Paper SM07 – PHUSE EU 2020

Python for Word Processing

Katja Glaß, Bayer AG, Berlin, Germany

ABSTRACT

Programming languages like Python can provide functionality which is hardly available for implementation in SAS as available open source tools are available. Utilizing such functions will enable opportunities which had not been available before.

This paper will describe how Bayer is implementing a Word Processing tool to automate creation of Word documents. The tool had been developed for creating Patient Narratives but will support other use cases as well. The paper will show how such a tool can be implemented and how a SAS macro interface to this functionality can be provided.

INTRODUCTION

Processes should be automated as much as possible to enable higher efficiency. Standardizations are very important to support automation. By having standard processes and tools, a global maintenance and enhancements of tools can be provided. Then all users can benefit from this as updates could be made available to all users through standard processes.

The SAS environment with many core functions is ideal to perform data processing including mappings, perform analysis and create reports. The options to create macros and functions allow tool developments which can put complex tasks into simple functionality blocks. SAS also provides LUA and Java integrations. With Viya many other languages can easily be integrated into the SAS environment.

Why implementing something in another programming language? Some tasks can be performed more efficient in other languages like Python provides many machine learning options and modules. Code complexity and maintenance might be easier in other languages as well. But the major argument to use another language is that there are many open source tools available, which provide huge functionality blocks. Using such tools can save a lot of time and effort because you don't need to re-implement them in SAS.

THE USE CASE

There had been a project to create patient narratives. Patient narratives are summaries typically about serious adverse event reactions of patients where the information should be available in written text form and not as a table. Next to textual descriptions some key information should be displayed. Sometimes tables should also be included to provide a better overview of information.

Subject identifier: 100012345

Age: 84

Sex: Male

Race: White

Investigational product administered: Treatment

Date of randomization: 01JUL2018

Narrative criteria: SAEs leading to permanent discontinuation of study medication

Country: Germany

Exposure		
Actual treatment	Start date	End date
xxx	xxx	xxx
xxx	xxx	xxx

Subject 100012345 had the following diagnosis or diagnoses: xxx, xxx and xxx. Details of all medical history and concomitant medications of interest are tabulated below.

The subject received the first dose of treatment on 10JUL2018. Details of study treatment intake including titrations are displayed below. Details on vital signs are also provided below.

On 10DEC2018, xxx day(s) after starting study treatment, the subject was reported with **lung neoplasm malignant** which the investigator considered serious because it met the following seriousness criterion or criteria: "Other medically important serious event; Requires or prolongs hospitalization". The investigator also considered this event to be severe in intensity, not related to study treatment, and not related to protocol-

Figure 1: Example Patient Narrative

CONSIDERATIONS

There are various ways how these narratives can be created.

- Manual Approach – all content is created manually by a human generating all these files
- Automation – creating narratives through automated processes
- Natural Language Processing – train an algorithm to create narratives depending on the input data

Even though natural language processing would be a fascinating field, it has been decided to go with automated processes. There are standard sentences for general adverse events which could easily be defined with classical methods. The specifics from the study should ideally also be described, but as every study has a different specific, it is difficult to provide enough training data for a natural language processing algorithm. A classical algorithm is also more transparent and for this we decided to check out classical automation.

When looking at the task to create narratives, the following challenges needs to be solved:

- Define specific layout, fonts, font styles, indents.
- Identify the text blocks which should change dynamically depending on clinical study data.
- Identify the text block content origin, sometimes content might come from a "yes" in the database.
- Handle conditional text parts or complete conditional paragraphs.
- Support tables as well as tables are sometimes requested to support the text.

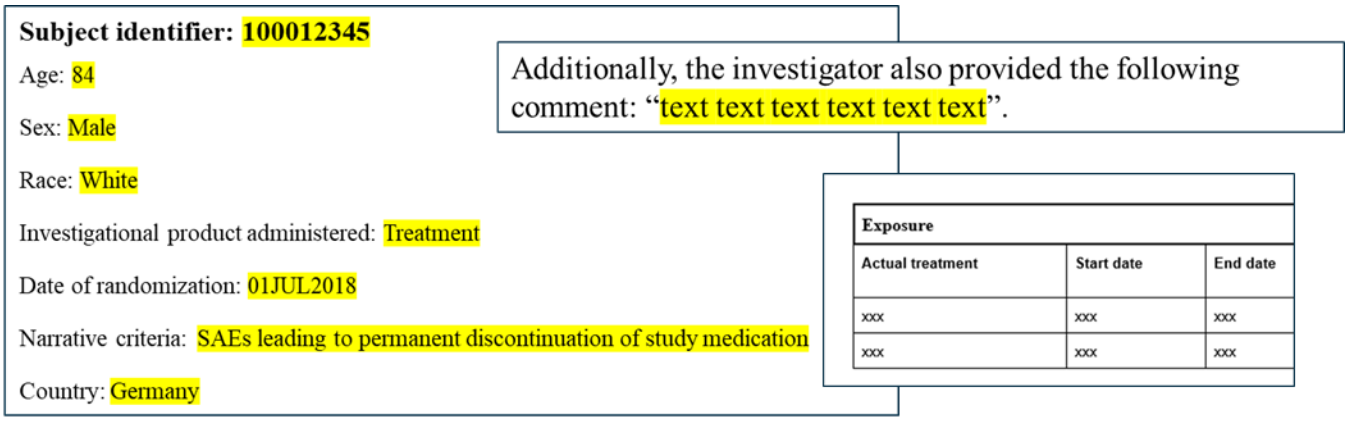

Figure 2: Highlighted Challenges

EXPERIMENTATION

Various implementations has been tested to investigate the ideal solution. For the first study, the patient narratives have been implemented completely in SAS. All texts and tables have been included in the final RTF documents with PROC REPORTs and using ODS TEXT statements. The code was quite long and hard to maintain. Continuous change requests to layout and content had been difficult to implement. It was clear that this is not the ideal solution.

At Bayer, we have an inhouse Word tool available. This tool can update Word documents according provided instructions, e.g. "replace this with that" and is callable directly from SAS. Text parts can be exchanged with other texts, tables can be filled, content coming from other documents can be included. There are commands available to remove text, sections and areas. This tool is been used to create validation documentation. By using various custom tags like {subjidn}, we had been able to reduce the complexity of the SAS program drastically. This tool is using a Microsoft Word file as input template, all special layout and format requirements can be applied easily in Microsoft Word. Unluckily our tool has issues when it comes to repetitions, so when an abstract should be repeated for each adverse event of a patient with all related details, these complex text blocks still had to be produced through SAS. The instruction-based approach was very good, but not the ideal solution.

The ideal solution was coming from a colleague. He was having issues to enter Chinese characters in our SAS environment for the dynamic texts and was looking for a solution outside of SAS. He found an open source tool for Python called "python-docx-template". This tool is also working with a word template. Furthermore, it uses a standard template language to perform data-driven updates. By having a closer look into this tool, we figured out that this fits our needs perfectly. The following section will describe the tool and how it has been implemented at Bayer.

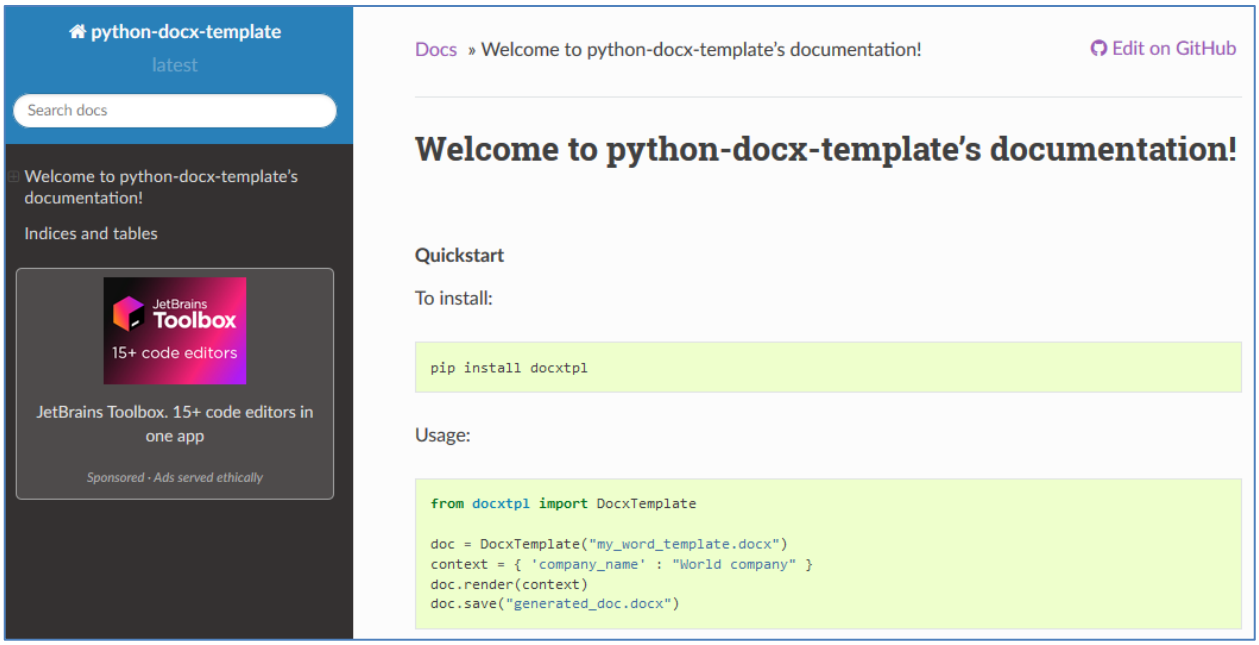

Figure 3: Screenshot from python-docx-template documentation

PYTHON-DOCX-TEMPLATE FUNCTIONALITY

The open source tool Pyhton-Docx-Template is using Jinja2 as template language. This is used in the Django web framework as template language. This template language defines the structure and programming commands to be able to create outputs just based on this template elements and the corresponding data.

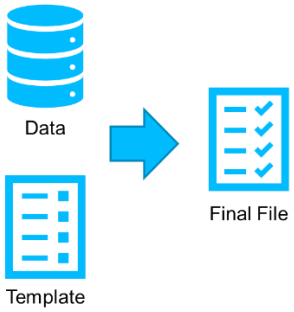

Figure 4: Tool Functionality Schema

DATA STRUCTURE

The data itself can easy be imported to Python in JSON format. There are two major classes of the data available. The first one is an object. For our use case we store all one-observational information of analysis datasets (ADS) into simple objects. The ADSL object for one subject can for example look like the following:

```
"ADSL" :
    {
        "RACE"    : "ASIAN",
        "RFENDT"  : "01JUN2018",
        "RFSTDT"  : "01MAR2018",
        "SEX"     : "F",
        "SUBJIDN" : "123456789"
    }
```

The second important type is an array, where multiple items of the same type are stored into. This is used whenever a subject can have multiple observations within a dataset. This is when a subject has multiple adverse events, multiple medical history entries and many more. The following extract shows an example how such an array can look like in JSON:

```
"ADAE" :
[
    {"ADURN" : "13", "AEREL" : "Y", "AESER" : "N", "AETERM" : "METRORRHAGIA"},
    {"ADURN" : "42", "AEREL" : "Y", "AESER" : "N", "AETERM" : "PROLONGED ..."},
    {"ADURN" : "31", "AEREL" : "Y", "AESER" : "N", "AETERM" : "PROLONGED ..."}
]
```

TEMPLATE ELEMENTS

Next to the data, the template elements must be included into the Word template including static text parts and the final required style and layout of the content.

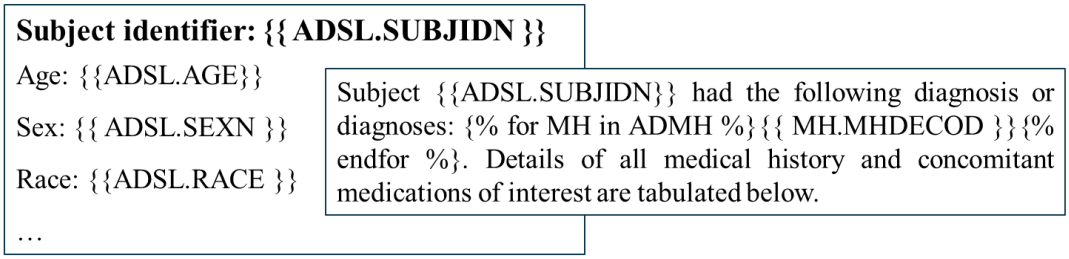

Figure 5: Example Template Elements

The Template language has various functionality available. The base functionality which will cover most of what is required is the simple text replacement with data content. Depending on whether the data is an object or an array of objects, either the direct attributes can be replaced, or a loop is needed in the template where then single elements can be used. The following table show how it looks in the template and provides additionally an example:

	Object	**Array with Objects**
Template:	{{ <data>.<variable> }}	{% for item in <data> %} {{ item.<variable>}} {% endfor %}
Example:	{{ ADSL. SUBJIDN }}	{% for ae in ADAE %} {{ ae.AESTDTL}} {% endfor %}

Conditional logic is also very important for the use case. Text parts or complete paragraphs should be omitted for some cases. The following example shows hot wo omit a text part if a dataset does not exist:

```
{% if <DATA> %} ... {% endif %}
```

By adding the “p”-tag for paragraph, a complete paragraph can be omitted. This this case the followed paragraph is only included in the output if the data is available:

```
{%p if <DATA> %} ... {%p endif %}
```

The typical if-then-else if-else conditions are also available in the template language and are using different names:

```
{%p if <DATA> %}
...
{%p elif <DATA2> %}
...
{%p else %}
...
{%p endif %}
```

Filters are special functions which can be applied to the text which should be replaced. The template language comes with a lot of default filters like lower to apply the lower-case functionality, upper to make characters in upper case, min to get the minimum value from a list and much more. Apart from default filters also custom filters can be created. For SAS programmers a lower-case filter named “lowcase” might be more intuitive as well as “upcase” to create upper cases.

An important custom filter for our use case is a filter to replace missing values with another text. For example if a patient has no race assigned, then there should not be printed “Race: .”, instead “Race: unknown” should be created. Another requirement is to use alternative labels. The data values are sometimes for example in CDISC “M/F” where for the sex “Male” or “Female” should be rendered into the resulting document. “N(o)” and “Y(es)” values should also

quite often be translated with a different text, for example as “serious” or “not serious”. To support this, a custom filter can be created. For incomplete dates it might also be nice to remove the minus characters which might be available. The following example shows how custom filters can be applied in the Word template file.

```
{{DEMO.RACE | missing("unknown") | lowcase}}
{{DEMO.SEX | label(„SEX")}}
{{ mh.MHSTDTL | compress("-") }}
```

There are tags available to create dynamic columns and rows. For this the flexible and generic tables are also no issue by using this tool. The following example displays how dynamic rows can be added. In this case each medical history item appears in a separate row. The formatting of the resulting table will be exactly as the template input format is, which makes the styling very easy.

Fixed Example Table				
Subject Identifier	**Term**	**Start Date**	**End Date**	**Status**
{%tr for item in ADMH %}				
{{ item.SUBJIDN }}	{{ item.MHDECOD }}	{{ item.MHSTDTL }}	{{ item.MHENDTL }}	{{ item.MHENRTPT }}
{%tr endfor %}				
This is my personal Footnote				

Figure 6: Example Template Syntax for a table

The Jinja2 template language is very powerful and enables a processing based on data. Many options are available. The Python-docx-template enables this template language for Word and enables in this way how a data-driven approach to program Word outputs.

IMPLEMENTATION

This tool fits our requirements for the narrative creation, and we investigated how to apply it in our environment. There is a Word-Template containing the template language elements which are the commands for the final rendition. The clinical data is collected as required, modified if necessary and exported to JSON. Additionally, we export some metadata in JSON format to support variable label usages as well as alternative labels. We maintain a python program which performs the python processing by calling the tool with our data. Finally, we maintain an interface from SAS to the Python program. The implementation is not complex and described hereinafter.

THE PYTHON PROGRAM

The Python program is running on the command line using three parameters as input and one as output. The inputs are the data in JSON format, the metadata in JSON format and the Word Template file containing the template commands. The output parameter contains the file name prefixes, as per subject one Word output is created. The command line call could look like the following:

```
python call_pete_wordupdate.py
    --meta <path>/meta.json
    --data <path>/data.json
    --template <path>/template_study12345.docx
    --outprefix <outpath>/study12345_subject_;
```

The program contains additional custom filters, a special post processing for line breaks, the general setup, a loop through all subjects where the docx-template is applies for each subject to create one output per subject. Finally, various errors are caught and easier to understand messages are printed to the log. Finally, the code is very light weight with only 180 lines of code.

In Python we can create definitions which are like functions or macros in SAS. The core definition containing the parameters looks like the following:

```
def pete_wordupdate(infile_meta, infile_data, infile_template, outfile_prefix):
```

Special filters are defined as definition as well. In the following example a missing filter is implemented. This filter receives next to the “value” coming from the data a parameter “replacement” from the filter (template element can be

{{ ADSL.SEX | missing(“unknown”) }} for example). When the value parameter is available, so not missing, not none and not a dot, then the original value is printed, otherwise the replacement value is printed.

```
def missing(value, replacement):
    if value is not None and len(value.strip()) > 0 and value != ".":
        return value
    else:
        return replacement
```

The environment setup is quite straight forward by using a simple definition. Custom filters can be added when using not the default environment.

```
jinja_env = jinja2.Environment()
jinja_env.filters['missing'] = missing
```

Reading JSON into Python can be done with available packages quite easy. An error handling would be recommended as errors could occur when the files are not valid.

```
file_meta = open(infile_meta, 'r')
file_data = open(infile_data, 'r‘)
try:
    metadata = json.load(file_meta)
    data = json.load(file_data)
except json.decoder.JSONDecodeError:
    print("ERROR: Invalid JSON files.")
```

The core step of the program is the looping through all subjects which are available in data[id]. We assign the template to a macro variable, perform then the template processing with the render function and finally save the output.

```
for id in data:
    print("- PETE_WORDUPDATE: Process identifier - " + id)
    tpl=DocxTemplate(infile_template)
    try:
        tpl.render(data[id],jinja_env)
    except ...:
        ...
    tpl.save(outfile_prefix + subject + '.docx')
```

These are the core parts for the implementation of the python program.

CONNECTING SAS WITH THE PYTHON PROGRAM

The next question is how to connect the SAS environment with the Python program. We have decided to run Python in a docker container. Docker containers package applications into containers which are portable and stable. This allows us to have a fixed and stable version of Python as well as from the python-docx-template language. When other Python tools come up which needs packages in different versions, these will have no influence on our environment. With server SSH and SCP commands we can upload files to our Python-Docker-Container, download files back to our SAS environment (SCP) and execute the python program through the command line (SSH).

The developers of the patient narratives are Statistical Analysts which are SAS experts. Our colleagues, our data and our results should all be available in our SAS environment, we have decided to create a SAS macro which is used as frontend and using the docker connection to run Python as backend. The SAS macro creates JSON files of containing SAS datasets and metadata datasets. The macro connects to the docker container, sends the file, executes the Python program, prints the Python log to the SAS log and receives finally the files. The macro is very generic and can be used for many use cases. The narrative use-case is our pilot where we maintain also some supporting macros.

The macro has a parameter for optional sub-setting. Especially during development, it is better to test the process with just one or two subjects. As issues in the template are very likely when people are just learning the template language.

The narratives process is based on a Microsoft Word template available as well as study data. Depending on the Microsoft Word template and the data, the metadata can be derived with a macro calls “%Pete_CreateMeta”. The user might update the Microsoft Word template and the metadata, e.g. to include additional alternative labels. Finally, the macro “%Pete_WordUpdate” is called with the data, metadata and the Microsoft Word template to create the single narrative files.

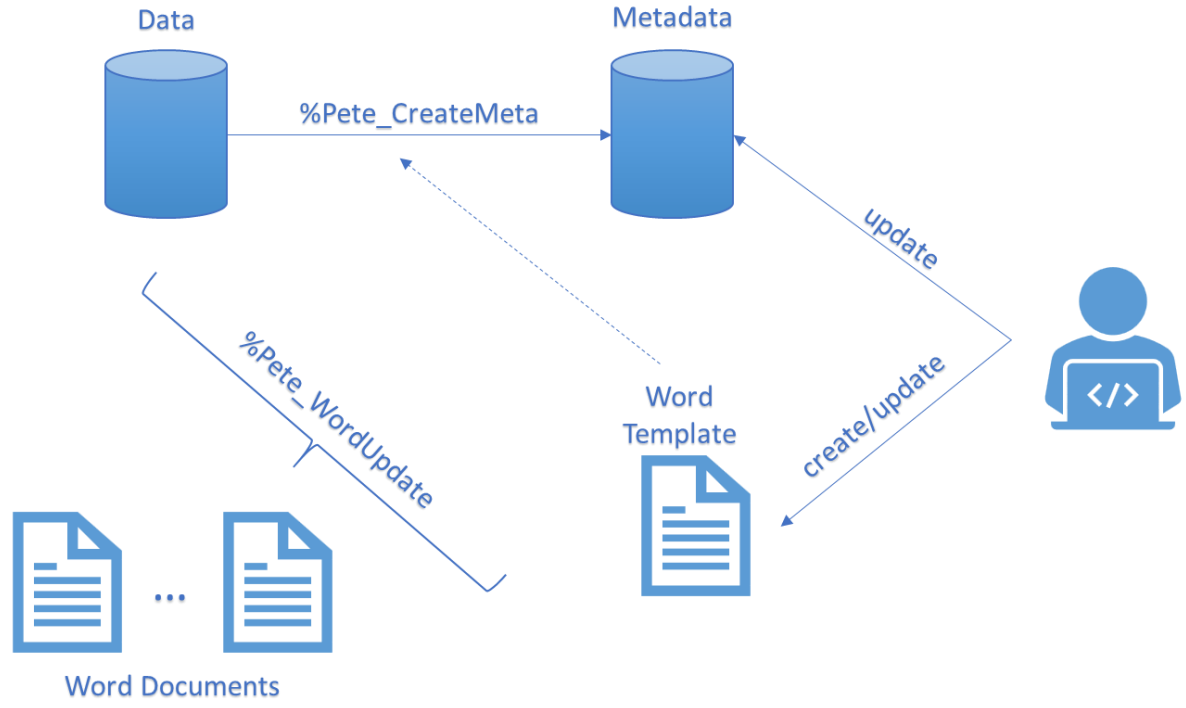

Figure 7: Narratives Creation Macros - Process Flow

CONCLUSION

We learned a lot about how we can apply open source solutions in other programming languages within our clinical environment. The connection to SAS was very important for us, as we want to have one process flow triggered and controlled by our SAS environment. We decided on a docker-container for our structure to be able to have a specific version for the required Python packages which allows us a one-time-validation as the environment is fixed and for this will behave always the same.

There are a lot of open source solutions available for various topics. By finding one which fits perfectly our use case, we had been able to implement an easy to maintain tool. We did not have to re-invent an own template language. Using Jinja as template language and to render data content with the python-docx-template tool enabled us to create a nice standard process. Finally, the tool is very generic, and can be implemented for other tasks as well.

We had been able to up-skill some colleagues. By using another programming language, the hurdles to perform more tasks in other languages shrink. So more opportunities of available tools can be taken. The main users will currently not be trained to become Python programmers – at least at the beginning – and for this the usage via SAS macro was die ideal solution here. The colleagues do not need to learn a new programming language to create narratives. Of course, they must learn the template language. Luckily, this is very intuitive and by providing training and enough examples, users feel very comfortable with this.

We are looking forward for new opportunities and additional use cases.

REFERENCES

[1] Python-docx-template documentation
https://docxtpl.readthedocs.io/en/latest/

[2] Python-docx-template source code on GitHub
https://github.com/elapouya/python-docx-template

[3] Jinja Template language documentation
https://jinja.palletsprojects.com/en/2.11.x/templates

[4] Docker Homepage
https://www.docker.com/

CONTACT INFORMATION

Contact the author at:

Katja Glaß
Bayer AG
Sellerstraße 32
13353 Berlin
Germany
Katja.Glass@Bayer.com