python-docx-template Documentation

Release 0.9.x

Eric Lapouyade

May 08, 2023

Contents

Quickstart

To install using pip:

```
pip install docxtpl
```

or using conda:

```
conda install docxtpl --channel conda-forge
```

Usage:

```
from docxtpl import DocxTemplate

doc = DocxTemplate("my_word_template.docx")
context = { 'company_name' : "World company" }
doc.render(context)
doc.save("generated_doc.docx")
```

CHAPTER 1

Introduction

This package uses 2 major packages :

- python-docx for reading, writing and creating sub documents
- jinja2 for managing tags inserted into the template docx

python-docx-template has been created because python-docx is powerful for creating documents but not for modifying them.

The idea is to begin to create an example of the document you want to generate with microsoft word, it can be as complex as you want : pictures, index tables, footer, header, variables, anything you can do with word. Then, as you are still editing the document with microsoft word, you insert jinja2-like tags directly in the document. You save the document as a .docx file (xml format) : it will be your .docx template file.

Now you can use python-docx-template to generate as many word documents you want from this .docx template and context variables you will associate.

CHAPTER 2

Jinja2-like syntax

As the Jinja2 package is used, one can use all jinja2 tags and filters inside the word document. Nevertheless there are some restrictions and extensions to make it work inside a word document:

2.1 Restrictions

The usual jinja2 tags, are only to be used inside the same run of a same paragraph, it can not be used across several paragraphs, table rows, runs. If you want to manage paragraphs, table rows and a whole run with its style, you must use special tag syntax as explained in next chapter.

Note:

a 'run' for Microsoft Word is a sequence of characters with the same style. For example, if you create a paragraph with all characters of the same style, MS Word will create internally only one 'run' in the paragraph. Now, if you put in bold a text in the middle of this paragraph, word will transform the previous 'run' into 3 different 'runs' (normal - bold - normal).

2.2 Extensions

2.2.1 Tags

In order to manage paragraphs, table rows, table columns, runs, special syntax has to be used:

```
{%p jinja2_tag %} for paragraphs
{%tr jinja2_tag %} for table rows
{%tc jinja2_tag %} for table columns
{%r jinja2_tag %} for runs
```

By using these tags, python-docx-template will take care to put the real jinja2 tags (without the *p*, *tr*, *tc* or *r*) at the right place into the document's xml source code. In addition, these tags also tell python-docx-template to **remove** the paragraph, table row, table column or run where the tags are located.

For example, if you have this kind of template:

```
{%p if display_paragraph %}
One or many paragraphs
{%p endif %}
```

The first and last paragraphs (those containing `{%p ... %}` tags) will never appear in generated docx, regardless of the `display_paragraph` value.

Here only:

```
One or many paragraphs
```

will appear in generated docx if `display_paragraph` is True, otherwise, no paragraph at all are displayed.

IMPORTANT : Always put space after a starting tag delimiter and a space before the ending one :

Avoid:

```
{%if something%}
{%pif display_paragraph%}
```

Use instead:

```
{% if something %}
{%p if display_paragraph %}
```

IMPORTANT : Do not use `{%p`, `{%tr`, `{%tc` or `{%r` twice in the same paragraph, row, column or run. Example :

Do not use this:

```
{%p if display_paragraph %}Here is my paragraph {%p endif %}
```

But use this instead in your docx template:

```
{%p if display_paragraph %}
Here is my paragraph
{%p endif %}
```

This syntax is possible because MS Word considers each line as a new paragraph (if you do not use SHIFT-RETURN).

2.2.2 Display variables

As part of jinja2, one can used double braces:

```
{{ <var> }}
```

if `<var>` is a string, `\n`, `\a`, `\t` and `\f` will be translated respectively into newlines, new paragraphs, tabs and page breaks

But if `<var>` is a *RichText* object, you must specify that you are changing the actual 'run':

```
{{r <var> }}
```

Note the `r` right after the opening braces.

VERY IMPORTANT : Variables must not contains characters like <, > and `&` unless using *Escaping*

IMPORTANT : Always put space after a starting var delimiter and a space before the ending one :

Avoid:

```
{{myvariable}}
{{rmyrichtext}}
```

Use instead:

```
{{ myvariable }}
{{r myrichtext }}
```

2.2.3 Comments

You can add jinja-like comments in your template:

```
{#p this is a comment as a paragraph #}
{#tr this is a comment as a table row #}
{#tc this is a comment as a table cell #}
```

See tests/templates/comments_tpl.docx for an example.

2.2.4 Split and merge text

- You can merge a jinja2 tag with previous line by using `{%-`
- You can merge a jinja2 tag with next line by using `-%}`

A text containing Jinja2 tags may be unreadable if too long:

```
My house is located {% if living_in_town %} in urban area {% else %} in countryside {
↪% endif %} and I love it.
```

One can use *ENTER* or *SHIFT+ENTER* to split a text like below, then use `{%-` and `-%}` to tell docxtpl to merge the whole thing:

```
My house is located
{%- if living_in_town -%}
 in urban area
{%- else -%}
 in countryside
{%- endif -%}
 and I love it.
```

IMPORTANT : Use an unbreakable space (*CTRL+SHIFT+SPACE*) when a space is wanted at line beginning or ending.

IMPORTANT 2 : `{%- xxx -%}` tags must be alone in a line : do not add some text before or after on the same line.

2.2.5 Escaping delimiters

In order to display `{%`, `%}`, `{{` or `}}`, one can use:

```
{_%, %_}, {_{ or  }_}
```

2.2.6 Tables

Spanning

You can span table cells horizontally in two ways, by using `colspan` tag (see tests/dynamic_table.py):

```
{% colspan <var> %}
```

<*var*> must contain an integer for the number of columns to span. See tests/test_files/dynamic_table.py for an example.

You can also span horizontally within a for loop (see tests/horizontal_merge.py):

```
{% hm %}
```

You can also merge cells vertically within a for loop (see tests/vertical_merge.py):

```
{% vm %}
```

Cell color

There is a special case when you want to change the background color of a table cell, you must put the following tag at the very beginning of the cell:

```
{% cellbg <var> %}
```

<*var*> must contain the color's hexadecimal code *without* the hash sign

CHAPTER 3

RichText

When you use `{{ <var> }}` tag in your template, it will be replaced by the string contained within *var* variable. BUT it will keep the current style. If you want to add dynamically changeable style, you have to use both : the `{{r <var> }}` tag AND a `RichText` object within *var* variable. You can change color, bold, italic, size, font and so on, but the best way is to use Microsoft Word to define your own *character* style (Home tab -> modify style -> manage style button -> New style, select 'Character style' in the form), see example in *tests/richtext.py* Instead of using `RichText()`, one can use its shortcut : `R()`

The `RichText()` or `R()` offers newline, new paragraph, and page break features : just use `\n`, `\a`, `\t` or `\f` in the text, they will be converted accordingly.

There is a specific case for font: if your font is not displayed correctly, it may be because it is defined only for a region. To know your region, it requires a little work by analyzing the document.xml inside the docx template (this is a zip file). To specify a region, you have to prefix your font name this that region and a column:

```
ch = RichText('TEST', font='eastAsia:')
```

Important : When you use `{{r }}` it removes the current character styling from your docx template, this means that if you do not specify a style in `RichText()`, the style will go back to a microsoft word default style. This will affect only character styles, not the paragraph styles (MSWord manages this 2 kind of styles).

IMPORTANT : Do not use 2 times `{{r` in the same run. Use RichText.add() method to concatenate several strings and styles at python side and only one `{{r` at template side.

Important : `RichText` objects are rendered into xml *before* any filter is applied thus `RichText` are not compatible with Jinja2 filters. You cannot write in your template something like `{{r <var>|lower }}`. Only solution is instead to do any filtering into your python code when creating the `RichText` object.

3.1 Hyperlink with RichText

You can add an hyperlink to a text by using a Richtext with this syntax:

```
tpl=DocxTemplate('your_template.docx')
rt = RichText('You can add an hyperlink, here to ')
rt.add('google',url_id=tpl.build_url_id('http://google.com'))
```

Put `rt` in your context, then use `{{r rt}}` in your template

CHAPTER 4

Inline image

You can dynamically add one or many images into your document (tested with JPEG and PNG files). just add {{ <var> }} tag in your template where <var> is an instance of doxtpl.InlineImage:

```
myimage = InlineImage(tpl, image_descriptor='test_files/python_logo.png',␣
↪width=Mm(20), height=Mm(10))
```

You just have to specify the template object, the image file path and optionally width and/or height. For height and width you have to use millimeters (Mm), inches (Inches) or points(Pt) class. Please see tests/inline_image.py for an example.

CHAPTER 5

Sub-documents

A template variable can contain a complex subdoc object and be built from scratch using python-docx document methods. To do so, first, get the sub-document object from your template object, then use it by treating it as a python-docx document object. See example in *tests/subdoc.py*.

Since docxtpl V0.12.0, it is now possible to merge an existing .docx as a subdoc, just specify its path when calling method *new_subdoc()*

```
tpl = DocxTemplate('templates/merge_docx_master_tpl.docx')
sd = tpl.new_subdoc('templates/merge_docx_subdoc.docx')
```

See *tests/merge_docx.py* for full code.

CHAPTER 6

Escaping

By default, no escaping is done : read carefully this chapter if you want to avoid crashes during docx generation.

When you use a `{{ <var> }}`, under the hood, you are modifying an **XML** word document, this means you cannot use all chars, especially <, > and `&`. In order to use them, you must escape them. There are 4 ways :

- `context = { 'var':R('my text') }` and `{{r <var> }}` in the template (note the `r`),
- `context = { 'var':'my text'}` and `{{ <var>|e }}` in your word template
- `context = { 'var':escape('my text')}` and `{{ <var> }}` in the template.
- enable autoescaping when calling render method: `tpl.render(context, autoescape=True)` (default is autoescape=False)

See tests/escape.py example for more informations.

Another solution, if you want to include a listing into your document, that is to escape the text and manage `\n`, `\a`, and `\f` you can use the `Listing` class :

in your python code:

```
context = { 'mylisting':Listing('the listing\nwith\nsome\nlines \a and some paragraph␣
↪\a and special chars : <>&') }
```

in your docx template just use `{{ mylisting }}`

With `Listing()`, you will keep the current character styling (except after a `\a` as you start a new paragraph).

CHAPTER 7

Replace docx pictures

It is not possible to dynamically add images in header/footer, but you can change them. The idea is to put a dummy picture in your template, render the template as usual, then replace the dummy picture with another one. You can do that for all medias at the same time. Note: the aspect ratio will be the same as the replaced image Note2 : Specify the filename that has been used to insert the image in the docx template (only its basename, not the full path)

Syntax to replace dummy_header_pic.jpg:

```
tpl.replace_pic('dummy_header_pic.jpg','header_pic_i_want.jpg')
```

The replacement occurs in headers, footers and the whole document's body.

CHAPTER 8

Replace docx medias

It is not possible to dynamically add other medias than images in header/footer, but you can change them. The idea is to put a dummy media in your template, render the template as usual, then replace the dummy media with another one. You can do that for all medias at the same time. Note: for images, the aspect ratio will be the same as the replaced image Note2 : it is important to have the source media files as they are required to calculate their CRC to find them in the docx. (dummy file name is not important)

Syntax to replace dummy_header_pic.jpg:

```
tpl.replace_media('dummy_header_pic.jpg','header_pic_i_want.jpg')
```

WARNING : unlike replace_pic() method, dummy_header_pic.jpg MUST exist in the template directory when rendering and saving the generated docx. It must be the same file as the one inserted manually in the docx template. The replacement occurs in headers, footers and the whole document's body.

CHAPTER 9

Replace embedded objects

It works like medias replacement, except it is for embedded objects like embedded docx.

Syntax to replace embedded_dummy.docx:

```
tpl.replace_embedded('embedded_dummy.docx','embedded_docx_i_want.docx')
```

WARNING : unlike replace_pic() method, embedded_dummy.docx MUST exist in the template directory when rendering and saving the generated docx. It must be the same file as the one inserted manually in the docx template. The replacement occurs in headers, footers and the whole document's body.

Note that *replace_embedded()* may not work on other documents than embedded docx. Instead, you should use zipname replacement:

```
tpl.replace_zipname(
    'word/embeddings/Feuille_Microsoft_Office_Excel1.xlsx',
    'my_excel_file.xlsx')
```

The zipname is the one you can find when you open docx with WinZip, 7zip (Windows) or unzip -l (Linux). The zipname starts with "word/embeddings/". Note that the file to be replaced is renamed by MSWord, so you have to guess a little bit...

This works for embedded MSWord file like Excel or PowerPoint file, but won't work for others like PDF, Python or even Text files : For these ones, MSWord generate an oleObjectNNN.bin file which is no use to be replaced as it is encoded.

CHAPTER 10

Get Defined Variables

In order to get the missing variables after rendering use

```
tpl=DocxTemplate('your_template.docx')
tpl.render(context_dict)
set_of_variables = tpl.get_undeclared_template_variables()
```

IMPORTANT : You may use the method before rendering to get a set of keys you need, e.g. to be prompted to a user or written in a file for manual processing.

CHAPTER 11

Multiple rendering

Since v0.15.0, it is possible to create `DocxTemplate` object once and call `render(context)` several times. Note that if you want to use replacement methods like `replace_media()`, `replace_embedded()` and/or `replace_zipname()` during multiple rendering, you will have to call `reset_replacements()` at rendering loop start.

CHAPTER 12

Microsoft Word 2016 special cases

MS Word 2016 will ignore `\t` tabulations. This is special to that version. Libreoffice or Wordpad do not have this problem. The same thing occurs for line beginning with a jinja2 tag providing spaces : They will be ignored. To solve these problem, the solution is to use Richtext:

```
tpl.render({
    'test_space_r' : RichText('          '),
    'test_tabs_r': RichText(5*'\t'),
})
```

And in your template, use the {{r notation:

```
{{r test_space_r}} Spaces will be preserved
{{r test_tabs_r}} Tabs will be displayed
```

CHAPTER 13

Jinja custom filters

`render()` accepts `jinja_env` optional argument : you may pass a jinja environment object. By this way you will be able to add some custom jinja filters:

```
from docxtpl import DocxTemplate
import jinja2

def multiply_by(value, by):
   return value * by

doc = DocxTemplate("my_word_template.docx")
context = { 'price_dollars' : 5.00 }
jinja_env = jinja2.Environment()
jinja_env.filters['multiply_by'] = multiply_by
doc.render(context,jinja_env)
doc.save("generated_doc.docx")
```

Then in your template, you will be able to use:

```
Euros price : {{ price_dollars|multiply_by(0.88) }}
```

CHAPTER 14

Command-line execution

One can use *docxtpl* module directly on command line to generate a docx from a template and a json file as a context:

```
usage: python -m docxtpl [-h] [-o] [-q] template_path json_path output_filename

Make docx file from existing template docx and json data.

positional arguments:
  template_path    The path to the template docx file.
  json_path        The path to the json file with the data.
  output_filename  The filename to save the generated docx.

optional arguments:
  -h, --help       show this help message and exit
  -o, --overwrite  If output file already exists, overwrites without asking
                   for confirmation
  -q, --quiet      Do not display unnecessary messages
```

See tests/module_execute.py for an example.

CHAPTER 15

Examples

The best way to see how it works is to read examples, they are located in *tests/* directory. Docx test templates are in *tests/templates/*. To generate final docx files:

```
cd tests/
python runtests.py
```

Generated files are located in *tests/output* directory.

If you are not sure about your python environment, python-docx-template provides Pipfiles for that:

```
pip install pipenv (if not already done)
cd python-docx-template (where Pipfiles are)
pipenv install --python 3.6 -d
pipenv shell
cd tests/
python runtests.py
```

CHAPTER 16

Share

If you like this project, please rate and share it here : http://rate.re/github/elapouya/python-docx-template

Functions index

Functions documentation

CHAPTER 17

Indices and tables

- genindex
- modindex
- search