

The output:

```
<top>
  <a>1</a>
  <b>2.2</b>
  <c>three</c>
</top>
```

Notice the top-level container: this is actually an XML requirement that there be exactly one outer tag (in this case, it is called 'top') containing the content. The 'top' tags surround the input table. If we want, we can just output the value:

```
>>> xd.XMLDumpValue(example)
```

The output:

```
<a>1</a>
<b>2.2</b>
<c>three</c>
```

This isn't legal XML by itself, but it can be part of a larger XML document composed piecewise.

There are actually a number of options for the XMLDumper: each option is ored in. For example, if we want pretty-printed XML and strict XML (with the header):

```
>>> xd = XMLDumper(sys.stdout, XML_DUMP_PRETTY | XML_STRICT_HDR )
>>> xd.XMLDumpKeyValue('top', example)
```

The output:

```
<?xml version="1.0" encoding="UTF-8"?>
<top>
    <a>1</a>
    <b>2.2</b>
    <c>three</c>
</top>
```

In this case, we output the XML header which, strictly speaking, is needed to be a standard conforming XML document. Currently, we only support version 1.0 and UTF-8 (namespaces are coming in a future release).

Attributes and Folding

Attributes are a critical part of any XML document: the XML tools here use a default convention that all keys that start with '_' are to be placed as attributes:

```
>>> a = { "chapter": { '_length': 100, '_pages':200, 'text': 'hello' } }
>>> xd.XMLDumpKeyValue("top", a)
```

The output:

```
<?xml version="1.0" encoding="UTF-8"?>
<top>
    <chapter length="100" pages="200">
        <text>hello</text>
    </chapter>
</top>
```

Notice that the keys '_length' and '_pages' got turned into attributes in the output XML because they started with '_'. This process is called *folding* and allows attributes to be represented simply in key-value structures. If you aren't comfortable with this, consider the following analogy: In UNIX, all files that start with a '.' are treated specially in an 'ls'. In the PicklingTools, all keys that start with '_' are treated specially in XML processing.

You can turn this folding feature off easily enough:

```
>>> xd=XMLDumper(sys.stdout, XML_DUMP_PRETTY | XML_DUMP_PREPEND_KEYS_AS_TAGS)
>>> xd.XMLDumpKeyValue("top", a)
```

The output:

```
<top>
    <chapter>
        <_length>100</_length>
        <_pages>200</_pages>
        <text>hello</text>
    </chapter>
</top>
```

You can also change the prepend_char to be anything you want in the constructor to XMLDumper (see help(xmldumper)).

Many people using XML support the convention that simple data should be in attributes and only structure (lists, dictionaries) should be in tags. The XML_DUMP_SIMPLE_TAGS_AS_ATTRIBUTES option allows you to do just that:

```
>>> xd=XMLDumper(sys.stdout,XML_DUMP_PRETTY|XML_DUMP_SIMPLE_TAGS_AS_ATTRIBUTES)
>>> xd.XMLDumpKeyValue("top", a)
```

The output:

```
<top>
    <chapter length="100" pages="200" text="hello">
    </chapter>
</top>
```

This option allows all simple data to sit in attributes.

All the options for XMLDumper are below. Some of them make more sense when coupled with the XMLLoader (see next section):

```
# Options for dictionaries -> XML
#  If XML attributes are being folded up, then you may
#  want to prepend a special character to distinguish attributes
#  from nested tags: an underscore is the usual default.  If
#  you don't want a prepend char, use XML_DUMP_NO_PREPEND option
XML_PREPEND_CHAR = '_'


# When dumping, by DEFAULT the keys that start with _ become
# attributes (this is called "unfolding").  You may want to keep
# those keys as tags.  Consider:
#
#   { 'top': { '_a':'1', '_b': 2 }}
#
# DEFAULT behavior, this becomes:
#   <top a="1" b="2"></top>       This moves the _names to attributes
#
# But, you may want all _ keys to stay as tags: that's the purpose of this opt
#   <top> <_a>1</_a> <_b>2</b> </top>
XML_DUMP_PREPEND_KEYS_AS_TAGS = 0x100

# Any value that is simple (i.e., contains no nested
# content) will be placed in the attributes bin:
#  For examples:
#    { 'top': { 'x':'1', 'y': 2 }} ->  <top x="1" y="2"></top>
```

```
XML_DUMP_SIMPLE_TAGS_AS_ATTRIBUTES = 0x200

# By default, everything dumps as strings (without quotes), but those things
# that are strings lose their "stringedness", which means
# they can't be "evaled" on the way back in.  This option makes
# Vals that are strings dump with quotes.
XML_DUMP_STRINGS_AS_STRINGS = 0x400

# Like XML_DUMP_STRINGS_AS_STRINGS, but this one ONLY
# dumps strings with quotes if it thinks Eval will return
# something else.  For example in { 's': '123' } : '123' is
# a STRING, not a number.  When evaled with an XMLLoader
# with XML_LOAD_EVAL_CONTENT flag, that will become a number.
XML_DUMP_STRINGS_BEST_GUESS = 0x800

# Show nesting when you dump: like "prettyPrint": basically, it shows
# nesting
XML_DUMP_PRETTY = 0x1000

# Arrays of POD (plain old data: ints, real, complex, etc) can
# dump as huge lists:  By default they just dump with one tag
# and then a list of numbers.  If you set this option, they dump
# as a true XML list (<data>1.0/<data><data>2.0</data> ...)
# which is very expensive, but is easier to use with other
# tools (spreadsheets that support lists, etc.).
XML_DUMP_POD_LIST_AS_XML_LIST = 0x2000

# When dumping an empty tag, what do you want it to be?
# I.e., what is <empty></empty>
# Normally (DEFAULT) this is an empty dictionary 'empty': {}
# If you want that to be empty content, as in an empty string,
# set this option: 'empty': ""
# NOTE: You don't need this option if you are using
# XML_DUMP_STRINGS_AS_STRINGS or XML_DUMP_STRINGS_BEST_GUESS
XML_DUMP_PREFER_EMPTY_STRINGS = 0x4000

# When dumping dictionaries in order, a dict BY DEFAULT prints
# out the keys in sorted/alphabetic order and BY DEFAULT an OrderedDict
# prints out in the OrderedDict order.  The "unnatural" order
# for a dict is to print out in "random" order (but probably slightly
# faster).  The "unnatural" order for an OrderedDict is sorted
# (because normally we use an OrderedDict because we WANTS its
# notion of order)
XML_DUMP_UNNATURAL_ORDER = 0x8000

# Even though illegal XML, allow element names starting with Digits:
# when it does see a starting digit, it turns it into an _digit
# so that it is still legal XML
XML_TAGS_ACCEPTS_DIGITS  = 0x80

# Allows digits as starting XML tags, even though illegal XML.
# This preserves the number as a tag.
XML_DIGITS_AS_TAGS = 0x80000


# When dumping XML, the default is to NOT have the XML header
# <?xml version="1.0">:  Specifying this option will always make that
# the header always precedes all content
XML_STRICT_HDR = 0x10000
```

Python and the XMLLoader

The XMLLoader reads XML and converts it to a Python dictionary: this is the inverse operation of the XMLDumper. (Note this assumes the type of XML we are processing is key-value kind of XML, not document XML).

The online docs are always helpful:

```
>>> import xmlloader
>>> help(xmlloader)
```

Let's start with a simple example. In a file named 'example.xml', we will put the following XML:

```
<top>
  <a>1</a>
  <b>2.2</b>
  <c>three</c>
</top>
```

To process this file and turn it into a dictionary:

```
>>> from xmlloader import *
>>> example = file('example.xml', 'r')
>>> xl = StreamXMLLoader(example, 0)  # 0 = All defaults on options
>>> result = xl.expectXML()
>>> print result
{'top': {'a': '1', 'c': 'three', 'b': '2.2'}}
```

We can match the *pretty print* nature of the original XML using the pretty module (which comes with the PicklingTools):

```
>>> from pretty import pretty
>>> pretty(result)
{
    'top':{
        'a':'1',
        'b':'2.2',
        'c':'three'
    }
}
```

From the previous section, we know that all XML has to have exactly one outermost container: in this case, the 'top' key. Many times, when translating from XML to a dictionary, the outer most container is superfluous. There is a simple option to 'drop' the outer most container:

```
>>> example = file('example.xml', 'r')
>>> xl = StreamXMLLoader(example, XML_LOAD_DROP_TOP_LEVEL)
>>> result = xl.expectXML()
>>> pretty(result)
```

```
{
    'a':'1',
    'b':'2.2',
    'c':'three'
}
```

You might notice that the values above are strings and not integers or floats: the default when turning XML into a dict is to just keep whatever string of content was in the XML as, well, a string. Using the eval function built-in to Python, we can turn these strings into their appropriate values. Or, we can use the XML_LOAD_EVAL_CONTENT option (which uses eval, but is a little bit smarter):

```
>>> example = file('example.xml', 'r')
>>> xl = StreamXMLLoader(example, XML_LOAD_DROP_TOP_LEVEL | XML_LOAD_EVAL_CONTENT)
>>> result = xl.expectXML()
>>> pretty(result)
{
    'a':1,
    'b':2.2,
    'c':'three'
}
```

This brings the keys to real values. Internally, the XMLLoader uses eval (which can be a security problem if you XML from untrusted sources), but is a little bit smarter: it only keeps the result of the eval if the entire output would be consumed in a tag. For example <tag>123 #12</tag> should will stay a string using XML_LOAD_EVAL_CONTENT, even though plain eval would return 123. And this is good! We don't want to lose any content!:

```
>>> xml_text = '<tag>123 #12</tag>'
>>> xl = XMLLoader(xml_text, XML_LOAD_EVAL_CONTENT)
>>> xl.expectXML()
{'tag':'123 #12'}
```

Notice the example above shows the difference between XMLLoader and StreamXMLLoader: the former takes input from a string, the latter takes input from a stream. You might also note that in every example we have created a new XMLLoader: if we didn't, the loader would just read from where we left off in the previous input. Rule of thumb: create a new XMLLoader for each XML document to process.

Attributes and the XMLLoader

There are about 4 ways of dealing with attributes in XML when converting to Python dictionaries.

1. Put them in a special '__attrs__' sub-table: the default
2. Unfold them: use XML_LOAD_UNFOLD_ATTRS
3. Unfold them, but drop the _: use XML_LOAD_NO_PREPEND_CHAR
4. Ignore them: use XML_LOAD_DROP_ALL_ATTRS option

Consider the following XML file (book.xml):

```
<?xml version="1.0" encoding="UTF-8"?>
<top>
    <chapter length="100" pages="200">
       <text>hello</text>
```

```
   </chapter>
</top>
```

We will convert this XML to a dict using the default way of handling attributes: stick the attributes in a special table called '__attrs__':

```
>>> book = file('book.xml','r')
>>> xl = StreamXMLLoader(book, XML_LOAD_DROP_TOP_LEVEL)
>>> result = xl.expectXML()
>>> pretty(result)
{
    'chapter':{
        '__attrs__':{
            'length':'100',
            'pages':'200'
        },
        'text':'hello'
    }
}
```

Just like normal XML, we can turn the strings into real values using XML_LOAD_EVAL_CONTENT:

```
>>> book = file('book.xml','r')
>>> xl = StreamXMLLoader(book, XML_LOAD_DROP_TOP_LEVEL | XML_LOAD_EVAL_CONTENT)
>>> result = xl.expectXML()
>>> pretty(result)
{
    'chapter':{
        '__attrs__':{
            'length':100,
            'pages':200
        },
        'text':'hello'
    }
}
```

This method makes it clear which values are attributes and which values are tags:

```
>>> print result['chapter']['__attrs__']['length']
>>>    # attributes of chapter are under chapter/__attrs__ table
```

Another different way to handle attributes (if you don't like the above) is to use the special character '_' in front of tags to indicate those came from the attributes section. With the XML_LOAD_UNFOLD_ATTRS option, the attributes get *unfolded* into the table of interest, as special keys starting with an '_':

```
>>> book = file('book.xml','r')
>>> xl = StreamXMLLoader(book, XML_LOAD_DROP_TOP_LEVEL |
...                            XML_LOAD_EVAL_CONTENT | XML_LOAD_UNFOLD_ATTRS)
>>> result = xl.expectXML()
>>> pretty(result)
{
    'chapter':{
        '_length':100,
        '_pages':200,
        'text':'hello'
    }
}
```

It's still pretty obvious what keys are attributes:

```
>>> print result['chapter']['_length']
>>>    # The attributes of chapter all start with an _
```

Of course, you can change the prepend character in the constructor of XMLLoader (see help(XMLLoader)), or you can get rid of it altogether with XML_LOAD_NO_PREPEND_CHAR:

```
>>> book = file('book.xml','r')
>>> xl = StreamXMLLoader(book, XML_LOAD_DROP_TOP_LEVEL |
...                            XML_LOAD_EVAL_CONTENT | XML_LOAD_UNFOLD_ATTRS | XML_LOAD_
>>> result = xl.expectXML()
>>> pretty(result)
{
    'chapter':{
        'length':100,
        'pages':200,
        'text':'hello'
    }
}
```

Finally, you can just drop all your attributes using the XML_LOAD_DROP_ALL_ATTRS:

```
>>> book = file('book.xml','r')
>>> xl = StreamXMLLoader(book, XML_LOAD_DROP_TOP_LEVEL |
...                            XML_LOAD_EVAL_CONTENT | XML_LOAD_DROP_ALL_ATTRS )
>>> result = xl.expectXML()
>>> pretty(result)
{
    'chapter':{
        'text':'hello'
    }
}
```

A list of options is available below (or look in xmlloader.py):

```
###################### OPTIONS for XML -> dictionaries

# ATTRS (attributes on XML nodes) by default becomes
# separate dictionaries in the table with a
# "__attrs__" key.  If you choose to unfold, the attributes
# become keys at the same level, with an underscore.
# (thus "unfolding" the attributes to an outer level).
#
# For example:
#   <book attr1="1" attr2="2>contents</book>
# WITHOUT unfolding  (This is the DEFAULT)
#   { 'book' : "contents",
#     '__attrs__' : {'attr1'="1", "attr2"="2"}
#   }
# WITH unfolding:  (Turning XML_LOAD_UNFOLD_ATTRS on)
#   { 'book' : "contents",
#     '_attr1':"1",
#     '_attr2':"2",
#   }
XML_LOAD_UNFOLD_ATTRS = 0x01


# When unfolding, choose to either use the XML_PREPEND character '_'
# or no prepend at all.  This only applies if XML_LOAD_UNFOLD_ATTRS is on.
#   <book attr1="1" attr2="2>contents</book>
# becomes
#  { 'book': "content",
#    'attr1':'1',
#    'attr2':'2'
#  }
# Of course, the problem is you can't differentiate TAGS and ATTRIBUTES
# with this option
XML_LOAD_NO_PREPEND_CHAR = 0x02

# If XML attributes are being folded up, then you may
# want to prepend a special character to distinguish attributes
# from nested tags: an underscore is the usual default.  If
# you don't want a prepend char, use XML_LOAD_NO_PREPEND_CHAR option
XML_PREPEND_CHAR = '_'

# Or, you may choose to simply drop all attributes:
# <book a="1">text<book>
#   becomes
# { 'book':'1' }   # Drop ALL attributes
XML_LOAD_DROP_ALL_ATTRS = 0x04

# By default, we use Dictionaries (as we trying to model
# key-value dictionaries).  Can also use ordered dictionaries
# if you really truly care about the order of the keys from
# the XML
XML_LOAD_USE_OTABS = 0x08

# Sometimes, for key-value translation, somethings don't make sense.
# Normally:
#   <top a="1" b="2">content</top>
# .. this will issue a warning that attributes a and b will be dropped
# because this doesn't translate "well" into a key-value substructure.
#   { 'top':'content' }
#
# If you really want the attributes, you can try to keep the content by setting
# the value below (and this will suppress the warning)
#
#  { 'top': { '__attrs__':{'a':1, 'b':2}, '__content__':'content' } }
#
# It's probably better to rethink your key-value structure, but this
# will allow you to move forward and not lose the attributes
XML_LOAD_TRY_TO_KEEP_ATTRIBUTES_WHEN_NOT_TABLES = 0x10

# Drop the top-level key: the XML spec requires a "containing"
# top-level key.  For example: <top><l>1</l><l>2</l></top>
# becomes { 'top':[1,2] }  (and you need the top-level key to get a
# list) when all you really want is the list:  [1,2].  This simply
# drops the "envelope" that contains the real data.
XML_LOAD_DROP_TOP_LEVEL = 0x20

# Converting from XML to Tables results in almost everything
# being strings:  this option allows us to "try" to guess
# what the real type is by doing an Eval on each member:
# Consider: <top> <a>1</a> <b>1.1</b> <c>'string' </top>
# WITHOUT this option (the default) -> {'top': { 'a':'1','b':'1.1','c':'str'}}
# WITH this option                  -> {'top': { 'a':1, 'b':1.1, 'c':'str' } }
# If the content cannot be evaluated, then content simply says 'as-is'.
# Consider combining this with the XML_DUMP_STRINGS_BEST_GUESS
# if you go back and forth between Tables and XML a lot.
XML_LOAD_EVAL_CONTENT = 0x40

# Even though illegal XML, allow element names starting with Digits:
# when it does see a starting digit, it turns it into an _digit
# so that it is still legal XML
XML_TAGS_ACCEPTS_DIGITS = 0x80

# Allows digits as starting XML tags, even though illegal XML.
# This preserves the number as a tag.
XML_DIGITS_AS_TAGS = 0x80000

# When loading XML, do we require the strict XML header?
# I.e., <?xml version="1.0"?>
# By default, we do not.  If we set this option, we get an error
# thrown if we see XML without a header
XML_STRICT_HDR = 0x10000
```

Lists and the XMLLoader/XMLDumper

Lists present some interesting challenges when converting back and forth. XML supports lists, but Python lists and XML lists are very different beasts in a few areas.

Consider: by default, multiple entries of the same tag in XML form a list:

```
<top>
   <ch>text1</ch>   # list item 1
   <ch>text2</ch>   # list item 2
</top>
```

There is an easy, obvious way to convert this into Python lists, and it works well:

```
>>> xl = XMLLoader("<top><ch>text1</ch><ch>text2</ch></top>")
>>> xl.expectXML()
{ 'top': { 'ch': ['text1', 'text2'] } }
```

What happens, though, if there is only one tag in the XML list?:

```
<top>
  <ch>text1</ch>   # list item?  plain data?
</top>
```

In the absence of extra information, 'ch' becomes a plain string: in XML, the only way to signal a list is with multiple entries, which don't exist here:

```
>>> xl = XMLLoader("<top><ch>text1</ch></top>", 0)
>>> xl.expectXML()
{'top': {'ch': 'text1' }}
```

In a full XML world, where schemas abound, the solution might be to have a schema to enforce this. Unfortunately, the XML tools here assume we *just* have the information of the table itself: there is no extra information. So, we have to make due with what XML gives us. In this case, we use the convention of having a special attribute type__ to indicate that 'ch' is a list:

```
<top>
  <ch type__="list">text1</ch>
</top>
```

Adding this attribute forces the XML translation to keep ch as a list:

```
>>> xl = XMLLoader("<top><ch type__='list'>text1</ch></top>", 0)
>>> xl.expectXML()
{'top': {'ch': ['text1']}}
```

For consistency, you can *always* put it on the first entry to tag an entity as a list:

```
<top>
 <ch type__="list">text1</ch>
 <ch>text2</ch>
</top>
```

And when you translate:

```
>>> xl = XMLLoader("<top><ch type__='list'>text1</ch><ch>text2</ch></top>", 0)
>>> xl.expectXML()
{'top': {'ch': ['text1', 'text2']}}
```

But you don't have to put the type__ tag on if you have multiple entries: in that case the tools can easily figure out if it's a list or not. It's really only if you have a *single lonesome tag* that you need the special type__ to force a list. (but ONLY put in on the first entry: if you put it on all the entries, you won't get what you expect).

How do you represent an empty list? Use an empty tag with the type__ attribute:

```
<top>
  <ch type__="list"/>
</top>
```

Translating, you'll see, yes, 'ch' becomes an empty list:

```
>>> xl = XMLLoader("<top><ch type__='list'/></top>", 0)
>>> xl.expectXML()
{'top': {'ch': []}}
```

There's another way in which XML lists and Python lists differ: all XML lists have to be *named*, whereas Python has the notion of *anonymous* lists and dictionaries. Consider in Python:

```
>>> a = { 'top': [ [1,2,3],['a','b'], {'a':1} ] }
```

The 'top' list contains two anonymous lists and one anonymous dictionary. Basically, because lists can hold anything (including other lists), the content inside the list exists without a name. Python programmers normally think of that as just a['top'][0], a['top'][1], etc. using indices, so they don't care that the inner lists don't have names.

In XML, all tags *HAVE* to have a name. To preserve the XML notion of lists, all entries should have the same name as well (we saw above that XML enforces lists by having repeated tags). This means we can't use 0,1,2, etc. like a Python programmer would. The XML needs a name for the anonymous list: the tools use 'list__':

```
>>> xd = XMLDumper(sys.stdout, XML_DUMP_PRETTY)
>>> a =  { 'top': [ [1,2,3],['a','b'], {'a':1} ] }
>>> xd.XMLDumpKeyValue('a', a)
<a>
    <top>
        <list__>1</list__>
        <list__>2</list__>
        <list__>3</list__>
    </top>
    <top>
        <list__>a</list__>
        <list__>b</list__>
    </top>
    <top>
        <a>1</a>
    </top>
</a>
```

The convention that the toolset uses is that "list__" will be the name XML uses to correspond to the anonymous Python lists. In the case of the dictionary, being inside a list like that makes it "obvious" it's a dictionary, so we don't need any special mechanism for that. To be sure this converts back faithfully:

```
>>> x = """
...   <a>
...       <top>
...           <list__>1</list__>
...           <list__>2</list__>
...           <list__>3</list__>
...       </top>
...       <top>
...           <list__>a</list__>
...           <list__>b</list__>
...       </top>
...       <top>
...           <a>1</a>
...       </top>
...   </a>
... """
>>> xl = XMLLoader(x, 0)
>>> xl.expectXML()
{'a': {'top': [['1', '2', '3'], ['a', 'b'], {'a': '1'}]}}
```

There is one final corner case:

```
>>> a= { 'top': [ {} ] }
```

An empty dictionary inside a list (so the dictionary is anonymous). Any keys in a dict usually offer enough information for the XML tools to figure out that it's a dictionary. In this case, since there are no keys, we need a special key to indicate an anonymous dictionary:

```
>>> xd.XMLDumpKeyValue('a', {'top': [ {} ] } )
<a>
    <top type__="list">
        <dict__>
        </dict__>
    </top>
</a>
```

In fact, you can use 'dict__' to name an anonymous dictionary, and the tools will do the right thing:

```
>>> x = """
... <a>
...       <top type__="list">
...           <dict__>
...           </dict__>
...       </top>
...   </a>
... """
>>> xl = XMLLoader(x, 0)
>>> xl.expectXML()
{'a': {'top': [{}]}}
```

In fact, adding the 'dict__' in the anonymous list works just fine, it's just clumsier:

```
>>> x = """
... <a>
...   <top>
...     <dict__><a>1</a></dict__>
...   </top>
...   <top>
...     <dict__/>
...   </top>
... </a>
... """
>>> xl = XMLLoader(x, 0)
>>> xl.expectXML()
{'a': {'top': [{'a': '1'}, {}]}}
```

Array Disposition

The ArrayDisposition parameter of the XMLLoader and XMLDumper is often misunderstood. Basically, the array disposition tells the XMLLoader/XMLDumper how to deal with arrays of POD: POD stands for Plain Old Data, meaning data like ints, floats, complexes (In hard core C, POD is quick and easy to manipulate). POD arrays are very important for efficient processing of lots of scientific data as they are are stored efficiently as contiguous data in memory.

Python lists *are not* POD arrays: lists have to deal with heterogeneous data (i.e., [1, 2.2, 'three']) and thus aren't as efficient for storing large amounts of data. In a crunch, however, POD arrays can be stored as Python lists.

There are five different array dispositions (number 4 is new to PicklingTools 1.3.0):

1. ARRAYDISPOSITION_AS_NUMERIC : Assume all array data is using the Python Numeric module which keeps arrays of POD:

   ```
   >>> import Numeric
   >>> a = Numeric.array([1,2,3], 'f') # POD array of floats
   ```

   The Python Numeric module may or may not be installed on your platform: many versions of RedHat Linux allow an RPM to be installed. If you use older XMPY (pre 4.0), Numeric is installed by default. Newer XMPY should use NumPy (see next bullet).

   As of 1.5.x series, Numeric is out of maintenance and we tend to prefer NumPy.

2. ARRAYDISPOSITION_AS_NUMPY : Assume all array data is using the Python NumPy module which keeps arrays of POD:

   ```
   >>> import numpy
   >>> a = numpy.array([1,2,3], 'f') # POD array of floats
   ```

   The Python numpy module may or may not be installed on your platform: many versions of RedHat Linux allow an RPM to be installed. NumPy really only works with XMPY greater than version 4.0.

3. ARRAYDISPOSITION_AS_LIST: Turn all array POD data into a list. This is the most inefficient way to store POD arrays, but it is the most compatible, as all versions of Python support the Python list:

   ```
   >>> l = [1,2,3]  # Not stored as anything special: uses
   >>>              # overhead of lists which is not the most
   >>>              # efficient way to store lots of POD
   ```

4. ARRAYDISPOSITION_AS_ARRAY: Most versions of Python have the array module:

   ```
   >>> import array
   >>> a = array.array('f', [1,2,3]) # POD array of floats
   >>>                               # NOTE! Python arrays
   >>>                               # have different interfaces
   >>>                               # than Numeric arrays
   ```

   The Python array module doesn't support complex data, but is available on almost Pythons as a default module.

5. ARRAYDISPOSITION_AS_NUMERIC_WRAPPER: This is new to simple array class which wraps the Python array, but retains the interface of the Numeric array and also supports complex data.

   ```
   >>> from simplearray import SimpleArray as array
   >>> a = array([1,2,3], 'D')   # POD array of complex doubles:
   >>>                           # works like Numeric arrays but
   >>>                           # just a simple Python class
   >>>                           # that wraps Python array module
   ```

In general, the array disposition indicate what the XMLLoader/XMLDumper will try to do with arrays of POD. The XMLLoader and XMLDumper do slightly different things based on the array disposition.

1. In the XMLDumper case, if the Python dictionary contains any POD data (Numeric array, NumPy array, Python array, or Numeric array wrapper), it will dump it as Numeric data UNLESS the array disposition is AS_LIST, in which case it will dump it as a list. Here's an example using the Numeric wrapper array, and how it dumps:

```
>>> from simplearray import SimpleArray as array
>>> e = { 'data': array([1,2,3], 'D'), 'time':'12:00' }
>>>
>>> import sys
>>> from xmldumper import *
>>> xd = XMLDumper(sys.stdout, e, XML_DUMP_PRETTY,
...                ARRAYDISPOSITION_AS_NUMERIC_WRAPPER)
>>> xd.XMLDumpKeyValue('top', e)
```

   The output:

```
<top>
    <data arraytype__="D">(1+0j),(2+0j),(3+0j)</data>
    <time>12:00</time>
</top>
```

   The POD array dumps a long list of comma-separated values (CSV), with an attribute indicating what the original type of the data was. But, if we use the default array disposition (which is ARRAYDISPOSITION_AS_LIST), that array will be turned into a plain list, and all POD array information will be lost (including the type tag):

```
>>> xd = XMLDumper(sys.stdout, XML_DUMP_PRETTY)
>>>                             # Default is ARRAYDISPOSITION_AS_LIST)
>>> xd.XMLDumpKeyValue('top', e)
```

   The output is:

```
<top>
    <data>(1+0j)</data>
    <data>(2+0j)</data>
    <data>(3+0j)</data>
    <time>12:00</time>
</top>
```

   Although we lose some information with this array disposition, this format is very compatible with many XML tools (as the notion of XML lists is well understood by those tools).

2. From the XMLLoader's point of view: it will only try to convert POD arrays if it actually encounters POD arrays! If the XML has only plain XML lists, the array disposition doesn't matter:

```
<!-- XMLLoader won't care what the array disposition is for this
     data, because all key values are standard lists: there is
     no special tags or anything indicating otherwise --!>
<top>
    <data>(1+0j)</data>
    <data>(2+0j)</data>
    <data>(3+0j)</data>
    <time>12:00</time>
</top>

<!---- XMLLoader *will care* because it sees the special
       arraytype__ tag below, so it knows that array POD --!>
<top>
    <data arraytype__="D">(1+0j),(2+0j),(3+0j)</data>
    <time>12:00</time>
</top>
```

When loading, the XMLLoader will follow the array disposition: all array POD data will be converted to either a Numeric array, Python array, Numeric wrapper array or plain Python list. As an example for Numeric Wrapper array:

```
>>> x = """
... <top>
...     <data arraytype__="D">(1+0j),(2+0j),(3+0j)</data>
...     <time>12:00</time>
... </top>
... """
>>> from xmlloader import *
>>> xl = XMLLoader(x, 0, ARRAYDISPOSITION_AS_NUMERIC_WRAPPER)
>>> xl.expectXML()
{'top': {'data': array([(1+0j),(2+0j),(3+0j)], 'D'), 'time': '12:00'}}
```

In the above example, the POD array was preserved, because we set the array disposition to load all POD arrays using the Numeric wrapper (simplearray.py). If we don't specify an array disposition, it uses the ARRAYDISPOSITION_AS_LIST as the default:

```
>>> x = """
... <top>
...    <data arraytype__="D">(1+0j),(2+0j),(3+0j)</data>
...    <time>12:00</time>
... </top>
... """
>>> from xmlloader import *
>>> xl = XMLLoader(x, 0) # Default is ARRAYDISPOSITION_AS_LIST
>>> xl.expectXML()
{'top': {'data': [(1+0j), (2+0j), (3+0j)], 'time': '12:00'}}
```

In the above case, the array was converted to a Python list: it preserves the data, but not the original type of the POD data (was the data complex float or complex double?) or the fact that is was a POD array.

The real reason for ArrayDisposition is because Python doesn't have a "good" standard POD array:

1. Numeric arrays are a standard at many places of work, but they aren't installed as standard
2. Python array don't handle complex data, which is a non-starter for some kinds of processing. They also don't do array operations (vector add, multiply, etc.) AND the serialization for Python arrays has changed between 2.6 and 2.7, so they are incompatibile between Pythons.
3. The Numeric wrapper handles complex, but still doesn't have the vector operations
4. Python lists are standard, but are a poor way to store large amounts of data

5. The NumPy package is fairly standard on most modern distributions, but you may still have to go out of your way to get it. In general NumPy is the best choice, as NumPy is in maintenance and fairly standard across multiple platforms.

C++ doesn't have this problem: the Array<T> is the standard way to deal with POD arrays. AS_NUMERIC and AS_PYTHON_ARRAY are handled the same (using the C++ Array<T> class). The AS_LIST is offered as a compatibility option and will convert POD arrays to the equivalent Python List (the C++ Arr()).

In XML, the 'arraytype' tags are the following:

```
s: 1 byte signed char
S: 1 byte unsigned char
i: 2 byte signed char
I: 2 byte unsigned char
l: 4 byte signed char
L: 4 byte unsigned char
x: 4 byte signed char
X: 4 byte unsigned char
f: 4 byte float
d: 8 byte double
F: 8 byte complex (2 4-byte floats)
D: 16 byte complex (2 8-byte doubles)
```

Thus:

```
<s arraytype__='S'>1,2,3</s>
```

Is an array of unsigned 1 byte integers. These typetags correspond to the C++ Val type tags.

When building arrays from Python, look at the Numeric typecodes from Numeric (print Numeric.typecodes) or array (help array).

Note that from a Python dictionary perspective, the arrays are usually printed as Numeric arrays. Their typecodes:

```
>>> import Numeric
>>> print Numeric.typecodes
{'Integer': '1sil', 'UnsignedInteger': 'bwu', 'Float': 'fd', 'Character': 'c', 'Complex'
```

Why didn't we use Numeric typecodes as the standard for the XML type tags? Unfortunately, the Numeric typecodes don't have a 8-byte unsigned integer, and depending on the type of machine (32-bit or 64-bit), the 'l' typecode may be (resp.) a 4-byte integer or an 8-byte integer. The Numeric typecodes are unfortunately inconsistent. The Python array typecodes have similar problems (the tags aren't guaranteed to be a x-bytes, and there are no complex typecodes). The Val typecodes (listed above) are always guaranteed to be exact number of bytes and they support complex data.

Back and Forth Between XML and Python Dictionaries

The XMLLoader and XMLDumper have been written in such a way to allow you to convert back and forth between XML and Python dictionaries and not lose information (if your XML is key-value XML). The previous sections made it look easy, but there are a lot of places where it be tricky.

1. lists: Lists can be problematic (see above discussion), but the solutions to those problems are outlined in the previous section.
2. floating point numbers: With any floating point data, the number of places you print can be important. The 'default' printing of Python is usually not good enough, so the *pretty* module has gone to great lengths (and the XMLDumper and XMLLoader both import the *pretty*

module) to make sure that floating point numbers are handled responsibly: floats are printed with 7 places, doubles are printed with 16 places. Both the C++ and Python versions of pretty should behave exactly the same way.

3. array disposition: When going back and forth between dictionaries and XML, be judicious with the array disposition, or you may lose information. See the previous section for more details on POD data.
4. dropping the top-level: In XML, the top-level document is frequently irrelevant. Converting back and forth between XML and dicts you will probably want to drop the top-level.

There are two tools for converting between XML and Python dictionaries from the command line: they are xml2dict.py and the dict2xml.py (or in the C++ area, xml2dict and dict2xml: the C++ version is significantly faster (60x), but will have to be compiled. As usual, the C++ and Python interfaces are exactly the same). Sample usage:

```
# From a UNIX prompt
% cd /fullpath/to/PicklingTools130/Python
% cat INPUT.XML

<?xml version="1.0" encoding="UTF-8"?>
<root>
  <list__>
    <data arraytype__="d">100.0,200.0</data>
    <data arraytype__="D">(100-100j),(600+1j)</data>
    <here>1</here>
  </list__>
  <list__>7</list__>
  <list__>(1-2j)</list__>
</root>

% python xml2dict.py INPUT.XML

[
  {
      'data':[
          array([100.0,200.0], 'd'),
          array([(100-100j),(600+1j)], 'D')
      ],
      'here':1
  },
  7,
  (1-2j)
]
```

The options on there were chosen to try to make it easy to go back and forth between the two representations without losing any info.

C++ and the XMLLoader and XMLDumper

The C++ version is remarkable similar to the Python version: in almost all respects, their behaviors, their interfaces, their options, and even their names should be exactly alike. Even though we said this earlier in this document, it is worth saying again: there has been considerable effort to make the C++ and Python versions of the XMLLoader and XMLDumper to be as close to the same as possible (for ease of maintenance). Thus, the Python and C++ should be almost interchangeable.

The C++ version is significantly faster (60x), but the Python will be easier to use.

Some notable differences: the C++ version deals with Vals instead of Python objects, C++ uses OTabs instead of OrderedDict.

Consider the following simple C++ example for XMLDumper:

```
// Includes needed
#include <iostream>
#include "xmldumper.h"

// C++ code
int main()
{
  Val v = Tab("{'a':1, 'b':2.2, 'c':'three'}");
  XMLDumper xd(std::cout, XML_DUMP_PRETTY | XML_STRICT_HDR);
  xd.XMLDumpKeyValue("top", v);
}

/* Output:
<top>
   <a>1</a>
   <b>2.2</b>
   <c>three</c>
</top>
*/
```

Like the XMLDumper of Python, the options are specified the same and the behavior and interfaces are essentially the same. Instead of using "sys.stdout", we use C++ stream std::cout.

There are multiple examples in the baseline of using this the XMLLoader and XMLDumper: take a look at xml2dict.cc and dict2xml.cc and the Makefiles to see examples of how to compile and use C++.

Different Types of Keys Of Dictionaries

Most of the discussions above assume the keys of dictionaries are strings. They can be other types:

1. **numbers:**

   If a key is an int, it will be converted to a tag with "0" around it. Unfortunately, an XML tag of <0> is not legal. By default, the tools will error out saying that keys with only numbers would be illegal XML. There are two options which help mitigate this: XML_TAGS_ACCEPTS_DIGITS and XML_DIGITS_AS_TAGS (yes, they are closely named). The first option (XML_TAGS_ACCEPTS_DIGITS) turns a dict key that starts with digits into an _digit. In other words, `0` would become _`0`. This gives legal XML, but doesn't translate back from XML to dict well. The second option (XML_DIGITS_AS_TAGS) allows "illegal" XML where `0` becomes the tag <0>: on other words, the numberness is preserved, even though the underlying XML is strictly speaking illegal.

   For conversions between Midas 2k OpalTables and XML, we suggest using the XML_DIGITS_AS_TAGS conversion.

2. **tuples and other types:**

   Right now, having a tuple as a key in a dict will cause "undefined" translations. As The Python will likely error out, and the C++ will try to convert the tuple to a string with varying degrees of success.

In general, we suggest keeping keys as strings to keep the XML translations clean and well-defined.

Python C-Extension Module: New In PicklingTools 1.4.1

The XMLtools for Python (`xmldumper.py` and `xmlloader.py`) was originally written all in raw Python: this was on purpose it would be easy to include the Python "as-is" without any special build process: *import xmldumper* and you are ready to go. The C++ routines, however, are significantly faster than the Python routines (character based I/O is usually much faster in C/C++ than Python). Take a look at the output of `xmltimingstest.py`:

```
% python xmltimingschecks.py
Time to create big table 0.00489687919617
Time to deepcopy big table 0.0030460357666
...time to convert PyObject to Val ... 0.00101280212402
...time to convert PyObject to Val and back... 0.00146102905273
Time for Python XMLDumper to dump an XML file 0.0433909893036
Time for C XMLDumper to dump an XML file: 0.00465703010559
 -----------
 *Warning: This version of Python doesn't support ast.literal_eval, so XML_LOAD_EVAL_CON
Time for Python Loader to load an XML file 0.61283493042
Time for C Ext Loader to load an XML file 0.00649094581604
```

From these numbers, it's easy to see that dict to XML C++ conversion routines are about an order of magnitude faster than their Python equivalents, and the XML to dict C++ conversion routines are about two orders of magnitude faster than their Python equivalents. Although it is nice to have raw Python solution to convert between dicts and XML (for smaller tables), for any larger tables, the extra speed of C++ may be essential.

The guts of the C++ converters are available in the Python C-Extension module `pyobjconvert`, but the real interface most Python users will use is `cxmltools`.

Building the pyobjconvert Python C-Extension Module

The README describes how to create the pyobjconvert module, which has the XML conversion wrappers:

```
CReadXMLFromStream, CReadXMLFromString, CReadXMLFromFile
CWriteXMLToStream, CWriteXMLToString, CWriteXMLToFile
```

This C extension module for Python increases the speed of the XML to dict conversion by 60x-100x and the dict to XML conversion by 6-10x.

The user probably doesn't want to use the pyobjconvert module directly (as it has a different API than the previous xmltools.py): instead, the user will *import cxmltools* which brings all the appropriate definitions in and the interfaces/names are converted to interfaces that are consistent with the xmldumper.py and xmlloader.py.

HOW TO BUILD

1. Check 'setup.py':

   Make sure it includes the paths to the code in *PicklingTools141/C++* and *PicklingTools141/C++/opencontainers1_7_5/include* (of course, the version numbers may change in later releases).

   By default, this should work, but these are relative paths from the PicklingTools main directory. You may want to move those directories.

2. Once you are sure those are correctly set-up, type:

```
% python setup.py build   # % is the UNIX prompt
```

This starts the build process and builds the C extension module for you. You should see something like this (and notice that it creates three sub-directories):

```
creating build
creating build/temp.linux-x86_64-2.4
gcc -pthread -fno-strict-aliasing -DNDEBUG -O2 -g -pipe -Wall -Wp,-D_FORTIFY_SOUR
gcc -pthread -fno-strict-aliasing -DNDEBUG -O2 -g -pipe -Wall -Wp,-D_FORTIFY_SOUR
creating build/lib.linux-x86_64-2.4
c++ -pthread -shared build/temp.linux-x86_64-2.4/pyobjconvertmodule.o build/temp.
```

3. Underneath PythonCExt should be three subdirs with names "something" like below:

```
build
build/temp.linux-x86_64-2.4
build/lib.linux-x86_64-2.4
```

Take a look at the *build/lib.linux-x86_64-2.4* dir: under there should be a `pyobjconvert.so` file. This is the file that contains your library.

Note these names aren't likely to be the same on your installation. The 'x86' means the machine is a 64-bit installation: yours may be a 32-bit installation and would be 'i686'. The '2-4' means this is for Python 2.4; you are probably using a newer version of Python like 2.6 or 2.7.

Use the appropriate names for your system.

4. Set your PYTHONPATH so it picks up the .so when you import:

```
% setenv PYTHONPATH "/full/path/to/PicklingTools141/PythonCExt/build/lib.linux-x8

% python
>>> import pyobjconvert  # without PYTHONPATH, it probably won't find your .so
>>> dir(pyobjconvert)
['CReadFromXMLFile', 'CReadFromXMLStream', 'CReadFromXMLString', 'CWriteToXMLFile
```

5. Try it out! An easy way to see if it works is to run the xmltimingtools.py script which shows the relative times of xmldumper vs. C XMLDumper, etc:

```
% python xmltimingschecks.py
Time to create big table 0.00489687919617
Time to deepcopy big table 0.0030460357666
...time to convert PyObject to Val ... 0.00101280212402
...time to convert PyObject to Val and back... 0.00146102905273
Time for Python XMLDumper to dump an XML file 0.0433909893036
Time for C XMLDumper to dump an XML file: 0.00465703010559
 -----------
 *Warning: This version of Python doesn't support ast.literal_eval, so XML_LOAD_E
Time for Python Loader to load an XML file 0.61283493042
Time for C Ext Loader to load an XML file 0.00649094581604
```

Note the C XMLDumper is about 10x faster than the Python version and the C XMLLoader is about 100x faster than the Python version

Surprisingly, converting from PyObject->Val->PyObject (which accomplishes a deep copy) is faster than the Python deepcopy (!)

6. To use the C version of the XML tools, try using `cxmltools`

The `cxmltools` requires the 'PicklingTools141/Python' to be on the Python path along with the extension module:

```
% setenv PYTHONPATH "${PYTHONPATH}:/full/path/PicklingTools141/Python"
% python
>>> from cxmltools import *   # make sure cxmltools.py is on PYTHONPATH
...                           # as is

>>> d = {'a':1, 'b':2 }
>>>
>>> a = WriteToXMLFile(d, 'top')   # Don't forget the "top"
>>> print a
<?xml version="1.0" encoding="UTF-8"?>
<top>
  <a>1</a>
  <b>2</b>
</top>

>>> res = ReadFromXMLString(a)
>>> print res
{'a': 1, 'b': 2}
```

In case the `cxmltools` aren't built, you can *import xmltools* and get the same behavior as above, just not as fast!

Note that the `cxmltools` does NOT have everything the `xmltools` has: it only has the simplified wrappers (listed below). These are the same simplified wrappers that the xmldumper/xmlloader have as well. Really, the only thing you *don't* have are the classes (XMLDumper/XMLLoader) that implements the conversions: all the functionality is still available, but through the easier-to-use wrappers.

For converting from XML to dict, the simplified wrappers are:

```
ReadFromXMLFile(filename, options, array_disp, prepend_char);
ReadFromXMLStream(stream, options, array_disp, prepend_char);
ReadXMLString(xml_string, options, array_disp, prepend_char);

defaults:
     options=XML_STRICT_HDR | XML_LOAD_DROP_TOP_LEVEL | XML_LOAD_EVAL_CONTENT
     array_disp=AS_NUMERIC
     prepend_char=XML_PREPEND_CHAR
```

For converting from dict to XML, the simplified wrappers are:

```
WriteToXMLFile(dict_to_convert, filename, top_level_key, options,
               array_disp, prepend_char)
WriteToXMLStream(dict_to_convert, stream, top_level_key, options,
                 array_disp, prepend_char)
WriteToXMLString(dict_to_convert, top_level_key, options,
                 array_disp, prepend_char)

defaults:
     top_level_key=None   (this should probably always be "top" instead)
     options=XML_DUMP_PRETTY | XML_STRICT_HDR | XML_DUMP_STRINGS_BEST_GUESS
     array_disp=AS_NUMERIC
     prepend_char=XML_PREPEND_CHAR
```

Note that the `WriteToXMLString` returns the string of interest, whereas the WriteToXMLFile/Stream instead both write to the given entity (and return None).

Conclusion

There are two sets of Python conversion routines: *xmltools.py* and *cxmltools.py*. The former allows a raw Python solution so you can get going immediately. The latter requires more work to build and use correctly, but gives routines that are significantly faster.

The XML Tools described herein have a particular job: allow dictionaries and XML to be used relatively interchangeably. When there is not a clear translation, it's probably worth stepping back and re-evaluating if XML or dict is the better solution. Some of the questionable conversions:

1. Does the XML have content and tags freely interspersed? These don't map well to dicts and maybe the user is using the "document" part of XML heavily.
2. Do dicts have complex keys (like tuples)? These don't map well to XML, and the user is maybe relying on the complex nature of dicts within Python.

If, on the other hand, your XML or dicts are used strictly for key-value type relationships (with a straight-forward use of attributes in XML), then these conversions make sense and may solve the problem for you. These tools have been written with a particular usage in mind and hopefully fit your bill.

Appendix A:

What features of XML we support:

1. hex escape sequences: Sequences such as &#x2A; are supported as of 1.3.1
2. DTDs: Usually, a DTD has a <!SOMETHING ... > format: we don't enforce or use the DTD at all, but it can be read---it is simply ignored. 1.3.0 couldn't recognize DTDs and 1.3.1 recognizes but ignores them.
3. namespaces: Neither 1.3.0 nor 1.3.1 recognize namespaces. 1.3.1 can at least parse XML with namespaces (by recognizing the : in names), but there is no support beyond that. A later release will embrace namespaces fully.
4. comments: XML comments can be interspersed in more places: they reduce to nothing. A future release may try to preserve the comment in a Python #
5. version: only 1.0
6. encoding: BUG: We specify UTF-8, but currently only works with ASCII. This won't be a problem unless you use any non-ASCII chars.