Paper SM09

Using Python to automate the Reviewer's Guide

Matt Metherell, PHASTAR, Portsmouth, United Kingdom
Simon Purkess, PHASTAR, London, United Kingdom

ABSTRACT

Writing Reviewer's Guides can be a very time-consuming part of the submission process, and a lot of what we want to include is already stored in SAS datasets, or other structured documents such as spreadsheets. Python has the ideal package to update Word documents with this data automatically, as it edits Word documents at an XML level. So, taking the CSDRG template as a starting point we can use the python-docx package to easily find a specific paragraph and enter specific text or a table.

INTRODUCTION

In this paper we will demonstrate how to simply and instantly populate the list of supplemental variables directly from the SUPPxx datasets, enter the P21 issues and our responses, list all the inclusion/exclusion criteria and automate a surprising number of other elements to greatly streamline the process of creating the SDTM Reviewer's Guide.

THE REVIEWER'S GUIDE

These are documents to help an independent reviewer from the regulatory agencies to navigate the study submission. It summarises key points from the protocol, key statistical analysis, and anything else deemed relevant.

An important feature is that they have a clearly-defined structure that requires specific information in specific sections, so we are able to automate some of these.

Reviewer's Guides are written for the benefit of external regulatory agencies like the FDA, PMDA or EMA. Templates are freely downloadable from the phuse working group, which also includes guidance.
https://advance.phuse.global/display/WEL/Clinical+Study+Data+Reviewer%27s+Guide+%28cSDRG%29+Package

PYTHON-DOCX

This is a python package that reads and saves a Word document purely as a metadata object. It sees each document as a collection of paragraphs. It sees each paragraph as a collection of 'runs' of text or a picture or table. It sees each table as a collection of rows, and each of those as a collection of cells.

PREREQUISITES

At the top of our program we will need to install and import the following libraries:

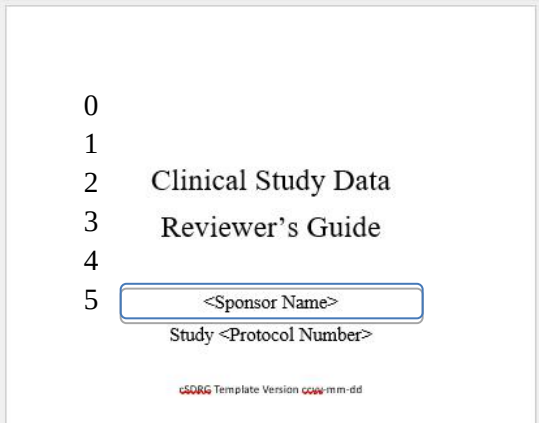
```
pip install python-docx pyreadstat

#For Word editing
from docx import Document
from docx.shared import Pt
from copy import deepcopy

#For reading SAS datasets
import pyreadstat as prs

#For reading Excel spreadsheets
from openpyxl import load_workbook

#For checking directory contents
from os import listdir
from os.path import isfile, join

#For general python tools
import pandas as pd
import numpy as np
```

PARAGRAPHS

Once we open a document, we can navigate through the list of paragraphs to extract the data we are after. In the below example we select the sixth paragraph (counting begins at 0 in python) by using the *paragraphs* attribute of the *document* object. Then the *text* attribute of this paragraph contains the text we see in the Word document: "<Sponsor Name>".

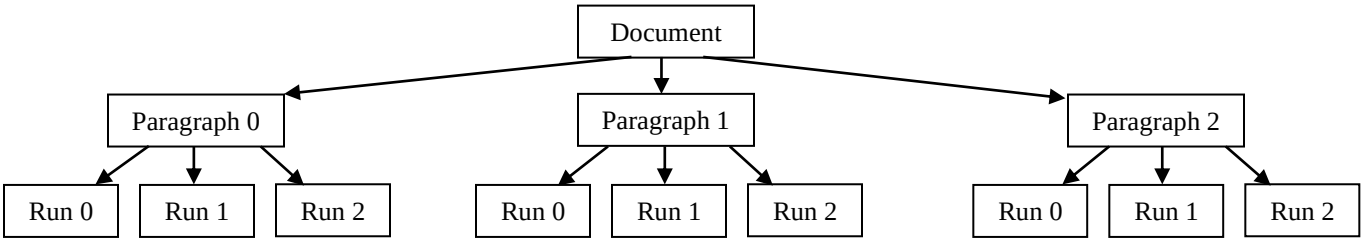
```
#Import original CDISC template
document = Document('Word/csdrg-
template.docx')

#Select paragraph
paragraph = document.paragraphs[5]

#Display paragraph text
print(paragraph.text)
```

```
    <Sponsor Name>
```



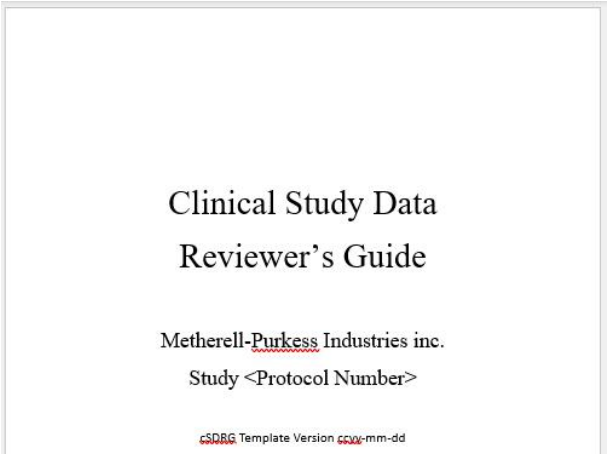
RUNS OF TEXT

A paragraph of text is made up of runs.
A run is a piece of text that has consistent attributes, so anytime the font or size changes, it's a new run.
Any time underlining, emboldening or any other stylization starts or ends, it's also a new run.
The hierarchy is summarized as follows:



To edit text, it is at the run level that we typically make changes. The following code extends the previous example by updating the content of the only (0^{th}) run of text in the paragraph. Before doing this, it is important to choose a font and size, which is done here by extracting the attributes of the pre-existing run. Any text we add in future will have these attributes until we reset them. The fact we extracted them from the run that we are editing is merely a matter of convenience for this example.

```
#Import original CDISC template
document = Document('Word/csdrg-
template.docx')

#Select paragraph
paragraph = document.paragraphs[5]

#Select run to update
run = paragraph.runs[0]

#Extract attributes
font = run.font

#Update text
run.text = "Metherell-Purkess Industries inc."

#Save document
document.save("Word/csdrg-updated.docx")
```



In some cases things are simpler though. If the only formatting that is present is the paragraph styling, then we can ignore the run, and edit *paragraph.text*.

TABLES

Tables exist directly under the document in the metadata hierarchy. They exist alongside paragraphs without changing the order of the paragraph. When editing the text in a formatted table, we don't need to worry about runs, as the text styling is already present in the template tables in the Reviewer's Guide template.

The following example is a check that we have identified the correct table in the template, by returning the value of a cell from the header. First we identify the table within the document, and then a cell within the table. The cell has two arguments: row (0 for the header row) and then column (0 for the first column, 1 for the second etc).

```
#Identify inclusion/exclusion table
ti_table = document.tables[7]

#Check contents to confirm correct table
ti_table.cell(0,1).text

       'Category'
```

Protocol/ Amendment Version	Category	IETESTCD	Full Text of Criterion

The following example updates a value in the table. To do this we identify the row, and then the cell within that row, and then update the value of that cell's *text* attribute. Finally we save the document to make the update visible when we open the file.

```
#Identify row to be update
row = ti_table.rows[1]

#Identify the cells of this row
row_cells = row.cells

#Update specific cell(s) in row
row_cells[0].text = "Final"

#Save document
document.save("Word/csdrg-updated.docx")
```

Protocol/ Amendment Version	Category	IETESTCD	Full Text of Criterion
Final			

TRIAL DESIGN MODEL (TDM) DATASETS

There are typically five of these, and they are described in sections 2.3.x of the CSDRG.
2.3.1: TA – Trial Arms
2.3.2: TE – Trial Elements
2.3.3: TV – Trial Visits
2.3.4: TI – Trial Inclusion/Exclusion Criteria
2.3.5: TS – Trial Summary

At a minimum, the contents of TI will be placed in Appendix I so we shall populate this as our next example, but it is not uncommon for a table of Visits to be included as an appendix, which will be a tabulation of the TV dataset. Extending the table example earlier, we can populate it with the contents of a SAS dataset, using the *pandas* package to read it in, the *numpy* package to create an array based on the contents and the looping over the length of the array to populate the table using the *python-docx* package.

```
#Identify table to be updated
ti_table = document.tables[7]

#Read in TI SAS dataset
ti_df = pd.read_sas('SAS/ti.sas7bdat', encoding="utf-8")

#Convert to array
ti_arr = ti_df.to_numpy()
```

```
#Enter each row of TI contents into new row of table
for studyid, domain, ietestcd, ietest, iecat, tivers in ti_arr:
    row_cells = ti_table.add_row().cells
    row_cells[0].text = tivers
    row_cells[1].text = iecat
    row_cells[2].text = ietestcd
    row_cells[3].text = ietest

#Save document
document.save("Word/csdrg-updated.docx")
```

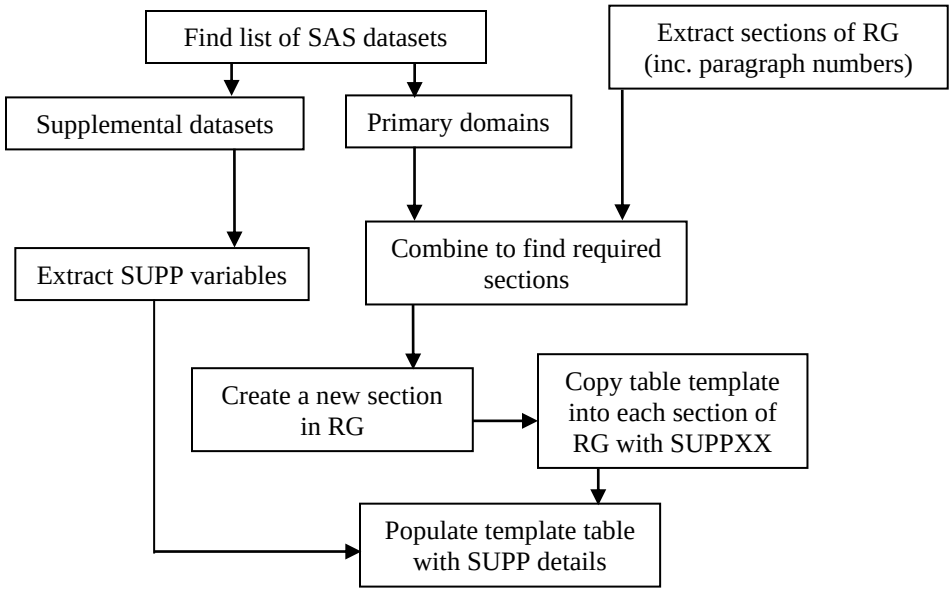
Protocol/ Amendment Version	Category	IETESTCD	Full Text of Criterion
V1.0	INCLUSION	IN01	Subject is aged ≥ 18 years of age at the screening visit.
V1.0	EXCLUSION	EX01	Treatment with any investigational product within the last 30 days.

SUPPLEMENTAL VARIABLES

As alluded to earlier, every other domain has its own section in the Reviewer's Guide as 3.4.x. This contains info about the type of data, and links to other datasets, and a list of all the variables in its supplemental dataset. This list of supplemental variables can be easily extracted and placed into the CSDRG in the section relating to their primary domain.

We will also create this subsection if it doesn't exist in the template and add it to the table in section 3.4 with the appropriate hyperlink to 3.4.x.

This combination of tasks is complicated, so below is an overview of all the functions we need to create:



FIND LIST OF SAS DATASETS

Using the *os* package we can find the name of files in a directory. In the example below, we look only at those that contain '.sas7bdat' in their filename. We then store the filename in an array of domain abbreviations, called *ds*.

```
#Find list of Datasets
mypath = 'SAS'
onlyfiles = [f for f in listdir(mypath) if isfile(join(mypath, f))]
sasfiles = [x for x in onlyfiles if '.sas7bdat' in x]
ds = []
for word in sasfiles:
    ds.append(word.upper())
```

We can then split this into two separate arrays; one for the primary domains (*domain*) and one for the supplemental datasets (*supp*). This is because we will need Reviewer's Guide sections for each of the primary domains, but a table only needs to be copied for all of those with supplemental datasets.

```
#Separate primary domains from supplementals
domain = []
supp = []
for i in range(len(ds)):
    if 'SUPP' in ds[i]:
        supp.append(ds[i])
    else:
        domain.append(ds[i])
```

For all of the primary domains, we will need the dataset label for the section title, so we shall create an array (*names*) that collects these from each of the filenames in the *domain* array, using the *pyreadstat* package. As we collect these in the same order, we can then join these arrays together at the end.

```
#Find names of Datasets
names = []
for word in domain:
    df, meta = prs.read_sas7bdat("SAS/"+word)
    names.append(meta.file_label)

#Join together domain and names
array_new = np.stack((domain, names), axis=1)
```

From here we want to join the list of supplemental filenames to the relevant primary domains, which is a task akin to an SQL join, which is best achieved using the *pandas* package, requiring both arrays to be converted into *pandas* dataframes.

```
#Convert to pandas dataframe to merge with supplemental datasets
df_new = pd.DataFrame({'domain': domain, 'ds_label': names })
df_supp = pd.DataFrame({'supp': supp})
df_supp['domain'] = df_supp['supp'].str[4:]

#Merge
df_ds = pd.merge(df_new, df_supp, how="outer", on=["domain"])

#Remove file extension from domain column
df_ds['domain'] = df_ds['domain'].str[:2]
```

This then gives us something like the following output:

	domain	ds_label	supp
0	AE	Adverse Events	SUPPAE.SAS7BDAT
1	DA	Drug Accountability	NaN
2	DM	Demographics	NaN
3	DS	Disposition	NaN
4	EG	ECG Test Results	SUPPEG.SAS7BDAT
5	EX	Exposure	NaN
6	FA	Findings About Events or Interventions	NaN
7	IE	Inclusion/Exclusion Criteria Not Met	NaN
8	LB	Laboratory Test Results	SUPPLB.SAS7BDAT
9	SE	Subject Elements	NaN
10	SV	Subject Visits	NaN
11	VS	Vital Signs	SUPPVS.SAS7BDAT

EXTRACT SUPPLEMENTAL VARIABLES

This function will extract the supplemental variables of a given dataset, and create a *pandas* array that lists the QNAMs and QLABELs.

```
def supp_extract(fname):

    #Read in SUPP dataset
    file = 'SAS/'+fname
    df = pd.read_sas(file, encoding="utf-8")

    #Find unique values
    return pd.unique(df[['QNAM', 'QLABEL']].values.ravel())
```

And this function can be called as follows:

```
supp_extract('suppae.sas7bdat')
```

```
array(['AEVAR1', 'Supplemental AE Variable 1', 'AEVAR2',
       'Supplemental AE Variable 2', 'AEVAR3',
       'Supplemental AE Variable 3'], dtype=object)
```

But we can call this on all entries in an array of supplemental dataset names as we shall see later on in this paper.

EXTRACT SECTIONS FROM REVIEWER'S GUIDE

The below function will check every paragraph for those that have some form of *Heading* styling, and collecting the text of these in an array called *headings*. This is then interrogated for those that begin with '3.4.' as these are the sections containing domain information. These headings are collected in the array *dom_head*, and the portion containing the domain abbreviation in *dom_abbr* (for comparing against SAS datasets), the domain name in *dom_name* and the paragraph number in *para_num*.

The template obviously has these sections in predefined paragraphs, but by searching the text value of the headings we can use this on a reviewer's guide that has already been populated, perhaps for an interim analysis.

```
def domain_headings(paragraphs):

    #Initiate arrays
    global headings, dom_head, dom_abbr, dom_name, para_num
    headings = []
    dom_head = []
    dom_abbr = []
    dom_name = []
    para_num = []

    for i in range(len(paragraphs)):
        paragraph = paragraphs[i]
        if paragraph.style.name.startswith('Heading'):

            #create list of headings
            headings.append(paragraph.text)
            if paragraph.text.startswith('3.4.'):
                useful_text = paragraph.text.split("\t",1)[1]

                #create array of dataset details and paragraph number
                dom_head.append(paragraph.text)
                dom_abbr.append(useful_text.split(" ",2)[0])
                dom_name.append(useful_text.split(" ",2)[2])
                para_num.append(int(i))
```

Then we can call this over all paragraphs in the document and combine these arrays into a dataframe like we did above.

```
domain_headings(document.paragraphs)
rg = pd.DataFrame({'heading': dom_head, 'abbr': dom_abbr, 'name': dom_name, 'para':
para_num })
```

	heading	domain	name	para
0	3.4.1\tAE – Adverse Events	AE	Adverse Events	115
1	3.4.2\tDS – Disposition	DS	Disposition	117
2	3.4.3\tEX – Exposure	EX	Exposure	119
3	3.4.4\tDataset – Dataset Label	Dataset	Dataset Label	121

COMBINE TO FIND REQUIRED SECTIONS

We can do this by merging our two dataframes to give us a complete list of all domains that we have and will need.

```
result = pd.merge(rg, df_ds, how="outer", on=["abbr"])
```

	heading	domain	name	para	ds_label	supp
0	3.4.1\tAE – Adverse Events	AE	Adverse Events	115.0	Adverse Events	SUPPAE.SAS7BDAT
1	3.4.2\tDS – Disposition	DS	Disposition	117.0	Disposition	NaN
2	3.4.3\tEX – Exposure	EX	Exposure	119.0	Exposure	NaN
3	3.4.4\tDataset – Dataset Label	Dataset	Dataset Label	121.0	NaN	NaN
4	NaN	DA	NaN	NaN	Drug Accountability	NaN
5	NaN	DM	NaN	NaN	Demographics	NaN
6	NaN	EG	NaN	NaN	ECG Test Results	SUPPEG.SAS7BDAT
7	NaN	FA	NaN	NaN	Findings About Events or Interventions	NaN
8	NaN	IE	NaN	NaN	Inclusion/Exclusion Criteria Not Met	NaN
9	NaN	LB	NaN	NaN	Laboratory Test Results	SUPPLB.SAS7BDAT
10	NaN	SE	NaN	NaN	Subject Elements	NaN
11	NaN	SV	NaN	NaN	Subject Visits	NaN
12	NaN	VS	NaN	NaN	Vital Signs	SUPPVS.SAS7BDAT

This is handily ordered so that all the existing sections are at the top (ordered by paragraph number), and all of the new ones are grouped below. We can therefore impute the paragraph numbers that these new sections will have (by adding 2 to every subsequent row) and also the heading (concatenating 3.4.x with the ds_label value).

```
#loop through the rows using iterrows()
for index, row in result.iterrows():
    if row['para'] == row['para']: #check if populated
        counter=row['para']
    else:
        counter = counter+2
        row['para'] = counter
        row['heading'] = "3.4."+str(index)+"\t"+row['domain']+" - "+row['ds_label']
    result.loc[index, 'para'] = row['para']
    result.loc[index, 'heading'] = row['heading']

#Convert paragraph values to integers
result['para']=result['para'].astype(int)
```

	heading	domain	name	para	ds_label	supp
0	3.4.1\tAE – Adverse Events	AE	Adverse Events	115	Adverse Events	SUPPAE.SAS7BDAT
1	3.4.2\tDS – Disposition	DS	Disposition	117	Disposition	NaN
2	3.4.3\tEX – Exposure	EX	Exposure	119	Exposure	NaN
3	3.4.4\tDataset – Dataset Label	Dataset	Dataset Label	121	NaN	NaN
4	3.4.4\tDA - Drug Accountability	DA	NaN	123	Drug Accountability	NaN
5	3.4.5\tDM - Demographics	DM	NaN	125	Demographics	NaN
6	3.4.6\tEG - ECG Test Results	EG	NaN	127	ECG Test Results	SUPPEG.SAS7BDAT
7	3.4.7\tFA - Findings About Events or Intervent...	FA	NaN	129	Findings About Events or Interventions	NaN
8	3.4.8\tIE - Inclusion/Exclusion Criteria Not Met	IE	NaN	131	Inclusion/Exclusion Criteria Not Met	NaN
9	3.4.9\tLB - Laboratory Test Results	LB	NaN	133	Laboratory Test Results	SUPPLB.SAS7BDAT
10	3.4.10\tSE - Subject Elements	SE	NaN	135	Subject Elements	NaN
11	3.4.11\tSV - Subject Visits	SV	NaN	137	Subject Visits	NaN
12	3.4.12\tVS - Vital Signs	VS	NaN	139	Vital Signs	SUPPVS.SAS7BDAT

CREATE NEW REVIEWER'S GUIDE SECTIONS

Now we can create any additional sections needed. As we can see from the *result* dataframe, we have a template section ready to be copied. If we copy this to the end of this list we can then rename them based on the *heading* column.

Unfortunately, copying paragraphs is not trivial in *python-docx*. Therefore we create a function that does this by creating a new paragraph at the end of the document, applying the styles of our source paragraph to it, copying the text value, and then moving this paragraph to where we require it. This is done with the following code:

```
def copy_paragraph(frm, to, head):
    #frm is the paragraph to copy
    old_paragraph = document.paragraphs[frm]

    #to is the paragraph it will be pasted AFTER
    position_paragraph = document.paragraphs[to]

    #Create paragraph at the end
    end_paragraph = document.add_paragraph()

    #Move to final location
    position_paragraph._p.addnext(end_paragraph._p)

    #Final paragraph is the reference to the end product
    final_paragraph = document.paragraphs[to+1]

    #Update paragraph style and copy text across
    final_paragraph.style = old_paragraph.style
    final_paragraph.text = old_paragraph.text
```

We can then call this for the entries in our *result* dataframe that don't have a paragraph number, which are all grouped together at the end. We call this once for the header and once for the "(Text here)" paragraph, and therefore it's useful to also add in some code to rename the sections while we got through them. The following code adds this to the function and then calls it over the dataframe.

```
    #If copying a header rename for new section
    if final_paragraph.style.name.startswith('Heading'):
        title = final_paragraph.text = head

#Convert dataframe to numpy array
result_n = result.to_numpy()

#Iterate over this array
for heading, domain, name, para, ds_label, supp in result_n:

    #For datasets without an existing section
    if pd.isna(name):
        copy_paragraph(115, para, heading) #heading
        copy_paragraph(116, para+1, heading) #paragraph text
```

At some point we want to remove the template section entirely, but for now we will remove just one paragraph after it to make every heading consistently two paragraphs after the previous. This has recorded which paragraph we need to remove later.

```
    #Remove Template heading and paragraphs
    if domain == "Dataset":
        del_para = para

document.paragraphs[del_para+1]._element.getparent().remove(document.paragraphs[del_pa
ra+1]._element)
```

COPY TEMPLATE TABLE INTO EACH SECTION

We need a similar function to copy table templates, which is thankfully simpler than copying paragraphs. This allows us to select a table and place a copy of it after a paragraph of our choice.

```
def copy_table_after(table, paragraph):
    tbl, p = table._tbl, document.paragraphs[paragraph]._p
    new_tbl = deepcopy(tbl)
    p.addnext(new_tbl)
```

Because of the way we populate the tables from datasets by adding a new row to the table for each row in the dataset, we need to remove the blank rows from the template table before we call this macro. We also need to define the template table just once, as once we have copied it, it will become the 5th table rather than the 4th. We therefore create a variable that will increment as we add a new table.

```
#Counter to track which table we have added
#Begins as template table
table_count=4

#Identify template table
tab = document.tables[table_count]

#Remove blank rows
tab._tbl.remove(tab.rows[1]._tr)
tab._tbl.remove(tab.rows[1]._tr)
tab._tbl.remove(tab.rows[1]._tr)
tab._tbl.remove(tab.rows[1]._tr)
```

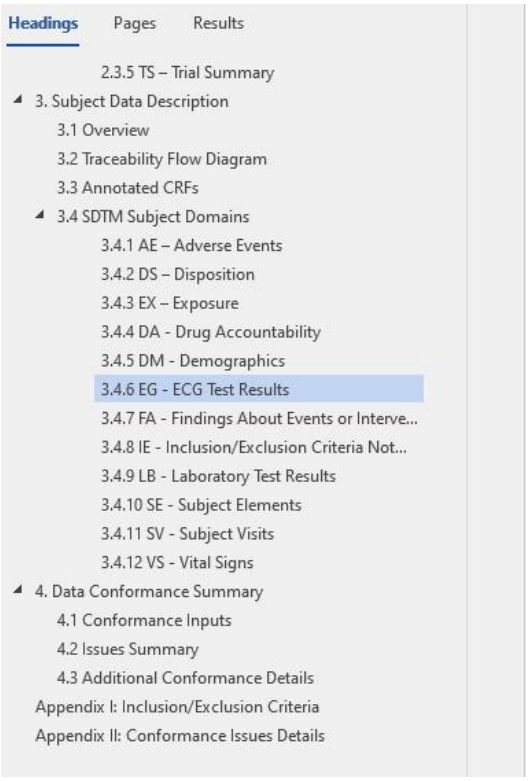
From here it is just a matter of iterating over our array to copy the table template, extracting the SUPP variables as we've seen above, and copy them into the table, as we've also done above.

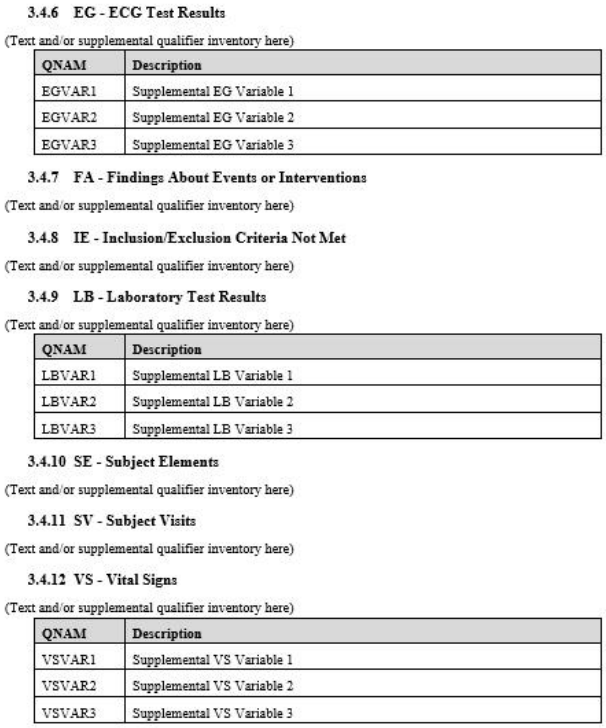
```
#Counter to track which table we have added
table_count=4

#Iterate over this array
for heading, abbr, name, para, names, supp in result2_n:

    #For sections with a supplemental dataset
    if pd.isna(supp)==0:

        #Copy table template after the heading and text
        copy_table_after(tab, para+1)

        #Reference to current table
        current_table = document.tables[table_count]
        table_count = table_count + 1

        #Extract SUPP variables into an array
        arr = supp_extract(supp)

        #Enter contents of array into new row of table
        for i in range(len(arr)):
            if i%2==0:
                row_cells = current_table.add_row().cells
                row_cells[0].text = arr[i]
            else:
                row_cells[1].text = arr[i]
```

We are now almost done. Once we remove the remaining template section and table, we will have what we are after.

```
#Remove template RG section
document.paragraphs[del_para]._element.getparent().remove(document.paragraphs[del_para
]._element)
document.paragraphs[del_para]._element.getparent().remove(document.paragraphs[del_para
]._element)

#Remove template table
document.tables[table_count]._element.getparent().remove(document.tables[table_count].
_element)
```

As well as the page being populated as we want, the headings in the navigation panel are also correct.





3.4.6 EG - ECG Test Results

(Text and/or supplemental qualifier inventory here)

QNAM	Description
EGVAR1	Supplemental EG Variable 1
EGVAR2	Supplemental EG Variable 2
EGVAR3	Supplemental EG Variable 3

3.4.7 FA - Findings About Events or Interventions

(Text and/or supplemental qualifier inventory here)

3.4.8 IE - Inclusion/Exclusion Criteria Not Met

(Text and/or supplemental qualifier inventory here)

3.4.9 LB - Laboratory Test Results

(Text and/or supplemental qualifier inventory here)

QNAM	Description
LBVAR1	Supplemental LB Variable 1
LBVAR2	Supplemental LB Variable 2
LBVAR3	Supplemental LB Variable 3

3.4.10 SE - Subject Elements

(Text and/or supplemental qualifier inventory here)

3.4.11 SV - Subject Visits

(Text and/or supplemental qualifier inventory here)

3.4.12 VS - Vital Signs

(Text and/or supplemental qualifier inventory here)

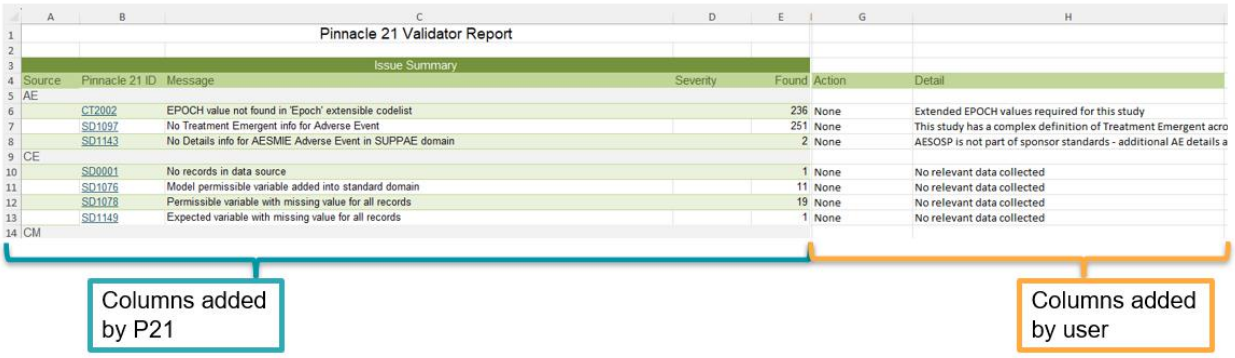
QNAM	Description
VSVAR1	Supplemental VS Variable 1
VSVAR2	Supplemental VS Variable 2
VSVAR3	Supplemental VS Variable 3

CONFORMANCE ISSUES (PINNACLE 21)

The validation issues identified by P21 Community edition will need to be included in section 4.2, along with our responses. As the validation report is in a standard format, if we simply add an additional column containing these responses, all the relevant information can instantly be included.

PINNACLE 21 EXCEL REPORT

After running a validation, Pinnacle 21 Community edition produces an XLSX document containing a list of validation messages. It's then the user's job to write a response to each message, explaining why the issue exists (e.g. an issue with the raw data, or if the study is programmed according to sponsor standards which deviate from CDISC standards). After the responses have been written, the "Issue Summary" tab of the XLSX file will look something like this:



Pinnacle 21 Validator Report

Issue Summary

Source	Pinnacle 21 ID	Message	Severity	Found	Action	Detail
AE						
	CT2002	EPOCH value not found in 'Epoch' extensible codelist		236	None	Extended EPOCH values required for this study
	SD1097	No Treatment Emergent info for Adverse Event		251	None	This study has a complex definition of Treatment Emergent acro
	SD1143	No Details info for AESMIE Adverse Event in SUPPAE domain		2	None	AESOSP is not part of sponsor standards - additional AE details a
CE						
	SD0001	No records in data source		1	None	No relevant data collected
	SD1076	Model permissible variable added into standard domain		11	None	No relevant data collected
	SD1078	Permissible variable with missing value for all records		19	None	No relevant data collected
	SD1149	Expected variable with missing value for all records		1	None	No relevant data collected
CM						

Columns added by P21

Columns added by user

Now we have the information we want to include in the section 4.2 table, we can use Python code to import from Excel, format the data and export into a table in Word.

OPENPYXL

This is a package that we can use to interface between Excel and Python – in this case we will only be reading in from Excel to Python, but the package can also be used to write from Python to Excel if required.

DATA IMPORT

Similarly to python-docx, openpyxl sees an XLSX file as a hierarchy of objects: the entire file is a workbook, which consists of one or more sheets, each containing a 2D array of cells. The code below navigates us to the sheet object called 'Issue Summary' which contains our validation comments and responses:

```
#Import openpyxl library for XLSX, and docx for DOCX
from openpyxl import load_workbook

#Create Workbook data structure
wb = load_workbook(filename = 'Excel\p21 report.xlsx')

#Select sheet of interest - Issue Summary
sheet = wb['Issue Summary']
```

REFERENCING CELLS

There are two ways to reference the contents of an individual cell within a sheet object:

```
sheet['A1'].value #Method 1: Letter/number coordinate reference
sheet.cell(1,1).value #Method 2: Row/column coordinate reference
```

Method 1 uses a letter/number coordinate system that's identical to the way cells are named in an Excel spreadsheet, however Method 2's row/column coordinate system can be easier to work with in Python code as it gives two separate integer parameters which can be looped through without any extra processing. To improve readability,

we've created a shorthand function for this paper – the call `c(row,col)` is equivalent to `sheet.cell(row,col).value`:

```
def c(r,c,worksheet=sheet): #Shorthand function for cell references
    return(worksheet.cell(r,c).value)
```

CREATING A TABLE

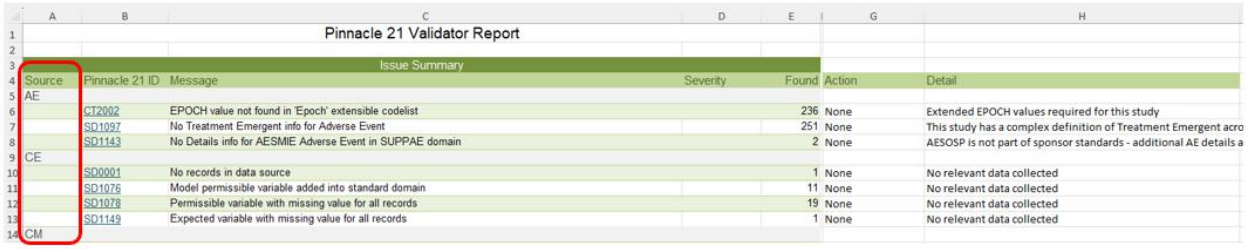
A new table can be added to the document using the python-docx function add_table():

```
#Create table object
p21table = document.add_table(rows=1, cols=4)
#Add header row
header=p21table.rows[0].cells
header[0].text="Dataset"
header[1].text="Diagnostic Message"
header[2].text="Count"
header[3].text="Explanation"
```

If your template already includes a blank "Issues Summary" table, you can instead create a reference to the existing table to add to.

Now our program is connected to both the Pinnacle 21 XLSX and report and the Reviewer's Guide table, we can loop through each row of the report, restructure the data and output to the next row of the table.

One hurdle to overcome with reading in the report row-by-row is that the dataset names in Column A are only populated at the top of each section:



	A	B	C	D	E	G	H
1			Pinnacle 21 Validator Report				
2							
3			Issue Summary				
4	Source	Pinnacle 21 ID	Message	Severity	Found	Action	Detail
5	AE						
6		CT2002	EPOCH value not found in 'Epoch' extensible codelist		236	None	Extended EPOCH values required for this study
7		SD1097	No Treatment Emergent info for Adverse Event		251	None	This study has a complex definition of Treatment Emergent acro
8		SD1143	No Details info for AESMIE Adverse Event in SUPPAE domain		2	None	AESOSP is not part of sponsor standards - additional AE details a
9	CE						
10		SD0001	No records in data source		1	None	No relevant data collected
11		SD1076	Model permissible variable added into standard domain		11	None	No relevant data collected
12		SD1078	Permissible variable with missing value for all records		19	None	No relevant data collected
13		SD1149	Expected variable with missing value for all records		1	None	No relevant data collected
14	CM						

To get around this, we can store the latest value seen in a variable called `dset` as we loop through the report:

```
#Populate rows of P21 messages
dset=""
for i in range(5,sheet.max_row+1):
    #Dataset name is only given once per section so it must be retained
    if c(i,1) is not None:
        dset=c(i,1)
    else: #P21 message row
        row_cells=p21table.add_row().cells
        row_cells[0].text=dset
        row_cells[1].text=f'{c(i,2)}: {c(i,3)}'
        row_cells[2].text=str(c(i,5))
        row_cells[3].text=str(c(i,8))
```

Code notes:

- `max_row` is an openpyxl sheet attribute which gives the index of the final row in the sheet, which we can use to stop the loop at the end of the report. Note that Python's `range(start,end)` function ends one row before the `end` parameter, so we must add 1 to include the final row.
- `add_row()` adds a new row to the table, and the `cells` attribute points us to individual cells within the row which we can access individually from left to right as `row_cells[0]`, `row_cells[1]` etc.

- f-string notation allows us to concisely combine and format multiple strings into one variable, by inserting existing variables into the string in curly brackets. If you're more familiar with traditional string concatenation, the line `f'{c(i,2)}: {c(i,3)}'` is equivalent to `c(i,2) + ': ' + c(i,3)`.

After saving our results, we have our final table:

Dataset	Diagnostic Message	Count	Explanation
AE	CT2002: EPOCH value not found in 'Epoch' extensible codelist	236	Extended EPOCH values required for this study
AE	SD1097: No Treatment Emergent info for Adverse Event	251	This study has a complex definition of Treatment Emergent across the two phases of the study, so this derived at ADaM level
AE	SD1143: No Details info for AESMIE Adverse Event in SUPPAE domain	2	AESOSP is not part of sponsor standards - additional AE details are stored in AEDESC01/AEDESC02
CE	SD0001: No records in data source	1	No relevant data collected
CE	SD1076: Model permissible variable added into standard domain	11	No relevant data collected
CE	SD1078: Permissible variable with missing value for all records	19	No relevant data collected
CE	SD1149: Expected variable with missing value for all records	1	No relevant data collected

CONCLUSION

Any data stored in a standard way can easily be included in the Reviewer's Guide. If it's not already included, this paper will give the reader the necessary detail to add new automations in addition to the surprising number that are already present in a tool that saves the user hours when writing the SDTM Reviewer's Guide.

ACKNOWLEDGMENTS

Much of this functionality is already provided by P21 Enterprise Edition™. If you're using this to validate your study, then use the Reviewer's Guides that they generate.
https://www.pinnacle21.com

RECOMMENDED READING

For more information on the python-docx package, we recommend reading the documentation at https://python-docx.readthedocs.io.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Matt Metherell / Simon Purkess
PHASTAR
2D Bollo Lane
London W4 5LE
Email: matthew.metherell@phastar.com / simon.purkess@phastar.com
Web: www.phastar.com