PyDocX Documentation

Release dev

PyDocX Team

Jul 06, 2017

Contents

CHAPTER 1

Installation

Python & OS Support

PyDocX is supported and tested with CPython versions 2.6, 2.7, 3.3, 3.4, and pypy.

PyDocX is supported and tested with Linux and Windows.

Install using pip

```
$ pip install pydocx
```

Upgrade using pip

```
$ pip install -U pydocx
```

CHAPTER 2

Usage

Converting files using the command line interface

Using the `pydocx` command, you can specify the output format with the input and output files:

```
$ pydocx --html input.docx output.html
```

Converting files using the library directly

If you don't want to mess around having to create exporters, you can use the `PyDocX.to_html` helper method:

```
from pydocx import PyDocX

# Pass in a path
html = PyDocX.to_html('file.docx')

# Pass in a file object
html = PyDocX.to_html(open('file.docx', 'rb'))

# Pass in a file-like object
from cStringIO import StringIO
buf = StringIO()
with open('file.docx') as f:
   buf.write(f.read())

html = PyDocX.to_html(buf)
```

Of course, you can do the same using the exporter class:

```
from pydocx.export import PyDocXHTMLExporter

# Pass in a path
```

```
exporter = PyDocXHTMLExporter('file.docx')
html = exporter.export()

# Pass in a file object
exporter = PyDocXHTMLExporter(open('file.docx', 'rb'))
html = exporter.export()

# Pass in a file-like object
from cStringIO import StringIO
buf = StringIO()
with open('file.docx') as f:
   buf.write(f.read())

exporter = PyDocXHTMLExporter(buf)
html = exporter.export()
```

Currently Supported HTML elements

- tables
  - nested tables
  - rowspans
  - colspans
  - lists in tables
- lists
  - list styles
  - nested lists
  - list of tables
  - list of pragraphs
- justification
- images
- styles
  - bold
  - italics
  - underline
  - hyperlinks
- headings

HTML Styles

The export class `pydocx.export.PyDocXHTMLExporter` relies on certain CSS classes being defined for certain behavior to occur.

Currently these include:

- class `pydocx-insert` -> Turns the text green.
- class `pydocx-delete` -> Turns the text red and draws a line through the text.
- class `pydocx-center` -> Aligns the text to the center.
- class `pydocx-right` -> Aligns the text to the right.
- class `pydocx-left` -> Aligns the text to the left.
- class `pydocx-comment` -> Turns the text blue.
- class `pydocx-underline` -> Underlines the text.
- class `pydocx-caps` -> Makes all text uppercase.
- class `pydocx-small-caps` -> Makes all text uppercase, however truly lowercase letters will be small than their uppercase counterparts.
- class `pydocx-strike` -> Strike a line through.
- class `pydocx-hidden` -> Hide the text.
- class `pydocx-tab` -> Represents a tab within the document.

Additionally, several list styles are defined based off the attribute values listed at: http://officeopenxml.com/WPnumbering-numFmt.php

- class `pydocx-list-style-type-cardinalText` -> (1, 2, 3, 4, etc.)
- class `pydocx-list-style-type-decimal` -> (1, 2, 3, 4, etc.)
- class `pydocx-list-style-type-decimalEnclosedCircle` -> (1, 2, 3, 4, etc.)
- class `pydocx-list-style-type-decimalEnclosedFullstop` -> (1, 2, 3, 4, etc.)
- class `pydocx-list-style-type-decimalEnclosedParen` -> (1, 2, 3, 4, etc.)
- class `pydocx-list-style-type-decimalZero` -> (01, 02, 03, etc.)
- class `pydocx-list-style-type-lowerLetter` -> (a, b, c, etc.)
- class `pydocx-list-style-type-lowerRoman` -> (i, ii, iii, etc.)
- class `pydocx-list-style-type-none` -> List style is removed
- class `pydocx-list-style-type-ordinalText` -> (1, 2, 3, 4, etc.)
- class `pydocx-list-style-type-upperLetter` -> (A, B, C, etc.)
- class `pydocx-list-style-type-upperRoman` -> (I, II, III, etc.)

Exceptions

There is only one custom exception (`MalformedDocxException`). It is raised if either the `xml` or `zipfile` libraries raise an exception.

CHAPTER 3

Conformance

Open Office XML is standardized by ECMA-376.

To the greatest degree possible, PyDocX intends to conform with this and subsequent standards.

17.9 Numbering

Section	Description	Implemented
17.9.1	abstractNum	true
17.9.2	abstractNumId	true
17.9.3	ilvl	true
17.9.4	isLgl	false
17.9.5	lvl (override)	false
17.9.6	lvl	true
17.9.7	lvlJc	false
17.9.8	lvlOverride	false
17.9.9	lvlPictPulletId	false
17.9.10	lvlRestart	false
17.9.11	lvlText	false
17.9.12	multiLevelType	false
17.9.13	name	false
17.9.14	nsid	false
17.9.15	num	true
17.9.16	numbering	true
17.9.17	numFmt	true
17.9.18	numId	true
17.9.19	numIdMacAtCleanup	false
17.9.20	numPicBullet	false
17.9.21	numStyleLink	false
17.9.22	pPr	false
17.9.23	pStyle	false
17.9.24	rPr	false
17.9.25	start	false
17.9.26	startOverride	false
17.9.27	styleLink	false
17.9.28	suff	false
17.9.29	tmpl	false

Deviations

In some cases, it was necessary to deviate from the specification. Such deviations should be only done with justification, and minimally. All intended deviations shall be documented here. Any undocumented deviations are bugs.

Missing val attribute in underline tag

- In the event that the `val` attribute is missing from a `u` (`ST_Underline` type), we treat the underline as off, or none. See also http://msdn.microsoft.com/en-us/library/ff532016%28v=office.12%29.aspx

  > If the val attribute is not specified, Word defaults to the value defined in the style hierarchy and then to no underline.

CHAPTER 4

Extending PyDocX

Customizing the HTML Exporter

Basic HTML exporting is implemented in `pydocx.export.html.PyDocXHTMLExporter`. To override default behavior, simply extend the class and implement the desired methods. Here are a few examples to show you what is possible:

```
class MyPyDocXHTMLExporter(PyDocXExporter):

    def __init__(self, path):
        # Handle dstrike the same as italic
        self.export_run_property_dstrike = self.export_run_property_italic

        super(MyPyDocXHTMLExporter, self).__init__(path=path)

    # Perform specific pre-processing
    def export(self):
        self.delete_only_FOO_text_nodes()
        return super(MyPyDocXHTMLExporter, self).export()

    def delete_only_FOO_text_nodes(self):
        # Delete all text nodes that match 'FOO' exactly
        document = self.main_document_part.document
        for body_child in document.body.children:
            if isinstance(body_child, wordprocessing.Paragraph):
                paragraph = body_child
                for paragraph_child in paragraph.children:
                    if isinstance(paragraph_child, wordprocessing.Run):
                        run = paragraph_child
                        for run_child in run.children[:]:
                            if isinstance(run_child, wordprocessing.Text):
                                text = run_child
                                if text.text == 'FOO'
                                    run.children.remove(text)
```

```
    # Don't display head
    def head(self):
        return
        # The exporter expects all methods to return a generator
        yield  # this syntax causes an empty generator to be returned

    # Ignore page break
    def get_break_tag(self, br):
        if br.is_page_break():
            pass
        else:
            return super(MyPyDocXHTMLExporter, self).get_break_tag(br)

    # Do not show deleted runs
    def export_deleted_run(self, deleted_run):
        return
        yield

    # Custom table tag
    def get_table_tag(self, table):
        attrs = {
            'class': 'awesome-table',
        }
        return HtmlTag('table', **attrs)

    # By default, the HTML exporter wraps inserted runs in a span with
    # class="pydocx-insert". This example overrides that method to skip
    # that behavior by jumping to the base implementation.
    def export_inserted_run(self, inserted_run):
        return super(PyDocXExporter, self).export_inserted_run(inserted_run)

    # Hide hidden runs
    def export_run(self, run):
        properties = run.effective_properties
        if properties.vanish:
            return
        elif properties.hidden:
            return
        results = super(MyPyDocXHTMLExporter, self).export_run(run)
        for result in results:
            yield result
```

Implementing a new exporter

If you want to implement an exporter for an unsupported markup language, you can do that by extending `pydocx.export.base.PyDocXExporter` as needed. For example, this shows how you might create a custom exporter for the FML, or Foo Markup Language:

```
class PyDocXFOOExporter(PyDocXExporter):

    # The "FOO" markup language denotes breaks using "\"
    def export_break(self):
        yield '\\'

    def export_document(self, document):
```

```
        yield 'START OF DOC'
        results = super(PyDocXFOOExporter, self).export_document(self, document)
        for result in results:
            yield result
        yield 'END OF DOC'

    # Text must be wrapped in ()
    def export_text(self, text):
        yield '({0})'.format(text.text)

    # Tables are denoted by [ ]
    def export_table(self, table):
        yield '['
        results = super(PyDocXFOOExporter, self).export_table(self, table)
        for result in results:
            yield result
        yield ']'

    # Table rows are denoted by { }
    def export_table_row(self, table_row):
        yield '{'
        results = super(PyDocXFOOExporter, self).export_table_row(self, table_row)
        for result in results:
            yield result
        yield '}'

    # Table cells are denoted by < >
    def export_table_row(self, table_cell):
        yield '<'
        results = super(PyDocXFOOExporter, self).export_table_cell(self, table_cell)
        for result in results:
            yield result
        yield '>'
```

The base exporter implementation expects all methods to return a generator. For this reason, it is not possible to have an empty method (`pass`) or have a method that just returns `None`. The one caveat is the special syntax that causes a method to return an empty generator:

```
def empty_generator():
    return
    yield
```

This implementation is consistent with the "only generators" rule, and is actually computationally faster than returning an empty list.

CHAPTER 5

Export Mixins

Export mixins provide standardized optional overrides for specific use cases. They exist in `pydocx.export.mixins`. Each mixin is defined as a class in its own module.

Detect faked superscript and subscript

Useful if you want runs of text that are styled smaller (relative to surrounding text) and positioned either above or below the surrounding text to be treated as super/subscript.

Example usage:

```
from pydocx.export.mixins import FakedSuperscriptAndSubscriptExportMixin


class CustomExporter(
    FakedSuperscriptAndSubscriptExportMixin,
    PyDocXHTMLExporter,
):
    pass
```

CHAPTER 6

Enumerated List Detection

The default behavior in PyDocX is to convert “faked” enumerated lists into “real” enumerated lists.

A “faked” enumerated list is a sequence of paragraphs in which the numbering has been explicitly typed. Additionally, the spacing across levels is manually set either using tab characters, or indentation. For example:

```
1. Apple
2. Banana
   a. Chiquita
   b. Dole
3. Carrot
```

Conversely, a “real” enumerated list is a sequence of paragraphs in which the numbering, and spacing, is automatic:

1. Apple
2. Banana
    (a) Chiquita
    (b) Dole
3. Carrot

Supported enumeration sequences

- arabic numberals: 1, 2, 3, ...
- uppercase alphabet characters A, B, C, ..., Z, AA, AB, ... AZ, ...
- lowercase alphabet characters a, b, c, ..., z, aa, ab, ... az, ...
- uppercase Roman numberals: I, II, III, IV, ...
- lowercase Roman numberals: i, ii, iii, iv, ...

Supported enumeration patterns

- Digit followed by a dot plus space: “1. ”, “A. ”, “a. ”, “I. ”, “i. “
- Surrounded by parentheses: “(1)”, “(A)”, “(a)”, “(I)”, “(i)”
- Digit followed by a parenthesis: “1)”, “A)”, “a)”, “I)”, “i)”

How to disable enumerated list detection

Extend the exporter to override the `numbering_span_builder_class` class variable as follows:

```
from pydocx.export.numbering_span import BaseNumberingSpanBuilder

class CustomExporter(PyDocXHTMLExporter):
    numbering_span_builder_class = BaseNumberingSpanBuilder
```

CHAPTER 7

Development

Installing requirements

Using pip

```
$ pip install -r requirements/docs.txt -r requirements/testing.txt
```

Using terrarium

Terrarium will package up and compress a virtualenv for you based on pip requirements and then let you ship that environment around.

```
$ terrarium install requirements/*.txt
```

Building the documentation locally

1. Install the documentation requirements:

   ```
   $ pip install -r requirements/docs.txt
   ```

2. Change directory to `docs` and run `make html`:

   ```
   $ cd docs
   $ make html
   ```

3. Load HTML documentation in a web browser of your choice:

   ```
   $ firefox docs/_build/html/index.html
   ```

Running tests

1. Install the development requirements:

```
$ pip install -r requirements/testing.txt
```

2. Run `make test lint` in the project root. This will run `nosetests` with coverage and also display any `flake8` errors.

```
$ make test lint
```

To run all tests against all supported versions of python, use `tox`.

Running tests with tox

`tox` allows us to use one command to run tests against all versions of python that we support.

Setting up tox

1. Decide how you want to manage multiple python versions.
   (a) System level using a package manager such as `apt-get`. This approach will likely require adding additional `apt-get` sources in order to install alternative versions of python.
   (b) Use pyenv to manage and install multiple python versions. After installation, see the pyenv command reference.
2. Install `tox`.

```
$ pip install tox
```

3. Configure tox.

Running tox

Now that you have `tox` setup, you just need to run the command `tox` from the project root directory.

```
$ tox
```

Getting involved

The PyDocX project welcomes help in any of the following ways:

- Making pull requests on github for code, tests and documentation.
- Participating on open issues and pull requests, reviewing changes

Coding Standards

- All python source files **must** be PEP8 compliant.
- All python source files **must** include the following import declaration at the top of the file:

```
from __future__ import (
    absolute_import,
    print_function,
    unicode_literals,
)
```

Unicode Data

- All stream data is assumed to be a UTF-8 bytestream unless specified otherwise. What this means is that when you are writing test cases for a particular function, any input data you define which would have otherwise have come from a file source must be encoded as UTF-8.

Release process

PyDocX adheres to Semantic versioning v2.0.0.

1. Update CHANGELOG.
2. Bump the version number in __init__.py on master.
3. Tag the version.
4. Push to PyPI

```
make release
```

CHAPTER 8

Plugins

You may find yourself needing a feature in PyDocX that doesn't exist in the core library.

If it's something that should exist, the PyDocX project is always open to new contributions. Details of how to contibute can be found in *Development*.

For things that don't fit in the core library, it's easy to build a plugin based on the *Extending PyDocX* and *Export Mixins* sections.

If you do build a plugin, edit this documentation and add it below so that other developers can find it.

Available Plugins

Plugin	Description
pydocx-resize-images	Resizes large images to the dimensions they are in the docx file
pydocx-s3-images	Uploads images to S3 instead of returning Data URIs

CHAPTER 9

Release Notes

dev

- Internal links and anchors are now retained. Thanks, sunu! #222

0.9.10

- No longer error when processing margin positions with decimal points.

0.9.9

- Rect elements now correctly handle image data

0.9.8

- Textboxes can now contain tables.
- Pict elements can now contain Rect elements.

0.9.7

- Text colors other than black and white are no longer ignored
- Textboxes have been implemented. We no longer lose the content inside of them.
- Markup compatibility has been implemented. We always use the Fallback for AlternateContent tags.

0.9.6

- Fixed issue in PyDocX CLI tool and added new test cases for the same

0.9.5

- Simple and Complex field hyperlinks now support bookmarks / internal anchors

0.9.4

- Faked lists inside tables are correctly converted to real lists

0.9.3

- Headings inside a complex field no longer fail to ignore styles

0.9.2

- Fixed issue where multiple complex fields in the same paragraph would cause content to disappear.

0.9.1

- Added EmbeddedObject support with Shape

0.9.0

- Implemented complex and simple field hyperlinks.
- This includes a significant change to the API. The export methods are now all called twice. The results are discarded in the first pass. In first pass (`self.first_pass == True`), you can now track information that will be used to make decisions in the second pass. The notable example where this technique is used is implementing complex fields. Because the export methods are called twice, some exporter extensions that perform lossly operations on the document structure may need to ignore processing during the first pass.
- The function signature of the `get_hyperlink_tag` has changed. It previously accepted a `Hyperlink` instance. Now it only accepts `target_uri`.

0.8.5

- Styled whitespace is no longer ignored. Previously, this would result in certain configurations with words grouped together without spaces.

0.8.4

- Headings now preserve italic, webHidden and vanish styles

0.8.3

- Decimal font sizes are now handled properly

0.8.2

- Paragraphs that have numbering definitions with a level number format of None are no longer considered list items.

0.8.1

- Headings in lists no longer break numbering. By default, in the HTML exporter, headings in lists are represented using the "strong" tag, regardless of the heading level.

0.8.0

- Note: This release consists of significant changes to the internal API and is not backwards compatible with prior versions
- Removed `ConvertRootUpperRomanListToHeadingMixin`
- Fixed issue where the same image referenced multiple times would not display correctly after the first instance
- Removed the preprocessor and re-implemented the functionality into the exporter
- Re-implemented the exporter into a top-down generator algorithm
- Implemented the necessary object classes for each element type (Paragraph, Run, Text, etc)
- Implemented enumerated list detection and conversion to numbering lists

0.7.0

- Added support for python 3.4
- Added support for pypy
- No longer adding list-style-type attribute to ordered list tags. We are now using a class to indicate these.

- Faked sub/super handling is no longer handled by default. Instead, that handling is implemented in a new mixin class. See `pydocx.export.mixins`
- `pydocx.wordml` and `pydocx.openxml` have been merged into `pydocx.openxml.packaging` to better mirror the MS implementation structure.
- `pydocx.models.styles` has been moved to `pydocx.openxml.wordprocessing.*`
- `pydocx.managers.styles` has been merged into `pydocx.openxml.wordprocessing.style_definition_part`
- Added `XmlCollection` field type, now used by `openxml.wordprocessing.styles.Style`
- Implemented several model classes for Numbering.
- Added numbering property to the numbering definitions part.
- XmlModels now define their own tags
- Simplified importing PyDocX
- Header processing now occurs in the exporter rather than the pre-processor
- PyDocXExporter.heading signature has changed from accepting heading_level which was an HTML tag to accepting heading_style_name which is the raw style name of the heading.
- The `convert_root_level_upper_roman` option has been replaced with an optional mixin `pydocx.export.mixins.ConvertRootUpperRomanListToHeadingMixin`.
- Preprocessor no longer manages table membership. Instead, that is handled in the base iterative parser.
- `ConvertRootUpperRomanListToHeadingMixin` would fail for paragraphs that had no properties.

0.6.0

- Moved parsers to export module
- Renamed DocxParser to PyDocXExporter
- Renamed Docx2Html to PyDocXHTMLExporter
- Eliminated all improper usages of the find_first utility function
- Added support for NumberingDefinitionsPart to the WordprocessingDocumentFactory

0.5.1

- Fixed issue #116 - Don't assume the first sz of an rPr actually is a direct child of that rPr.

0.5.0

- Moved CLI to __main__
- Moved tests to root-level module

0.4.4

- Specify charset in rendered HTML
- Added support for using defusedxml to mitigate XML vulnerabilities.

0.4.3

- Allow a file-like object to be passed into the DocXParser constructor.
- Added basic support for footnotes.

0.4.2

- Fixed a problem with calculating image sizes

0.4.01

- Take into account run position and size to apply superscript and subscript tags to runs that would look like they have superscript and subscript tags but are being faked due to positioning and sizing.

0.4.00

- External images are now handled. This causes a backwards incompatible change with all handers related to images.

0.3.23

- Added support for style basedOn property

0.3.22

- Fixed a bug in which the run paragraph mark properties were used as run properties (pPr > rPr within a style definition)
- Fixed a bug in which the run paragraph properties defined a global style identifier, any of those styles defined globally were ignored.
- Fixed a bug which allowed run properties to reference paragraph properties, and paragraph properties to reference run properties. Such instances are now ignored.

0.3.21

- We are once again supporting files that are missing images.

0.3.20

- Fixed a problem with list nesting. We were marking list items as the first list item in error.

0.3.19

- Added support for python 3.3
- Fixed a problem with list nesting with nested sublists that have the same ilvl.

0.3.18

- Fixed an issue with marking runs as underline when they were not supposed to be.

0.3.17

- Fixed path issue on Windows for Zip archives
- Fixed attribute typo when attempting to generate an error message for a missing required resource

0.3.16

- CHANGELOG.md was missing from the MANIFEST in 0.3.15 which would cause the setup to fail.

0.3.15

- Use inline span to define styles instead of div
- Use ems for HTML widths instead of pixels
- If a property value is `off`, it is now considered disabled

0.3.14

- Use paths from `_rels/.rels` instead of hardcoding

0.3.13

- Significant performance gains for documents with a large number of table cells.
- Significant performance gains for large documents.

0.3.12

- Added command line support to convert from docx to either html or markdown.

0.3.11

- The non breaking hyphen tag was not correctly being imported. This issue has been fixed.

0.3.10

- Found and optimized a fairly large performance issue with tables that had large amounts of content within a single cell, which includes nested tables.

0.3.9

- We are now respecting the `<w:tab/>` element. We are putting a space in everywhere they happen.
- Each styling can have a default defined based on values in `styles.xml`. These default styles can be overwritten using the `rPr` on the actual `r` tag. These default styles defined in `styles.xml` are actually being respected now.

0.3.8

- If zipfile fails to open the passed in file, we are now raising `MalformedDocxException` instead of `BadZipFIle`.

0.3.7

- Some inline tags (most notably the underline tag) could have a `val` of `none` and that would signify that the style is disabled. A `val` of `none` is now correctly handled.

0.3.6

- It is possible for a docx file to not contain a `numbering.xml` file but still try to use lists. Now if this happens all lists get converted to paragraphs.

0.3.5

- Not all docx files contain a `styles.xml` file. We are no longer assuming they do.

0.3.4

- It is possible for `w:t` tags to have `text` set to `None`. This no longer causes an error when escaping that text.

0.3.3

- In the event that `cElementTree` has a problem parsing the document, a `MalformedDocxException` is raised instead of a `SyntaxError`

0.3.2

- We were not taking into account that vertical merges should have a continue attribute, but sometimes they do not, and in those cases word assumes the continue attribute. We updated the parser to handle the cases in which the continue attribute is not there.
- We now correctly handle documents with unicode character in the namespace.
- In rare cases, some text would be output with a style when it should not have been. This issue has been fixed.

0.3.1

- Added support for several more OOXML tags including:
  - caps
  - smallCaps
  - strike

  - dstrike
  - vanish
  - webHidden

More details in the README.

0.3.0

- We switched from using stock `xml.etree.ElementTree` to using `xml.etree.cElementTree`. This has resulted in a fairly significant speed increase for python 2.6
- It is now possible to create your own pre processor to do additional pre processing.
- Superscripts and subscripts are now extracted correctly.

0.2.1

- Added a changelog
- Added the version in `pydocx.__init__`
- Fixed an issue with duplicating content if there was indentation or justification on a p element that had multiple t tags.