PharmaSUG 2020 - Paper AD-171

Clinical Database Metadata Quality Control with SAS® and Python

Craig Chin and Lawrence Madziwa, Fred Hutch

ABSTRACT

A well-designed clinical database requires well-defined specifications for Case Report Forms (CRFs), Field Attributes, and Data Dictionaries. The specifications are passed on to the Electronic Data Capture (EDC) Programmers, who program the clinical database. How can a study team ensure that the source specifications are complete, and the resulting clinical database metadata match the source specifications? This paper presents two approaches. Initially, we used SAS® to read in the specifications and clinical database metadata, and to provide comparison checks. Then, we converted the project in Python in order to build a user-friendly tool that allows customers to run the checks themselves. These reports have improved the quality of our clinical databases, as well as saved each study team several hours of back and forth between the specification development and EDC programming.

INTRODUCTION

There are many reasons to ensure consistent study database metadata, including:

- Study database documentation for validation
- Study build errors could affect data collection and/or analysis
- Implementing study build corrections takes time and effort

Our organization uses Medidata RAVE EDC system to design, collect, and store our clinical study databases. The study team collectively develops the study database through an iterative process. This process includes two study metadata sources: the study team specifications defined in the Study Build Specification (SBS) and the study build metadata in the Study Design Specification (SDS).

Converting the SBS and SDS documents to a database of metadata provides a source for quality control reports, as well as additional tools to increase study database development efficiency.

THE STUDY DATABASE DEVELOPMENT PROCESS

STUDY BUILD SPECIFICATIONS

The study team develops the SBS, an Excel workbook which includes metadata that defines the study database, that also includes CRFs, edit checks, and data dictionary. For each study CRF, the workbook contains an individual worksheet that contains the database specifications such as field name, field response type, and format. Figure 1 shows a selection of an SBS worksheet for a typical study Demographics CRF.

Figure 1. SBS Worksheet for Demographics CRF

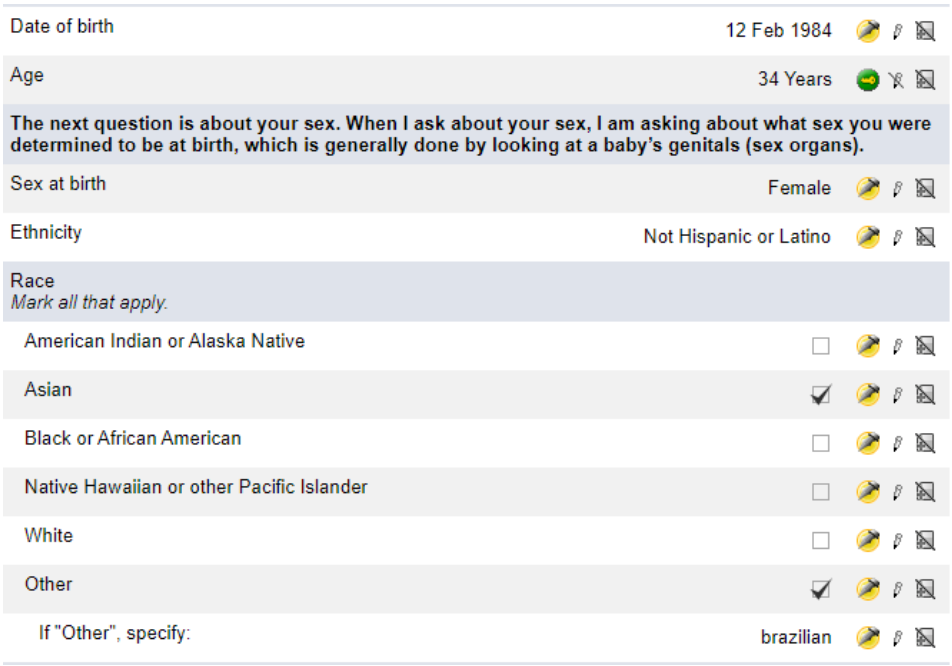
	D	E	F	G	H
1	Field OID	Item Text (Field Label)	SAS Label	Response Type (Control Type)	Format
3	BRTHDAT	Date of birth	Date of Birth	DateTime	dd- MMM- yyyy
4	AGE	Age	Age	Text	3
5	AGEU	Age unit	Age Unit	Text	$5
6	LABEL4	**When I ask about your sex, I am asking about what sex you were determined to be at birth, which is generally done by**		Text	
7	SEXBRTH	Sex at birth	Sex at Birth	RadioButton	$1
8	ETHNIC	Ethnicity	Ethnicity	RadioButton (Vertical)	1
9	LABEL1	Race *Mark all that apply.*		Text	
10	RACEAMIND	American Indian or Alaska Native	American Indian or Alaska Native	CheckBox	1
11	RACEASIAN	Asian	Asian	CheckBox	1
12	RACEAFRAM	Black or African American	Black or African American	CheckBox	1
13	RACEHAWAII	Native Hawaiian or other Pacific Islander	Native Hawaiian or other Pacific Islander	CheckBox	1
14	RACECAUC	White	White	CheckBox	1
15	RACEOTH	Other	Race Other	CheckBox	1
16	RACEOSP	If "Other", specify:	Race Other Specify	LongText	$200

BRA | Chemistry - Screen | Chemistry - FU | Con Med | CMYN | Demographics

EDC PROGRAMMING

The EDC programmer uses the SBS to program the Medidata RAVE study build. The specifications are reflected in the eCRF appearance in RAVE, as well as the underlying database. Figure 2 is a screenshot of the Demographics eCRF in RAVE for data collection.

Figure 2. Demographics eCRF in RAVE



ANNOTATED ECRF FOR DATA SET USERS

Figure 3 shows the annotated Demographics eCRFs with a selection of its metadata elements.

Figure 3. Annotated Demographics eCRF

	Field Name	Data Type	Units	Values	Pre-Filled Values	Include Field OID
2	BRTHDAT	dd- MMM-yyyy				BRTHDAT
3	AGE	3				AGE
5	SEXBRTH	$1		M = Male F = Female		SEXBRTH
6	ETHNIC	1		1 = Hispanic or Latino 2 = Not Hispanic or Latino		ETHNIC
8	RACEAMIND	1				RACEAMIND
9	RACEASIAN	1				RACEASIAN

STUDY DESIGN SPECIFICATIONS (STUDY BUILD METADATA)

Upon the completion of the study build, for documentation purposes, the EDC programmer generates the Study Design Specifications (SDS) from RAVE. The SDS is the study build metadata in XML format (readable in Excel), with all eCRF field metadata in the SDS Fields worksheet. Figure 4 shows the SDS Fields worksheet with a selection of the Demographics field metadata.

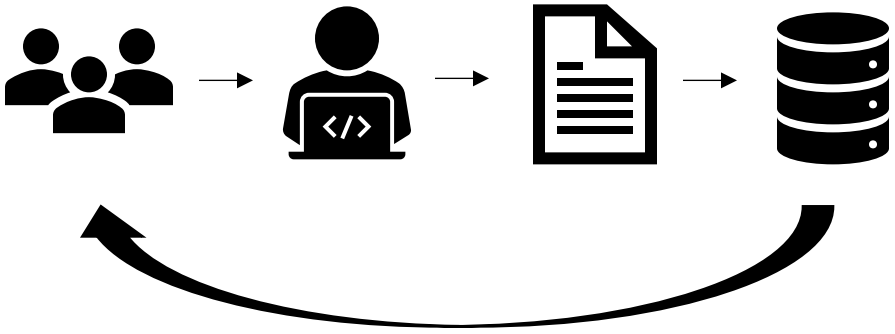
Figure 4. SDS Fields worksheet with Demographics eCRF metadata

FormOID	FieldOID	VariableOID	DataFormat	DataDictionaryName	ControlType	PreText
DM	BRTHDAT	BRTHDAT	dd- MMM- yyyy		DateTime	Date of birth
DM	AGE	AGE	3		Text	Age
DM	AGEU	AGEU	$5		Text	Age unit
DM	LABEL4				Text	The next question is about your sex. When I ask about your sex, I am asking about what sex you were determined to be at birth, which is generally done by looking at a baby's genitals (sex organs).
DM	SEXBRTH	SEXBRTH	$1	SEX	RadioButton	Sex at birth
DM	ETHNIC	ETHNIC	1	ETHNIC	RadioButton (Vertical)	Ethnicity
DM	LABEL1				Text	Race Mark all that apply.
DM	RACEAMIND	RACEAMIND	1		CheckBox	American Indian or Alaska Native
DM	RACEASIAN	RACEASIAN	1		CheckBox	Asian
DM	RACEAFRAM	RACEAFRAM	1		CheckBox	Black or African American
DM	RACEHAWAII	RACEHAWAII	1		CheckBox	Native Hawaiian or other Pacific Islander
DM	RACECAUC	RACECAUC	1		CheckBox	White
DM	RACEOTH	RACEOTH	1		CheckBox	Other
DM	RACEOSP	RACEOSP	$200		LongText	If "Other", specify:

THE PROCESS FOR STUDY DEVELOPMENT AND UPDATES

Any study database changes are reflected in an updated SBS, so these development steps are repeated for initial development (user acceptance and functional testing), production, and post-production modifications. Figure 5 shows a visual representation of the study build development process.

Figure 5. Study Build Development Process: Study Team develops SBS -> EDC Programming -> eCRF / Clinical Database -> SDS. Repeat as necessary.



The study build development process is effective but has opportunity for inconsistencies between the source SBS and the eventual study build SDS. For example, the SBS may be missing key metadata, or the SDS may reveal EDC programming errors. Fortunately, these two study metadata sources have many elements that can be directly compared and are easily converted to a database.

SAS PILOT IMPLEMENTATION – CONVERTING SBS AND SDS TO SAS DATA SETS

Converting the source SBS and SDS files is straightforward using SAS, using some basic SAS procedures and programming methods.

CONVERTING AN INDIVIDUAL EXCEL WORKSHEET TO A SAS DATA SET

The IMPORT procedure is used in conjunction with the FILENAME statement to reference EXCEL workbook/worksheets and output to a SAS data set. To reference a workbook's specific worksheet, use the option SHEET= with an appended "n" character after the worksheet name:

```
filename SBSFILE "/devel/opsprog/sbs_sds/pharmasug2020/SBSexample.xlsx"
  encoding='utf-8';

PROC IMPORT OUT= democrf_metadata
  DATAFILE= SBSFILE
  DBMS=xlsx REPLACE ;
  sheet="DEMOGRAPHICS"n;
  GETNAMES=no;
  datarow=2;
RUN;
```

For the SBS, the SAS data set of the individual eCRF contains the worksheet columns as variables, which are named, typed, and formatted accordingly in a subsequent DATA step.

The SDS workbook structure is more straightforward. It contains the FIELDS worksheet of all eCRF metadata and is converted to a SAS data set similarly using PROC IMPORT:

```
filename SDSFILE " /devel/opsprog/sbs_sds/pharmasug2020/SDSexample.xlsx"
  encoding="utf-8";

PROC IMPORT OUT= sds_fields
            DATAFILE= SDSFILE
            DBMS=xlsx REPLACE;
            sheet="FIELDS";
            GETNAMES=no;
            datarow=2;
RUN;
```

USING THE SQL PROCEDURE TO DERIVE A DELIMITED LIST OF WORKSHEETS AND THE TOTAL NUMBER OF WORKSHEETS

The SQL procedure is used in combination with the LIBNAME statement to access the workbook and its metadata as SAS DICTIONARY table views:

```
libname SBSXLSX xlsx"/devel/opsprog/sbs_sds/pharmasug2020/SBSexample.xlsx"
  access=readonly inencoding='utf-8';

proc sql noprint;
  create table SBS_SHEETS as
  select memname from dictionary.tables
  where upcase(libname)="SBSXLSX"
;
quit;
```

The LIBNAME references the full path and filename of the SBS file, using the XLSX engine. The worksheet names are saved as SAS data set SBS_SHEETS from SAS view DICTIONARY.TABLES.

PROC SQL is then used to create macro variables for a delimited list of eCRF worksheet names and the total number of eCRF worksheets from the SBS_SHEETS data set (excluding worksheets that are not related to eCRFs):

```
proc sql noprint;
   select memname
          into :tab_list
          separated by '|'
          from sbs_sheets
          where memname not in ("CRFS", "FOLDERS", "DICTIONARY")
          ;
   select count(memname)
          into :tab_count
          from sbs_sheets
          where memname not in ("CRFS", "FOLDERS", "DICTIONARY")
          ;
quit;
```

USING A %DO LOOP TO READ IN ALL WORKSHEETS INTO A STANDARDIZED DATA SET

Using the macro variables of the total number and eCRF list, we created a macro that loops through the eCRFs, reads them in using the previously described PROC IMPORT, standardizes the variables in a DATA step, and creates a metadata data set (SBS_VARIABLES) of all eCRF variables using the APPEND procedure:

```
%macro convert_sheet_to_dataset
  %let counter=0;
  %do %until (&counter = &tab_count);
    %let counter=%eval(&counter + 1);
    %let tab_current=%scan(%bquote(&tab_list), &counter, %bquote(|));

    <PROC IMPORT eCRF worksheet TAB_CURRENT to eCRF data set tab_&counter>
    <DATA STEP to standardize eCRF data set tab_&counter>

    proc append data=tab_&counter base=sbs_variables;
    run;

  %end;
%mend convert_sheet_to_dataset;
```

SAS PILOT IMPLEMENTATION – COMPARING THE TWO STUDY DATABASE METADATA SOURCES

With SAS data sets of eCRF metadata from the SBS and SDS files, we can easily compare several field level aspects of the study build:

1. Identify eCRF/fields that exist in only one metadata source, possibly due to field name issues
2. Compare metadata items for consistency at the eCRF/field level
    a. Data Format
    b. Data Dictionary Name
    c. Response Type
    d. Item Text

e. Log Field indicator

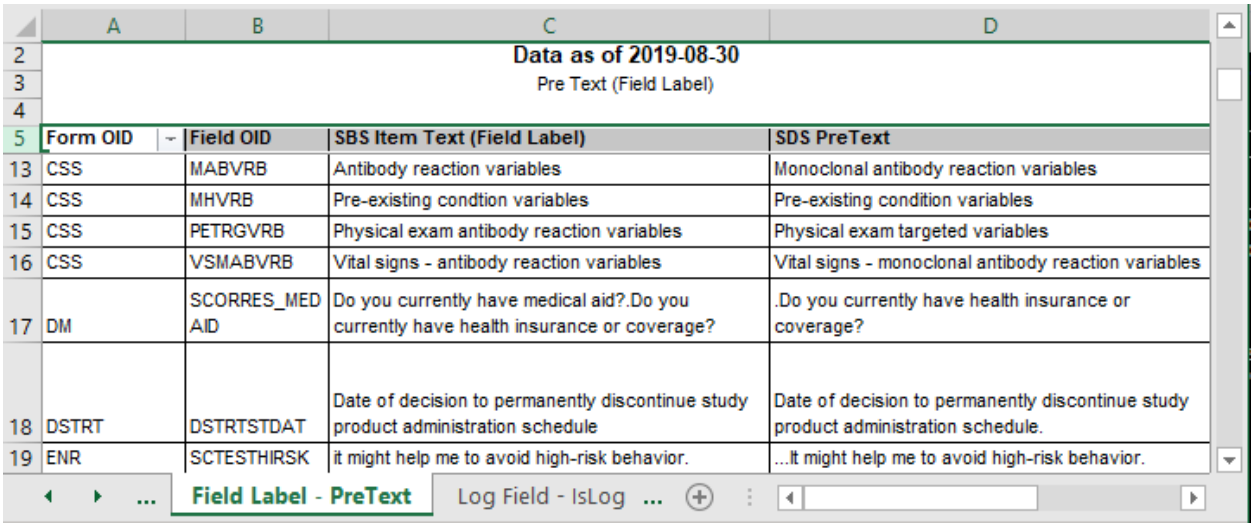
f. SAS Label

g. Required field indicator

h. Review groups

Quality Control reports of flagged issues are provided to the Clinical Data Manager (CDM) of the study team. Figure 6 shows a sample spreadsheet of field label discrepancies between the SBS (field label) and SDS (PreText) by eCRF and field.

Figure 6. Quality Control Report Example



	A	B	C	D
2			Data as of 2019-08-30	
3			Pre Text (Field Label)	
4				
5	Form OID	Field OID	SBS Item Text (Field Label)	SDS PreText
13	CSS	MABVRB	Antibody reaction variables	Monoclonal antibody reaction variables
14	CSS	MHVRB	Pre-existing condtion variables	Pre-existing condition variables
15	CSS	PETRGVRB	Physical exam antibody reaction variables	Physical exam targeted variables
16	CSS	VSMABVRB	Vital signs - antibody reaction variables	Vital signs - monoclonal antibody reaction variables
17	DM	SCORRES_MEDAID	Do you currently have medical aid?.Do you currently have health insurance or coverage?	.Do you currently have health insurance or coverage?
18	DSTRT	DSTRTSTDAT	Date of decision to permanently discontinue study product administration schedule	Date of decision to permanently discontinue study product administration schedule.
19	ENR	SCTESTHIRSK	it might help me to avoid high-risk behavior.	...It might help me to avoid high-risk behavior.

SAS PILOT IMPLEMENTATION – NEXT PHASE

The SAS pilot implementation provided useful quality control (QC) of the study database. Using the study build process, the study team updated the study build as necessary and provided feedback to the Clinical Programmers. Several improvements to the QC process were requested:

1. Flexibility - SBS was not intended as a database source; individual eCRF worksheets had varied format and content.
2. Expandability – the study team identified several more items for comparison.
3. Sensitivity and specificity – some inconsistencies were flagged due to extraneous text (e.g. special text characters, html tags) that could be ignored; identifying common patters and filtering in SAS implementation was done through trial and error using SAS text functions. Improved filtering was needed.
4. Accessibility - Clinical Programmers generated the report on demand after each SBS/study database update. Study team had to request updated reports after each iteration. Moreover, not all members of the study team had immediate access to SAS.
5. Scalability - code needed to be able to adapt to multiple studies. Study metadata QC process desired for multiple studies in various stages of development.
6. Timing – SBS/SDS QC process only discovered issues after the study build implemented. QC of the SBS before handoff to EDC programmers could save additional time.

To address these needs, we needed an implementation that was immediately accessible on demand, and to all stakeholders. It needed to be agile – with a robust text handling functionality, and capable, within the Excel framework, of recognizing features such as struck out cells. All this needed to be wrapped into a

friendly user interface that enabled entry validation. Python, with its myriad of ready-to-use modules, was a handy candidate

PYTHON IMPLEMENTATION OVERVIEW

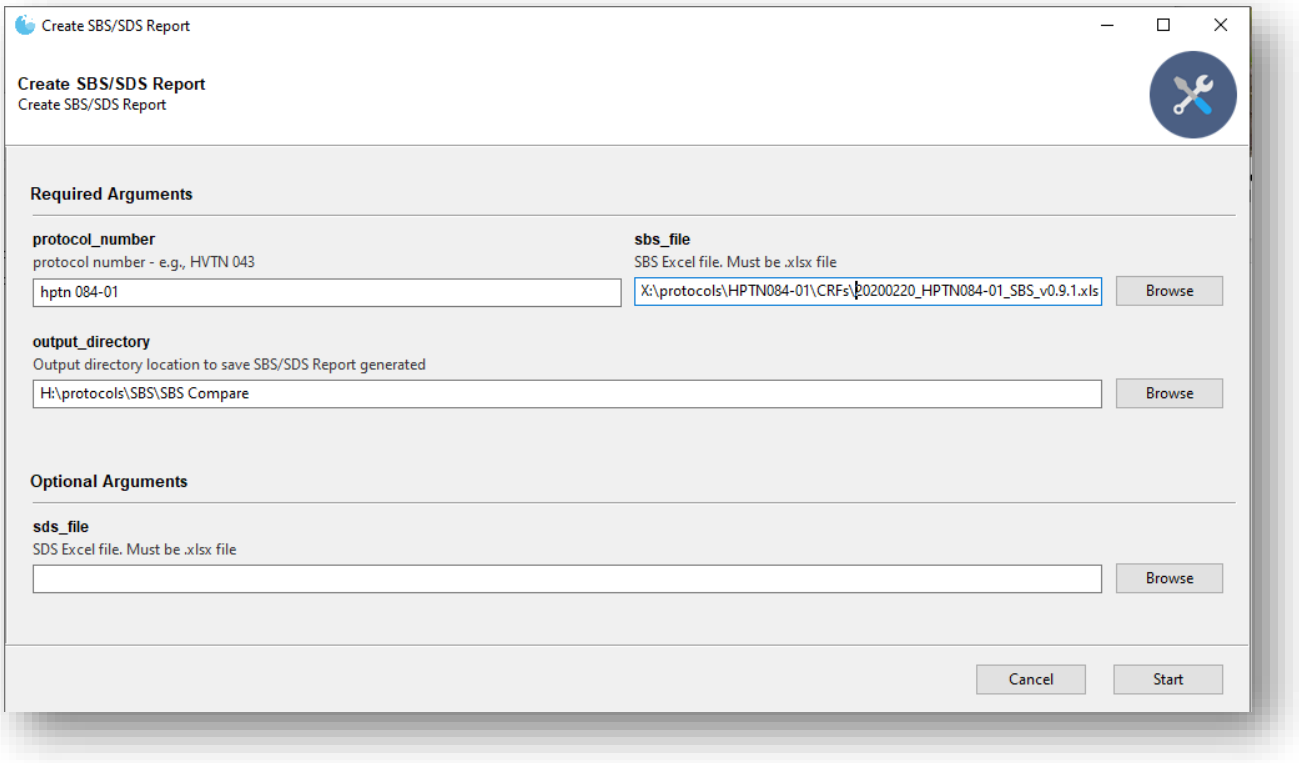
Python offers a rich suite of modules for data handling and transformation. For this project, we made use of the pandas module for reading in and reshaping data. For data input and interface display, we used the Gooey module. Finally, we packaged the code into an executable that can be distributed to end users. Essential code snippets for the translation, including how to create a user interface and the executable, are provided in subsequent sections as well as in the appendix.

PYTHON IMPLEMENTATION – DEVELOPMENT PROCESS

The Python implementation carries over all the SAS-implemented checks. For certain checks, we also programmatically added feedback in a Comments field to inform the user why particular items were flagged.

Code translation involves two stages. The first is a direct comparison between the SBS and SDS documents. The second is a check for internal consistency within the SBS document itself. For simplicity and ease of code maintenance, each is implemented as its own Python function; however, both would be handled via the same user interface. Which call is run depends on the inputs provided. If a path to the SDS document is provided, the tool runs an SBS/SDS comparison; otherwise, it does an internal review of the SBS document. A screenshot of the interface is provided below.

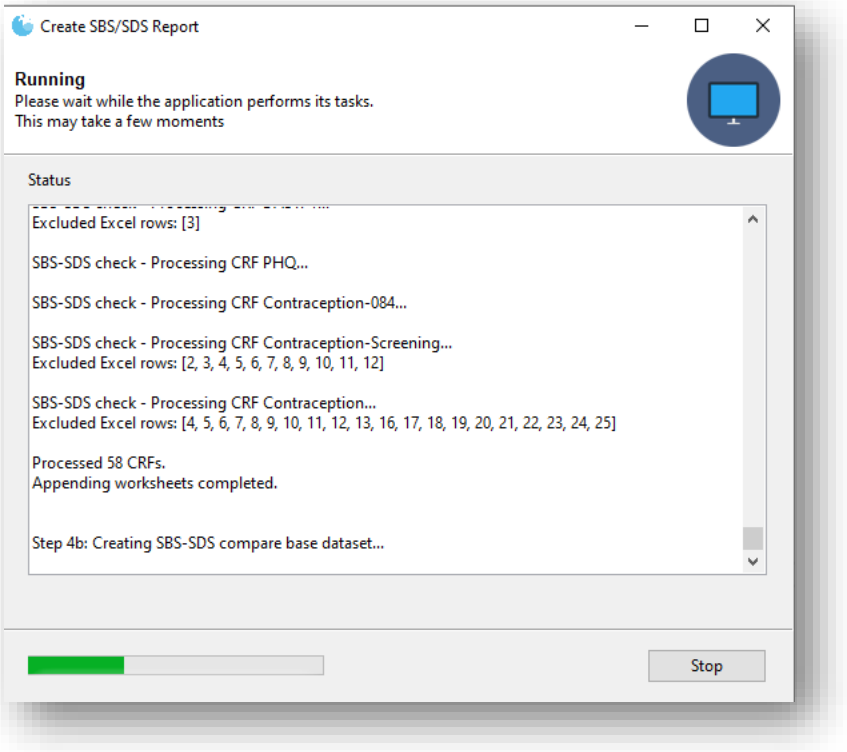
Figure 7. Gooey User Interface screenshot



The first 3 inputs: the protocol number, the SBS file, and an output directory are required. The SDS file is optionally given, depending on what report the user needs generated. The user then clicks the ‘Start’ button in the lower right corner to initiate the report. Shown below is a snapshot during a typical run process. From this point until the program is fully executed, the interface takes over as the standard

output, with all print statements logged through it. Thus, via carefully selected print statements, the developer can allow the user to view/monitor progress as the code runs.

Figure 8. Progress Monitoring/Display



Finally, each process generates a report in Excel that is opened as soon as it is created, which brings it to the user's immediate attention.

PYTHON IMPLEMENTATION – DETAILS

The main task is programmatically reading and manipulating Excel files, including formatting and styling the reports to have a more professional look. The primary modules used for this are pandas and openpyxl. As mentioned previously, the Gooey module handles and regulates user input, as well as managing how the code is run once initiated. Below is a screenshot of the imported modules.

Figure 9. Imported modules

```
import pandas as pd
import numpy as np
import os
import json
from gooey import Gooey, GooeyParser
from openpyxl import Workbook
from openpyxl.styles import Alignment, NamedStyle, Font, Color, PatternFill
from openpyxl import load_workbook
from datetime import date
from openpyxl.utils import get_column_letter
from SBS_Internal_review import internal_review
from datetime import datetime
import xlrd
```

The program is run as two main functions. The main function imports the second and initiates the user interface when called. (See the third from last statement in the Figure 9 above). This paper will focus on four elements of the implementation:

- reading in pertinent Excel files through python,
- allowing for code flexibility,
- an overview of setting up the user interface, and
- creating an executable that can be made available to users.

Admittedly, data transformation is really the greatest component of this project. It is briefly addressed in the Appendix with useful resources cited, but it can be a whole topic on its own and has books written on it. A particularly great reference is Wes McKinney's Python for Data Analysis[1].

READING IN THE SBS EXCEL DOCUMENT

The SBS document consists of over 70 tabs, most of which are CRFs. The forms have a standard definition, though variations have been encountered across studies. Fortunately, the columns of most interest are constant in name and type.

There are several Python modules that can read in Excel files: xlwt writes to older versions of Excel (xls extension); xlrd reads those versions, while openpyxl handles more recent Excel versions (xlsx extension). Under the hood, pandas calls these modules to read and write Excel files. Its read_excel method requires at a minimum a path and filename to an Excel workbook as an argument. For example, the following code reads in the Dictionary tab of the SBS Excel document:

```
SBS_Data_Dictionary = pd.read_excel(SBSFile, sheet_name="Dictionary")
```

Reading in the CRFs is an extension of the above. However, because most of the tabs are CRFs, reading all of them at once is easier by first identifying and excluding the few tabs that are not CRFs. First, we collect all the tabs in the SBS Excel document into a list. Then we filter that list by removing non-CRF tabs. What remains will be SBS CRF tabs, which we can then iterate over:

```
sbs_tabs = [sheet_name for sheet_name in SBS_xl.sheet_names]

CRF_tabs = [
    tab
    for tab in sbs_tabs
    if tab.upper()
    not in (
        "CRFS",
        "FOLDERS",
```

```
        "DICTIONARY",)
]
```

The two lists above were created via a Python list comprehension. A list comprehension is a convenient way to create a list with one code statement. The iteration is performed below. In it, each CRF tab is read and appended to an initially empty list (df_list). As each tab is processed, a CRF count variable keeps track of the number of CRFs. During processing, the output of the print statement is displayed through the GUI interface, alerting the user to the progress, as is shown in Figure 8 above. Upon completion, the final number of CRFs iterated over is also displayed.

```
crf_count = 0
df_list = []
for tab in CRF_tabs:
    crf_count += 1

    tab_name = df
    df = pd.read_excel(
        SBSFile,
        sheet_name=tab,
        header=0,
    )
    print(f"SBS-SDS check - Processing CRF {tab_name}... ")
    df["Source_Tab"] = "{}".format(tab_name)

    df_list.append(df)
```

For illustration, two ways are presented to handle Python strings. The first print statement uses a more modern approach, known as **f-strings**. In order to combine Python variables with strings, it requires placing the required string in f-prefixed quotation marks – single or double – with the Python variables placed in curly brackets. An alternative older approach places the desired string in quotes, with variable positions represented by empty sets of curly brackets. The variable names placed outside, in order, as arguments to an attached format statement. Where possible, the former method is preferred as it requires less typing and is also easier to read. The one caveat is f-strings are available in Python versions 3.6 and later.

```
print(f"Processed {crf_count} CRFs.")
print("Appending worksheets completed.")
```

The final step creates a data frame out of df_list object. It uses pandas' **concat** method, which takes a sequence, such as a list, or a mapping, such as a dictionary or a data frame object, returning a data frame:

```
sbs_variables = pd.concat(df_list, sort=False, ignore_index=True)
```

HANDLING INPUT DATA VARIATIONS GRACEFULLY

In an ideal world, the Excel files to be parsed are best created in a standard way with static metadata elements. In the real world, the data generating processes evolve. Python offers a utility to allow some amount of variation – try/catch statements. Below is an example of a function used in the tool to 'clean out' Excel cell contents. It takes a given string, removes or compresses out specified unwanted

characters, and returns either a stripped string void of those characters, or a number if what remains after compressing are only digits. An example use would be to find format size. In the SBS document, character field variable formats use a dollar-sign prefix. The dollar signs can be ignored when extracting variable width.

```
def ncompress(string, chars):
    """This function returns string with chars removed"""
    want = "".join([char for char in string if char not in chars and
                char is not None])
    try:
          if float(want):
                return float(want)
    except ValueError:
          return want.strip()
```

The code uses a most basic type implementation of the **try/catch** Python construct, where failures in the try block are set to be handled by their type in the exception block.

Another example is given below. At the end of the run, the program immediately opens the generated Excel report. If this fails for any reason, a note is written to the user interface screen telling the user where to find it, along with the error. It is possible to have multiple except statements, where each exception prior to the very last must specify the type of error they will handle, such as ValueError above. It is also possible to use nested try/catch blocks.

```
try:
    print("Opening up SBS/SDS Report…")
    os.system(f'start "excel" "{rpt_workbook}"')
except Exception as e:
    print(f"Workbook can be found in location: {rpt_workbook}")
    print(f"Error {e}")
```

SETTING UP THE USER INTERFACE

Setting up the user interface using the Gooey module[2] is relatively straightforward. Gooey also produces interfaces with a more modern feel than other comparable Python packages. One approach is to make the main calls into Python functions, with the required inputs – the SBS Excel file, the study name and the output destination – as arguments. Gooey has various widgets available for each input type. For example, text input is handled by the "TextField" widget. File selection, which allows browsing to a required file, is made possible through the "FileChooser" widget. Selection of an output directory is facilitated by the "DirChooser" widget. These are convenient as they minimize user input and also errors. In addition, for some of these widgets, validation is available. For example, if only more modern versions of Excel are acceptable as file inputs, one can include validation for that widget, so the program does not proceed if supplied a file with the wrong extension. See the Appendix for a validation example.

```
@Gooey(program_name="Create SBS/SDS Report", advanced=True)
def parse_args():
    stored_args = {}
    script_name = os.path.splitext(os.path.basename(__file__))[0]
    args_file = f"{script_name}-args.json"
    if os.path.isfile(args_file):
        with open(args_file) as data_file:
            stored_args = json.load(data_file)

    parser = GooeyParser(description="Create SBS/SDS Report")
    parser.add_argument(
        "protocol_number",
        action="store",
        default=stored_args.get("protocol_name"),
```

```
        widget="TextField",
        help="Protocol Name - e.g., HVTN 043. Used in output report name
and report title.",
    )
```

The widgets described above are handled in parameters which are part of a decorated parse_args function. A decorator is a Python function that takes as input another function, returning a modified version of that input function. While the details are beyond this paper's scope, it suffices to know we will need the @Gooey decorator when extracting the arguments in the parse_args function, as shown above. Inside the inner function, we create a GooeyParser object, to which expected arguments are added. The code above shows how we would add the TextField widget, displayed as protocol_number in Figure 7. See the Appendix for details on adding 3 other widgets.

```
 args = parser.parse_args()
    with open(args_file, "w") as data_file:
        json.dump(vars(args), data_file)
    return args


if __name__ == "__main__":
    conf = parse_args()
    print("Output Directory:", conf.output_directory)
    print("SBS File:", conf.sbs_file)
    print("SDS File:", conf.sds_file)
    print("Protocol Number:", conf.protocol_name)
    if conf.sds_file is not None:
        print("Running SBS/SDS Compare since both SBS and SDS were
supplied...")
        process_data(
            conf.sbs_file, conf.sds_file, conf.output_directory,
conf.protocol_name
        )
    else:
        print()
        print("Running internal SBS review since no SDS supplied...")
        internal_review(conf.sbs_file, conf.output_directory,
conf.protocol_name)
```

We end by storing inputs to a json file, so that in subsequent runs of the code, the last entered inputs are cached and provided as defaults.

Finally, the line: if __name__ == "__main__" indicates that we would like to run the code that follows, if the code is executed directly from that script. Otherwise, if the code in that script is imported while running another module, then it would not be the main script but a dependency of it, and the subsequent code would not be run.

SAMPLE REPORT OUTPUT

Below is a sample of the SBS Internal Review report, which is one of two reports generated. It opens to a cover page, which shows a categorized summary of issues found. It also displays the time it was created, the study for which it was run, a link to the SBS file analyzed, as well as a summary of the number of checks performed. The user can navigate to the other tabs for a more detailed listings of issues found, if

any. Ideally, users then address the issues raised and regenerate the report until no more problems are cited.

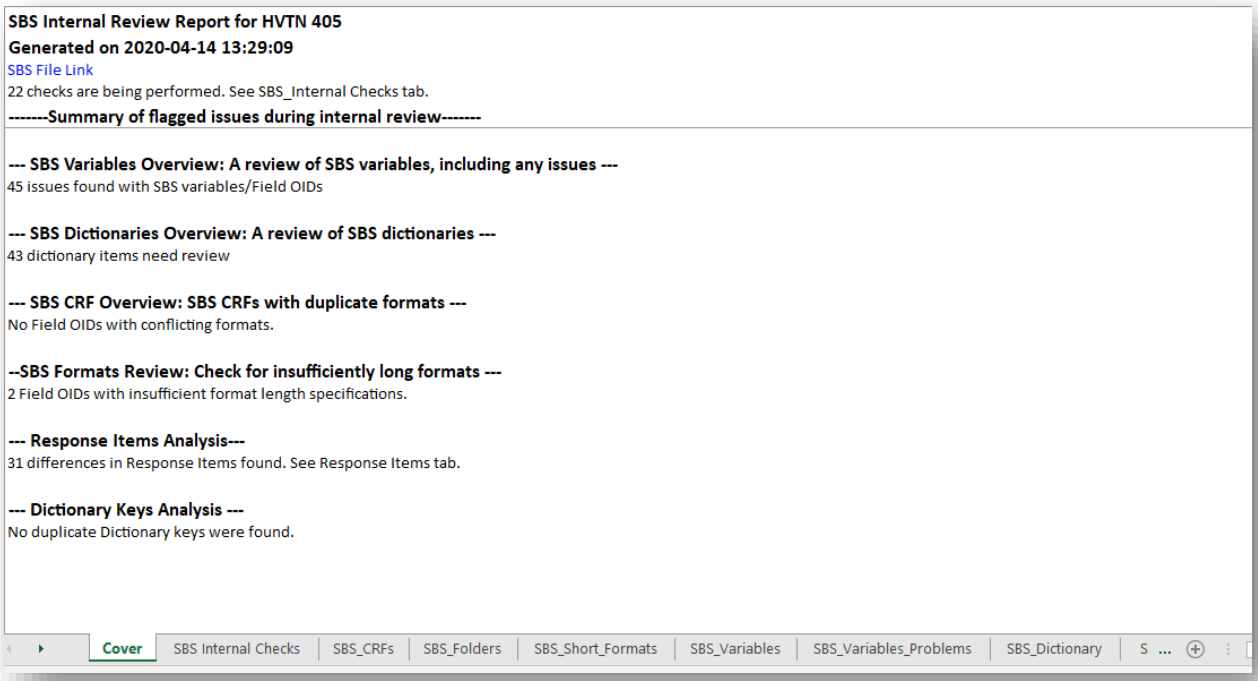


Figure 10. Sample SBS Internal Review Report

DEPLOYMENT

Once completed, the tool can be deployed to users in one step using the pyinstaller module. After installing this module via Python's "pip install", one only needs to open a command prompt, navigate to the directory containing script to be converted into an executable, and give the following command:

```
pyinstaller <python script name>.py --onefile
```

The onefile argument allows all dependencies to be packaged up in one file, which is more portable. You can then provide users a link to the executable from a website or Sharepoint site, for example.

PYTHON IMPLEMENTATION – FUTURE AND CHALLENGES

The project remains a work in progress. We continue working with our customers to learn about their processes. We seek to identify any tedious manual checks during the spec build/design phases and automate them to create efficiency and minimize human error. The challenge is to achieve a level of standardization in the SBS/SDS creation while still allowing users some level of form-free editing. This enables users/reviewers of the documents to include their edits, while simultaneously making it possible for an automated process to provide ongoing and readily available quality control. For example, SBS users prefer to strikeout rows in the SBS document that needed correction. Striking out rows instead of deleting them serves as documentation. Programmatically, this required reading in and ignoring strikethroughs, which was not a straightforward implementation, but which eventually accommodated.

CONCLUSION

Using SAS and Python, Clinical Programming has created tools to extract and analyze metadata from study database specifications to generate quality control reports. With the aid of various Python modules,

our Clinical Data Managers can now run the interactive tool and generate the reports independently, as needed. The implementation of these tools into the study database development and validation process has saved time and helped ensure a high-caliber study database for data collection and analyses.

REFERENCES

1. McKinney, Wes. 2018. *Python for Data Analysis*, 2nd Edition. Sebastopol, CA, O'Reilly
2. Kiehl, Chris. 2020. Gooey v1.0.3 https://github.com/chriskiehl/Gooey

ACKNOWLEDGMENTS

The authors would like to thank their colleagues at Fred Hutch/SCHARP for their inspiration, support, and feedback.

RECOMMENDED READING

- *Python for Data Analysis*

CONTACT INFORMATION

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

Craig Chin
SCHARP / Fred Hutch
craig@fredhutch.org

Lawrence Madziwa
SCHARP / Fred Hutch
lmadziwa@fredhutch.org

APPENDIX

Below is code used to create the user interface using the Gooey module:

```
@Gooey(program_name="Create SBS/SDS Report", advanced=True)
def parse_args():
    stored_args = {}
    script_name = os.path.splitext(os.path.basename(__file__))[0]
    args_file = f"{script_name}-args.json"
    if os.path.isfile(args_file):
        with open(args_file) as data_file:
            stored_args = json.load(data_file)
    parser = GooeyParser(description="Create SBS/SDS Report")
    parser.add_argument(
        "protocol_name",
        action="store",
        default=stored_args.get("protocol_name"),
        widget="TextField",
        help="Protocol Name - e.g., HVTN 043. Used in output report name
and report title.",
    )

    parser.add_argument(
        "sbs_file",
        action="store",
        default=stored_args.get("sbs_file"),
```

```
        widget="FileChooser",
        help="Study Build Specification. This Excel (xlsx) document is
created by Clinical Data Managers to define "
             "the database to be programmed in RAVE. For Global Library
Build Spec, click on Help Tab on top left",
        gooey_options={
            "validator": {
                "test": "user_input.endswith('xlsx')",
                "message": "SBS file must be an Excel file with the .xlsx
extension.",
            }
        },
    )
    parser.add_argument(
        "-sds_file",
        widget="FileChooser",
        help="Study Database Specification. This describes the database as-
built. It is exported from RAVE. See Help "
             "Tab (top left) for instructions for how to export the current
SDS from RAVE. Must be .xlsx file.",
        gooey_options={
            "validator": {
                "test": "user_input.endswith('xlsx')",
                "message": "SDS file must be an Excel file with the .xlsx
extension.",
            }
        },
    )

    parser.add_argument(
        "output_directory",
        action="store",
        default=stored_args.get("output_directory"),
        widget="DirChooser",
        help=f"Output directory location to save the generated reports.",
    )

    args = parser.parse_args()
    with open(args_file, "w") as data_file:
        json.dump(vars(args), data_file)
    return args


if __name__ == "__main__":
    conf = parse_args()
    print("Output Directory:", conf.output_directory)
    print("SBS File:", conf.sbs_file)
    print("SDS File:", conf.sds_file)
    print("Protocol Number:", conf.protocol_name)
    if conf.sds_file is not None:
        print("Running SBS/SDS Compare since both SBS and SDS were
supplied...")
        process_data(
            conf.sbs_file, conf.sds_file, conf.output_directory,
conf.protocol_name
        )
    else:
```

```
        print()
        print("Running internal SBS review since no SDS supplied...")
        internal_review(conf.sbs_file, conf.output_directory,
conf.protocol_name)
```