Paper 3009-2019

How SAS® and Python Enhance PDF – Going Beyond Basic ODS PDF

Sy Truong, Pharmacyclics, Inc.
Jayant Solanki, University at Buffalo - SUNY

ABSTRACT

SAS has vast capabilities in accessing and manipulating the core business information within large organizations. This is coupled with powerful ODS tools to generate PDF Reports. However, there are limitations to the manipulation of an existing PDF file. Python has access to many libraries that can facilitate manipulation of PDF bookmarks, hyperlinks and the content of an existing PDF file. This paper describes a project that combines the strengths of both SAS and Python for PDF editing. This includes:

- **Bookmarks –** Renaming and re-arranging parent/child bookmarks
- **Hyperlinks or Annotations –** Adjusting internal page links and external file PDF links
- **Styles -** Controlling zoom views of the pages and fonts sizes
- **Text Editing –** Manipulating text content of the PDF file

The Users of this PDF Tool are SAS programmers. Thus, a SAS macro is used to allow users to specify the PDF inputs. The %pdftools macro described in this paper reads the Excel file and other sources to gather business rules. After performing series of certain checks on the PDF inputs, it then integrates with a Python program which then apply corrections to the PDF file. There are many tools that can be used to generate and manipulate PDF reports. The most effective solution is to utilize the strength of each tool in a unique way to formulate an easy to use, yet effective at achieving an optimal PDF report.

PDF OVERVIEW AND PROJECT DEFINITIONS

The Portable Document Format (PDF) was originally created by Adobe Systems Inc. PDF standard *ISO 32000-2* defines the open-source standards to present documents which may contain a combination of text and images.

Unlike a MS Word document, the content delivery in a PDF file is more accurate and consistent across multiple platforms and software. This is due to the presence of **Carousel Object System** *(COS) objects*. CO*S objects,* or simply *objects* are the fundamental building blocks of a PDF file. A PDF file is a collection of thousands of objects. There are eight main types of objects which includes:

1. **Boolean Object:** This is a Boolean data object similar to one in other programming languages. It stores either a *true* or *false* value.
2. **Numeric Object:** PDF standard has two types of numerical object: integer or floating points values.

3. **String Object:** This object stores sequence of 8-bit bytes representing characters. It can be written in two ways: as a sequence of characters enclosed by parentheses: **(** and **)** or as hexadecimal data enclosed by single angle brackets: **<** and **>**.
4. **Name Object:** This object uniquely defined sequence of characters preceded by forward slash: **/**
5. **Array Object:** This object stores heterogenous collection of objects in one-dimensional format inside square brackets separated by white spaces: **[** and **].**
6. **Dictionary Object:** This object represents a key-value pair, logically similar to dictionary variable in other programming languages. The key here always represents a *Name Object***.** However, the value can be any type of object such as String, Stream, Array or even a Null**.** The pair is enclosed by double angle brackets: **<<** and **>>**.
7. **Stream Object**: Pages in PDF file stores content as an arbitrary sequence of 8-bit bytes in Stream objects. The byte sequence represents collection of images, texts and font types and its description.
8. **Null Object:** This object is analogous to Null or None values in other programming languages. Setting a value of any other objects to Null signifies the absence of value for that object.

Some examples depicting different objects in a PDF file:

```
% Example depicting different Objects present in PDF standard
% a simple Dictionary Object incorporating different types of Objects
<<
      /Type /Page % /Type and /Page is a Name Object, /Type is a Key and
/Page is a Value
      /Author (John Doe) % John Doe is a String Object
      /MediaBox [0 0 612 792] % [0 0 612 792] is an Array Object
      /Resources
            <<
                  /Font <</F1 2 0 R>>
            >>
      /Rotate 90 % 90 is a Numeric Object
      /Parent 4 0 R % 4 0 R is a reference to an Object
      /NewWindow true % true is a Boolean Object
      /Contents
            << % Stream Object attribute dictionary
                /Filter     /FlateDecode
                /Length     35
            >>
            stream
                  quick fox jumped over the lazy dog % Stream Object data
            endstream
>>
```

There are two ways to declare an object in a PDF file:

1. **Direct Object**: It is created directly (inline) in the file.
2. **Indirect Object**: It is called in (indirectly) by using reference, which in case will always be the object number.

To use an indirect object, we must first define an object using the object number. The following small example illustrates the difference between Direct and Indirect objects.

```
<</Name (I am Groot)>> % a direct object
3 0 obj         % object Number 3, generation 0, another direct object
<<
     /Name (I am Groot)
>>
endobj

<</Name 3 0 R>> % an indirect object reference
4 0 obj          % another indirect object reference
<<
     /Name 3 0 R
>>
endobj
```

Each object can be uniquely identified in each PDF file, having unique generation ID. Every object is then mapped using cross-reference table present in PDF file.

The Four Sections of a PDF file

Shown in the Figure 2-1, every PDF file has 4 distinct sections:

1. **Header:** This section stores information about the version of the PDF standard.
2. **Trailer:** This section stores a dictionary object which direct a PDF Reader where to start rendering the PDF file.
3. **Body:** This is the content section of the PDF which contains all the 8 different types of objects we have described.
4. **Cross-reference table:** This is the last section of the PDF file that provides random access to every object defined in the PDF file.

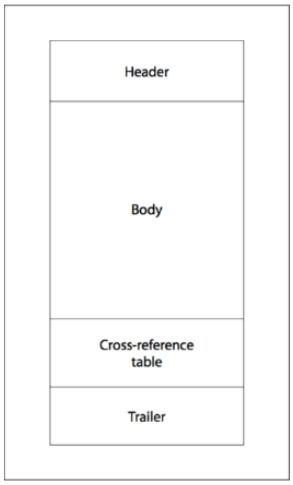

Figure 2-1. Four Sections of PDF

Document structure of a PDF file

The PDF content visible to a User is organized as a Document tree. Document tree starts with a **Root** object, called *Catalog*. Shown in Figure 2-2, the **Root** object is a dictionary object which has two important child objects:

1. **Outlines:** This key has value which references to the Outline or Bookmark Tree of a PDF file.
2. **Pages:** This key has value which references to the Page Tree of a PDF file.

An example showing sample Catalog

```
% Catalog object Example
1 0 obj
<<
      /Type /Catalog
      /Pages 22 0 R % Root object
of Page Tree
      /PageMode /UseOutlines
      /Outlines 23 0 R % Root
object of Outline Tree

>>
endobj
```

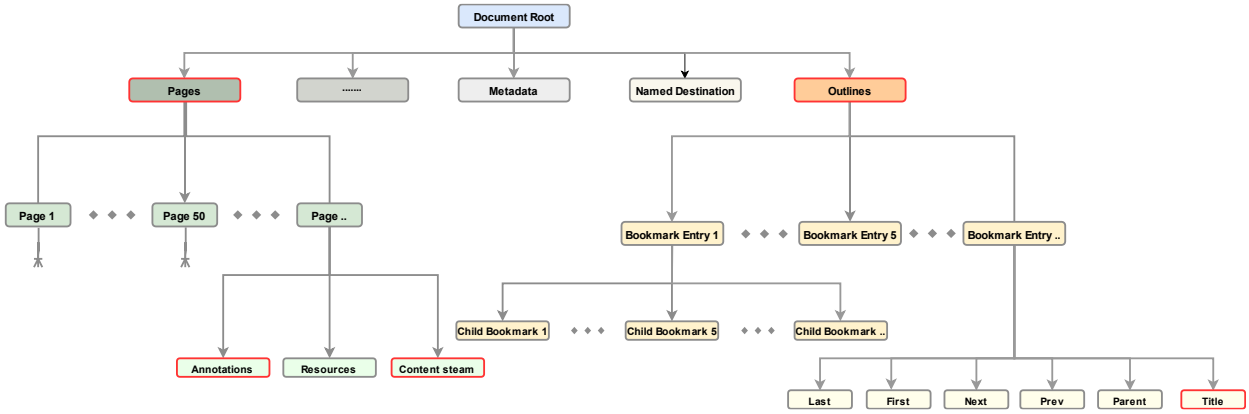

Figure 2-2. The Document tree of a PDF file

The Outline Tree

Outline tree helps a User to navigate interactively in a PDF file using bookmarks. Bookmarks usually have hierarchical structure, i.e., presence of child bookmark objects which have their own sub-trees. The hierarchy serves as a visual table of content for a PDF file. Clicking a bookmark can have following affects:

a. Opens an internal destination page.
b. Opens an external PDF file or performs an action, usually expanding a child bookmark sub-tree.

Root of this tree is an Outline dictionary and is stored as a value for *Outlines* key. Following code example shows standard for describing a bookmark in a PDF file:

```
0 obj % declaration of Root object for the Outlines key
<<
      /First      12 0 R % indirect reference to its first child bookmark
      /Last 22 0 R % indirect reference to its last child bookmark
      /Count      11 % count of its child bookmarks
>>
endobj
12 0 obj % declaration of first child bookmark of the Root object
<<
      /Title      (Story of Groot) % title name of the child bookmark which
is visible to the User
      /Parent     11 0 R % indirect reference to Root object, which is the
parent of this child
      /Next 19 0 R % indirect reference to next sibling bookmark
      /First      13 0 R % indirect reference to its own first child
bookmark, shows presence of sub-tree
      /Last 18 0 R % indirect reference to its last child bookmark
      /Count      -6 % count of its child bookmarks, negative sign directs
the PDF reader to collapse those child bookmarks unless expanded by user
      /Dest [33 0 R /XYZ 0 792 0] % indirect reference to destination page in
the PDF file which will be opened when the title is clicked by user
>>
endobj
```

The Page Tree

All pages in the PDF file are the children of the child object *Pages* present in Document tree. Root of the Page tree is an object stored as a value for */Pages* key. All the pages present in the PDF file are the children of Page Tree. Below code shows the declaration of a Page object:

```
10 0 obj % page object declaration
<<
 /Type /Page
 /Parent 5 0 R % points to the Page Tree Root object
 /Resources 3 0 R
 /Contents 9 0 R % contains the content of the page which will be displayed
in PDF reader
 /Annots [35 0 R] % points to the sole annotation object link present in the
page
>>
endobj
```

Each child page of a Page tree encapsulates following important dictionary keys:

a. **Content or Content stream:** This key has the direct or indirect reference to stream object which has certain sets of instructions used for interpreting content display in the page. It has instructions for creating tables, text formatting, font formatting and annotations or links appearances. Content stream has pairs of *operand* and *operator*, with operand preceding an operator. Following code snippet shows Content stream declaration in a PDF file:

```
9 0 obj
<<
   /Length 31 /Filter /FlateDecode
>>
stream %displaying a blue colored text
   BT % begin instructions
   0 0 1 rg    % 0 0 1 is an operand, tells the PDF reader to parse
operand 0 0 1 as RGB value
   (Story of Groot) Tj % Tj is an operator, tells the PDF Reader that
operand is a text
   ET % end of instructions
endstream
endobj
```

b. **Resources:** It is a dictionary containing information about media which is present in the page, i.e., image data, font data or audio data. Key name is */Resources*.

c. **Annotations:** Annotations are clickable actions in PDF. This dictionary key holds an Array object that contains indirect references to all the clickable locations present in the page. They are used for visiting destination pages, perform executable actions like playing an audio or video, opening a note, launching an external application and opening non-PDF files. Annotations' Key name is */Annots*. There are several actions available in PDF standard, four important actions which concerns to us are:

   I. **GoTo:** This action takes the User to destination page of the currently opened PDF file.
   II. **GoToR:** This action takes the User to destination page of external or remote PDF file.
   III. **URI:** Like hyperlink in Webpage, it resolves the uniform resource identifier in PDF file.
   IV. **Launch:** This action opens a non-PDF file or an application.

GoTo and **GoToR** actions provide additional controls like setting specific location and magnification factor or Inherent-zoom of destination page, which is defined using */D* or*/Dest* key. Below code snippet shows example of different annotations:

```
35 0 obj % action for internal
Link or GoTo action
<<
   /Type /Annot
   /Subtype /Link
   /Rect [171 600 220 630] %
location of the rectangular box
which encloses the linked texts
   /A
   <<
         /Type /Action
         /S /GoTo % action type is
GoTo
         /D [39 0 R /XYZ 0 10000
0] % /D is destination to jump,
can have array object or string
object as a value
         % format is [page /XYZ
left top zoom], page is designated
page object number, left and top
are coordinates and zoom is
magnification
   >>
   /Border [0 0 0] % 0 0 0 means
no border around the linked text
>>
endobj

36 0 obj % action for external
Link or GoTo-Remote action
<<
   /Type /Annot
   /Subtype /Link
   /Rect [171 700 220 730]
   /A
   <<
         /Type /Action
         /S /GoToR % action type
is GoToR
         /F (ADRG.pdf) % file path
including file name
         /D [51 0 R /XYZ 0 10000
0]
         /NewWindow true
   >>
   /Border [0 0 0]
>>
endobj

37 0 obj % action for opening an
application or another file
<<
   /Type /Annot
   /Subtype /Link
   /Rect [171 800 220 830]
   /A
   <<
         /Type /Action
         /S /Launch % action type
is Launch
         /F (adsl.xpt) % file path
including file name
   >>
   /Border [0 0 0]
>>
endobj
```

All the issues in the Define.pdf are around Bookmarks, Annotations and Content stream. One of the best aspects of PDF standard is separation of Content stream from annotations and bookmarks. This separation has helped us to solve the issues in Define.pdf with little interference with the overall content of the PDF file.

DEFINE.pdf ISSUES

Before any trip, it is wise to have a roadmap to know where you are going, in order to get to your destination. In the domain of clinical trials and electronic submissions to a regulatory agency; the Define.pdf/xml function as a road map to the data and summary reports that SAS produces. SAS is a powerful tool for managing the metadata in which the Define.pdf is reporting upon. It can also be used to generate the Define.pdf. SAS can effectively fix issues in define.xml, however, there are instances where there are errors or imperfections within the PDF which SAS cannot remedy. In this example, SAS organizes

the metadata information in an Excel file. This file is uploaded to Pinnacle 21 Enterprise solution, which then generates the define.pdf. Upon initial review of the bookmarks, the reviewer immediately sees some issues in the PDF bookmarks as shown in Figure 3-2:

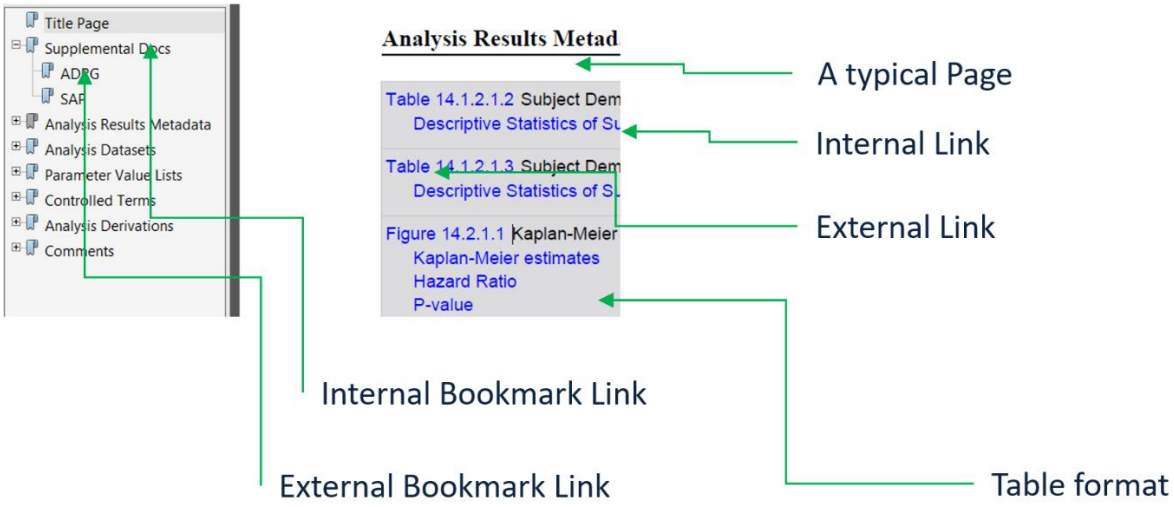

Figure 3-1, A typical PDF page, with annotations links and bookmarks links

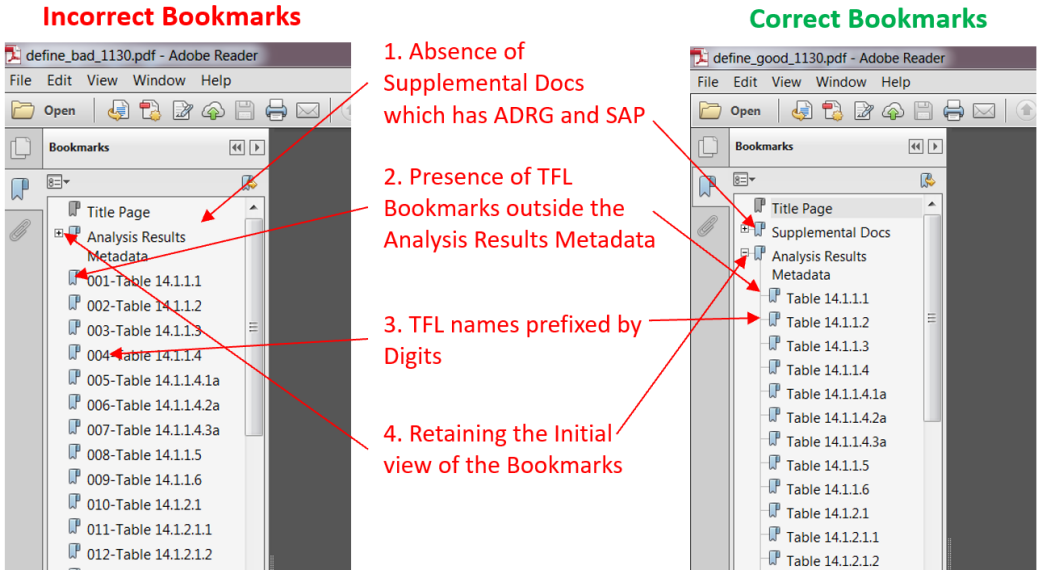

Figure 3-2, Issues present in bookmarks of Define.pdf

APPLYING FIXES

The issues shown above requires a solution to perform the following actions:

1. Insert a new bookmark at a position with a standard name such as “Supplemental Docs”, and have it link to an external file.
2. A series of bookmarks for summary tables appear to be at the wrong level. They need to be moved to become a child of the parent bookmark named “Analysis Results Metadata”.

3. The user defined bookmarks have a special prefix such as “001-“ which were used to sort all the summary tables. This prefix text label needs to be removed.
4. When the user first opens the PDF file, the many parent/child elements within the bookmarks are collapsed. The requirement for this define.pdf is for all the bookmarks to be expanded so the user can see all the children bookmarks upon initial view.

The bookmark is only a small part of the PDF and already, the project has identified a series of issues. Some items are enhancements which can be considered nice to have but others are essential for a successful submission. The challenge then is to identify a solution since SAS does not have the capabilities to read and manipulate an existing PDF file. Our team consists of mainly SAS programmers, so the challenge was to identify a solution outside of SAS as described in the next section.

TOOLS AND APPROCHES

The above issues made us aware of the core requirements for the tool to be developed:

1. **Bookmarks**:
   - Manipulate titles of the bookmark objects and several of its properties.
   - Add or delete child level bookmarks which can be external or internal bookmarks.
2. **Annotations**:
   - Identify and categorize all annotations in the current PDF file.
   - Add or delete annotation objects at page level and manipulate the destinations.
   - Correctly map internal links with TFL (Tables Figures and Listings) headings in the ANASPEC (Analysis Specifications) file.
3. **Text Manipulations:**
   - Decode content from the Content stream object of the page.
   - Perform regular expression (regex) pattern search to identify certain keywords in all pages.
   - Edit keywords to fix the related issues and add it back to content stream.
   - Identify names of annotations from Content stream and map it with the annotation objects.

Several design constraints were also present. The tool to be developed should not have any third-party dependencies. This would have required installation on the targeted System in which it will run on. The tool should be flexible enough for incorporating methods in correcting unforeseen issues. This section further describes the “PDF Tools” projects through all the existing tools evaluated. It illustrates the strengths and weaknesses of related tools and how the combination of SAS and Python emerged as the final combination for tools of choice.

There are numerous tools developed under different programming languages that can perform generic manipulations of PDF documents; such as: splitting a PDF into separate PDFs, deleting a page, extracting texts from pages, converting a Page into bitmap, etc. We tested several tools on a sample Define.pdf file and found out that they barely cover our core requirements. *CPDF* from Coherent Graphics Ltd, can only add or delete internal bookmarks. P*DFtk* from PDF Labs was designed around extracting content from the

pages of a PDF. There was no way it could manipulate the PDF's content. *PDFMiner* is a library written in Python used for extracting information from PDF files and is incapable of modifying the file. Like *PDFMiner*, *MuPDF* is also catered towards extracting text from PDF file. *iText* developed by iText Group NV, is an extensive library based on Java and C# programming languages. It can create and manipulate PDF file on object level. However, it has language dependencies which needed installation and has a significant learning curve. We settled upon *PyPDF2* from Phaseit Inc and Mathieu Fenniak, which is a python-based library. This library can do PDF manipulations at the object level. Except for PyPDF2 and iText, most of the solutions mentioned were developed in closed ecosystem with no support intended for community-based development; hence there was no way we could have extended their capabilities to cover all the issues identified in Define.pdf.

Why Python?

We have seen that PDF file is a collection of objects, and all objects are interlinked using Dictionary objects using key-value pairs under a Document tree. Everything in Python is an object which is analogous to PDF object. Dictionaries and List in Python are mutable and are easy to create. This helps us in creating complex multi-dimensional dictionaries and list in Python using only few lines of code. In terms of readability and preparedness, Python is a go-to language for a beginner. Hence the maintenance and expansion of the tool can easily be taken by anyone who previously had no background in Python language.

Methodology

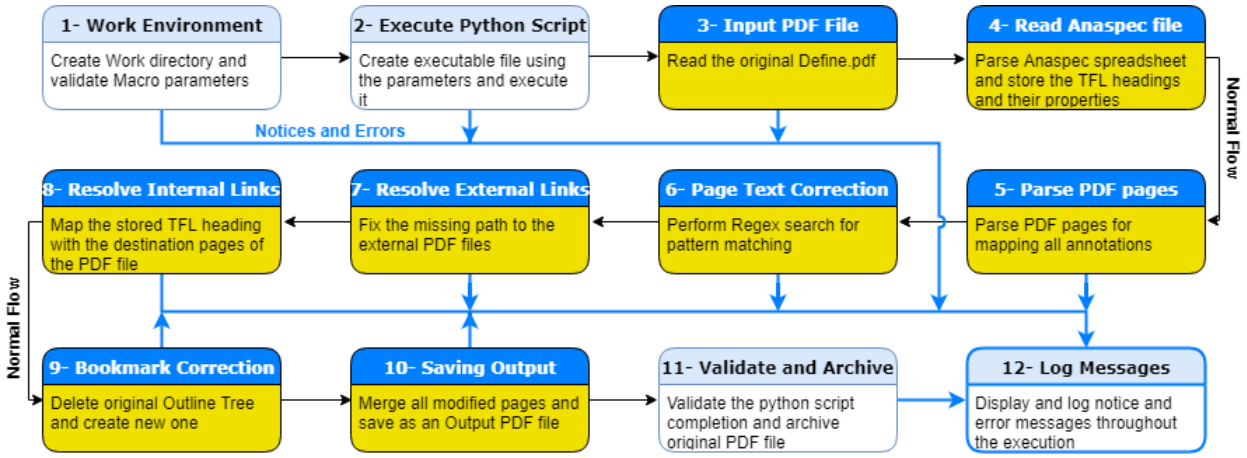

Figure 4-1, Workflow of our operation on Define.pdf

Figure 4-1 presents the workflow of our tool. Blocks 1, 2, 11 and 12 are part of SAS macro %pdftools, and rest of the blocks are part of the Python script which we developed using PyPDF2. In block 1 and 2, the macro accepts and validates the parameters for the paths to Define.pdf and supporting Excel file. On successful validation the macro creates and executes a Windows batch file using provided parameters and moves to next stage where operation is then handled by Python script.

Block 3 stage reads Define.pdf file and parses the PDF Document tree. A pythonic representation of the Document tree is then created. Each node in this separate tree is a Python dictionary object having key-

value pairs based on the original objects present in the Document tree. This separate tree helps us to easily traverse to targeted nodes and modify their key-value pairs based on our core requirements. In Block 4, the script reads the Excel file for storing TFL information. This is the same Excel file which is used by Pinnacle 21 for getting Metadata information. Block 5 is responsible for retrieving all the annotation objects present in the PDF file. This stage identifies the external and internal broken links by looking for malformed annotation object dictionaries.

Blocks 6 to 9 are responsible for performing corrections on the PDF file. In block 6, the byte stream content, stored as a value of the */Contents* key in the page dictionary object is decoded to reveal the content of the page. Subsequently regex is used for identifying certain text patterns. On positive identification, manipulations are done on them based on our business requirement. The modified content is encoded and saved back to the */Contents* key. In the next two stages, block 7 and 8, the missing destination path for the broken annotations are corrected using the help of TFL information read previously. The last stage in the corrections, block 9, using original Outline tree, creates new Outline tree with prefixes removed from the title, deletion of redundant child bookmarks, and addition of missing external bookmarks. The newer Outline tree replaces the original one. Following code snippet in Python shows how the content was edited in */Contents* key:

```
content = pageRef['/Contents'].getObject()#getting value of /Contents key
Content = ContentStream(content, pageRef)#decoding the content bytestream
for count in range(0, len(Content.operations)):
    operands, operator = Content.operations[count]
    try:
        for index in range(0, len(patterns)):
            pattern = patterns[index]
            altText = altTexts[index]
            if operator == b_('Tj') and re.search(pattern, operands[0],
re.M|re.I): #using regex to match pattern
                text = operands[0]
                if altText == None:#remove annotation action from the text
                    replaceString =  text
                    temp, others  = Content.operations[count-1]
                    temp[2] = NumberObject(0)# changing [0,0,1] to [0,0,0],
blue to black color, RGB
               elif pattern == r'\bPage \d+$': #adding Page X of TotalPages
                    replaceString = text+altText
                else:
                    replaceString = re.sub(pattern, altText, text)# looking
for *-Table occurrence in the string, replace the pattern with given string
                operands[0] = TextStringObject(replaceString)
    except Exception as e:
        print ('WARNING: problem in deciphering the stream page' +str(e))
        print(traceback.format_exc())#print stacktrace of the error
pageRef.__setitem__(NameObject('/Contents'), Content)
```

In block 10 stage, the modified pages are merged to form an output PDF object, which is then written onto the disk as a new PDF file. In block 11, the SAS macro checks the presence of new PDF file, on positive match, the macro renames and move the original PDF file to an Archive folder. The output PDF file is renamed as Define.pdf and a separate copy of the output PDF file is also added to the Archive folder and

is paired with the Original Define.pdf using timestamps. Block 12 is the log module, which continuously receives the messages from previous blocks and pipes it to the SAS log output.

LOG MESSAGING

This section describes the method of sending messages from the Python program to the SAS log as the program is executed. It demonstrates how this seemingly small task can significantly enhance the usability of the tools for end users.

This will be illustrated through some code as shown here:

```
*** Apply the Python program update with Analysis Result Spec ***;
curline = '"' || strip("&pdftools") || '\Miniconda2\python.exe" "' ||
          strip("&pdftools") || '\pdfeditor\src\pdftools.py" ' ||
          '--specFile "' || strip("&anaspec") || '" ' ||
          '--inFile "define.pdf" --outFile "define_good.pdf" ';
put curline;

*** Run the Python program and pipe the results to SAS log ***;
filename tasks pipe "&fixpdf";
data _null_;
   infile tasks lrecl=1500 truncover;
   length logLines $200;
   input logLines $char200.;
   put logLines;
run;
```

Example Python Code:

```
### Reads bookmarks and return a tuple array ###
def get_toc(pdf_path):
    infile = open(pdf_path, 'rb')
    parser = PDFParser(infile)
    document = PDFDocument(parser)

    toc = list()
    for (level,title,dest,a,structelem) in document.get_outlines():
        toc.append((level, title))
    return toc

print ("NOTE: *** You are currently running Python PDFTools version 1.0 ***")
      sys.stdout.flush()
      NumofPages = input.getNumPages()
print ("NOTE: *** %d pages processed Successfully ***"%(NumofPages))
sys.stdout.flush()
print ("NOTE: *** Processing Document "+ args.inFile +" ***")
```

What is shown in the SAS log is:

```
NOTE: *** 386 pages processed Successfully ***
NOTE: *** Processing Document define.pdf ***
```

Example error checking Python:

```
#checking if the CONFIG file has been passed by the user
if args.configFile: # if passed then check below for extension
    if(not (re.search(r'.xlsx', args.configFile, re.M|re.I))): #check if the
file has proper extension format.
        print ("ERROR: *** Cannot find .xlsx extension for the config file
***")
        print ("-h or --help for more running the script in proper format")
        print ("Exiting now")
        progExit()
    print ("NOTE: *** Fetching operation list from the Configuration
spreadsheet ***")
    try:
        oprMap = readConfExcel(filename =args.configFile)
    except Exception as e:
        print ('ERROR: *** Some error occurred while reading the config
spreadsheet ***')
        print(traceback.format_exc())
```

CONCLUSION

This will summarize the importance of how DEFINE.pdf is used as part of an electronic submission to the FDA and how there are challenges in the way it is produced. This paper illustrates how automating the corrections and updates to the PDF can more efficiently produce the final PDF as compared to a manual process. The programs presented shows the gains of time savings leading to a more accurate final PDF report. This paper summarizes how integrating different languages with different strengths can result in efficient solutions.

REFERENCES

SAS Documentation, SAS Institute Inc.
PDF standard ISO 32000-2, 2008, Adobe Inc.
PDF Explained by John Whitington, O'Reilly
pdfmark Reference Manual, Adobe Inc.
PyPDF2, Phaseit, Inc. and Mathieu Fenniak
iText, iText Group NV
Coherent PDF Tools (CPDF), Coherent Graphics Ltd
MuPDF, Artifex
PDFtk, PDF Labs
PDFMiner, Yusuke Shinyama