Paper 3009-2019

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

Sy Truong, Pharmacyclics LLC; Jayant Solanki, Pharmacyclics LLC

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 different PDF 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 when opened on a PDF reader
- **Text Editing –** Manipulating text content of pages in a PDF file

The primary Users of this PDF Tool are SAS programmers. Thus, a SAS macro has been written which allows users to specify the PDF inputs, perform validation checks and provide log messages throughout the tool runtime. The %pdftools macro described in this paper reads the Excel file and other sources to gather business rules. After performing series of checks on the PDF inputs, it then integrates with a Python program which then apply corrections to the PDF file. There are numerous 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. The PDF standard *ISO 32000-2* defines the open-source standards used 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*. COS *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 include:

1. **Boolean Object:** This is a Boolean data object which is similar to ones 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-point 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, 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 a reference, which in this 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 a unique generation ID. Every object is then mapped using cross-reference table present in PDF file.

The Four Sections of a PDF File

As shown in the Figure 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 directs the PDF Reader to the starting point of a PDF file rendering.
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.

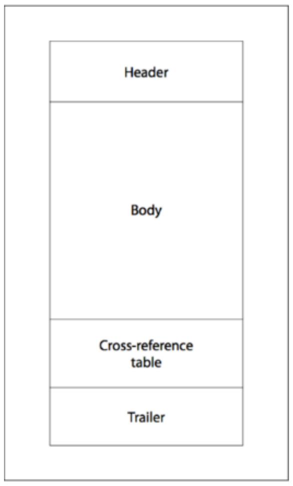


Figure 1, Four Sections of PDF

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, 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
```

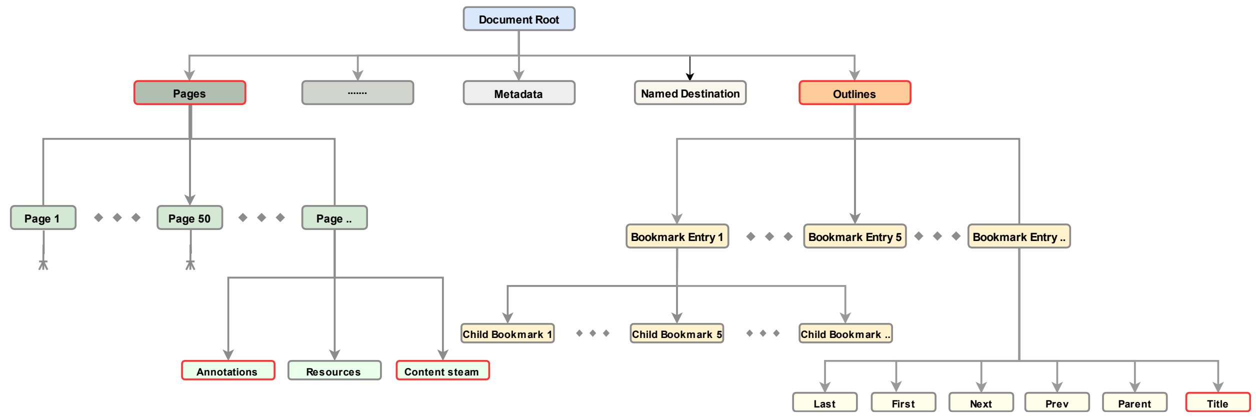


Figure 2, The Document tree of a PDF file

The Outline Tree

Outline tree helps a User to navigate interactively in a PDF file using bookmarks. Bookmarks usually have a 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:

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

The 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 considered the children of the child object *Pages* represented in Document tree. The Root of the Page tree is an object stored as a value for */Pages* key. All the pages 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:

1. **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
   ```

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

3. **Annotations:** Annotations are clickable actions in PDF. This dictionary key holds an Array object that contains indirect references to all the clickable locations found in the page. They are used for visiting destination pages, perform executable actions such as playing an audio or video, opening a note, launching an external application and opening non-PDF files. Annotation's Key name is */Annots*. There are several actions available in PDF standard, four important actions which concerns to us are:
    a. **GoTo:** This action takes the User to destination page of the currently opened PDF file.
    b. **GoToR:** This action takes the User to destination page of external or remote PDF file.
    c. **URI:** Like hyperlink in Webpage, it resolves the uniform resource identifier in PDF file.
    d. **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 for 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. Figure 3 highlights some important components associated with Define.pdf which helps the Reviewer to easily navigate to different portions of the document. Upon initial review of the Define.pdf, the reviewer immediately sees some critical issues in the PDF document as shown in Figure 4 (a, b and c):

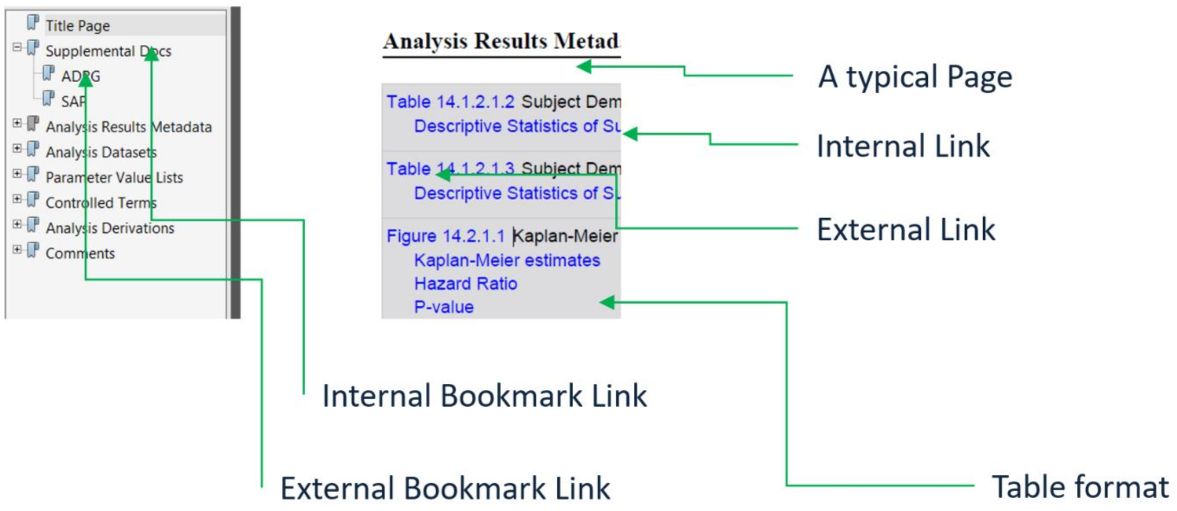
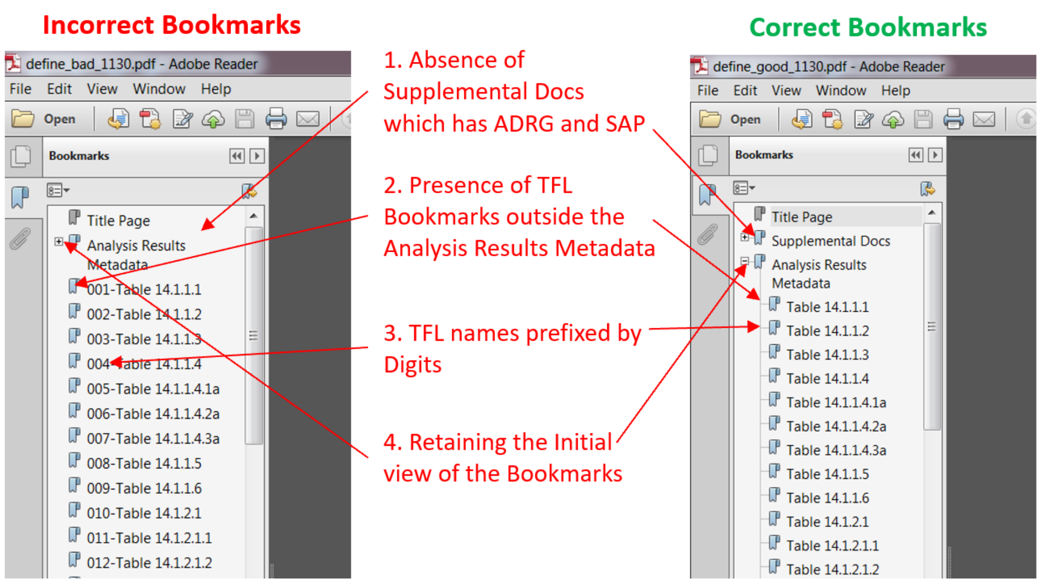


Figure 3, A typical PDF page, with annotations links and bookmarks links


Incorrect Bookmarks
Correct Bookmarks
define_bad_1130.pdf - Adobe Reader
define_good_1130.pdf - Adobe Reader
File Edit View Window Help
Open
Bookmarks

Incorrect Bookmarks:

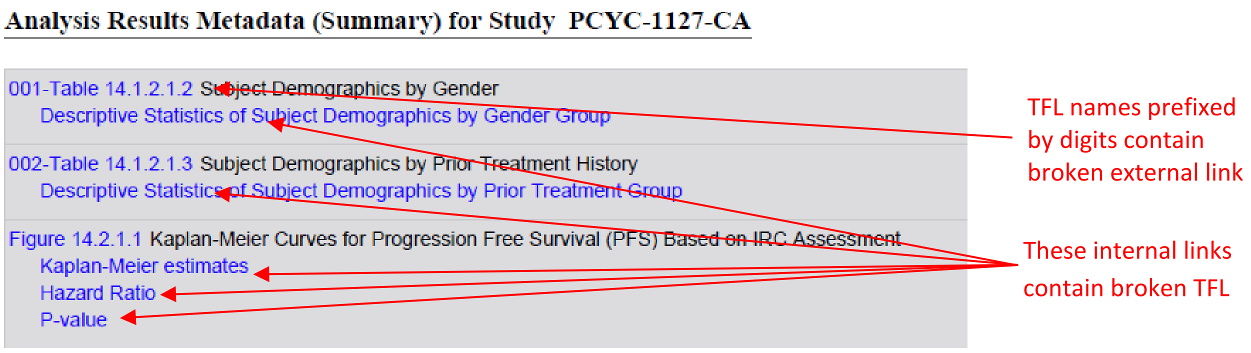
- Title Page
- Analysis Results Metadata
- 001-Table 14.1.1.1
- 002-Table 14.1.1.2
- 003-Table 14.1.1.3
- 004-Table 14.1.1.4
- 005-Table 14.1.1.4.1a
- 006-Table 14.1.1.4.2a
- 007-Table 14.1.1.4.3a
- 008-Table 14.1.1.5
- 009-Table 14.1.1.6
- 010-Table 14.1.2.1
- 011-Table 14.1.2.1.1
- 012-Table 14.1.2.1.2

1. Absence of Supplemental Docs which has ADRG and SAP
2. Presence of TFL Bookmarks outside the Analysis Results Metadata
3. TFL names prefixed by Digits
4. Retaining the Initial view of the Bookmarks

Correct Bookmarks:

- Title Page
- Supplemental Docs
- Analysis Results Metadata
  - Table 14.1.1.1
  - Table 14.1.1.2
  - Table 14.1.1.3
  - Table 14.1.1.4
  - Table 14.1.1.4.1a
  - Table 14.1.1.4.2a
  - Table 14.1.1.4.3a
  - Table 14.1.1.5
  - Table 14.1.1.6
  - Table 14.1.2.1
  - Table 14.1.2.1.1
  - Table 14.1.2.1.2

Figure 4-a, Issues present in bookmarks of Define.pdf



Analysis Results Metadata (Summary) for Study PCYC-1127-CA

001-Table 14.1.2.1.2 Subject Demographics by Gender
Descriptive Statistics of Subject Demographics by Gender Group

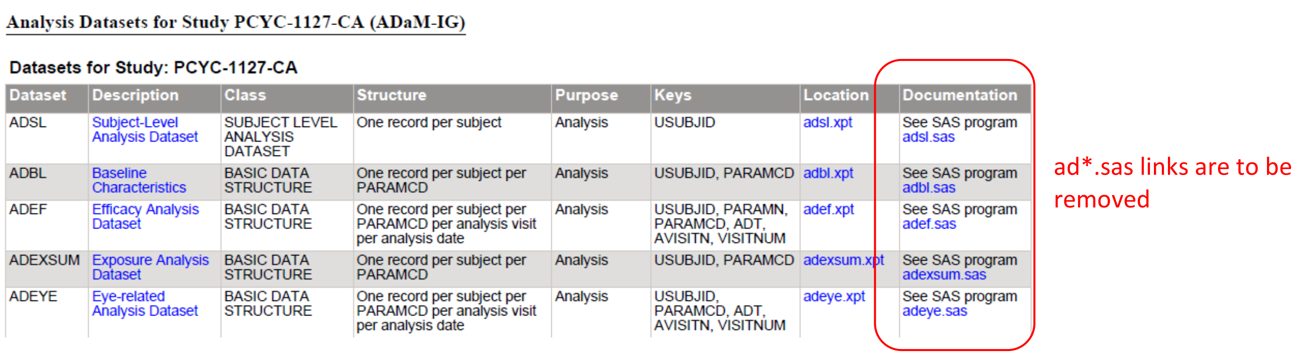
002-Table 14.1.2.1.3 Subject Demographics by Prior Treatment History
Descriptive Statistics of Subject Demographics by Prior Treatment Group

Figure 14.2.1.1 Kaplan-Meier Curves for Progression Free Survival (PFS) Based on IRC Assessment
Kaplan-Meier estimates
Hazard Ratio
P-value

TFL names prefixed by digits contain broken external link

These internal links contain broken TFL

Figure 4-b, Issues present in TFL summary pages



Analysis Datasets for Study PCYC-1127-CA (ADaM-IG)

Datasets for Study: PCYC-1127-CA

Dataset	Description	Class	Structure	Purpose	Keys	Location	Documentation
ADSL	Subject-Level Analysis Dataset	SUBJECT LEVEL ANALYSIS DATASET	One record per subject	Analysis	USUBJID	adsl.xpt	See SAS program adsl.sas
ADBL	Baseline Characteristics	BASIC DATA STRUCTURE	One record per subject per PARAMCD	Analysis	USUBJID, PARAMCD	adbl.xpt	See SAS program adbl.sas
ADEF	Efficacy Analysis Dataset	BASIC DATA STRUCTURE	One record per subject per PARAMCD per analysis visit per analysis date	Analysis	USUBJID, PARAMN, PARAMCD, ADT, AVISITN, VISITNUM	adef.xpt	See SAS program adef.sas
ADEXSUM	Exposure Analysis Dataset	BASIC DATA STRUCTURE	One record per subject per PARAMCD	Analysis	USUBJID, PARAMCD	adexsum.xpt	See SAS program adexsum.sas
ADEYE	Eye-related Analysis Dataset	BASIC DATA STRUCTURE	One record per subject per PARAMCD per analysis visit per analysis date	Analysis	USUBJID, PARAMCD, ADT, AVISITN, VISITNUM	adeye.xpt	See SAS program adeye.sas

ad*.sas links are to be removed

Figure 4-c, Issues present in Analysis dataset pages

Figure 4 highlights core issues which are present in every Define.pdf file generated by Pinnacle21 software. In addition to these core issues, the project has identified a series of smaller issues. Some of these findings can be considered “nice to have” enhancements, but others are essential for a successful submission. The challenge then is to identify a solution, since SAS does not have the capabilities to manipulate an existing PDF file. SAS ODS does provide a solution to produce a PDF along with capabilities of customizing bookmarks. However, this project required the ability to edit and update an existing PDF file. Our team consists of mainly SAS programmers, so it was a difficult to identify a solution outside of SAS as described in the next section.

TOOLS AND APPROACHES

The above issues made us aware of the core requirements which the tool must have in order to achieve:

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 the page level and manipulate the destinations
   - Correctly map internal links with TFL (Tables Figures and Listings) headings in the ANASPEC (Analysis Specifications) file
   - Correctly map external links with TFL headings to the respective PDF files
   - Fix the destination page numbers in the external links mapped to the respective PDF files
   - Modify the inherent zoom factor for every links
3. **Text Manipulations:**
   - Decode content from the content stream object of the page
   - Perform regular expression (regex) pattern search to identify certain keywords upon all pages
   - Edit keywords to fix the related issues and add it back to content stream
   - Identify names of annotations from the content stream and map it with the annotation objects

Several design constraints were also stipulated upon this project. The tool to be implemented 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 to handle future corrections upon unforeseen issues. This section further describes the “PDF Tools” project which started with an evaluation of tools and solutions. The evaluation illustrates the strengths and weaknesses of related tools and how the combination of SAS and Python emerged as the best final combination.

There were numerous tools developed under different programming languages that can perform generic manipulations of PDF documents. This include tasks 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 our

Define.pdf file and found that they did not fully meet all 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. It had not solutions for manipulating 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. We also evaluated *PDFMiner* and *MuPDF* which also catered towards extracting text from PDF file. Another tool evaluated was *iText,* developed by iText Group NV, is an extensive library based on Java and C# programming languages. It can create and manipulate PDF file at the object level. However, it has language dependencies which needed installation and required a significant learning curve. Upon review, 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 a closed ecosystem with no support intended for community-based development; hence there was no way we could easily extended their capabilities to cover all the specific issues identified in Define.pdf.

Why Python?

A PDF file is a collection of objects, and that all objects are interlinked using Dictionary objects with key-value pairs under a Document tree. In a similar manner, we can create object type Data Structures, which can represent similar key-value pairs present in the PDF file. Dictionaries and Lists in Python are mutable and are easy to create. This helps us in creating complex multi-dimensional dictionaries and lists in Python using only few lines of code. In terms of approachability and readability, Python is a go-to language for a beginner. Hence the maintenance and expansion of the tool developed in Python can easily be taken by anyone who previously have little background in the Python language.

Methodology

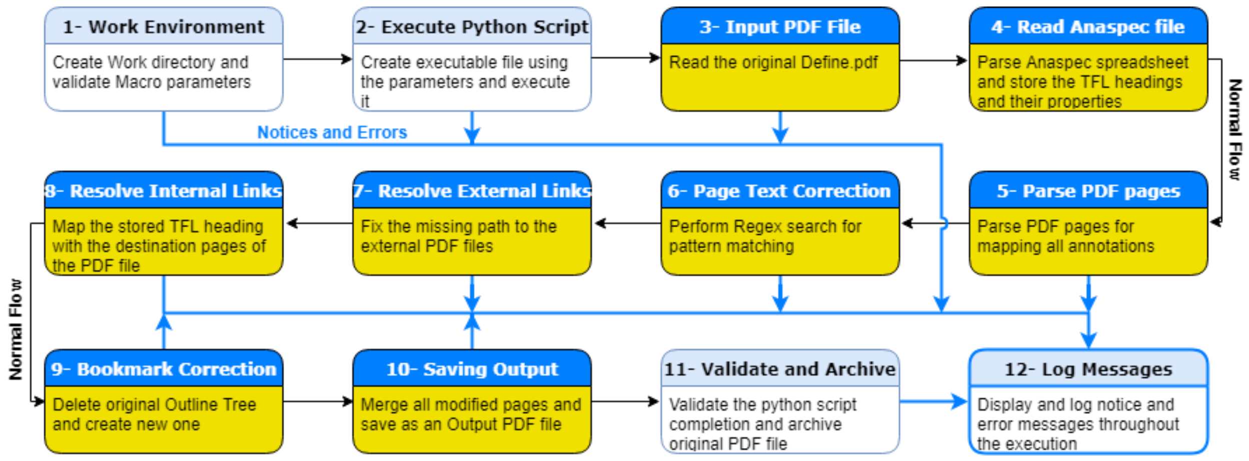

1- Work Environment
Create Work directory and validate Macro parameters
2- Execute Python Script
Create executable file using the parameters and execute it
3- Input PDF File
Read the original Define.pdf
4- Read Anaspec file
Parse Anaspec spreadsheet and store the TFL headings and their properties
Normal Flow
Notices and Errors
8- Resolve Internal Links
Map the stored TFL heading with the destination pages of the PDF file
7- Resolve External Links
Fix the missing path to the external PDF files
6- Page Text Correction
Perform Regex search for pattern matching
5- Parse PDF pages
Parse PDF pages for mapping all annotations
Normal Flow
9- Bookmark Correction
Delete original Outline Tree and create new one
10- Saving Output
Merge all modified pages and save as an Output PDF file
11- Validate and Archive
Validate the python script completion and archive original PDF file
12- Log Messages
Display and log notice and error messages throughout the execution

Figure 5, Workflow of our operation on Define.pdf

Figure 5 presents the workflow of our tool. The blocks represented in steps 1, 2, 11 and 12 are components of the %pdftool SAS macro; while the rest of the steps are comprised of the Python script which has been developed using PyPDF2.

STEP 1-2, Setting up Work Environment and executing Python script: The macro accepts and validates the user specified parameters for the paths to Define.pdf and supporting Excel file. Upon a successful validation, the macro creates and executes a Windows batch command file using macro parameters; moving the process to next stage where file processing is handled by the Python script. Messages generated by the Python script is continuously being read and displayed by the Macro in the form of Notices and Errors. The following SAS code illustrates how the BAT file which invoked the Python program is piped into the SAS log.

```
*** 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;
```

STEP 3, Reading the Define.pdf file: Python script reads the Define.pdf file and parses thru 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 upon the original objects currently in the Document tree. This separate tree helps us to easily traverse through the targeted nodes and modify their key-value pairs based on predefined core requirements. Below is a Python snippet for reading the PDF file and returning the corresponding Python dictionary object:

```
input = PdfFileReader(open(args.inFile, "rb")) # reading the input pdf file
#PdfFileReader is function provided PyPDF2 to read and convert the PDF file
into Python #Dictionary Object
```

STEP 4, Reading the ANASPEC Excel file: The script also reads the Excel file for storing TFL information. This is the same Excel file which is used by the Pinnacle 21 tool to acquire the Metadata information. Metadata information includes file paths to different PDF files, TFL heading and sub-headings. Below is a Python snippet for reading the Excel file:

```
def readSpecExcel(filename = None):
    filemap = {} # dictionary for storing the file mapping
    subLinks = {}# dictionary for storing the sub-links for the Table and
                 # Figures
    try:
        wb = load_workbook(filename=filename, read_only=True)
        ws = wb.active
        rows = ws.rows
        for row in rows:
            if (row[0].value!=None):
                if (str(row[0].value).strip() not in subLinks):
                    subLinks[str(row[0].value).strip()] = []
                filemap[str(row[0].value).strip()] = \
                str(row[3].value).strip()
                subLinks[str(row[0].value).strip()]
                .append(str(row[6].value).strip())
    except Exception as e:
        print ("ERROR: *** ANASPEC file is missing in the working directory
        provided by the User ***")
        progExit()
    return filemap, subLinks
```

STEP 5, Parsing Define.pdf pages: This step is responsible for retrieving all the annotation objects present in every pdf page of Define.pdf. Since the page content displays link names with blue colored texts, and every page has content and annotations dictionary stored separately; we must make sure that the link names and link properties in the annotations are mapped correctly. This step also identifies external and internal broken links by looking for malformed annotation object dictionaries. Figure 6 and code snippets below illustrates this step:

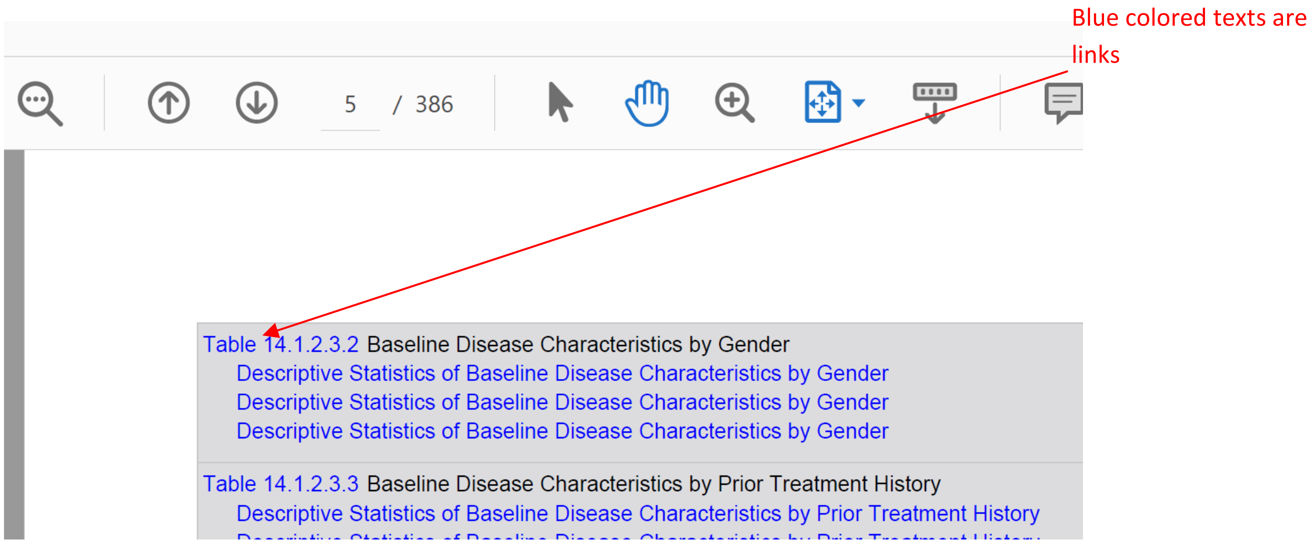


Table 14.1.2.3.2 Baseline Disease Characteristics by Gender
Descriptive Statistics of Baseline Disease Characteristics by Gender
Descriptive Statistics of Baseline Disease Characteristics by Gender
Descriptive Statistics of Baseline Disease Characteristics by Gender

Table 14.1.2.3.3 Baseline Disease Characteristics by Prior Treatment History
Descriptive Statistics of Baseline Disease Characteristics by Prior Treatment History

Figure 6, A page in Define.pdf illustrating links present in the content

5.1 - Below is the Adobe PDF code for Figure 6:

```
2129 0 obj % Page Object
<<
   /Type/Page
   /MediaBox[0 0 612 792]
   /Resources
      <<
         /Font<</F3 849 0 R/F1 2 0 R>>
      >>
   /Rotate 90
   /Annots [2130 0 R 2131 0 R 2132 0 R 2133 0 R 2134 0 R 2135 0 R 2136 0 R
   2137 0 R 2138 0 R 2139 0 R] % contains all internal and External links
   /Contents 2148 0 R % content shown in Figure 6
   /Parent 2069 0 R
>>
Endobj
```

5.2 – Python code for parsing the page content and getting link names:

```
def parseContent(page, debug=False):
    content = page['/Contents'].getObject()
    content = ContentStream(content, page)#creating Contentstream class \
    #instance
    linkNames = []
    flag = False
    for operands,operator in content.operations:
        try:
            if operator==b_('rg') and operands == [0,0,1]:#blue colored text
                flag = True
                text = operands[0]
            if operator == b_('Tj') and flag: # first parameter, word to be \
            #searched, second parameter, word series were word may be present
                flag = False
                linkNames.append(operands[0])
        except:
            print ('WARNING: *** problem in deciphering the stream page ***')
    #add some error control here, what if nothing is found
    return linkNames
```

5.3 – Mapping Link names with their Annot properties:

```
#Each page will have equal count of blue-colored text and Annots elements.
#This helps in mapping link name and link properties
def findLink(page, debug=False):
    processesLinks = {}#stores the mapped links
    page = input.getPage(index)
    linkNames = parseContent(page)#parsing the Contentstream, looking for \
    #text with blue formatting
    key = -1# helps in matching link name and its link properties
    if('/Annots' in page):# proceed only if page has links
        for annot in page['/Annots']:
            try:
                path = None
                key = key + 1
                linkName = linkNames[key]
                obj = annot.getObject()
                if(obj['/A']['/S'] == '/GoToR' or obj['/A']['/S'] == \
                '/Launch' or obj['/A']['/S'] == '/URI'):#look for external \
                #links only
                #store the link properties here
                    path = obj['/A']['/F']
                    processesLinks[annot.idnum] = {'Name' : linkName ,
                    'Path'  : path, 'Dest' : None, 'Type' : obj['/A']['/S'],
                    'ObjectId' : annot.idnum}
                else:#for /GoTo, internal links
                #store the link properties here
                    path = obj['/A']['/D'].getObject()
                    processesLinks[annot.idnum] = {'Name' : linkName ,
                    'Path' : path, 'Dest' : None, 'Type' : obj['/A']['/S'],
                    'ObjectId' : annot.idnum}
            except:
                print ("ERROR: *** Fatal error occurred while looking for
                links inside the input pdf, exiting now ***")
                print(traceback.format_exc())
                progExit()
    return processesLinks
```

STEP 6, Correcting the Page Text using Regex: In this step, 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. Upon positive identification, manipulations are applied to them based on our business rules. The modified content is encoded and saved back to the `/Contents` key. Below is the Python snippet for correcting page number in the PDF file:

```
def removeText(page, patterns, altTexts):
    content = page['/Contents'].getObject()#fetch the value from the content
#key
    Content = ContentStream(content, pageRef)#creating contentstream class
#instance
    Operations = Content.operations #fetch the list of (operand, operator)
#tuple
    #now iterate through the list
    for item in Operations:
        operands, operator = item # fetching operand and operator
        #now check for pattern in the operands based on operator type, for
#example
        if operator == b_('Tj') and re.search(pattern, operands[0],
re.M|re.I): # tj is one of the key used by Adobe PDF for formatting txt
            if pattern == r'\bPage \d+$': # check for Page Number
                replaceString = text+altText
                operands[0] = TextStringObject(replaceString)#replace the
operand with the #alternate Page number
    pageRef.__setitem__(NameObject('/Contents'), Content)#add the modified
#text back to Content key
```

STEP 7, Correcting external links: This step takes care of correcting the external links. It performs operations like deleting old links, creating new one, fixing the destination pages and correcting inherent zoom. It uses TFL metadata fetched from the Anaspec Excel file. Below are two code snippets, illustrating the process:

7.1 – Removing the existing Link:

```
def removeExtLinks(page, objectId=None, debug=False):
    if "/Annots" in page:# Annots key has all the links of particular page
        temp = page["/Annots"][:]
        if (objectId == None):#delete all the external links
            del page['/Annots']
            page[NameObject('/Annots')] = ArrayObject()
            for annot in temp:
                obj = annot.getObject()
                if(obj['/A']['/S'] == '/GoToR' or obj['/A']['/S'] == \
                '/Launch' or obj['/A']['/S'] == '/URI'):#look for external
                # links only
                    if debug:
                        print ("NOTE: *** Deleted object id
                        "+str(annot.idnum)+" ***")
                    continue
                else:
                    page[NameObject('/Annots')].append(annot)
            if debug:
                print ("NOTE: *** Deleted all the external links ***")
        else:#delete the particular indirect object, or the external link
            # print page
            del page["/Annots"]
            page[NameObject('/Annots')] = ArrayObject()
            for annot in temp:
                if (annot.idnum == objectId):
                    if debug:
                        print ("NOTE: *** Deleted object id "+str(objectId)+"
                        ***")
                    continue
                else:
                    page[NameObject('/Annots')].append(annot)
    return page
```

7.2 – Adding new external link:

```
def addGOTOR(output, page, path, rect, border=None, destination = 0,
debug=False):
    C = [0,0,1]#border color
    CObject = ArrayObject([NameObject(n) for n in C])
    sublink = DictionaryObject()
    sublink.update({
    NameObject('/S'): NameObject('/GoToR'),#Using GoToR type
    NameObject('/F'): TextStringObject(path),#correct file path given here
    NameObject('/NewWindow true /D') :
    ArrayObject([
       NumberObject(int(destination)-1), NameObject('/XYZ'),
       NameObject(0), NameObject(10000), NameObject(0.0)
    ])#inherent zoom, last number should be zero, also tells PDF Reader to
    #open link in new window
    });
    lnk = DictionaryObject()
    lnk.update({
       NameObject('/Type'): NameObject('/Filespec'),
       NameObject('/Subtype'): NameObject('/Link'),
       NameObject('/Rect'): rect,
       NameObject('/H'): NameObject('/I'),
       NameObject('/Border'): ArrayObject(borderArr),
       NameObject('/C'): CObject,
       NameObject('/A'): sublink
    })
    lnkRef = output._addObject(lnk)# creating new object for indirect
    #reference
    if "/Annots" in page:
        page['/Annots'].append(lnkRef)
    else:
        page[NameObject('/Annots')] = ArrayObject([lnkRef])
    return output, page
```

STEP 8, Correcting internal links: Similar to the previous step, this step repeats the process for internal links.

STEP 9, Bookmark corrections: In this step, the script, 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. The 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 stack trace of the error
pageRef.__setitem__(NameObject('/Contents'), Content)
```

STEP 10, Saving Output: During this step, the modified pages are merged to form an output PDF object, which is then written back onto the disk as a new PDF file.

STEP 11, Validating new Define.pdf and archiving the original Define.pdf: The SAS macro checks for the presence of a new PDF file. Upon a positive match, the macro renames and move the original PDF file to an Archive folder. The output PDF file is then renamed to 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.

STEP 12: The log module, which continuously receives the messages from previous steps 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.

The following SAS code example illustrates the log messaging approach:

```
*** 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;
```

The following Python code example works in conjunction with the SAS 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 +" ***")
```

The following is then generated in the SAS log for the users to review:

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

The following Python code perform error checking:

```
# 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

SAS is a very powerful platform for managing metadata and generating reports upon many business processes. This paper presents a specific example where SAS can be used to generate a Define.pdf report documenting information on an electronic submission for a clinical trial to be submitted a regulatory agency such as the FDA. The specific example selected illustrates how a very niche project containing many constraints and requirements that even SAS cannot fully have a complete solution for. The challenge was then to identify a solution which not only can manage and report on the metadata, but also update and correct the information in the PDF formatted report, once the PDF has been created. There are a multitude of solutions which was evaluated but what emerged as the best solution was the integration of SAS and Python along with its accompanied open source libraries. The leveraging of the power of SAS along with the flexibility, maintainability and scripting agility of the Python language allowed for the manipulation and corrections to the final PDF report. Perhaps future releases of SAS will incorporate enhanced functionality that would enable users to more universally manipulate any PDF file generated by SAS or thru other tools. In the meantime, creative integration approaches such as the one presented in this paper can be applied.

PDF as a file format is cross platform and is universal, however, in this case, it is used for a very specific purpose. The Define.pdf project is also unique to the pharmaceutical industry with intricate requirements which comes with a highly regulated industry. The approach presented in this paper shows how the synthesis of very different programming languages of SAS and Python can be combined to form a unique solution that successfully accomplish the task. In addition to having SAS execute operating system commands and using pipes to communicate with external tools; perhaps future API can be developed to open up SAS to other programming languages such as Python. In a dynamic and constantly evolving technology environment, the best solution is not necessarily applying one set of technologies; rather a creative integration of multiple technologies and programming languages is needed to truly provide the most effective and optimal solution that can meet any specific challenge.

REFERENCES

- SAS Documentation, SAS Institute Inc.
- PDF standard ISO 32000-2, 2008, Adobe Inc.

- PDF Explained by John Whitington, O'Reilly
- Pdfmark Reference Manual for Adobe PDF standard, Adobe Inc.
- PyPDF2: A pure Python library for performing wide array of PDF file manipulations, Phaseit, Inc. and Mathieu Fenniak
- iText: Harness the power of PDF with the iText PDF Engine, iText Group NV
- Coherent PDF Tools (CPDF): Command line Tools for wide range of professional and robust modification of PDF files, Coherent Graphics Ltd
- MuPDF: Lightweight PDF, XPS, and E-book viewer, consists of a software library, command line tools, and viewers for various platforms, Artifex
- PDFtk: A simple tool for doing everyday things with PDF documents, PDF Labs
- PDFMiner: A tool for extracting textual information from PDF files, Yusuke Shinyama

CONTACT INFORMATION

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

Sy Truong
Principal System Analyst, Pharmacyclics LLC
(669) 224-1107
struong@pcyc.com
https://www.linkedin.com/in/sy-truong-979b382

Jayant Solanki, MS (CS)
Statistical Programmer I, Pharmacyclics LLC
(716) 598-9115
jsolanki@pcyc.com
https://www.linkedin.com/in/jayantsolanki