Processing XML Text with Python and ElementTree – a Practical Experience

Radovan Garabík

Ľudovít Štúr Institute of Linguistics
Slovak Academy of Sciences
Bratislava, Slovakia

Abstract

In this paper, we evaluate the use of XML format as an internal format for storing texts in linguistic corpora, and describe our experience in using the ElementTree Python XML parser in the Slovak National Corpus.

1 Introduction

XML format, despite its shortcomings, is attracting more and more attention as a format for text representation in corpus linguistics. XML is intended as a free extensible mark-up language for the description of richly structured textual information. The exact method of data description is unspecified and is usually designed according to specific requirements.

The Text Encoding Initiative (TEI) project[1] tries to establish a common XML schema for the general-purpose encoding of textual data. Following the relative success of SGML-based CES (Corpus Encoding Standard), an XML version of it was proposed[2] as a standard to store corpus compatible data.

XML as such gained quite a lot of popularity among different corpora (and corpus linguists); some of them use different XML schemas[3], but many of them use the XCES format.[4]

2 Information Hierarchy in Text Documents

Logically, we can design a rather complicated hierarchy for a document, consisting of sections, each with its heading, each section consisting of subsections (each of those eventually with a heading of its own), then divided into paragraphs. Other types of texts (such as poems) can have different, often more complicated structure. We are talking now only about the structure of information flow in a document, not about other linguistic information (like sentence boundaries). When considering the features (styles) of common word processing and desktop publishing systems, one would expect that this kind of structure is present and in common use.

However, looking at actual texts that come into corpora, we find this kind of structure only very rarely. The overwhelming majority of word processing DTP software users do not use the facility offered by the software to create (or use those already existing) logical styles to format the document, but apply physical text attributes to the document parts instead – so, for example, the headers are distinguished from the rest of the text only by changes in font size or font weight. This makes it almost impossible to use universal tools to extract logical structure from the documents. Often, only very basic structure can be identified and kept in the corpus.

3 Various Levels of Text Representation

There are actually two different ways of putting texts into the XCES format. One way is to use XML tags to mark up the hierarchical structure of text flow and typographical information. The other way is to use XML to organise basic structural elements of the texts (usually words) together with additional linguistic information into a rigid structure for further processing – in this way, we are using XML format as a (rather inefficient) way of emulating a tabular format.

4 Why Python

Our programming language of choice is Python[5], a high level object oriented programming language with a very clean syntax. Typically, using Python for software development leads to very short deployment times when compared with others, better promoted languages. The clarity of the syntax also contributes to very few language-oriented bugs in the software, leaving more time for debugging and optimalisation of the algorithms used. Python also has an excellent standard library, covering most of the routine programming tasks connected with interfacing various levels of the operating system, user interaction and robust data manipulation. There are also many other external libraries (modules) covering more specialised tasks, and connecting to existing libraries in other programming languages (most notably C and C++) is easy, insofar as programming in C or C++ is easy.

The disadvantages of using Python stem mostly from the fact that it is an interpreted language, with the consequent negative effects relating to speed of execution. While several Python compilers, optimisers and JIT-compilers have been designed, at least theoretically, only Psyco[6] seems mature enough for production use, and its performance gain is not very impressive – thanks to Python's dynamic nature.

5 Structure of Data in the Slovak National Corpus

Texts coming into the corpus are put into a hierarchical structure, each level corresponding to a different stage of text conversion and processing. Initially, texts are stored in the *Archive* in their original format. The texts are then converted into common text format, keeping some typographic information present in the original sources. We call this level of text processing the *Bank*. The data are then cleaned up and additional linguistic information is added to them, and the files are placed in the next level called the *Corpusoid*. The final step in data processing is a level called simply the *Data*, where the data are converted into binary format for the corpus manager.

File format in the Bank is in fact a simple subset of XCES-conforming XML. The files from the *Archive* are converted into this common Bank-format and these files are then converted on their way to the *Corpusoid* In the Corpusoid, texts are already tokenised, tokens are grouped into sentences, and each token contains additional information about lemma and morphosyntactic categories. Therefore, XML is used here to implement this tabular-like structure.

6 Using ElementTree

ElementTree[7], by Fredrik Lundh, is a Python implementation of an XML structure representation, in DOM-like style. The whole tree structure is represented by an ElementTree object, which can be created from scratch or read from an existing XML file. Parsing an XML file can be done in one line of code:

```
tree = ElementTree.parse('filename.xml')
```

Similarly, writing in-memory representation of an XML structure to a file can be done in this way:

```
tree.write(file('output_filename.xml', 'w'), encoding='utf-8')
```

Each XML node is represented by a dictionary-like object of an *Element* class. It is possible to loop through children of the node, to find a given subnode, to query attributes of the node or to modify any of these in place. In order to start working with nodes, we have to create a reference to a top-level node in our XML structure:

```
root = tree.getroot()
```

`root` is now an *Element* object. Let's take as an example the following piece of an XML file:

```
<p style="plain">Paragraph with a <hi>highlighted</hi> word.</p>
```

This will be represented in Elementtree as an Element class with the following attributes (some are omitted for brevity):

```
element.name == 'p'
element.text == 'Paragraph with a '
element.attribs = {'style':'plain'}
element.tail = None
element.children = [hi_element]
```

where *hi_element* is another Element class:

```
element.name == 'hi'
element.text == 'highlighted'
element.attribs = {}
element.tail =  'word.'
element.children = []
```

The problem with this approach is obvious: while the text after the highlighted part in our example is logically and structurally on the same level as the rest of the text, in Elementtree XML representation it has been put into the `<hi>` element as a tail attribute, creating a lot of problems when trying to program a way of iterating through the text, because suddenly one has to be aware that parts of the text can be hidden in subordinate elements – and we have go into arbitrary depths.

In fact, as our experience in parsing the bank format shows, this problem is really intimidating. We had to use complicated solutions, often including careful recursion into subnodes, and we learned that it is almost impossible to modify the document structure in place, because one has to be careful about putting the tail elements into the correct places when eliminating, adding or otherwise modifying the children nodes.

Fortunately, we need not to deal with the texts on this level, the only thing we have to do with texts in the Bank is to tokenise them and transform them into the XCES Corpusoid files.

Looking on the bright side, ElementTree turned out to be a very useful representation of XCES files in the corpusoid. Each token is represented by a `<tok>` node, containing several subnodes describing the token. At the first stage, just after converting the text from bank into XCES format, there is just an `<orth>` subnode with original wordform as a text attribute:

```
<tok>
<orth>meč</orth>
</tok>
```

The text is then lemmatised and morphologically annotated. We are using the software described in [8, 9]. The system consists of an external executable program, expecting data in its own SGML encoded format, transforming it and writing the output into an SGML output file. In order to utilise the tagger in our system, we convert our XCES file into input format, run the tagger, then iterate through tokens in the output SGML file and fill in lemmas and morphosyntactic tags into XML elements.

After the analyser run, XML in the Corpusoid looks like this (indentation has been added for clarity):

```
<tok>
  <orth>meč</orth>
  <disamb>
    <base>meč</base>
    <ctag>SSis1</ctag>
  </disamb>
  <lex>
    <base>meč</base>
    <ctag>SSis1</ctag>
  </lex>
  <lex>
    <base>mečat'</base>
    <ctag>VMesb+</ctag>
  </lex>
</tok>
```

The results of the analyser run are stored in a sequence of `<lex>` elements. Each `<lex>` element describes one possible combination of a lemma (`base` node) and morphosyntactic tag (`ctag` node), corresponding to a given wordform. Out of these `<lex>` elements, one is chosen by a disambiguating module of the analyser as the right one for the given word, using statistical principles (see [8]), and is put into a `<disamb>` node.

Commented pseudocode (a valid python code) adding a `<lex>` element into the Elementtree corpusoid representation can look like this:

```
# tok variable refers to an element corresponding to
# a <tok> entry in XML file
#
# first, create a subnode of a <tok> node, with XML tag 'lex'
lex = SubElement(parent=tok, tag='lex')
# add newlines to make the XML look more pretty
lex.text = lex.tail = '\n'
# create a subnode of a <lex> node, with XML tag base
base = SubElement(parent=lex, tag='base')
# put the actual content into the <base> XML 'node'
base.text = lemma_from_analyser
# create a subnode of a <lex> node, with XML tag 'ctag'
ctag = SubElement(lex, 'ctag')
# put the actual content into the <ctag> XML node
ctag.text = tag_from_analyser
# that's all
```

It is possible to run other different analysers (e.g. semantic tagger) at this point; adding additional XML tags (i.e. subelements) into the `<tok>` node is really easy. Only if we need to modify the superior XML structure, we have to refrain from modifying the document in place because of the difficulties involved, and we should better create a new elementtree structure and create elements and subelements of it as needed.

7 Compatibility and Performance

ElementTree, having been written in pure Python, runs wherever Python can run, without any problems whatsoever. This includes almost all modern Unix operating systems together with Linux and MacOSX, and the Microsoft family of operating systems. Since XML has been designed from the beginning as a common format for textual data cross platform interchange, there are no problems at all in using documents transferred to/from other platforms. To avoid eventual problems with character encoding, we universally use UTF-8 encoding in NFKC canonical normalisation (as is the de-facto norm in the Unix world). The other, perfectly acceptable way would to use just ASCII encoding, and have non-ASCII characters represented as XML entities. Being written in Python, one could expect ElementTree not to perform sufficiently well. However, in addition to the pure Python version, there is an alternative cElementTree module written in C, with ElementTree-compatible API, much better performance and lower memory requirements. As our experience shows, the speed of parsing is sufficient even for pure Python version – on a modest 1200 MHz Pentium III

CPU, an average speed of parsing a completely annotated XML file is about 1200 tokens per second. The morphological tagger on the above configuration is able to analyse 250 tokens per second, so the total overhead of using the ElementTree Python-based solution is not bad at all. ElementTree, being DOM-like, not SAX-like, requires the whole parsed document to be present in computer memory; therefore the memory requirements are going to be important. For example, representation of fully annotated document of about 200 000 tokens (one of the biggest continuous texts present in the Slovak National Corpus), being 16 MB of size, takes 410 MB of memory. The C version gives much better results – parsing speed is about 80000 tokens per second, and the above mentioned document takes 62 MB of memory, which is perfectly adequate for modern computer systems.

There is also another implementation of the Python XML parsing library with API almost identical to ElementTree, called lxml[10], based on very fast libxml2 parsing library[11]. In addition to ElementTree capabilities, it exposes libxml2 and libxslt specific functionality, providing a way of handling XPath, Relax NG, XML Schema, XSLT and c14n. However, we did not evaluate this software.

8 Conclusion

Using Python has no doubt great advantages when used in general programming, especially considering its clean syntax, readability and extensive standard library and rich language features, all contributing to very rapid programming. Out of the different XML parser libraries existing for Python, ElementTree stands out because of its pure pythonic approach to the internal XML representation. Using ElementTree is not so straightforward during the first stages of text processing, with complex XML structures usually used to represent typographic information, but it really shines when processing and modifying already tokenised text, with linear sequence of tokens (or other text units represented as data described by XML tags). The approach described is successfully used in the Slovak National Corpus, where Python is the programming language of choice, used at almost all levels of text processing and conversions.

References

[1] http://www.tei-c.org/

[2] Ide, N., Bonhome, P., Romary, L., XCES: *An XML-based Encoding Standard for Linguistic Corpora*. In: Proceedings of the Second International Language Resources and Evaluation conference. Paris, European Language Resources Association (2000)

[3] Zakharov, V., Volkov, V.: *Morphological Tagging of Russian Texts of the XIXth Century*. In: Text, Speech and Dialogue. Proceedings of the 7th International Conference TSD 2004. Brno, Czech Republic: (2004) 235–242

[4] Przepiórkowski, A.: *The IPI PAN Corpus preliminary version*. Warszawa, Instytut Podstaw Informatyki PAN

[5] http://www.python.org/

[6] Rigo, A.: *Representation-based Just-in-time Specialization and the Psyco prototype for Python*. In: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation. Verona, Italy: (2004) 15–16

[7] http://effbot.org/

[8] Hajič, J., Hladká, B.: Czech Language Processing - POS Tagging. In: *Proceedings of the First International Conference on Language Resources and Evaluation*. Granada, Spain: (1998) 931–936

[9] Hajič, J., Hric, J., Kuboň, V.: Machine Translation of Very Close Languages. In: *Proceedings of the ANLP 2000*. Seattle, U.S.A. (2000) 7–12

[10] http://codespeak.net/lxml/

[11] http://xmlsoft.org/