abbyy_to_epub3 Documentation

Release 1.0

Deborah Kaplan

Nov 10, 2019

CONTENTS:

CHAPTER

ONE

ABBYY XML TO EPUB3

1.1 Introduction

This module transforms ABBYY XML documents, generated by ABBYY FineReader 10, into primitively accessible ePub 3. The code is optimized for ABBYY XML documents created by the Internet Archive, though it may work for other ABBYY XML as well.

CHAPTER

TWO

FEATURES

1. Unicode-compliant
2. Can handle left-to-right and right-to-left text.
3. Attempts to recognize running headers, footers, and decimal or page numbers. Level of confidence in fuzzy matching can be fine tuned in `config.ini`. Errs on the side of minimizing false positives.
4. Will use Kakadu image libraries if present, otherwise will fall back to Pillow.

CHAPTER

THREE

LIMITATIONS

1. Accessibility is inherently limited by the input ABBYY FineReader documents. If they are marked up with headings and other semantic markup, that structure will be incorporated into the ePub.
2. There is currently no functionality for image description.
3. The module can also transform ABBYY XML documents generated by ABBYY FineReader 6. However, those documents are not marked up with headings, so there is no structural navigation for accessibility.

CHAPTER

FOUR

REQUIREMENTS

- Python 3
- If running epubcheck, a Java Runtime environment
- If running DAISY Ace, NodeJS version >= 6.4.0
- If using Kakadu, install the binaries and add the your PATH and LD_LIBRARY_PATH

CHAPTER

FIVE

USAGE

From within a Python program:

```
from abbyy_to_epub3 import create_epub
book = create_epub.Ebook('docname')  # See *Assumptions* below.
book.craft_epub()
```

From the shell:

```
abbyy2epub docname     # See *Assumptions* below.
```

The available command line arguments are:

```
usage: abbyy2epub [-h] [-d] [--epubcheck level] [--ace level] docname

Process an ABBYY file into an EPUB

positional arguments:
  item_dir         The file path where this item's files are kept.
  item_identifier  The unique ID of this item.
  item_bookpath    The prefix to a specific book within an item.In a simple
                   book, usually the same as the item_identifier.

optional arguments:
  -h, --help   show this help message and exit
  -d, --debug  Show debugging information
  --epubcheck  Run EpubCheck on the newly created EPUB, given a severity level
  --ace  Run DAISY Ace on the newly created EPUB, given a severity level
```

CHAPTER

SIX

SYSTEM DEPENDENCIES

Epubcheck: If you'd like to run epubcheck, there are certain system dependencies. Depending on running environment, these may need to be manually installed. On Ubuntu, I installed these with:

```
sudo apt-get install default-jre libpython3-dev
```

DAISY Ace: If you'd like to run Ace, there are certain system dependencies. Read the installation instructions, but in a nutshell:

- Install NodeJS. *Important*: You need at least version 6.4.0, which is newer than the version in the package manager for many distributions. (E.g. versions of Ubuntu before 17.10 Artful). If you have an older version on your system and you can't upgrade, consider running NodeJS in an isolated environment such as nodeenv.
- Install Ace:

```
npm install @daisy/ace -g
```

- Create a configuration file for the user account who'll be running the code, in *~/.config/DAISY Ace/*. You can modify the configuration *per the documentation <https://daisy.github.io/ace/docs/config/>_* but be sure to add this block:

```
{
    "cli": {
        "return-2-on-validation-error": true
    }
}
```

CHAPTER

SEVEN

INSTALLATION

This package can be installed on your local system. From the directory containing setup.py:

```
pip install -r requirements.txt
python setup.py develop
pip install .
```

You can rebuild the documentation, which is generated with Sphinx.

```
cd docs
make html
```

CHAPTER

EIGHT

DEPLOYING AT THE INTERNET ARCHIVE

Before deploying, make sure you bump the version of the package in *__init__.py*. Then, run the *upload.sh* script in the root of the repository and enter the appropriate Internet Archive credentials when prompted.

You can test that the package has been installed correctly by going to https://devpi.archive.org or by running *$ pip3 install –upgrade -i https://petaboxdevpi:{PASSWORD}@devpi.archive.org/books/formats abbyy_to_epub3*.

Note that *petaboxdevpi:{PASSWORD}* is not needed inside IA network‘

CHAPTER

NINE

TESTING

Run `py.test` from the top-level app directory. Create new tests in the `tests` subdirectory.

CHAPTER

TEN

ASSUMPTIONS

An item may contain 1 or more books. In order to accommodate this subtlety and delineate between books, an *item_dir* and *item_identifier* are not sufficient to isolate a specific book. To circumvent this limitation, we require another identifier called the *item_bookpath* which acts as a prefix to the files of a specific book. Given a datanode and an *item_dir* of an item, all the constituent files for a book can be constructed using *item_identifier* and *item_bookpath* in the following ways:

- The *item_identifier* (the unique ID of this item)
- The *item_dir* is the file path where this items files are kept
- The *item_bookpath* is name of the particular book file, often the same as *item_identifier*

The structure is assumed to be:

- `scandata.xml` describes the structure of the book (metadata, pages numbers)
- `docname_abbyy.gz` unzips to `docname_abbyy`, an XML file generated by ABBYY.
- `docname_jp2.zip` unzips to a directory called `docname_jp2`, which includes a number of documents in the format `docname_####.jp2`.
- The scandata has hopefully marked up one leaf as 'Cover'. Failing that, we will use the first leaf marked 'Title', and failing that, the first leaf marked 'Normal'.
- There is a single global metadata manifest file for the entire item named `{item_identifier}_meta.xml`.
- All of the other book specific files follow the form `{item_bookpath}_{file}`. e.g. `{item_bookpath}_abbyy.gz`

CHAPTER

ELEVEN

FURTHER READING

Module documentation is available at Read The Docs.

CHAPTER

TWELVE

CONTRIBUTE

- Source code on GitHub
- Issue tracker

12.1 abbyy_to_epub3

12.1.1 abbyy_to_epub3 package

Submodules

abbyy_to_epub3.constants module

abbyy_to_epub3.create_epub module

class abbyy_to_epub3.create_epub.**ArchiveBookItem**(*item_dir*, *item_identifier*, *item_bookpath*)

Bases: object

Archive.org is a website which contains an archive of items composed of archived digital content. Archive.org items are distributed across a cluster of machines called datanodes. In order to access the files of an item, you need to know 4 things:

a) The Archive.org *item_identifier* (the unique ID of this item) e.g. https://archive.org/details/{item_identifier}

b) the datanode server address which hosts this item

c) the *item_dir* which is the file path on this datanode where this items files are kept

d) the name of the files within this *item_dir*

Certain archive.org items are specifically structured (file organizations, contents, names) to store and play Books. Every Archive Book Item contains the following files: - a jp2.zip containing all the scanned images of the book - an abbyy file containing the OCR'd plaintest of these scans - scandata.xml whose metadata describes the structure of the book

(metadata, pages numbers)

- meta.xml which describes the entire archive.org *item*

A complication is that Archive.org Book Items may contain 1 or more books. In order to accommodate this subtlety and delineate between books, an *item_dir* and *item_identifier* are not sufficient to isolate a specific book. To circumvent this limitation, we require another identifier called the *item_bookpath* which acts as a prefix to

the files of a specific book. Given a datanode and an *item_dir* of an Archive Book Item, all the constituent files for a book can be constructed using *item_identifier* and *item_bookpath* in the following ways:

- There is a single global metadata manifest file for the entire Archive Item named *{item_identifier}_meta.xml*.
- All of the other book specific files follow the form *{item_bookpath}_{file}*. e.g. *{item_bookpath}_abbyy.gz*

class abbyy_to_epub3.create_epub.**Ebook**(*item_dir*, *item_identifier*, *item_bookpath*, *debug=False*, *epubcheck=None*, *ace=None*)

Bases: *abbyy_to_epub3.create_epub.ArchiveBookItem*

Ebook is a utility for generating epub3 files based on Archive.org items. Holds extracted information about a book & the ebooklib EPUB object.

DEFAULT_ACE_LEVEL = 'minor'

DEFAULT_EPUBCHECK_LEVEL = 'warning'

craft_epub(*epub_outfile='out.epub'*, *tmpdir=None*)
Assemble the extracted metadata & text into an EPUB

craft_html()
Assembles the XHTML content.

Create some minimal navigation: * Break sections at text elements marked role: heading * Break files at any headings with roleLevel: 1 Imperfect, but better than having no navigation or monster files.

Images will get alternative text of "Picture #" followed by an index number for this image. Barring real alternative text for true accessibility, this at least adds some identifying information.

create_accessibility_metadata()
Set up accessibility metadata

extract_cover()
http://web.archive.org/web/20180416230000/https://www.safaribooksonline.com/blog/2009/11/20/best-practices-in-epub-cover-images/

extract_images()
Extracts all of the images for the text.

For efficiency's sake, do these all at once. Memory & CPU will be at a higher premium than disk space, so unzip the entire scan file into temp directory, instead of extracting only the needed images.

get_cover_leaf()
Try to find a cover image. If nothing is tagged as 'Cover', use the first page tagged 'Title'. If nothing is tagged as 'Title', either, use the first page tagged 'Normal'. Self.pages is an OrderedDict so break as soon as you find something useful, and don't search the whole set of pages.

identify_headers_footers_pagenos(*placement*)
Attempts to identify the presence of headers, footers, or page numbers

1. Build a dict of first & last lines, indexed by page number.
2. Try to identify headers and footers.

Headers and footers can appear on every page, or on alternating pages (for example if one page has header of the title, the facing page might have the header of the chapter name).

They may include a page number, or the page number might be a standalone header or footer.

The presence of headers and footers in the document does not mean they appear on every page (for example, chapter openings or illustrated pages sometimes don't contain the header/footer, or contain a modified version, such as a standalone page number).

Page numbers may be in Arabic or Roman numerals.

This method does not attempt to look for all edge cases. For example, it will not find: - constantly varied headers, as in a dictionary - page numbers that don't steadily increase - page numbers misidentified in the OCR process, eg. IO2 for 102 - page numbers with characters around them, eg. '~ 45 ~'

image_dim(*block*)
Given a dict object containing the block info for an image, generate a tuple of its dimensions: (left, top, right, bottom)

images_are_extracted()

is_header_footer(*block*, *placement*)
Given a block and our identified text structure, return True if this block's text is a header, footer, or page number to be ignored, False otherwise.

load_scandata_pages()
Parse the page-by-page scandata file. This stores page size, right or left leaf, and page type (eg copyright, color card, etc).

make_chapter(*heading*)
Create a chapter section in an ebooklib.epub.

make_image(*block*)
Given a dict object containing the block info for an image, generate the image HTML

set_metadata()
Set the metadata on the epub object

validate_a11y(*epub_file*, *level=None*)
Individual test failures are logged in EARL syntax https://daisy.github.io/ace/docs/report-json/ Structurally: "assertions": [

{ "@type": "earl:assertion", "earl:result": {

"earl:outcome": "fail"

}, "assertions": [

{ "@type": "earl:assertion", "earl:result": {

"earl:outcome": "fail", "html": "[The invalid HTML]"

}, "earl:test": {

"earl:impact": "serious", "help": {

"dct:description": "[Plain language error]"

},

}

}

], "earl:testSubject": {

"url": "cover.xhtml",

},

},

]

validate_epub(*epub_file*, *level=None*)

abbyy_to_epub3.parse_abbyy module

class abbyy_to_epub3.parse_abbyy.**AbbyyParser**(*document*, *metadata_file*, *metadata*, *paragraphs*, *blocks*, *debug=False*)

Bases: object

The ABBYY parser object. Parses ABBYY metadata in preparation for import into an EPUB 3 document.

And ABBYY document begins with a font and style information:

```
<documentData>
  <paragraphStyles>
    <paragraphStyle
      id="{idnum}" name="stylename"
      mainFontStyleId="{idnum}" [style info]>
    <fontStyle id="{idnum}" [style info]>
  </paragraphStyle>
  [more styles]
</documentData>
```

This is followed by the data for the pages.

```
<page>
    <block></block>
    [more blocks]
</page>
```

Blocks have types. We process types Text, Picture, and Table.

Text:

```
<page>
        <region>
        <text> contains a '\n' as a text element
        <par> The paragraph, repeatable
            <line> The line, repeatable
                <formatting>
                <charParams>: The individual character
```

Picture: we know the corresponding scan (page) number, & coordinates.

Table:

```
<row>
  <cell>
    <text>
      <par>
```

Each <*par*> has an identifier, which has a unique style, including the paragraph's role, eg:

```
<par align="Right" lineSpacing="1790"
    style="{000000DD-016F-0A36-032F-EEBBD9B8571E}">
```

This corresponds to a paragraphStyle from the <*documentData*> element:

```
<paragraphStyle
    id="{000000DD-016F-0A36-032F-EEBBD9B8571E}"
    name="Heading #1|1"
    mainFontStyleId="{000000DE-016F-0A37-032F-176E5F6405F5}"
```

(continues on next page)

(continued from previous page)

```
    role="heading" roleLevel="1"
    [style information]>
```

The roles map as follows:

Role name	role
Body text	text
Footnote	footnote
Header or footer	rt
Heading	heading
Other	other
Table caption	tableCaption
Table of contents	contents

etree = ''

find_namespace()
find the namespace of an XML document. Assumes that the namespace of the first element in the context is the namespace we need. This is more memory-efficient then parsing the entire tree to get the root node.

is_block_type(*blockattr*, *blocktype*)
Identifies if a block has the given type.

ns = ''

nsm = ''

parse_abbyy()
Parse the ABBYY into a format useful for *create_epub*. Process the the elements we will need to construct the EPUB: *paragraphStyle*, *fontStyle*, and *page*. We traverse the entire tree twice with *iterparse*, because lxml builds the whole node tree in memory for even tag-selective *iterparse*, & if we don't traverse the whole tree, we can't delete the unowned nodes. *fast_iter* makes the process speedy, and the dual processing saves on memory. Because of the layout of the elements in the ABBYY file, it's too complex to do this in a single iterative pass.

parse_block(*block*)
Parse a single block on the page.

parse_metadata()
Parse out the metadata from the _meta.xml file

process_pages(*elem*)
Iteratively process pages from the ABBYY file. We have to process now rather than copying the pages for later processing, because deepcopying an lxml element replicates the entire tree. The ABBYY seems to be sometimes inconsistent about whether these elements have a namespace, so be forgiving.

process_styles(*elem*)
Iteratively parse styles from the ABBYY file into data structures. The ABBYY seems to be sometimes inconsistent about whether these elements have a namespace, so be forgiving.

version = ''

abbyy_to_epub3.parse_abbyy.**add_last_text**(*blocks*, *page*)
Given a list of blocks and the page number of the last page in the list, mark up the last text block for that page in the list, if it exists.

abbyy_to_epub3.utils module

abbyy_to_epub3.utils.**dirtify_xml**(*text*)
Re-adds forbidden entities to any XML string. Could cause problems in the unlikely event the string literally should be '&amp'

abbyy_to_epub3.utils.**fast_iter**(*context*, *func*)
Garbage collect as you iterate to save memory Based on StackOverflow modification of Liza Daly's fast_iter

abbyy_to_epub3.utils.**gettext**(*elem*)
Given an element, get all text from within element and its children. Strips out file artifact whitespace (unlike etree.itertext).

abbyy_to_epub3.utils.**is_increasing**(*l*)
Given a list, return True if the list elements are monotonically increasing, and False otherwise.

abbyy_to_epub3.utils.**sanitize_xml**(*text*)
Removes forbidden entities from any XML string

Module contents

CHAPTER

THIRTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

INDEX

N

P

S

V