tei_transformer Documentation

Release 0.8.1

Tom McLean

February 15, 2016

Contents

Contents:

CHAPTER 1

Introduction

tei_transformer is a Python script for transforming a TEI-encoded critical edition into a pdf file. There are plenty of XSLT stylesheets to do something like this already, but using Python instead gives a secret advantage: we don't really lose out on anything, but it's unbelievably easy to customise things. We also don't have to restrict ourselves to the xml tree; it's very easy to bring in extra information or shift things about more easily. For example, a trick like adding a lemma note from an external data source for a person mentioned in the edition on their first appearance, then only indexing them on subsequent ones, is trivially easy rather than enormously complicated. (And is, in fact, something we do.)

1.1 Basic Usage

```
tei_transformer example.xml
```

This is pretty simple. The one proviso is that the script expects a folder called `resources` in the same directory as example.xml. This needs to contain a file called `personlist.xml` containing a list (in TEI-format) of people mentioned in the text and a BibLaTex file of references for citations called `references.bib`.

There's also plenty of optional files you can include for things like introductions. You can change things like the filenames of these by providing a file "config.yaml" in resources.

Of course, it's also possible to skip all of this; and fit it into your own chain of events; simply getting a .tex file is as simple as:

```
from tei_transformer.transform import ParserMethods

xmlpath = 'example.xml'
parsed = ParserMethods.parse(xmlpath)
tree = parsed.getroot().find('.//{*}body')
transformed_tree = ParserMethods.transform_tree(tree)
text = '\n'.join(transformed_tree.itertext()).strip()
```

However, your project's assumptions and requirements will almost certainly differ from the default assumptions, and it's definitely a good idea to muck about with things and see what happens. See *Customisation*, or consider just downloading the very simple source and manipulating it as you choose.

1.2 Installation

```
pip install tei_transformer
```

1.3 Requirements

Files are parsed using lxml:

```
pip install lxml
```

The tex file produced needs pdflatex[http://latex-project.org/ftp.html] to produce a pdf file. The installation of tex which you use will also need the `reledmac` package and the Perl script `latexmk`. Most installations will have these in any case.

CHAPTER 2

Customisation

The nature of a critical edition is such that you'll almost certainly have your own special requirements; and the nature of a TEI-encoding scheme is such that things are so very diverse it's hard to make any assumptions about how an encoding works.

Because of this, you are very likely to want to either give new instructions for transforming a particular type of tag, or override the existing ones to meet your own requirements.

Fortunately, the whole reason this project exists is to make it very easy to do so.

2.1 How things work

We use a standard parser, lxml, to make sense of a tei-encoded lxml file. This parser reads tags according to a Python class, TEITag. Each type of tag (`p`, `head`, `q`, etc) is assigned to a class inheriting from TEITag which defines a property, `target` that is the same as the tag's name, and also gives a method `get_replacement`, which is called to replace the tag in the new document with a string.

If the replacement is `None`, a tag is not replaced.

This seems quite complicated – and it can get as complicated as you like – but its usage is very simple. Here is, for example, is more or less the complete class `SoCalled`, which handles tags of the type `<soCalled>`

```
class SoCalled(TEITag):

target = 'soCalled'

def get_replacement(self):
        return "`%s'" % self.text
```

The only point which might need explanation is where `self.text` comes from; it is, of course, the text contained within the tag. Because the class SoCalled inherits from the class `TEITag`, and `TEITag` inherits from the class `LXML.etree.ElementBase`, all the methods available to `ElementBase` can be called to find out more about the tag. See the API documentation under `TEITag` to see what is available. These mean that any information from within the parse tree you want to find out is easily accessible.

For example, getting the attribute 'hello_world' for a tag is as simple as:

```
self.get('hello_world')
```

There is one proviso, though: unlike XSLT, tags are replaced one-at-a-time, rather than simultaneously. To make this a bit more logical, tags are not replaced in document order, but, weakly-sorted, by the number of descendants.

So you can always guarantee that a tag's parent is still accessible (with `self.getparent()`), but its children or siblings may have already been replaced with text.

Several methods are also available for tags beyond those defined by lxml.etree; again, see the API documentation. The big ones are `unwrap()`, which unwraps a tag, and `delete()`, which removes it without replacement.

2.2 Overriding an existing class, or adding a new one

Almost certainly, you'll be wanting to override things.

Tag handling classes are pretty simple, as above. What you want to do is rather than using the command deal with the classes making up the application instead.

Have a look at the source; it's very short and kept deliberately simple rather than hyper-efficient.

CHAPTER 3

API

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

t

T