sox: Streaming Output of XML for Python

A low-memory-usage XML generator

John W. Shipman
2012-06-29 11:54

Abstract

Describes a module allowing programs in the Python programming language to generate large XML files without large amounts of processor memory; includes its implementation as a literate program.

This publication is available in Web form¹ and also as a PDF document². Please forward any comments to tcc-doc@nmt.edu.

This work is licensed under a Creative Commons Attribution-NonCommercial 3.0 Unported License.

Table of Contents

1. When you need to generate large XML documents .............................................................. 2
2. Online files ........................................................................................................................... 3
3. Design rationale .................................................................................................................. 3
   3.1. Workflow for XML generation ...................................................................................... 3
   3.2. Design refinements: convenience features ................................................................. 4
   3.3. Context managers and the “with” construct .............................................................. 6
4. Interface to the sox module ............................................................................................... 7
   4.1. The SoxError class ....................................................................................................... 7
   4.2. The Sox class ................................................................................................................ 7
   4.3. The Elt class: An element in progress ....................................................................... 9
   4.4. Name validation ......................................................................................................... 9
5. sox.py: The code prologue ............................................................................................... 10
6. Imported modules .............................................................................................................. 10
7. Manifest constants ........................................................................................................... 10
   7.1. DEFAULT_ENCODING ............................................................................................... 10
   7.2. NAME_START_RANGES ......................................................................................... 10
   7.3. NAME_CHAR_RANGES ............................................................................................ 11
   7.4. ESCAPE_MAP ......................................................................................................... 11
8. Specification functions ..................................................................................................... 12
   8.1. content-processor ...................................................................................................... 12
   8.2. escaped-attribute ....................................................................................................... 12
   8.3. content-list ............................................................................................................... 12
   8.4. valid-comment .......................................................................................................... 13
   8.5. xml-name .................................................................................................................. 13

¹ http://www.nmt.edu/tcc/projects/sox/
³ http://creativecommons.org/licenses/by-nc/3.0/
1. When you need to generate large XML documents

This document describes a module for the Python programming language\(^4\) that helps in the generation of large XML documents without regard to processor memory limitations.

Sometimes, as when formatting a database query for Web presentation as XHTML, or generating an XSL-FO file destined for a PDF rendering, it is necessary to generate quite large XML documents. The author’s preferred techniques for reading and writing XML from Python are described in Python XML processing with \textit{lxml}\(^5\). The \texttt{etbuilder} module presented in that document is especially handy for XML generation, allowing generation of arbitrary XML structures with compact code. However, these methods require that the entire XML document reside as a tree in memory.

Hence, what the author needed was a module that generates XML sequentially, allowing the generation of quite large documents without significant memory requirements.

\(^4\)\url{http://www.python.org/}
\(^5\)\url{http://www.nmt.edu/tcc/help/pubs/pylxml/}
• Section 3, “Design rationale” (p. 3) discusses why the module works as it does.
• Section 4, “Interface to the sox module” (p. 7) describes how to use the module.
• Section 5, “sox.py: The code prologue” (p. 10) presents the code in the form of a lightweight literate
program.  

2. Online files

Files pertaining to this project:

• sox.zip: Zip archive of the sox/ directory.
• sox.py: The module containing the Sox and other classes.
• soxtest: A small example and test driver.
• sox.xml: Source for this document as DocBook-XML 4.3.

3. Design rationale

Conceptually, it is easy to generate well-formed XML: Every start tag must have a corresponding end
tag, and inner elements must end before the outer element's end tag.

To minimize our memory usage, we can write the start tag as soon as we start an element, and write
the content of the element as we go. However, to insure that the XML is well-formed, the sox.py
module will check that start tags and end tags match.

The obvious data structure to manage this checking is a stack.

• The stack is initially empty.
• When an element is started, the start tag is written immediately, and the element name is pushed
onto the stack.
• When an element's end tag is written, the top element of the stack is popped. Its name should match
the name of the end tag just written.
• If the stack is empty at the end, then the start and end tags are balanced.

The worst-case memory usage will occur when the file is deeply nested. The author can't imagine an
XML application that needs more than a few dozen levels of element nesting.

3.1. Workflow for XML generation

Here is the basic workflow for programs that use this interface:

1. Create an instance of class Sox and tell it where the output is going. Let's call this instance s.

2. To start an element, use the s.start(N, ...) method, where N is the tag name, e.g.,
“s.start("html", ...)”. The element's start tag is written to the output right away.

You can add various other arguments to this method call to specify attribute values and text content.
For example, this call:

    s.start("p", "Some text", id="x32")

---

6 http://www.nmt.edu/~shipman/soft/litprog/
7 http://www.nmt.edu/tcc/projects/sox/sox.zip
8 http://www.nmt.edu/tcc/projects/sox/sox.py
9 http://www.nmt.edu/tcc/projects/sox/soxtest
10 http://www.nmt.edu/tcc/projects/sox/sox.xml
11 http://www.nmt.edu/tcc/help/pubs/docbook43/
12 http://en.wikipedia.org/wiki/XML#Well-formedness_and_error-handling
would generate this output:

```xml
<p id="x32">Some text</p>
```

The `.start()` method will return a token, an instance of class `Elt`, that you will use later to generate the element's end tag. Let's call this instance `elt`. The `s.start()` method also pushes `elt` on its internal stack to use later to check for start/end tag balancing.

3. To add text content to this element, use a call to `s.write(...)` with string values. Those values will be written to the output right away.

4. Any calls to the `s.start()` method that you make before an outer element is finished will generate child elements of that outer element. All child elements must be completed before the parent element's end tag is written.

5. To finish an element, call the `elt.end()` method of the `elt` token that was returned by `s.start()`.

   Assuming that you have completed the generation of any child elements, `elt` should also be on top of the `sox` instance's internal stack, and that value will be popped off the stack. Then this method will write the element's end tag.

There is also a `s.leaf(N, ...)` method for generating empty elements and elements with no element children. It does not return a value, but immediately writes the element.

In order to insure well-formed XML output, the `sox` module will perform several other validity checks.

- Element names and attribute names must conform to XML naming rules. The first character must be a letter, underbar, or `:"`, or any of a number of Unicode character ranges specified in the XML standard. Additional characters can be any of those plus digits, `"-", "\"`, and several other Unicode character ranges.
- In attribute values, the characters `"&", "\<", and "\"` (double-quote) will be escaped in output as entities `"&lt;", "&amp;", and "&quot;`, respectively.

### 3.2. Design refinements: convenience features

Before we discuss the features of the `sox` module, let us look at another useful XML generation technology and see what features we can borrow from it.

The author has used the `etbuilder` module, described in *Python XML processing with lxml* quite extensively in a wide variety of XML generation applications. This module includes a multi-purpose element factory `E` that greatly streamlines XML generation. The `etbuilder` module was in turn based on the *An ElementTree Builder* by Fredrik Lundh.

In particular, calls to the `E()` factory allow an arbitrary number of positional and keyword arguments that are processed by type.

- To add text content within the element, supply that text as a positional argument of type `str`.
- To specify element attributes, use a positional argument of type `dict`. The key must be a valid XML attribute name, and the related value must be a valid XML attribute value.

   If you want to generate an attribute whose name is a valid Python identifier, you can also specify it as a keyword argument, e.g., `"id='p04'`.

- You can even include calls to `E()` inside calls to `E()`, to create child elements.

---

13 http://www.w3.org/TR/xml/
14 http://www.nmt.edu/tcc/help/pubs/pylxml/
15 http://effbot.org/zone/element-builder.htm
Here is a small, complete script that demonstrates how terse XHTML generation can be.

```python
#!/usr/bin/env python
import sys
from etbuilder import E, et

root = E.html(
    E.head(
        E.title("Page title here"),
    ),
    E.body(
        E.h1("Main title here"),
        E.hr(),
        E.p({'class': 'note'}, "Some text and some more text", id='p001'))
)

doc = et.ElementTree(root)
doc.write(sys.stdout, pretty_print=True)
```

And here is the output of that script:

```html
<html>
  <head>
    <title>Page title here</title>
  </head>
  <body>
    <h1>Main title here</h1>
    <hr/>
    <p id="p001" class="note">Some text and some more text</p>
  </body>
</html>
```

These are all good ideas, but we cannot implement all of them in the sox module.

We can't implement nested element calls as in E(). Because all Python arguments are evaluated before the call, we would have to save up all the child content until after the start tag gets written. That defeats our low-memory goal.

We can borrow the other two techniques, though.

1. String-type positional arguments will be added as child content after the start tag. All arguments must be Unicode, or strings convertible to Unicode with UTF-8 encoding. (Support for other encodings would be easy to add, if requested."

   Some other types (int, bool) will be run through `unicode()` and then treated as string content.

2. Dictionary arguments and keyword arguments will become attributes within the start tag, assuming that their names are valid XML names.

So here is our sox conversion of the above etbuilder example.

```python
import sys
import sox
s = sox.Sox(sys.stdout)
html = s.start("html")
head = s.start("head")
title = s.start("title", "Page title here")
title.end()
body = s.start("body")
```
3.3. Context managers and the “with” construct

Starting in Python 2.5, Python adds a “with” statement feature, and the concept of a context manager. The sox module supports this feature, and its use can simplify and clarify the structure of the Python code and its generated XML.

A context manager is any instance of a class that defines two special methods, .__enter__ and .__exit__, that do initialization and cleanup operation respectively.

Here is the general form of the with statement, at its simplest:

```
with E [as V]: suite
```

Execution of this construct proceeds like this.

1. Expression E is evaluated. It must produce a context manager.
2. The context manager’s .__enter__() method is called with no arguments (except self).
   
   If the expression is followed by “as”, the name V is bound to whatever value was returned by the .__enter__() method.
3. The suite (containing one or more statements) is executed.
4. The context manager’s .__exit__ method is called with argument list (self, exc_type, exc_value, traceback).

   If the suite completed without raising an exception, the last three arguments will all be None.

   If the execution of the suite raised an exception, the last three arguments will be: the exception type; the exception value; and a stack traceback. Consult the The Python Language Reference[^16] for the details of the exception case.

In general, you can have several clauses in a with statement separated by commas, so for example this is quite legal:

```
with c1() as c, xyz(15), qrz("Jack") as w5dau:
    f(c, w5dau, ",(5,9)"
```

The above example is equivalent to:

```
with c1() as c:
    with xyz(15):
        with qrz("Jack") as w5dau:
            f(c, w5dau, "(5,9)"
```

Since the purpose of a context manager is to ensure that cleanup operations are performed, and since the problem of XML stream generation is to ensure that tags are closed in the proper order, we can take

[^16]: http://docs.python.org/reference/

[^60]: Streaming Output of XML from Python

New Mexico Tech Computer Center
advantage of this construct by making the `Elt` class a context manager, whose `__exit__` method generates the element's closing tag.

Here, then, is a rewrite of our simple XHTML generation example using context managers.

```python
import sys
import sox
s = sox.Sox(sys.stdout)

with s.start("html"):
    with s.start("head"):
        s.leaf("title","Page title here"):
    with s.start("body"):
        s.leaf("h1","Main title here"):
        s.leaf("hr")
        s.leaf("p", {'class': 'note'}, "Some text", " and some more text", id='p001'):
```

### 4. Interface to the sox module

- Section 4.1, “The SoxError class” (p. 7): For exceptions generated by this module.
- Section 4.2, “The Sox class” (p. 7).
- Section 4.3, “The Elt class: An element in progress” (p. 9).

#### 4.1. The SoxError class

All errors generated by this module will raise an exception of class `SoxError`.

#### 4.2. The Sox class

To start generation of an XML file, use the constructor this way:

```python
from sox import Sox
...
sox = Sox(outFile=None)
```

The `outFile` argument may be any writeable file; the default value is `sys.stdout`.

Attributes and methods of this class:

```python
.soat(name, *p, **kw)
```

Start a new element of the given `name`. Additional positional arguments of type `unicode` will be added as text content after the start tag. Arguments of type `str` will be converted to Unicode using the UTF-8 encoding. Arguments of type `int`, or `bool` will be converted to Unicode and then added as text content.

Arguments of type `dict` will be translated into attributes in the start tag. Each keyword argument will be translated into a single attribute.

The method returns an instance of class `Elt` (see Section 4.3, “The Elt class: An element in progress” (p. 9)) that must be used to terminate the element later on. That same instance is also pushed on the `Sox` instance's internal stack.
If the element name or any attribute names are not valid according to the XML rules, this method will raise a `SoxError`. For the rules governing names, see Section 4.4, “Name validation” (p. 9).

Non-ASCII characters in attribute values will be replaced by XML character references. For example, code point `#x262e` will appear as “&#x262e;”. Additionally, in the values of attributes, these substitutions will be performed:

```
&lt; &lt;
& &amp;
" &quot;
```

`.end(elt)`

If the `elt` argument matches the one at the top of the Sox instance’s internal stack, the method writes an end tag for this element to the output file and pops that element off the stack.

If there is no match, the method raises a `SoxError`.

`.write(*p)`

The arguments must be Unicode strings, or any of the types that result in element content when passed to the `.start()` method (they may not be dictionaries). The content is written immediately to the output file.

`.leaf(*p, **kw)`

This method is for generating a “leaf” of the XML tree, that is, an element with no element children. The arguments are interpreted in the same way as arguments to the `.start()` method. If none of the arguments produced any textual content inside the element, you will get an empty tag; otherwise the result will place start and end tags around the content.

`.flush()`

Forces any buffered output to be written.

`.cleanup()`

If you want to be certain that all your elements have been closed, call this method. It checks that the internal stack is empty. If not, it raises a `SoxError`.

`.close()`

Calls the `.cleanup()` method and then closes the output file.

`.comment(text)`

Emits an XML comment to the output. The `text` argument is Unicode or a string that can be converted to Unicode via the UTF-8 encoding.

The comment text may not include two consecutive hyphen (`--`) characters. It may not start or end with a hyphen. The method will raise a `SoxError` if these rules are violated.

`.pi(target, args)`

Emits an XML processing instruction to the output. The `target` argument must be Unicode or convertible to Unicode via UTF-8, and also conform to the rules for XML names. The `args` text is Unicode or convertible to Unicode, and may not contain any instances of the string "?>".

`.doctype(name, pubId=None, sysId=None)`

Writes a `DOCTYPE` declaration.

- The `name` must be the XML element name of the root element.
- If neither of the optional arguments is given, the output looks like this:

```
<!DOCTYPE name>
```
If a pubId is given but not a sysId:

```xml
<!DOCTYPE name PUBLIC "pubId">  
```

If a sysId is given but not a pubId:

```xml
<!DOCTYPE name SYSTEM "sysId">  
```

If both the pubId and sysId are given:

```xml
<!DOCTYPE name PUBLIC "pubId" "sysId">  
```

### 4.3. The Elt class: An element in progress

An instance of this class is returned by the `S.start()` method as a token of elements whose start tag has been written but whose end tag has not.

**Elt(sox, tag)**

The constructor requires two arguments: the containing Sox instance and the element name.

**.end()**

Calls the `.end()` method of the Sox instance. This is the normal method of termination.

**.__enter__(self)**

Does nothing; its existence is required by the Python context manager protocol.

**.__exit__(self, exc_type, exc_value, traceback)**

Required by the Python context manager protocol. Calls `self.end()`.

### 4.4. Name validation

This package supports the definitions of valid element and attribute names as defined in the XML standard\(^\text{17}\). Names must consist of one character from set `NameStartChar` followed by zero or more characters from set `NameChar`.

**NameStartChar**

Names may start with “:”, “_”, any ASCII letter, or any member of several code point ranges from the Unicode standard:

- `#x00C0-#x00D6`
- `#x00D8-#x00F6`
- `#x00F8-#x2FF`
- `#x0370-#x037D`
- `#x037F-#x1FFF`
- `#x200C-#x200D`
- `#x2070-#x218F`
- `#x2C00-#x2FEF`
- `#x3001-#xD7FF`
- `#xF900-#xFDCF`
- `#xFDF0-#xFFFD`
- `#x10000-#xEFFFF`

\(^\text{17}\) [http://www.w3.org/TR/xml/](http://www.w3.org/TR/xml/)
**NameChar**

The set NameChar includes all the members of NameStartChar plus the digits 0-9, “-”, “.”, the `&middot;` character “·”, and characters from two additional ranges of Unicode code points:

- #x0300-#x036F
- #x203F-#x2040

---

**5. sox.py: The code prologue**

The implementation of the sox.py module follows.

```python
'''sox.py: Sequential output of XML with a low memory footprint

Do not edit this file directly. It is extracted automatically from the documentation:
http://www.nmt.edu/tcc/projects/sox/
'''
```

---

**6. Imported modules**

```python
# - - - - - I m p o r t s

We need the sys module for the standard I/O streams.

import sys
```

---

**7. Manifest constants**

Because Python has no constants *per se*, we use the variables whose names are in capital letters with “_” as the word separator.

```python
# - - - - - M a n i f e s t c o n s t a n t s

7.1. DEFAULT_ENCODING

This constant defines the default Unicode encoding we use for coercing str values to unicode.

```python
DEFAULT_ENCODING = 'utf-8'
```

---

7.2. NAME_START_RANGES

A list of the Unicode code point ranges that define the NameStartChar character set in the XML standard. See Section 4.4, “Name validation” (p. 9).

---

Note: For the full context, see the original source for the documentation of sox.py.
7.3. NAME_CHAR_RANGES

A list of Unicode code point ranges that define the NameChar character set, over and above the characters in NameStartChar that are automatically included.

```python
NAME_CHAR_RANGES = [
    (ord('-'), ord('-')),  
    (ord('.'), ord('.')),  
    (ord('0'), ord('9')),  
    (0x00B7, 0x00B7),    
    (0x0300, 0x036F),    
    (0x203F, 0x2040)  
]
```

7.4. ESCAPE_MAP

This dictionary’s keys are the characters that are not allowed within XML attribute values. Characters “<” and “&” are explicitly forbidden by the XML standard\(^\text{19}\). We also escape the double-quote character “""" because it is used to delimit the entire attribute value.

```python
ESCAPE_MAP = {  
    u'<' : u'&lt;',  
    u'&' : u'&amp;',  
    u'"' : u'&quot;'  
}
```

\(^\text{19}\)http://www.w3.org/TR/xml/
8. Specification functions

As part of the Cleanroom\(^\text{20}\) verification infrastructure, here are some specification functions used by various intended functions in the module. All intended function names contain two or more hyphenated words, to distinguish them from names in the actual program.

8.1. content-processor

This specification function defines a generic interface used by Sox.start() and other methods to process arguments that are to be made into Unicode content within an element.

```python
# content-processor(item, contentList, attribs) ==
# [ (contentList is a list) and (attribs is a dict) ->
#   if item is a dict ->
#     if (all keys of item are xml-names) and
#     (all values of item can be converted to Unicode) ->
#       attribs := attribs with the key-value pairs from
#       item added, with both converted to Unicode
#     else -> raise SoxError
#   else if item is Unicode or can be converted to Unicode ->
#     contentList +:= item as unicode
#   else -> raise SoxError ]
```

8.2. escaped-attribute

This function describes how the values of XML attributes are sanitized to make them valid.

```python
# escaped-attribute(v) ==
# v with "\," ",", and ",=" replaced by ",\&"," ",\&"," ",\&quot;"> 
```

8.3. content-list

The content-list function describes the positional arguments to methods such as Section 12, “Sox.start(): Start an element” (p. 16). The interpretation of each positional argument depends on its type.

```python
# content-list ==
# a list whose elements may be unicode, str, int, bool, or
# dict
# Describes positional arguments to methods like Sox.start().
```

\(^{20}\) http://www.nmt.edu/~shipman/soft/clean/
8.4. valid-comment

Describes values that can be made into XML comments.

```python
# valid-comment ==
# a unicode value that does not start or end with "-" and does
# not contain the string "--"
```

8.5. xml-name

Describes a valid XML name.

```python
# xml-name ==
# a name consisting of one NameStartChar followed by zero or
# more instances of NameChar, as defined in http://www.w3.org/TR/xml/
```

8.6. unicode-okay

```python
# unicode-okay(s) ==
# if type(s) is unicode or (s can be converted to unicode using
# DEFAULT_ENCODING) ->
# True
# else ->
# False
```

9. class SoxError

This is the exception thrown for all error conditions within this module.

```python
# - - - - - c l a s s  S o x E r r o r

class SoxError(Exception):
    '''Exception for all sox.py errors.'''
    ..
    pass
```
10. class Sox

An instance of this class represents a writer that emits XML to a given output stream. The Cleanroom intended functions here are more rigorous statements of the interface described in Section 4.2, “The Sox class” (p. 7).

```python
# - - - - - c l a s s  S o x

class Sox(object):
    '''Serial XML writer.

    Exports:
    Sox(outFile):
        [ outFile is a writeable file ->
        return a new Sox instance that writes to outFile
        with an empty tag stack ]
    .outFile: [ as passed to constructor ]

    For the specification functions used below, see Section 8.5, “xml-name” (p. 13) and Section 8.3, “content-list” (p. 12).

    .start(tag, *p, **attr):
        [ if (tag is a unicode xml-name) and
        (p is a content-list) and
        (attr is a dictionary whose keys are xml-names) ->
        outFile += a start tag with name (tag), attributes
        from (attr) and dict elements of p, and content
        from non-dict elements of p
        self := self with an Elt containing (tag) pushed on its
        stack
        return an Elt containing (tag)
        else -> raise SoxError ]
    .end(elt):
        [ elt is an Elt instance ->
        if elt is the top of self's stack ->
        outFile += end tag for elt.tag
        self := self with its stack popped
        else -> raise SoxError ]
    .write(*p):
        [ p is a sequence of unicode, or str values ->
        outFile += those values, with str values
        decoded via utf-8 ]
    .leaf(tag, *p, **kw):
        [ if (tag is an xml-name) and
        (p is a content-list) and
        (kw's keys are xml-names) ->
        outFile += an element with name (tag), attributes
        from (kw) and dict elements of p, and content
        (if any) from non-dict elements of p
        else -> raise SoxError ]
    .flush():
        [ self.outFile := self.outFile with all buffered output sent ]
```

New Mexico Tech Computer Center
.cleanup():
    [ if self's stack is empty ->
    I
    else -> raise SoxError ]
.close():
    [ if self's stack is empty ->
      self.outFile := self.outFile, closed
    else -> raise SoxError ]

See Section 8.4, “valid-comment” (p. 13) for the rules for the content of XML comments.

.pi(target, args):
    [ target and args are unicode or utf-8-encoded str values ->
      if (target is an xml-name) and (args does not contain the string "?>") ->
        self.outFile += an XML processing instruction with target (target) and arguments (args)
      else -> raise SoxError ]

Here is the class's internal state.

State/Invariants:
  _stack:
    [ a list of open elements as Elt instances, in the order they were opened ]

11. Sox.__init__()

The constructor has little to do except to remember its output file and initialize the element stack.

```python
# --- Sox.__init__ ---

def __init__(self, outFile=None):
    '''Constructor.
    ...
    if outFile is None:
        self.outFile = sys.stdout
    else:
        self.outFile = outFile
    self._stack = []
```
12. **Sox.start()**: Start an element

```python
# - - - S o x . s t a r t

def start(self, tag, *p, **kw):
    '''Start an element.
    '''

The first order of business is to check all the arguments. The `tag` argument must be an `xml-name`; this is checked by Section 13, “Sox._nameCheck(): Is this a valid xml-name?” (p. 17). Then we go through the positional arguments and the keyword arguments and, assuming all are valid, separate them into content (contentList, a list of Unicode values) and attributes (attribs, a dictionary); see Section 16, “Sox._sortParams(): Classify content and attribute arguments” (p. 19).

```python
#-- 1
# [ if tag is an xml-name ->
#   uTag := tag as unicode
# else -> raise SoxError ]
uTag = self._nameCheck(tag)

#-- 2
# [ if p and kw are valid ->
#   contentList := a list of unicode values made from
#   the non-dictionary elements of p
#   attribs := a dictionary made from the dictionary
#   elements of p, plus the elements of kw
# else -> raise SoxError ]
contentList, attribs = self._sortParams(p, kw)

Next we write the start tag: see Section 28, “Sox._startTag(): Write a start tag” (p. 28).

```python
#-- 3
# [ sys.stdout += (an XML start tag with name (tag)
#   and attributes made from (attribs)) ]
self._startTag(uTag, attribs)

Following the start tag comes the initial content, if any.

```python
#-- 4
# [ if contentList is nonempty ->
#   sys.stdout += concatenated elements of contentList
# else -> I ]
if len(contentList) > 0:
    self.outFile.write(u''.join(contentList).encode('utf-8'))

Finally, fabricate the Elt instance, push it on the stack, and also return it.

```python
#-- 5
# [ self._stack := a new Elt instance with name (tag)
#   return that Elt instance ]
elt = Elt(self, uTag)
```
13. Sox._nameCheck(): Is this a valid xml-name?

```python
def _nameCheck(self, name):
    '''Is (name) a valid XML name?

    if name is unicode or a UTF-8 encoded str ->
    if name is a valid xml-name ->
    return name as unicode
    else -> raise SoxError
    ...'''

    uName = self._unicodify(name)

    if len(uName) == 0:
        raise SoxError("The empty string is not a valid "
                      "XML name.")
    else:
        start = uName[0]
        rest = uName[1:]

    if not self._checkCharRange(start, NAME_START_RANGES):
        raise SoxError("The first character of %s is not "
                       "a NameStartChar")
```

First we coerce the name to Unicode; see Section 14, “Sox._unicodify(): Force to Unicode” (p. 18).
For the definition of a valid XML name, see Section 8.5, “xml-name” (p. 13): the name must have at least one NameStartChar character followed by zero or more NameChar characters.

For the tables of Unicode code point ranges that define the XML NameStartChar and NameChar categories, see Section 7.2, “NAME_START_RANGES” (p. 10) and Section 7.3, “NAME_CHAR_RANGES” (p. 11).
For the logic that checks whether a character is within one of those ranges, see Section 15, “Sox._checkCharRange(): Is this code point in a given set of intervals?” (p. 19).
"a valid XML NameStartChar." % uName)

#-- 4
# [ if the ordinal of any character in (rest) is not in
# any of the closed intervals in NAME_CHAR_RANGES ->
# raise SoxError
# else -> I ]
for k, c in enumerate(rest):
    if ((not self._checkCharRange(c, NAME_START_RANGES)) and
        (not self._checkCharRange(c, NAME_CHAR_RANGES))):
        raise SoxError("The character in position %s of "
        "%s is not a valid XML NameChar." % (k+1, uName))

#-- 5
return uName

14. Sox._unicodify(): Force to Unicode

#--- Sox._unicodify

    def _unicodify(self, s):
        """Insure that s is unicode.

        [ type(s) is unicode or str ->
          if type(s) is unicode ->
            return s
          else if s is unicode-okay ->
            return s as unicode
          else -> raise SoxError ]

        ...

    #-- 1
    if type(s) is unicode:
        return s

    #-- 2
    try:
        u = unicode(s, DEFAULT_ENCODING)
    except UnicodeDecodeError:
        raise SoxError("String %s cannot be converted to "
        "Unicode." % s)

    #-- 3
    return u
15. **Sox._checkCharRange(): Is this code point in a given set of intervals?**

```python
# - - - _ c h e c k C h a r R a n g e

def _checkCharRange(self, c, rangeTable):
    '''Is this character's code point in a given set of ranges?

    [ (c is a single Unicode character) and
      (rangeTable is a list of tuples of two integers (lo, hi)) ->
      if there is an entry in rangeTable such that
      lo <= ord(c) <= hi ->
      return True
      else ->
      return False ]

...'''
```

As you read this, are you thinking to yourself, gee, isn’t this a terribly inefficient way to do it? Shouldn’t we build a dictionary that maps all possible Unicode code points to bool values so we can find out whether a character is in a class in nearly constant time? Well, if so, just keep in mind the words of Dr. Donald Knuth, who once said, “...premature optimization is the root of all evil.”

```python
#-- 1
cp = ord(c)

#-- 2
for lo, hi in rangeTable:
    if lo <= cp <= hi:
        return True

#-- 3
return False
```

16. **Sox._sortParams(): Classify content and attribute arguments**

```python
# - - - S o x . _ s o r t P a r a m s

def _sortParams(self, p, kw):
    '''Check and process a content-list.

    [ (p is a sequence) and
      (kw is a dict) ->
      if (p is a valid content-list) and
      (the keys of kw and the dict elements of p are all
      xml-names) and
      (the values from kw and the values of dict elements
```

---

of p are all unicode-okay) ->
return (a list of unicode values made from
the non-dictionary elements of p, a dictionary made
from the dictionary elements of p as unicode
plus the elements of kw as unicode)
else -> raise SoxError ]
```

We start by building an empty dictionary in which to accumulate attribute key-value pairs, and an
empty list in which to accumulate content items.

```python
#-- 1
contentList = []
attribs = {}
```

The handling of the elements of p is by type. We process them by using a class dictionary `._typeRouter
whose keys are the valid types, and each related value is a function that conforms to the interface defined
in Section 8.1, "content-processor" (p. 12).

```python
#-- 2
# [ if the type of each element of p is a key in
# self._typeRouter ->
#   contentList += content derived from that element as
#   Unicode by self._typeRouter[that type], if any
#   attribs += attributes derived from that element as
#   Unicode by self._typeRouter[that type], if any
# else ->
#   contentList += (anything)
#   attribs += (anything)
#   raise SoxError ]
for param in p:
  #-- 2 body
  #-- 2.1
  # [ if type(param) is a key in self._typeRouter ->
  #   processor := the corresponding value
  # else -> raise SoxError ]
  try:
    processor = self._typeRouter[type(param)]
  except KeyError:
    raise SoxError("Argument %r is not a valid type 
    "for element content." % param)
  #-- 2.2
  # [ contentList += content derived from param as
  #   Unicode by processor, if any
  #   attribs += attributes derived from param as
  #   Unicode by processor, if any ]
  processor(self, param, contentList, attribs)
```

That takes care of all the positional arguments. Next, we fold the dictionary kw of keyword arguments
into attribs. We can't use the dict.update() method for this merger because we must check the
keys to make sure they are valid XML names, and convert the keys and the values to Unicode.
sox.py

```python
#-- 3
# [ if (all the keys of kw are xml-names) and
#   (all the values from kw are unicode-okay) ->
#   attrib += key-value pairs from kw, with keys
#   and values as unicode
# else ->
#   attrib += (anything)
# raise SoxError ]
for key, value in kw.items():
    #-- 3 body
    # [ if (key is an xml-name) and (value is unicode-okay) ->
    #   attrib[key as unicode] := value as unicode
    # else -> raise SoxError ]
    #-- 3.1
    # [ if key is an xml-name ->
    #   uniKey := key as unicode
    # else -> raise SoxError ]
    uniKey = self._nameCheck(key)
    #-- 3.2
    # [ if value is unicode or can be converted etc. ->
    #   uniValue := value, so converted
    # else -> raise SoxError ]
    uniValue = self._unicodify(value)
    #-- 3.3
    attribs[uniKey] = uniValue

#-- 4
return (contentList, attribs)
```

17. **Sox._unicodeProcessor()**

This function implements the interface described in Section 8.1, “content-processor” (p. 12) for unicode parameters. Unicode values are appended to the contentList.

```python
# - - - S o x . _ u n i c o d e P r o c e s s o r

def _unicodeProcessor(self, item, contentList, attribs):
    """Process unicode arguments
    [ type(item) is unicode ->
      contentList += item ]
    """
    contentList.append(item)
```
18. **Sox._strProcessor()**

This function implements the interface described in Section 8.1, “content-processor” (p. 12) for the `str` type. The value is coerced to Unicode (if possible) and appended to the `contentList`.

```python
# --- Sox._strProcessor
def _strProcessor(self, item, contentList, attribs):
    '''Process str arguments
    [ type(item) is str ->
    if item is unicode-okay ->
        contentList += item as unicode
    else -> raise SoxError ]
    ...
```

For the logic that coerces the value to Unicode, see Section 14, “Sox._unicodify(): Force to Unicode” (p. 18).

```python
#-- 1
# [ if item is unicode-okay ->
#  contentList += item as unicode
#  else -> raise SoxError ]
contentList.append(self._unicodify(item))
```

19. **Sox._intProcessor()**

This function implements the interface described in Section 8.1, “content-processor” (p. 12) for the `int` type. The value is coerced to Unicode and appended to the `contentList`.

```python
# --- Sox._intProcessor
def _intProcessor(self, item, contentList, attribs):
    '''content-processor for type int
    [ type(item) is int ->
    contentList += unicode(str(item)) ]
    contentList.append(unicode(str(item)))
```

20. **Sox._boolProcessor()**

This function implements the interface described in Section 8.1, “content-processor” (p. 12) for the `bool` type. We convert the value first to `int` because XML understands 0 and 1 but not strings like "True", then to Unicode.

```python
# --- Sox._boolProcessor
def _boolProcessor(self, item, contentList, attribs):
```
21. Sox._dictProcessor()

This function implements the interface described in Section 8.1, “content-processor” (p. 12) for the dict type. Assuming that all of the keys in the item are xml-names, we add the key-value pairs to attribs. We use Section 22, “Sox._attrEscape(): Escape forbidden characters in attribute values” (p. 23) to substitute standard escape sequences for certain forbidden characters in each attribute value.

```python
# -- S o x . _ d i c t P r o c e s s o r

def _dictProcessor(self, item, contentList, attribs):
    '''content-process for dictionaries

    [ type(item) is dict ->
      if all keys in attribs are xml-names ->
        attribs += key-value pairs from item
      else ->
        attribs += (anything)
        raise SoxError ]
    ...
    #-- 1
    for key, value in item.items():
        #-- 1 body
        # [ if key is an xml-name ->
        #   attribs[key] := value
        # else -> raise SoxError ]
        #-- 1.1
        # [ if key is an xml-name ->
        #   I
        # else -> raise SoxError ]
        self._nameCheck(key)
        #-- 1.2
        attribs[key] = value
```

22. Sox._attrEscape(): Escape forbidden characters in attribute values

```python
# -- S o x . _ a t t r E s c a p e

def _attrEscape(self, s):
    '''Escape forbidden attribute characters.
```
if s is unicode-okay ->
    return s as unicode, with any characters that are
    keys in ESCAPE_MAP replaced by the related values
else -> raise SoxError }

See Section 7.4, “ESCAPE_MAP” (p. 11) for the list of forbidden characters and their equivalents. We
use recursion because it’s cleaner. First, the basis case, then the recursive case.

```python
if len(s) == 0:
    return u''
else:
    first = self._unicodify(s[0])
    rest = s[1:]

    if first in ESCAPE_MAP:
        return ESCAPE_MAP[first]+self._attrEscape(rest)
    else:
        return first+self._attrEscape(rest)
```

23. Sox._typeRouter(): Mapping of content parameter
types to converter functions

This class dictionary relates the types of parameters that can be passed to Sox.start() (and similar
methods) to the corresponding conversion function. The functions in this table must implement the in-
terface described by Section 8.1, “content-processor” (p. 12).

```python
_typeRouter = {
    unicode: _unicodeProcessor,
    str: _strProcessor,
    int: _intProcessor,
    bool: _boolProcessor,
    dict: _dictProcessor
}
```
24. Sox._buildAttrs(): Build XML attributes from a dictionary

```python
# --- Sox._buildAttrs

def _buildAttrs(self, attribs):
    '''Format XML attributes.

    [ attribs is a dictionary whose keys are xml-names and
      whose values are all unicode-okay ->
      if attribs is empty ->
      return ""
      else ->
      return " " + (a space-separated list of items A=V
      where each A is a key in attribs as unicode and
      V is escaped-attribute(the related value as
      unicode)) ]
    '''

    if len(attribs) == 0:
        return u''
    else:
        result = [u'']

    for key, value in attribs.items():
        uniKey = key as unicode
```

We return the empty string if there are no attributes. Without this case, the start tag might look like “<name >”. If there are any attributes, we start the result list with a single empty string, which will cause the result to start with one space that separates the tag name from the first attribute in the start tag.

```python
#-- 1
if len(attribs) == 0:
    return u''
else:
    result = [u'']

#-- 2
# [ if (the keys of attribs are all xml-names) and
# (the values of attribs are all unicode-okay) ->
# result += unicode values of the form "K=V" where
# each K is an element of attrib as unicode
# and each V is the corresponding value as
# unicode
# else ->
# result += (anything)
# raise SoxError ]
for key, value in attribs.items():
    #-- 2 body
    # [ if (key is an xml-name) and (value is unicode-okay) ->
    # result += (a string of the form K=V where K
    # is (key) as unicode and V is (value) as
    # unicode with forbidden characters escaped
    # else -> raise SoxError ]
    #-- 2.1
    # [ if key is an xml-name ->
    # uniKey := key as unicode
```
The “K=V” item we append to the result list must actually be ASCII. We use the `unicode.encode()` function with the ‘xmlcharrefreplace’ option so that non-ASCII characters are escaped, e.g., code point 0xC0 will be rendered as “&192;”.

Now we have the attribute name and value in Unicode form.

The closing tag has the form “</tag>”, but the tag name must be converted to ASCII.
26. Sox.write()

```python
#--- Sox.write
def write(self, *p):
    '''Write content in the current element.
    '''

The arguments to this method are treated the same as the content arguments to Sox.start(): they are converted to Unicode, then to UTF-8 ASCII, then written to the output file.

We can use Section 16, “Sox._sortParams(): Classify content and attribute arguments” (p. 19) to do the work of converting our arguments to a list of Unicode strings. That method, however, allows dictionaries in its “p” argument, but this method stipulates that dictionaries are not allowed. However, if any dictionaries are present, the attribute dictionary returned by ._sortParams() will be nonempty.

```}

27. Sox.leaf(): Write an element with no element children

```python
#--- Sox.leaf
def leaf(self, tag, *p, **kw):
    '''Write an element with no element children.
    '''

Next, we write the elements of contentList to the output file, with non-ASCII characters escaped.

```
The argument processing is the same here as it is in Section 12, “Sox.start(): Start an element” (p. 16): see Section 13, “Sox._nameCheck(): Is this a valid xml-name?” (p. 17) and Section 16, “Sox._sortParams(): Classify content and attribute arguments” (p. 19).

```python
#-- 1
# [ if tag is an xml-name ->
#   uTag := tag as unicode
# else -> raise SoxError ]
uTag = self._nameCheck(tag)

#-- 2
# [ if p and kw are valid ->
#   contentList := a list of unicode values made from
#   the non-dictionary elements of p
#   attribs := a dictionary made from the dictionary
#   elements of p, plus the elements of kw
# else -> raise SoxError ]
contentList, attribs = self._sortParams(p, kw)
```

Because we know there will be no child elements, we can write the complete element right away. There are two cases, however. If there are no content elements, we can write an empty tag. If there is content, we write the start tag, then the content, then the end tag. See: Section 29, “Sox._emptyTag(): Write an empty XML tag” (p. 29); Section 28, “Sox._startTag(): Write a start tag” (p. 28); and Section 30, “Sox._endTag(): Write an XML end tag” (p. 29).

```python
#-- 3
# [ if contentList is empty ->
#   self.outFile += an empty tag with name (uTag)
#   and attributes made from (attribs)
# else ->
#   self.outFile += (an empty tag with name (uTag)
#   and attributes made from (attribs)) +
#   (concatenation of content) + (an end tag with
#   name (uTag)) ]
if len(contentList) == 0:
    self._emptyTag(uTag, attribs)
else:
    self._startTag(uTag, attribs)
    self.write(*contentList)
    self._endTag(uTag)
```

### 28. Sox._startTag(): Write a start tag

```python
#-- -- S o x _ s t a r t T a g

def _startTag(self, uTag, attribs):
    '''Write a start tag to the output.

    [ (uTag is a unicode xml-name) and
      (attribs is a dict with its keys unicode xml-names) ->
      self.outFile += an XML start tag with name (uTag)
```

```
See Section 24, “Sox._buildAttrs(): Build XML attributes from a dictionary” (p. 25) for the logic that converts an attribute dictionary into the XML form “name="value"”. That method’s return value has a leading space if there are any attributes; this is necessary to separate it from the tag. If there are no attributes, however, it returns an empty string, so you don’t get tags like <foo >.

```python
#-- 1
self.outFile.write("<%s%s>" % 
(uTag.encode('utf-8'), self._buildAttrs(attribs)))
```

29. Sox._emptyTag(): Write an empty XML tag

This method is nearly identical to Section 28, “Sox._startTag(): Write a start tag” (p. 28), except that it appends a slash “/” just inside the closing “>”.

```python
# - - - S o x . _ e m p t y T a g
def _emptyTag(self, uTag, attribs):
    """Write an empty XML tag.
    ""
    #-- 1
    self.outFile.write("<%s%s/>" % 
    (uTag.encode('utf-8'), self._buildAttrs(attribs)))
```

30. Sox._endTag(): Write an XML end tag

```python
# - - - S o x . _ e n d T a g
def _endTag(self, uTag):
    """Write an XML end tag.
    [ uTag is a unicode xml-name ->
      self.outFile += an XML end tag with name (uTag) ]
    ...
    self.outFile.write("</%s>" % uTag.encode('utf-8'))
```

31. Sox.flush()

```python
# - - - S o x . f l u s h
def flush(self):
    """Flush buffered output.
    ""
    self.outFile.flush()
```
32. Sox.cleanup(): Cleanup

This method insures that all open elements have been closed.

Note

The original design called for the destructor (__del__()) method to do this cleanup. However, a class's destructor is called only when its reference count goes to zero. If there are open elements, then each Elt instance on the stack will contain a reference to the parent Sox instance, so merely deleting the Sox instance (with a del statement, for example) does not enter the destructor, because the stack still contains references to the parent instance. Hence we ask the caller to use an explicit call here to insure closing of all open elements.

```python
# - - - Sox.cleanup

def cleanup(self):
    '''Insure all elements have been closed.
    ...
    if len(self._stack) > 0:
        if len(self._stack) == 1:
            text = ("One <%s> element was not closed." %
                    self._stack[0].tag)
        else:
            tagNames = ' '.join([s.tag
                                  for s in self._stack])
            text = ("Multiple unclosed elements: %s" % tagNames)
    raise SoxError(text)
    else:
        self.flush()
```

33. Sox.close(): Cleanup and close

```python
# - - - Sox.close

def close(self):
    '''Cleanup and close.
    ...
    #-- 1
    self.cleanup()
    #-- 2
    self.outFile.close()
```
34. **Sox.comment()**

```python
#--- Sox.comment

def comment(self, text):
    '''Emit an XML comment.
    '''
    The format of a comment is <!-- ... -->. The XML comment syntax forbids three hyphens in a row, so the text can't start or end with a hyphen. Also, two or more hyphens in a row are not allowed within the text.
    
    #-- 1
    # [ if text is unicode or can be coerced to unicode ->
    #    uText := text as unicode
    #  else -> raise SoxError ]
    uText = self._unicodify(text)

    #-- 2
    # [ if (text starts or ends with a hyphen) or
    #    (text contains u'--') ->
    #    raise SoxError
    #  else -> I ]
    if len(uText) > 0:
        if uText[0] == u'-':
            raise SoxError("Comment text %r: Cannot start "
                            "with a hyphen." % uText)
        elif uText[-1] == u'-':
            raise SoxError("Comment text %r: Cannot end "
                            "with a hyphen." % uText)
        elif u'--' in uText:
            raise SoxError("Comment text %r: Cannot contain "
                            "two consecutive hyphens." % uText)

    #-- 3
    self.outFile.write("<!--%s-->" % uText.encode('utf-8'))
```

35. **Sox.pi(): Write a processing instruction**

```python
#--- Sox.pi

def pi(self, target, args):
    '''Write an XML processing instruction.
    '''
    #-- 1
    # [ if (target is a valid xml-name) and
    #    (target and args are unicode or can be coerced to
    #     unicode) ->
    #     uTarget := target as unicode
    #     uArgs := args as unicode
```
Aside from the constraint that both parts must be Unicode and the target must a valid XML name, the standard\(^{22}\) also stipulates that the arguments part may not contain the string “?>”.

```python
#-- 2
if u'?>' in uArgs:
    raise SoxError("PI arguments %r: May not contain "
                  "'?>'." % uArgs)

#-- 3
self.outFile.write("<?%s %s?>" %
                  (uTarget.encode('utf-8'), uArgs.encode('utf-8')))
```

### 36. Sox.doctype()

```python
#-- S o x . d o c t y p e

def doctype(self, name, pubId=None, sysId=None):
    '''Write a DOCTYPE declaration.'''
```

There are four cases depending on the presence or absence of the two optional arguments.

<table>
<thead>
<tr>
<th>pubId</th>
<th>sysId</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>&lt;!DOCTYPE name&gt;</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>&lt;!DOCTYPE name SYSTEM &quot;sysId&quot;&gt;</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>&lt;!DOCTYPE name PUBLIC &quot;pubId&quot;&gt;</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>&lt;!DOCTYPE name PUBLIC &quot;pubId&quot; &quot;sysId&quot;&gt;</td>
</tr>
</tbody>
</table>

We’ll deal with these cases by building a list `pieceList` with the various pieces, then concatenating them.

```python
#-- 1
# [ pieceList := a list containing <!DOCTYPE name " ]
pieceList = [ "<!DOCTYPE %s" % name ]

#-- 2
if pubId is None:
    if sysId is not None:
        pieceList.append(' SYSTEM "%s" % sysId)
    else:
        pieceList.append(' PUBLIC "%s" % pubId)
else:
    pieceList.append(' PUBLIC "%s" "%s" %
                     (pubId, sysId))
```
37. **class Elt: One open element**

Each instance of this class represents one open element whose start tag has been written but whose end tag is pending. The **Sox** class maintains a stack of these instances in order to ensure start-end tag matching. The intended functions below are formalized versions of the interface defined in Section 4.3, “The Elt class: An element in progress” (p. 9).

```python
# - - - - - - - class Elt

class Elt(object):
    '''Represents a partially built XML element.

    Exports:
    Elt(sox, tag):
        [(sox is the containing Sox instance) and
        (tag is an xml-name) ->
          return a new Elt instance with those values ]
    .sox:  [ as passed to constructor ]
    .tag:  [ as passed to constructor ]
    .end():
        [ if self is the top of sox's stack ->
          sox := sox with its stack popped
        else -> raise SoxError ]
        __enter__(): [ I ]
        __exit__(): [ same as .end() ]

    ..."```

38. **Elt.__init__()**

```python
# - - - Elt.__init__

def __init__(self, sox, tag):
    '''Constructor.

    self.sox = sox
    self.tag = tag"
```

39. **Elt.end()**

A convenience method that calls the `.end()` method of the parent **Sox** instance.
def end(self):
    '''Terminate an open element.
    ..
    self.sox.end(self)

This method is required because the class is a context manager. It returns the instance, in case anyone cares.

This is the cleanup method of the context manager protocol discussed in Section 3.3, “Context managers and the “with” construct” (p. 6).

This script exercises the various functions of the module.

#!/usr/bin/env python
#================================================================
# soxtest: Test driver for sox.py
# # Do not edit this file directly. It is extracted automatically from
# # the documentation:
# # http://www.nmt.edu/tcc/projects/sox/
# #
# import sys
# import sox
def main():
    '''Main program.'''
    s = sox.Sox(sys.stdout)
    s.doctype("html", HTML_PUB_ID, HTML_SYS_ID)
    html = s.start("html")
    head = s.start("head")
    s.leaf("title", "Page title here")
    head.end()
    body = s.start("body")
    s.leaf("h1", "Main title here")
    s.leaf("hr")
    s.leaf("p", {'class': 'note'}, "Some text",
            " and some more text", id='p001')
    body.end()
    html.end()

if __name__ == '__main__':
    main()