Paper DH02

How Can Python Be Used in Clinical Trials?

Andras Kasa, UCB Biosciences GmbH, Monheim am Rhein, Germany
Matt Davies, Analytical Programming Services Limited, Cardiff, United Kingdom
Pierre Dostie, UCB Pharma S.A., Colombes, France

ABSTRACT

Once upon a time, Python™ opened the door and independently from the clinical trial field, enthusiastic programmers made several data manipulation and statistical packages in Python for their own purposes.
Although Python is greatly utilised in the field of AI and ML, with data evangelists appropriately convinced of its supernature, it is fairly underutilised in the clinical research field. We did some experiments and explored how hard it would be to use these Python packages for data manipulation, statistics calculation, output creation purposes, or for exploratory purposes such as creating dashboards.
Another interesting aspect is that Python is an unvalidated language, meaning its packages are constantly evolving which can be a concern since packages are heavily dependent on each other. We explored this aspect as well and tried to come up with a solution.

INTRODUCTION

Python is a multipurpose language that can be used for a wide range of applications, such as web development, data analysis and visualization, automation, GUI development, game development, mobile application development, machine learning and AI tasks, natural language processing tasks, generative AI, robotics, and cybersecurity.

Several research papers in the pharmaceutical field highlight the numerous advantages of using Python. One study concludes that Python is particularly valuable for automating complex and repetitive tasks, leading to significant time savings (1). The author of this paper demonstrates the practical application of Python by using it to modify SAS® programs in batches and streamline deliveries through a custom Python tool.
In another paper by the same author, the potential of Python in Natural Language Processing (NLP) is showcased. The author presents a tool that utilizes NLP techniques to verify program deliveries and identify instances of plagiarism (2). Other papers delve into exploring the capabilities of Python, such as the creation of specialized plots and the development of specific-purpose XML files (3) (4). There are also experiments creating basic summary tables with Python (5).
In this paper, we aim to provide a comprehensive overview of our research and highlight various examples of Python's utility. First, we will discuss the strengths and limitations of Python. Next, we will introduce the powerful data manipulation library, Pandas. Additionally, we will demonstrate a practical approach to generating define.xml files using Python. Finally, we will showcase three basic examples of how Python can be employed to create individual plots, basic dashboards, and PDF outputs.

ADVANTAGES OF USING PYTHON

Easy to Learn and Readable Syntax: Python has a clean and readable syntax that makes it easy to learn and understand, even for beginners. Its code is designed to be easily readable and expressive, which enhances productivity and reduces the time required for development and debugging.
Large and Active Community: Python has a large and active community of developers who contribute to its growth and development. This community provides extensive support, resources, and libraries, making it easier to find solutions to problems and accelerate development.
Cross-Platform Compatibility: Python is a cross-platform language, which means that Python code can run on various operating systems, including Windows, macOS, and Linux. This allows developers to write code once and run it on different platforms without major modifications.

DISADVANTAGES OF USING PYTHON

Performance: Since Python is a high-level and human-friendly language, Python code needs to go through many stages of abstraction before it can become executable machine code.
Memory Consumption: Python can consume more memory compared to other languages. This can be a concern for applications with limited memory access.

In conclusion, Python is a versatile language that can be used for a wide range of applications. Its simplicity, readability, and extensive library ecosystem make it an attractive choice for developers across various domains. Whether you are building web applications, analyzing data, automating tasks, or developing AI models, Python has the tools and resources to support your needs. Since it can be used in many fields, the only limit is your imagination.

In terms of data analysis and data visualization, there are several libraries that are useful for these tasks such as:

NumPy: It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays. NumPy is often used in conjunction with pandas for data manipulation and analysis.
Pandas: Pandas is widely used in data science and data analysis tasks. It allows you to perform tasks such as finding correlations between columns, calculating average, maximum, and minimum values, and cleaning data by deleting irrelevant or incorrect rows, merging, sorting or aggregating data. Pandas is built on top of NumPy.
Matplotlib: Matplotlib is a plotting library that provides a wide variety of static, animated, and interactive visualizations in Python. It allows you to create line plots, scatter plots, bar plots, histograms, and many other types of plots to explore and visualize your data.
Seaborn: Seaborn is a statistical data visualization library built on top of Matplotlib. It provides a high-level interface for creating informative and attractive statistical graphics. Seaborn simplifies the process of creating complex visualizations and offers a range of built-in themes and color palettes.
Plotly: It allows you to create a wide range of charts, including line plots, scatter plots, bar charts, area charts, histograms, box plots, and more. Plotly also supports 3D visualizations and maps.
Scikit-learn: scikit-learn is a machine learning library that provides a wide range of supervised and unsupervised learning algorithms. It includes tools for data preprocessing, feature selection, model evaluation, and more. Scikit-learn can be used in conjunction with pandas for tasks such as classification, regression, clustering, and dimensionality reduction.
Statsmodels: Statsmodels is a library that focuses on statistical modeling and econometrics. It provides a wide range of statistical models, including linear regression, time series analysis, generalized linear models, and more. Statsmodels integrates well with pandas and provides tools for statistical inference and hypothesis testing.

PANDAS

In this paper, we will not provide an in-depth overview of the Pandas package. Instead, we aim to highlight the main differences between using Pandas and SAS from the perspective of SAS programmers and we try to give a survival kit for those who are getting started with it. In the past few years, the authors of this paper, who are originally SAS programmers, have extensively experimented with Pandas. In this summary, we will outline our experiences with Pandas and summarize the key distinctions between Pandas and SAS.

GENERAL DIFFRENCES

As a Programming Language: Pandas is built on top of Python. On the other hand, SAS has its own proprietary programming language specifically designed for data analysis and statistical modeling.
Data Manipulation: Both languages provide wide range of functions and methods for data manipulation, such as filtering, sorting, merging, slicing or reshaping data. It is hard to decide which language is easier to learn since all of us started our data programmer career in the field of SAS programming.
Data Visualization: Pandas integrates well with other Python libraries like Matplotlib, Plotly or Seaborn, allowing for easy and customized data visualization. SAS also has its own data visualization capabilities which are easy to use but it may limit customized options compared to Python.
Community and Support: Both SAS and Pandas have a large and active community of users and developers, which means there are plenty of online resources, tutorials, and forums available for support. Due to the open-source nature of the Pandas community being more organic, people come from various fields of data science.
Cost: Pandas is an open-source library and is freely available for anyone to use. On the other hand, SAS is a commercial software suite that requires a license. SAS also provides data warehouse solutions, complex multi-user cloud based computational platforms and a high-performance AI analytics platform.
Validation: SAS is our gold standard since it is a validated software, while Pandas is also highly reliable but there is no guarantee of no calculation error would occur.

DATAFRAME

Pandas' main data object is the DataFrame, which is a two-dimensional tabular data structure consisting of rows and columns. Each row and column have its own index, starting from 0 by default, as is typical in Python. While rows and columns can be labeled, there is no concept of a variable name like in SAS. In some cases, Pandas may automatically assign subindexes to variables during certain procedures.

DIFFERENCES BETWEEN SAS AND PYTHON'S LOGIC

SLICING DATA, DIFFERENCES BETWEEN LOC AND ILOC

In Pandas, data can be sliced using column and row indices. In the example below, we create a 10x10 DataFrame and select the 2nd and 4th columns, as well as the rows from 2 to 7 (excluding the 7th row). This exercise utilizes the iloc functionality of Pandas, as shown in Figure1 below. Generally, when we use iloc, the endpoint of a range is not included.

```
import pandas as pd
import numpy as np

# Set a random seed for reproducibility
np.random.seed(123)

# Generate a 10x10 DataFrame with random integers
df = pd.DataFrame(np.random.randint(1, 100, size=(10, 10)), columns=['col'+str(i) for i in range(1, 11)])

# Select the 2nd and 4th columns, and the rows from the 2 to 7 (not inclusive)
df_slice = df.iloc[2:7, [1, 3]]

# Print the resulting DataFrame
print(df_slice)
```

```
   col2  col4
2    50    68
3    49    93
4    77     4
5    78    16
6    15    55
```

Figure1

While when we use the loc functionality of the Pandas we can slice a data upon the labels of the columns and if we are using indexes in the loc, the start and end points of range are included, see an example below in Figure2:

```
import pandas as pd
import numpy as np

# Set a random seed for reproducibility
np.random.seed(123)

# Generate a 10x10 DataFrame with random integers
df = pd.DataFrame(np.random.randint(1, 100, size=(10, 10)), columns=['col'+str(i) for i in range(1, 11)])

# Select the 2nd and 4th columns, and the rows from the 2 to 7 (not inclusive)
df_slice = df.loc[2:7, ['col2', 'col4'] ]

# Print the resulting DataFrame
print(df_slice)
```

```
   col2  col4
2    50    68
3    49    93
4    77     4
5    78    16
6    15    55
7    85    50
```

Figure2

LABELS, INDEXES

Surprisingly, two or more columns can have the same label in Pandas since the index of those can help to distinguish between them. In Figure3, we see an example of that when one accidently creates a DataFrame with similar labels. We can easily encounter this situation if we forget that our DataFrame already contains a column labeled as AVAL. For

a SAS programmer, this mindset might be unusual, but it's important to emphasize that Pandas uses indexes **and labels for variables rather than names.**

```
# Create two DataFrames with different column label.
df1 = pd.DataFrame ({'USUBJID': ['XC34-001-04159','XC34-001-08743','XC34-002-16523'],
                     'PARAM': [ 'LAB1','LAB2' ,'LAB1'], 'AVAL': [1,1,1.5] })
df2 = pd.DataFrame ({'USUBJID': ['XC34-001-04159','XC34-001-08743','XC34-002-16523'],
                      'AVAL2': [70,75,62] })
# Merge the two DataFrames by USUBJID
df = pd.merge (df1, df2, on=['USUBJID'])
#Rename the AVAL2 to AVAL
df = df.rename(columns={'AVAL2': 'AVAL'})
df
```

	USUBJID	PARAM	AVAL	AVAL
0	XC34-001-04159	LAB1	1.0	70
1	XC34-001-08743	LAB2	1.0	75
2	XC34-002-16523	LAB1	1.5	62

```
df_ren = df.rename(columns={'AVAL' : 'AVALX'})
df_ren
```

	USUBJID	PARAM	AVALX	AVALX
0	XC34-001-04159	LAB1	1.0	70
1	XC34-001-08743	LAB2	1.0	75
2	XC34-002-16523	LAB1	1.5	62

```
df.columns.values[2] = 'AVALX'
df
```

	USUBJID	PARAM	AVALX	AVAL
0	XC34-001-04159	LAB1	1.0	70
1	XC34-001-08743	LAB2	1.0	75
2	XC34-002-16523	LAB1	1.5	62

Figure3

DISPLAYING DATAFRAMES

Let's discuss a limitation of Pandas. As SAS programmers, we are accustomed to the ability to easily scroll through data since SAS provides us with an interface that allows us to navigate both vertically and horizontally within a dataset. This feature has been available in SAS for quite some time.
In contrast, when we call the name of a DataFrame in Jupyter Notebook by default, Pandas only displays the first 5 and the last 5 records if the DataFrame contains more than 10 records (See that in Figure4).

```
df2
```

	USUBJID	PARAM	token	Visit	AVAL
0	XC34-001-04159	Serum Potasium	1	Visit1	87
1	XC34-001-04159	Serum Potasium	1	Visit2	69
2	XC34-001-04159	Serum Potasium	1	Visit3	97
3	XC34-001-04159	Serum Potasium	1	Visit4	50
4	XC34-001-04159	Serum Potasium	1	Visit5	2
...	...	...	...	...	...
94	XC34-002-16523	Cholesterol	1	Visit7	46
95	XC34-002-16523	Cholesterol	1	Visit8	45
96	XC34-002-16523	Cholesterol	1	Visit9	57
97	XC34-002-16523	Cholesterol	1	Visit10	35
98	XC34-002-16523	Cholesterol	1	Visit11	18

Figure4

To display all rows or columns in Python, we can use the following command as shown in Figure5:

```
pd.set_option('display.max_rows', None)
df2
```

	USUBJID	PARAM	token	Visit	AVAL
0	XC34-001-04159	Serum Potasium	1	Visit1	87
1	XC34-001-04159	Serum Potasium	1	Visit2	69
2	XC34-001-04159	Serum Potasium	1	Visit3	97
3	XC34-001-04159	Serum Potasium	1	Visit4	50
4	XC34-001-04159	Serum Potasium	1	Visit5	2
5	XC34-001-04159	Serum Potasium	1	Visit6	18
6	XC34-001-04159	Serum Potasium	1	Visit7	46
7	XC34-001-04159	Serum Potasium	1	Visit8	45
8	XC34-001-04159	Serum Potasium	1	Visit9	57
9	XC34-001-04159	Serum Potasium	1	Visit10	35
10	XC34-001-04159	Serum Potasium	1	Visit11	18

Figure5

Unfortunately, when attempting to display a relatively large DataFrame in Pandas, Python may return an error message indicating insufficient memory and the inability to display the DataFrame. In such cases, alternative solutions are required. One option is to display only the header of the DataFrame using the .head() function. Another approach is to filter the data or use the .sample() function, which provides random samples of the DataFrame. Alternatively, one can export the DataFrame to a CSV or Excel file and thoroughly examine the data there.

WHAT CAN WE DO WHEN WE NEED TO MERGE LARGE DATAFRAMES?

One drawback of Pandas is that when you need to merge two relatively large DataFrames, you may encounter a memory issue, which can result in an error message from Python. To overcome this problem, you can utilize the following function, as shown in Figure6 below.

This function adheres to the principles of a Python function and is not specific to Pandas, although it utilizes Pandas commands within its implementation.

The purpose of this function is to efficiently handle the merging of large DataFrames. It achieves this by dividing the DataFrames into smaller chunks, performing the merge operation on each chunk, and finally combining the results. This approach effectively reduces memory usage and mitigates the risk of encountering memory errors when dealing with large datasets.

```python
# This function can help merging large dataframes
def merge_chunks(left_df, right_df, on, how='left', chunk_size=100000):
    """
    Perform a merge operation between the left and right dataframes using a split-apply-combine approach.

    :param left_df: left dataframe to merge
    :param right_df: right dataframe to merge
    :param on_cols: column(s) to merge on
    :param how: type of join to perform (default: 'left')
    :param chunk_size: size of each data chunk (default: 100000)
    :return: merged dataframe
    """
    left_chunks = [left_df[i:i+chunk_size] for i in range(0, len(left_df), chunk_size)]
    right_chunks = [right_df[i:i+chunk_size] for i in range(0, len(right_df), chunk_size)]

    merged_chunks = []
    for l_chunk, r_chunk in zip(left_chunks, right_chunks):
        merged = pd.merge(l_chunk, r_chunk, on=on, how=how)
        merged_chunks.append(merged)

    merged = pd.concat(merged_chunks)
    return merged
```

Figure6

TRANSPOSING DATA

When you apply the transpose() function on a DataFrame object in Pandas, it simply swaps the columns and rows. The Pandas documentation provides a simple example of this functionality (11). However, what if we want to transpose data selectively, such as transposing only certain variables to parameters or converting stacked values to an unstacked structure? Unlike SAS, Pandas offers two functions for these tasks. The melt() function is specifically designed to reshape data from a unstacked to stacked format, while the pivot() function is designed for the opposite task. In the following snippets, we will provide examples demonstrating the usage of these functions. Figure7 displays our original dataset. Figure8 displays when we will apply the melt function to transform the wide structure into a long one.

	USUBJID	2022-01	2022-02	2022-03	2022-04	2022-05	2022-06	2022-07	2022-08	2022-09	2022-10	2022-11	2022-12	2023-01	2023-02	2023-03	2023-04	2023-05	2023-06	2023-07	2023-08
0	JS01-001-00001	4	5	7	3	6	3	5	4	3	5	9	1	7	9	9	6	7	6	7	1
1	JS01-002-00002	9	1	6	7	7	9	9	5	5	1	2	10	1	8	6	5	8	3	4	6
2	JS01-003-00003	7	10	8	10	5	10	7	6	8	4	10	2	4	10	4	9	4	6	7	8
3	JS01-004-00004	2	2	7	7	5	10	1	10	4	9	6	4	4	9	8	1	2	8	3	4
4	JS01-005-00005	3	6	9	10	8	10	8	5	6	2	9	7	9	5	3	8	2	7	3	7

Figure7

```
# Transpose the dataset
transposed_data = pd.melt(data, id_vars='USUBJID', var_name='DATE', value_name='AVAL')
# Display the transposed dataset
transposed_data
```

	USUBJID	DATE	AVAL
0	JS01-001-00001	2022-01	4
1	JS01-002-00002	2022-01	9
2	JS01-003-00003	2022-01	7
3	JS01-004-00004	2022-01	2
4	JS01-005-00005	2022-01	3
5	JS01-001-00001	2022-02	5
6	JS01-002-00002	2022-02	1
7	JS01-003-00003	2022-02	10
8	JS01-004-00004	2022-02	2
9	JS01-005-00005	2022-02	6
10	JS01-001-00001	2022-03	7

Figure8

To revert back to the unstacked structure, we should utilize the pivot function, this demonstrated in Figure9.

```
# Pivot the transposed data - from long to wide.
pivoted_data = transposed_data.pivot(index='USUBJID', columns='DATE', values='AVAL')

# Reset the index and rename the column axis
pivoted_data.reset_index(inplace=True)
pivoted_data.columns.name = None

# Display the pivoted data
pivoted_data
```

	USUBJID	2022-01	2022-02	2022-03	2022-04	2022-05	2022-06	2022-07	2022-08	2022-09	2022-10	2022-11	2022-12	2023-01	2023-02	2023-03	2023-04	2023-05	2023-06	2023-07	2023-08
0	JS01-001-00001	4	5	7	3	6	3	5	4	3	5	9	1	7	9	9	6	7	6	7	1
1	JS01-002-00002	9	1	6	7	7	9	9	5	5	1	2	10	1	8	6	5	8	3	4	6
2	JS01-003-00003	7	10	8	10	5	10	7	6	8	4	10	2	4	10	4	9	4	6	7	8
3	JS01-004-00004	2	2	7	7	5	10	1	10	4	9	6	4	4	9	8	1	2	8	3	4
4	JS01-005-00005	3	6	9	10	8	10	8	5	6	2	9	7	9	5	3	8	2	7	3	7

Figure9

CREATING NEW VARIABLES BASED ON SPECIFIC CONDITIONS

Below, we will discuss two methods for creating new variables based on specific conditions. One useful method in pandas is the 'cut' function, which allows us to create a categorical variable based on a numeric value. To use this function, we first need to determine the boundaries of the categories and assign labels to them. We can then apply the 'cut' function as demonstrated in Figure10.

```
# copy our dataset into a new one, will do this 2 times since we would show 2 examples
example1 = transposed_data.copy()
example2 = transposed_data.copy()

# Categorize values of the AVAL- first solution

# Define the category ranges
categories = [0, 3, 7, float('inf')]
labels = ['1-3', '4-7', 'Above 7']

# Categorize the AVAL column
example1['CATEGORY'] = pd.cut(example1['AVAL'], bins=categories, labels=labels)
example1
```

Figure10

Another highly useful technique is the 'apply' method. With this method, we can create a temporary function known as a lambda function, which allows us to define specific conditions. In the example below, we can observe that the lambda function is used within the 'apply' method to create new variables based on these conditions. The lambda function is a compact way to define a function without explicitly naming it. It can take one or more arguments, and in this case, it takes a single argument 'x' which represents each value in the dataset. Within the lambda function, we can write our conditions using if-else statements. In the example, the 'else' branch opens a new 'if' branch, which can also have its own 'else' branch. This nesting of conditions allows us to create complex logic for variable creation. The 'apply' method, combined with the lambda function, provides a flexible and efficient way to manipulate data and create new variables based on specific conditions. Figure11 below showcases an example that demonstrates the usage of the 'apply' method with the lambda function. In this case, the lambda function is employed to apply a conditional calculation to each element or row of the DataFrame, resulting in the creation of new variables based on specific conditions.

```
# Categorize values of the AVAL- second solution, using the lambda function with the apply method.

# Define the lambda function for categorization
categorize = lambda x: '1-3' if x >= 1 and x <= 3 else ('4-7' if x >= 4 and x <= 7 else 'Above 7')

# Apply the lambda function to create the CATEGORY column
example2['CATEGORY'] = example2['AVAL'].apply(categorize)

example2
```

	USUBJID	DATE	AVAL	CATEGORY
0	JS01-001-00001	2022-01	4.0	4-7
1	JS01-002-00002	2022-01	9.0	Above 7
2	JS01-003-00003	2022-01	7.0	4-7
...	...	...	...	...
112	JS01-003-00003	2023-11	8.0	Above 7
113	JS01-004-00004	2023-11	1.0	1-3
114	JS01-005-00005	2023-11	1.0	1-3

115 rows × 4 columns

Figure11

Finally, we want to highlight that each condition in Pandas should relate to a logical operator. The code demonstrated in Figure12 for example will not work properly.

```
# The below code will fail to run since each conditions should be connected:

example3_filtered = example3[ ( 4 <= example3['AVAL'] < 9) ]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_68788\706073243.py in ?()
----> 3 # The below code will fail to run since each conditions should be connected:
      4
      5 example3_filtered = example3[ ( 4 <= example3['AVAL'] < 9) ]

~\Anaconda3\envs\STAN\lib\site-packages\pandas\core\generic.py in ?(self)
   1525     @final
   1526     def __nonzero__(self) -> NoReturn:
-> 1527         raise ValueError(
   1528             f"The truth value of a {type(self).__name__} is ambiguous. "
   1529             "Use a.empty, a.bool(), a.item(), a.any() or a.all()."
   1530         )

ValueError: The truth value of a Series is ambiguous. Use a.empty, a.bool(), a.item(), a.any() or a.all().
```

Figure12

Instead of this, we need to separate our condition into two separate conditions and use the '&' operator to combine them as shown in Figure13. As SAS programmers, we might not be accustomed to this approach.

```
example3_filtered = example3[ (4 <= example3['AVAL']) & (example3['AVAL']  < 9) ]
example3_filtered
```

	USUBJID	DATE	AVAL	CATEGORY
0	JS01-001-00001	2022-01	4	4-7
2	JS01-003-00003	2022-01	7	4-7
5	JS01-001-00001	2022-02	5	4-7
...	...	...	...	...
108	JS01-004-00004	2023-10	6	4-7
109	JS01-005-00005	2023-10	7	4-7
112	JS01-003-00003	2023-11	8	Above 7

66 rows × 4 columns

Figure13

CREATING DEFINE.XML WITH SAXON IN PYTHON

Creating Define.xml files is an essential task in the field of clinical trials. There are various approaches to accomplish this, such as using the SAS Clinical Toolkit or utilizing the Pinnacle 21 Enterprise.
Saxonica is a company that specializes in providing comprehensive solutions for XML workflows. According to their website, the company was founded in 2004 and their primary focus is the development of their own product called Saxon. Originally an open-source software, Saxon has evolved over time with continuous investment and has been commercialized. Currently, there are three available editions of Saxon for Python: SaxonC-HE (Home Edition which is the open-source version), SaxonC-PE (Professional Edition), and Saxon-EE (Enterprise Edition, both commercial products). Saxonica has an impressive customer base, including major banks and publishers.
Now, let's explore how to utilize the open-source edition of Saxon. We'll go through the process step by step. In essence, creating Define.xml with Saxonc is relatively straightforward. We need a stylesheet XSL file and a source XML document, which essentially contains the metadata of our trial in a predefined format.

To understand the concept of XSL, let's quote from the World Wide Web Consortium's (W3C) homepage (6):

"*What is XSL*?
XSL is a language for expressing style sheets. An XSL style sheet is, similar to CSS, a file that describes how to display an XML document of a given type."
"*How Does It Work?*
Styling requires a source XML document that contains the information that the style sheet will display. The style sheet itself describes how to display a document of a given type."

NOW, LET'S SEE HOW THIS IS IMPLEMENTED IN PRATICE USING PYTHON

The code snippet presented in Figure14 serves as an example showcasing the fundamentals of using Saxon for XML

transformations. In this code Xls2xml declares the stylesheet which will be used for the conversion. The file_xml is essentially our metadata in XML format, which is also readable and editable in Excel. The fileout variable points to the directory where our define.xml file will be stored.
First, we create a Xslt30Processor object by instantiating the PySaxonProcessor class, which represents a Saxon processor. The PySaxonProcessor class is part of the PySaxon library, which provides Python bindings for the Saxon XSLT and XQuery processor.
In the next step, we declare the xslproc_define variable, which will be used to perform the XSLT transformation.
Next, we compile the XSL stylesheet using the compile_stylesheet method of the Xslt30Processor object and store the compiled stylesheet in the executable object.

```
from saxonche import PySaxonProcessor

# Define the paths to the XSLT stylesheet and the input XML document
xls2xml_stylesheet = "C:/python_paper/stylesheet/xls2xml2-0.xsl"
file_xml = "C:/python_paper/spec/Specs.xml"
fileout = "C:/python_paper/result/define.xml"

# Create a Xslt30Processor object
proc_define = PySaxonProcessor(license=False)
xsltproc_define = proc_define.new_xslt30_processor()
# Compile the XSLT stylesheet
executable = xsltproc_define.compile_stylesheet(stylesheet_file=xls2xml_stylesheet)
# Load the input XML document
document = proc_define.parse_xml(xml_file_name=file_xml)
# Create define.xml based on a predefined stylesheet
executable.transform_to_file(output_file=fileout, xdm_node= document)
```

Figure14

After that, we parse the input XML file using the parse_xml method, which represents our metadata stored in a specific format like that used in Pinnacle 21. The parsed XML document is stored in the document object.
Below in Figure15, you can see an example of how a typical specification file may look like.

Instructions	Dataset Name	OID	Name	Description	Mandatory	Data Type	Length	Key Sequence
Multiple rows can be completed.	ADSL	ADSL.STUDYID	STUDYID	Study Identifier	Yes	text	200	1
DatasetName is used to specify which dataset the variable belongs to and should specify the SASDatasetName if present. Otherwise the Name should be specified.	ADSL	ADSL.USUBJID	USUBJID	Unique Subject Identifier	Yes	text	200	2
When creating SDTM/ADaM/SEND domains, the following rules should be followed when completing the 'Mandatory' column: If Core = 'Required' then Mandatory should = 'Yes'. If Core = 'Expected' or 'Permissible' then Mandatory should = 'No'.	ADSL	ADSL.SUBJID	SUBJID	Subject Identifier for the Study	Yes	text	200	
When creating SDTM/ADaM/SEND domains, the following rules should be followed when completing the 'Data Type' column: If Type = 'Char' then DataType should = 'date', 'datetime', 'time' or 'text'. If Type = 'Num' then DataType should = 'float' or 'integer'.	ADSL	ADSL.SITEID	SITEID	Study Site Identifier	Yes	text	200	
To include the Origin of a variable, go to the Origin Worksheet and follow the instructions.	ADSL	ADSL.SITEGRy	SITEGRy	Pooled Site Group y	Yes	text	200	
If a CodeList, Comment or Methods is referenced here, ensure to populate a row in the respective sheet for the element definition.	ADSL	ADSL.SITEGRyN	SITEGRyN	Pooled Site Group y (N)	Yes	integer	8	
	ADSL	ADSL.COUNTRY	COUNTRY	Country	No	text	3	
	ADSL	ADSL.ACOUNTRY	ACOUNTRY	Analysis Country	No	text	200	
	ADSL	ADSL.REGIONy	REGIONy	Geographic Region y	No	text	200	
	ADSL	ADSL.REGIONyN	REGIONyN	Geographic Region y (N)	No	integer	8	
	ADSL	ADSL.BRTHDTC	BRTHDTC	Date/Time of Birth	No	datetime		

... mentalDoc | Datasets | Variables | SupplementalQualifiers | Values | WhereClause | CodeLists | CodedItems | Methods | FormalExpression | ...

Figure15

Eventually, in the last step, we perform the XSLT transformation on the input XML document using the transform_to_file method of the executable object. This process creates the define.xml file.
The Python programming aspect of this task is relatively straightforward, as you can see. However, the real challenge lies in preparing a well-crafted XSL stylesheet file, which requires a good understanding of XML and XSL programming. Additionally, it is important to have a grasp of the fundamentals of SaxonC and its technology.
Once the XSL stylesheet and the metadata XML are properly standardized, the process should indeed run smoothly. In fact, with a relatively small investment, each company can develop its own define.xml creation tool that is specifically tailored to its unique needs.

HOW DID UCB IMPLEMENT THIS?

UCB has developed a Graphical User Interface (GUI) that integrates several functions for creating define.xml files. Python and its PySimpleGUI package were utilized for this purpose. With this tool, users can generate a define.xml file, as well as an HTML or PDF version of the define.xml. Additionally, users can easily add an ARM (Analysis Results Metadata) section to the define.xml by using our pre-defined ARM.xml file. Similar to the process described earlier, the ARM.xml file, along with the metadata XML file, can be converted into a submission-ready define.xml. Furthermore, the

define.xml file is editable in Excel, just like the specification input.
The tool also offers other useful features, such as the ability to generate define.pdf and define.html files in addition to the define.xml file. Furthermore, it provides validation of the define.xml against CDISC's standards.
Please refer to Figure16 below showcasing the GUI.

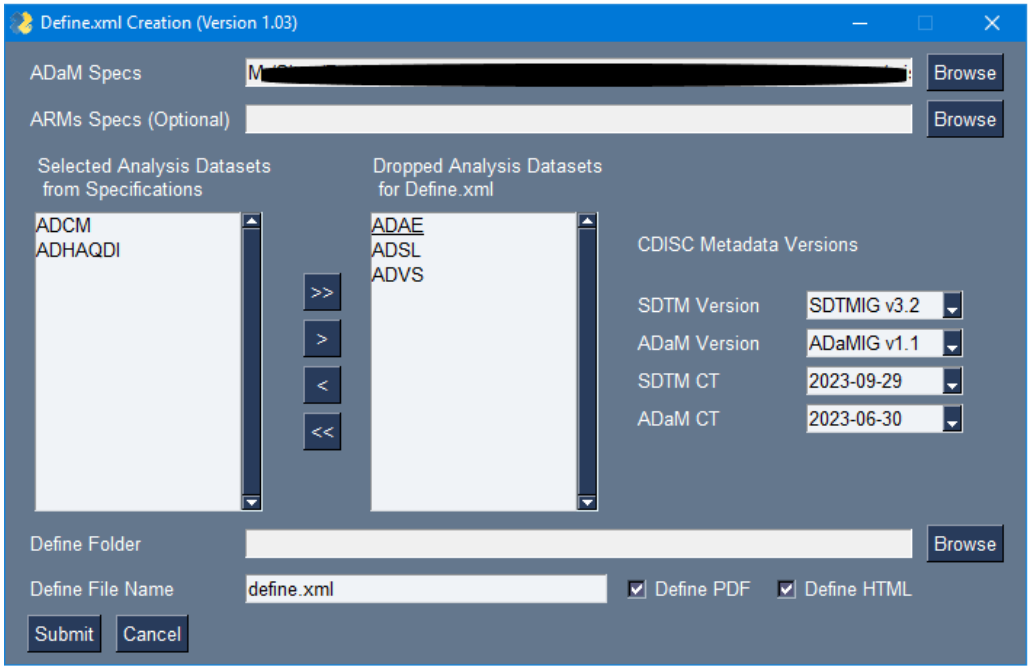

Figure16

To enhance our Define.xml tool's functionality and performance, our team of programmers continually implements suggested ideas and improvements.

UCB has plans to invest time and effort in developing similar tools in the future. These tools will assist with tasks such as checking delivery packages, ensuring submission readiness, bundle TFLs, and facilitating the daily tasks of statistical programmers.

INDIVIDUAL PLOTS CREATED WITH PLOTLY

As a team of programmers, we encountered a need to create comprehensive figures that would contain information related to three specific events: Medical History, Concomitant Medication, and Adverse Events. This requirement arose from a request made by UCB safety colleagues, who wanted a single output that consolidated all the event-related data for patients with a specific Adverse Event in a clinical program.
To address this requirement, we explored different approaches used in the industry. We came across examples in PharmaSUG (7) (8) (9) papers that showcased similar endeavors using tools like SAS or Tableau. Additionally, we had prior success in generating patient profile figures using R. Given this knowledge, we set out to create the consolidated figures to fulfill the request.
While we did not find any paper/offered solution in Python and Plotly-generated figures, it would be highly beneficial to utilize Plotly since its figures are interactive. As mentioned in the first section of the paper, Plotly is an amazing visualization library. All Plotly figures allow users to interact with the plot by zooming, panning, and hovering over data points to display additional information. Furthermore, it is easy to customize these figures, including legends, titles, axis scaling, and annotations. Additionally, these files can be exported in PNG, JPG, or HTML formats.

Turning our attention to Figure17, we can explore the visual representation of these figures. The patient ID and trial ID have intentionally been obscured.
In this figure, all three types of events (Medical History, Adverse Event, and Concomitant Medication) are displayed. Each event is represented by a line, and the color of the line indicates the type of event. The color of the Adverse Event lines remains the same regardless of the severity of the AE.
The events are evenly spaced horizontally and are ordered based on their start time. The x-axis represents the relative time to baseline, while the y-axis does not hold any specific meaning.
On the right top corner of the figure one can see the Plotly's default features such the zoom in/out, pan, box and lasso select, auto scale and the download buttons.

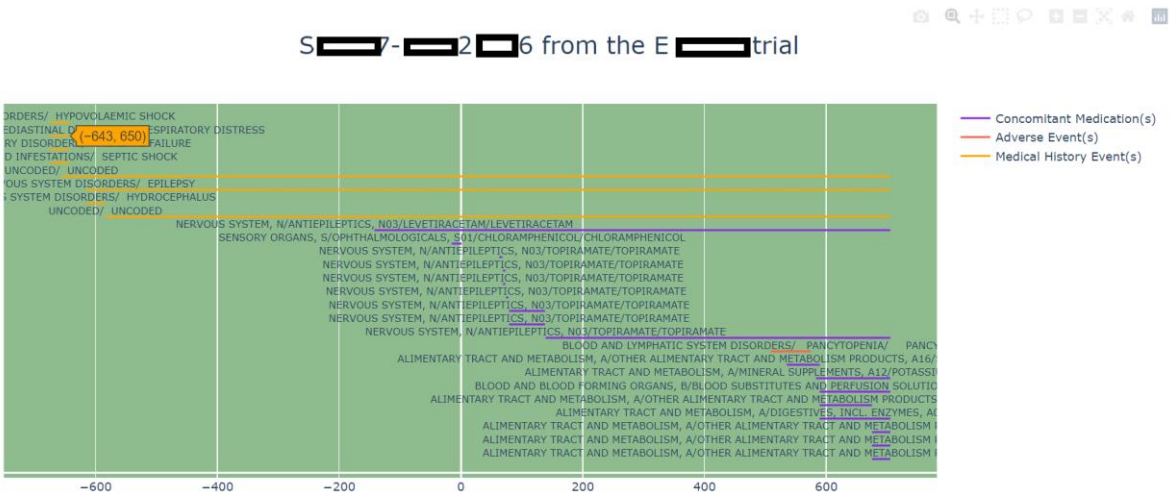

Figure17

In Figure18, we demonstrate how the appearance of the plot changes when someone zooms in. Notably, the annotations within the figure dynamically adjust to fit the new zoomed-in view, while the scaling of the x-axis may undergo slight modifications. This feature allows for the creation of diverse subfigures that present varying levels of detail. Such functionality proves particularly valuable when examining specific Adverse Events and analyzing the events that precede and follow them. It is important to note that the current stage of this visualization is a proof of concept. However, there are numerous potential avenues for further development, including the incorporation of dosing information and the integration of specific lab patterns into the figure, which can be indicated by color to signify their severity.

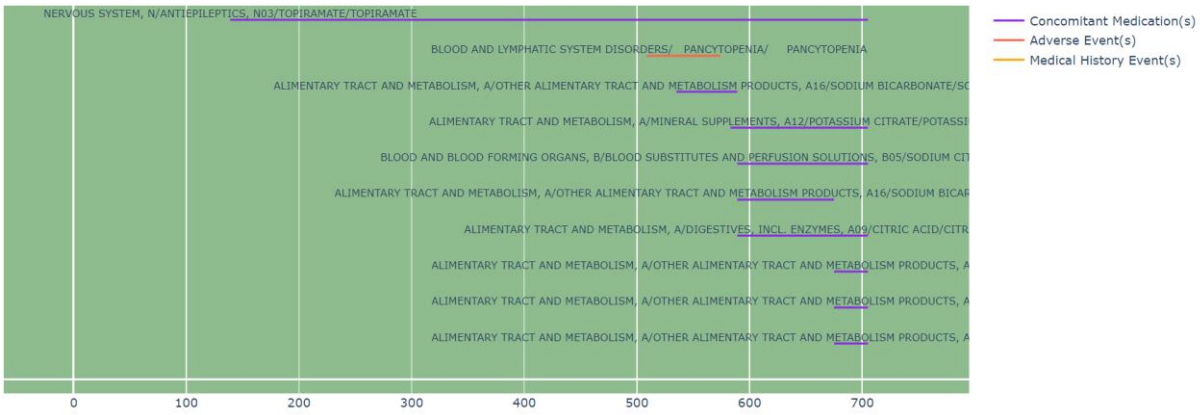

Figure18

HOW WERE THESE FIGURES CREATED?

To create a comprehensive figure that includes information related to three specific events (AE, CM, and MH), we followed a step-by-step process.

Data Preparation: First, we gathered all the data sources (AE, CM, and MH) and merged them into a single DataFrame. We also incorporated a separate DataFrame containing basic patient information, such as their start day on the trial. By doing this, we were able to calculate the relative day-to-baseline values for the start and end dates of all the events. In some cases, we had to impute the end date value to the end of the trial date or use other imputations.
The relative day of the start/end dates served as the X coordinate for the lines in the figure. The next step involved determining the Y coordinates.

Calculating Y Coordinates: To calculate the Y coordinates, we followed a specific process. First, we counted the number of events per patient, as this served as the foundation for the Y coordinate calculation. Then, we arranged the events for each patient in descending order based on their relative start day and assigned a sequence number to each event.
The Y coordinate calculation itself was quite straightforward. We multiplied each event's sequence number by the total

number of events. For instance, if there were 10 events, the first event would have a Y coordinate of 100, the second event would have a Y coordinate of 90, and so on. The Y coordinate for the end date's relative day was inherited from the Y coordinate assigned to the start date's relative day.
Currently, the distance between events depends on the number of events. However, it is also possible to use a fixed distance, such as 10 pixels. Nevertheless, using a flexible distance seems to provide enough space between the lines for annotation purposes.
Finally, we separated the different types of events again, as they would be used separately when generating the plot.

Plot Generation with Plotly: The plot was created using two nested loops. The outer loop iterated through the list of patients, treating each patient as a separate figure. The inner loop iterated through the different types of events.
By following this process, we were able to create a comprehensive figure that displayed the event-related data for each patient, with separate lines representing the different types of events.
Let's examine the code for the outer loops as depicted in Figure19.

```
import plotly.graph_objects as go
from plotly.offline import plot
from IPython.display import display

import numpy as np
# Define data points for each figure
figures = []
for ll in range(len(unique_figure)):

    #Handle x axis
    filtered_min_max = D_min_max.loc[D_min_max['FIGUREID'] == ll]
    Dmin  = filtered_min_max['Dmin']-300
    Dmax  = filtered_min_max['Dmax']+700
    filtered_min_max = filtered_min_max.reset_index()

    if not filtered_min_max.empty:
        PAT_TRI = str(filtered_min_max.at[0, 'USUBJID']) + ' from the ' + str(filtered_min_max.at[0, 'STUDYID']) +' trial'


    filtered_cm = cm_input.loc[cm_input['FIGUREID'] == ll]
    # Create a list with two elements, with X1 and X2 values respectively
    combined_list_cm_x = [[x1, x2] for x1, x2 in zip(filtered_cm['X1'], filtered_cm['X2'])]
    # Create a list with two elements, with Y1 and Y2 values respectively
    combined_list_cm_y = [[y1, y2] for y1, y2 in zip(filtered_cm['Y1'], filtered_cm['Y2'])]
    annotation_cm_list = list()
    annotation_cm_list = filtered_cm['CM_COL3'].tolist()
```

Figure19

It is a great example of the flexibility of Python data objects. At the beginning of the code, we initialize an empty list which serves as a container for the figures created for each patient. Python lists can store a wide range of data objects, including lists of lists, data frames, or any other data structure. This flexibility is shared with R.
Next, in the code, we determine the minimum and maximum values of the figures. These values will be used later to set up the figure's margin. The variable PAT_TRI is a string that combines the patient's unique identifier (USUBJID) with the patient's trial ID. This combined string will be displayed as the title of the figure.
After that, we filter the patient's Concomitant Medication data and create coordinate pairs (X1, X2, Y1, Y2) for the X and Y values. Each of these tuples represents a line, which corresponds to an individual event. We also create a list of annotations, which provides basic information about the Concomitant Medication (name of the medication and its coding). This list can include details such as the frequency and form of the medication.
The code example we have seen is repeated three times for different types of events. To gain a deeper understanding, let's now shift our focus to the inner loop, as illustrated in Figure20.

```
    # Create the figure
    fig = go.Figure(layout_yaxis=dict(
            tickmode='array',
            tickvals=[0],
            ticktext=[''],
        ))
# Add all the scatter plots to the same figure - CM section
    for j in range(len(combined_list_cm_x)):
        x_cm_vals = combined_list_cm_x[j]
        y_cm_vals = combined_list_cm_y[j]
        textcmj = list()
        textcmj=[annotation_cm_list[j], '']

        if j == 0:
            fig.add_trace(go.Scatter(x=x_cm_vals, y=y_cm_vals, mode='lines+text', name='Concomitant Medication(s)', text=text
        else:
            fig.add_trace(go.Scatter(x=x_cm_vals, y=y_cm_vals, mode='lines+text', name='', line=dict(color='blueviolet'), tex
```

Figure20

Here, we add each event to the figure line by line. This is the main purpose of the exercise, which involves creating the [X1, X2], [Y1, Y2] pairs in the outer loop. First, we create the figure object and then call the inner loop on a single patient's Concomitant Medication dataset. We add each event to the figure one by one. We differentiate between zero index and non-zero index lines because we only add the legend once to the figure. Otherwise, each line would have its own legend.
We repeat this process for Adverse Event and Medical History data, effectively adding all events to the figure. Finally, we add the title to the figure, set up the range and background color, and append the newly created figure to the list of figures. As a final step, we display all the figures and create an HTML format for them. Please refer to the examples below.
As you can see in Figure21, it is easy to create individual figures using this approach.

```
    fig.update_layout(
    title={
        'text': str(PAT_TRI),
        'font': {'size': 24},
        'x': 0.5,
        'xanchor': 'center'
        }
    )
    fig.update_xaxes(range=[Dmin, Dmax])
    # Set the background color
    fig.update_layout(
    plot_bgcolor='darkseagreen')


    figures.append(fig)

# Show all the figures
for ii, fig in enumerate(figures):
    fig.show()
    filename = f"C:/SafetyDashboard/HTML/SafetyDashboardPilot{ii}.html"
    plot(fig, filename=filename)
```

Figure21

DASHBOARD CREATION WITH DASH

The creators of Plotly have also developed a web application framework library called Dash, which bears similarities to R Studio's Flexdashboard/Shiny application framework. With Dash, someone has the flexibility to create a single HTML file that can be effortlessly shared with others, or an application that can be deployed on their computer or a server, such as Azure. For interactive applications that allow users to select data affecting displayed figures, a web application deployment is necessary.

The Dash website (10) provides useful examples and clear instructions on how to build one's own application. Additionally, there are several video courses available online. Similar to many GUI libraries, Dash offers a wide range of buttons, switches, sliders, and inputs that can be utilized to customize the appearance of an application or affect plots and output objects. Due to the extensive resources already available, we won't provide an in-depth overview of this package. Instead, we will showcase a basic example code that creates an application allowing users to choose between compounds, generating a stacked area chart illustrating the utilization of full-time equivalents FTEs over time. Our intention is to demonstrate the ease of using this package.

We also recognize the potential of utilizing Dash in the field of Rare Diseases, providing an interactive tool to assist clinicians or safety colleagues. We are currently conducting research into the seamless deployment of these applications, with the goal of making them readily accessible to all stakeholders.

Let's take a look at the code displayed in Figure22. We start by creating a Dash application using the dash.Dash(__name__) command. The colors dictionary defines three color values that will be used for styling the web page. The layout command sets the layout of the Dash application. First, we specify the background color, and then we create four elements on our page: html.H1 for the header, html.P for a sentence, dcc.Dropdown for the dropdown menu, and dcc.Graph for the stacked area chart component. The @app2.callback decorator defines the callback function that will update the stacked-area-chart figure based on the selected value in the project-dropdown. Finally, we define the callback function that takes the selected "Project" value as an input. It filters our DataFrame and uses the Plotly Express library to create a stacked area chart. It updates the layout of the chart, including the title and axis labels. Then, it returns the updated figure.

```
import plotly.express as px

app2 = dash.Dash(__name__)

colors = {
    'background': '#444444',
    'text': '#47BDEB',
    'text2': '#005A7D'
}

app2.layout = html.Div( style={'backgroundColor': colors['background']},
    children=[
        html.H1( children='Resource Allocation Over Time: Stacked Area Charts' , style={
            'textAlign': 'center',
            'color': colors['text']}),
        html.P( children='Please choose a compound:', style={
            'textAlign': 'left',
            'color': colors['text']} ),
        dcc.Dropdown(
            id='project-dropdown',
            options=[{'label': project, 'value': project} for project in projects],
            value=projects[0],  # Set the initial value to the first project
            style={
            'textAlign': 'left',
            'color': colors['text2']}
        ),
        dcc.Graph(
            id='stacked-area-chart' ,style={'backgroundColor': colors['background']}
        )
    ]
)

@app2.callback(
    Output('stacked-area-chart', 'figure'),
    Input('project-dropdown', 'value')
)

def update_figure(project):
    filtered_group = group[group['Project'] == project]

    fig = px.area(filtered_group, x='PARAM', y='AVAL', color='Study_Act__number', line_group='Study_Act__number')
    fig.update_layout(title=f"Stacked Area Chart - Project {project}")
    fig.update_layout(transition_duration=500,   font={ 'color': colors['text2'] },  legend=dict(
        title='Study'))
    fig.update_xaxes(tickmode='linear', dtick='M1', title='Month')
    fig.update_yaxes(title='FTEs')
    return fig
```

Figure22

Figure23 below demonstrates how the application looks.

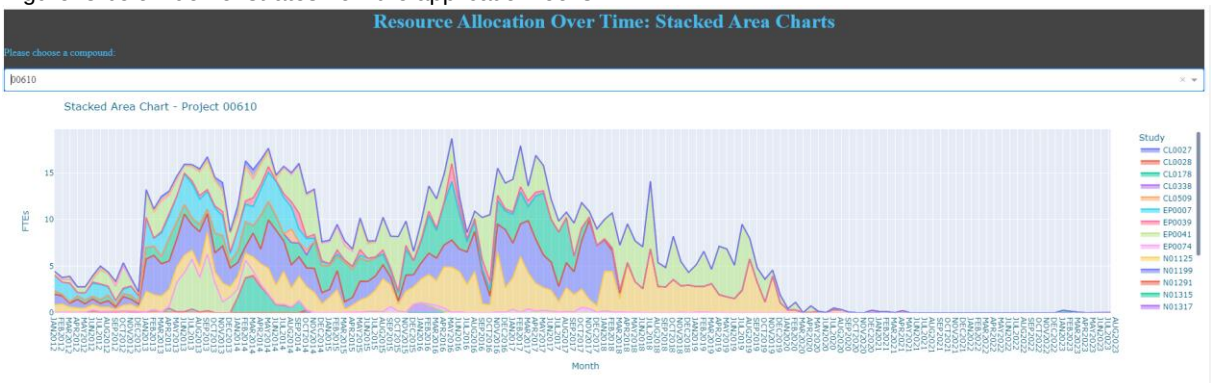

Figure23

It is important to note that all Plotly features are inherited. This means that you can hover over a certain area or select specific trials, and the figure will automatically update accordingly.

CREATING FIGURE OUTPUTS WITH PYTHON

One of the advantages of using SAS is its ability to easily generate various types of reports, including statistical tables, listings, and figures. However, when it comes to creating outputs with Python, we encountered some challenges. To address this, we conducted experiments to find a solution. Our focus was on creating plots using the Plotly library and exporting them in a submission-ready format. Unfortunately, the process was not as straightforward as we initially anticipated.

We adopted a two-step approach. Firstly, we utilized the ReportLab library to generate a canvas PDF file (see that in Figure24). After we create a separate PDF using a predefined HTML that we convert into PDF, and we overlay the two PDFs into one PDF.

The canvas file contains the essential elements of a typical report, such as a header containing basic information about the report for example, compound name, indication the report's confidentiality status, and the environment in which the program was executed. We also included the title and footnotes of the output in this PDF file.

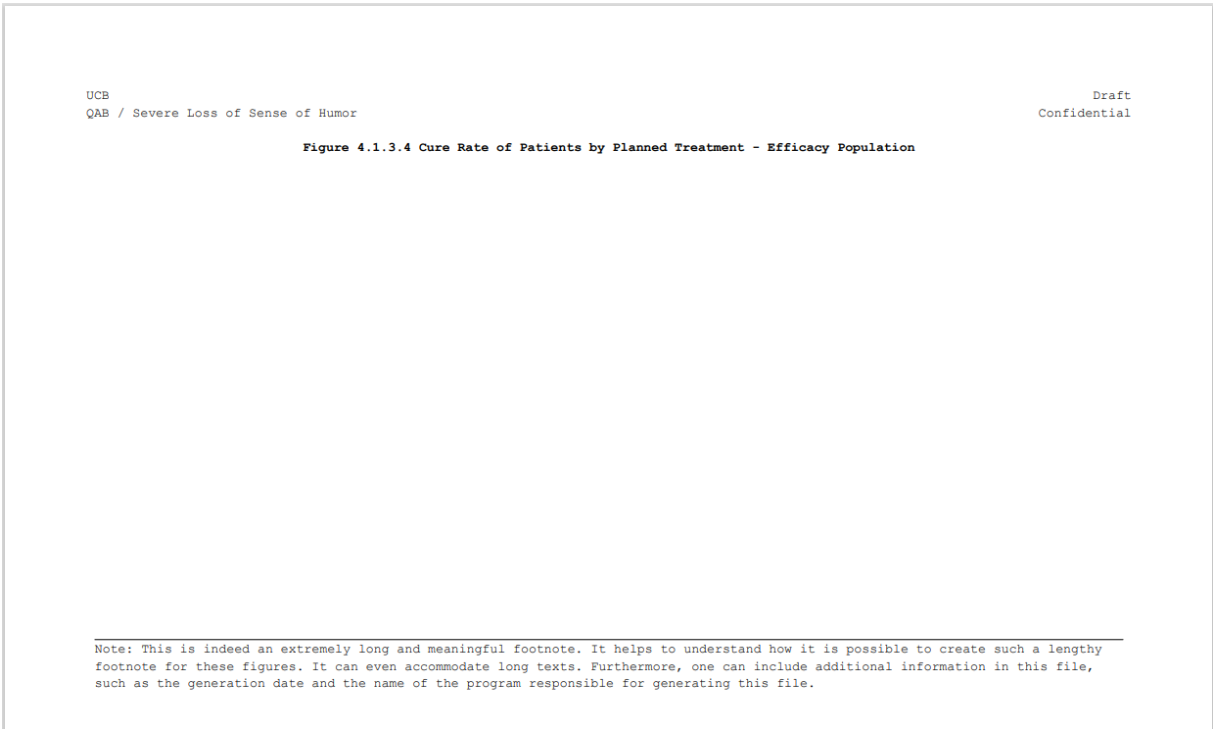

Figure24

These were generated by the code displayed in Figure25, below we describe how this has been done.

First, we create a function called add_header, which will be utilized later when we create a page template. This function basically creates a canvas and invokes the predefined styles with the getSampleStyleSheet function. After that, we declare several styles that will be used later in the header and title. Next, we declare the texts and use the wrap() method on each Paragraph object to determine the width and height of each paragraph when wrapped within the document's width and top margin. Once the dimensions of each paragraph are determined, the code uses the drawOn() method to draw each paragraph on the canvas at the appropriate position. The positions are calculated based on the dimensions of the document, the left margin, and the top margin.

We store the footnote texts in the my_text variable and create a new PDF with the SimpleDocTemplate function. The pagesize parameter sets the page size to landscape A4 format, and we define the margins. A frame is defined using the Frame function, specifying the position and dimensions of the frame. We then create a template object using the previously defined frame and call the add_header function. The doc object is updated with the template in the next step.

Lastly, we define the elements to be applied to the document, such as setting up the footer's vertical space with the Spacer function, configuring several aspects of the line that separates the footnote from the trunk of the document, and placing our footnote with the Paragraph command. The build method is called on the doc object, passing in a list of elements to be included in the document. In this case, the spacer, horizontal rule line, and paragraph are added.

By executing this code, a PDF document will be generated with the specified formatting and content.

```
def add_header(canvas, doc):
    canvas.saveState()
    styles = getSampleStyleSheet()

    # Create a new style for the left-aligned text
    left_aligned_style = ParagraphStyle('leftAligned', parent=styles['Normal'], alignment=0, fontName='Courier', fontSize=9)

    # Create a new style for the right-aligned text
    right_aligned_style = ParagraphStyle('rightAligned', parent=styles['Normal'], alignment=2, fontName='Courier', fontSize=9

    # Create a new style for the centered text
    centered_style = ParagraphStyle('centered', parent=styles['Normal'], alignment=1, fontName='Courier-bold', fontSize=9)

    header_left_1 = Paragraph("UCB", left_aligned_style)
    header_left_2 = Paragraph("QAB / Severe Loss of Sense of Humor", left_aligned_style)
    header_right_1 = Paragraph("Draft", right_aligned_style)
    header_right_2 = Paragraph("Confidential", right_aligned_style)

    w_left_1, h_left_1 = header_left_1.wrap(doc.width, doc.topMargin)
    w_left_2, h_left_2 = header_left_2.wrap(doc.width, doc.topMargin)
    w_right_1, h_right_1 = header_right_1.wrap(doc.width, doc.topMargin)
    w_right_2, h_right_2 = header_right_2.wrap(doc.width, doc.topMargin)

    header_left_1.drawOn(canvas, doc.leftMargin, doc.height + doc.topMargin - h_left_1)
    header_left_2.drawOn(canvas, doc.leftMargin, doc.height + doc.topMargin - h_left_1 - h_left_2)
    header_right_1.drawOn(canvas, doc.width + doc.leftMargin - w_right_1, doc.height + doc.topMargin - h_right_1)
    header_right_2.drawOn(canvas, doc.width + doc.leftMargin - w_right_2, doc.height + doc.topMargin - h_right_1 - h_right_2)

    title = Paragraph("Figure 4.1.3.4 Cure Rate of Patients by Planned Treatment - Efficacy Population", centered_style)
    w, h = title.wrap(doc.width, doc.topMargin)
    title.drawOn(canvas, (doc.width - w) / 2 + doc.leftMargin, doc.height + doc.topMargin - 4 * h)

my_text = "Note: This is indeed an extremely long and meaningful footnote. It helps to understand how it is possible to creat
doc = SimpleDocTemplate("example_flowable.pdf", pagesize=landscape(A4),
                        rightMargin=2*cm, leftMargin=2*cm,
                        topMargin=15*cm, bottomMargin=2*cm)

frame = Frame(doc.leftMargin, doc.bottomMargin, doc.width, doc.height, id='normal')
template = PageTemplate(id='test', frames=frame, onPage=add_header)
doc.addPageTemplates([template])

# Add a rule line and a spacer before the 'Note:' footnote
elements = [Spacer(1, 10), HRFlowable(width="100%", thickness=0.5, lineCap='round', color=colors.black, spaceBefore=0,
                                      spaceAfter=0, hAlign='CENTER', vAlign='BOTTOM', dash=None),
            Paragraph(my_text.replace("\n", "<br />"), normal_style)]
doc.build(elements)
```

Figure25

The trunk section is handled differently. Firstly, we require a Plotly figure object and a predefined HTML code (which will be included in the program attachments of this paper). We embed the Plotly HTML code within this HTML, and then convert it into a PDF using the xhtml2pdf package and its pisa function.
In Figure26, you can observe the appearance of the predefined HTML. Here, we define several styles and specify the dimensions for the different sections: the header_frame, content_frame, and footer_frame. While the content frame is the most important from our perspective, the position and size of the other frames are also significant. However, we will not populate the header or footer; they will solely serve as physical separators since they are already included in the other PDF file. The FIGURES section acts as the engine, as we convert our Plotly figure into HTML code and nest it there. This results in a formatted HTML that can be converted into a PDF file. You can refer to Figure27 to see this piece of code.

```html
<html>
<head>
<style>
    @page {
        size: a4 landscape;
        @frame header_frame {           /* Static Frame */
            -pdf-frame-content: header_content;
            left: 50pt; width: 742pt; top: 50pt; height: 40pt;
        }
        @frame content_frame {          /* Content Frame */
            left: 190pt; width: 594pt; top: 90pt; height: 346pt;

        }
        @frame footer_frame {           /* Another static Frame */
            -pdf-frame-content: footer_content;
            left: 50pt; width: 742pt; top: 772pt; height: 20pt;
        }
    }

</style>
</head>

<body>

    <!-- Content for Static Frame 'footer_frame' -->
    <div id="footer_content">Page <pdf:pagenumber>
        of <pdf:pagecount>
    </div>

    <!-- HTML Content -->
    {{ FIGURES }}
</body>
</html>

```

Figure26

```python
import base64
import plotly.express as px
from xhtml2pdf import pisa


def figure_to_base64(figures):
    images_html = ""
    for figure in figures:
        image = str(base64.b64encode(figure.to_image(format="png", scale=2)))[2:-1]
        images_html += (f'<img src="{image}"><br>')
    return images_html

def create_html_report(template_file, images_html):
    with open(template_file,'r') as f:
        template_html = f.read()
    report_html = template_html.replace("{{ FIGURES }}", images_html)
    return report_html

def convert_html_to_pdf(source_html, output_filename):
    with open(f"{output_filename}", "w+b") as f:
        pisa_status = pisa.CreatePDF(source_html, dest=f)
    return pisa_status.err

figures = [fig]

images_html = figure_to_base64(figures)
trunk_html = create_html_report("C:/SafetyDashboard/HTML/template.html", images_html)
convert_html_to_pdf(trunk_html, "trunk.pdf")
```

Figure27

The result of the code shown in Figure27 can be seen below in Figure28. It is simply a PDF without a title, header, or footer.

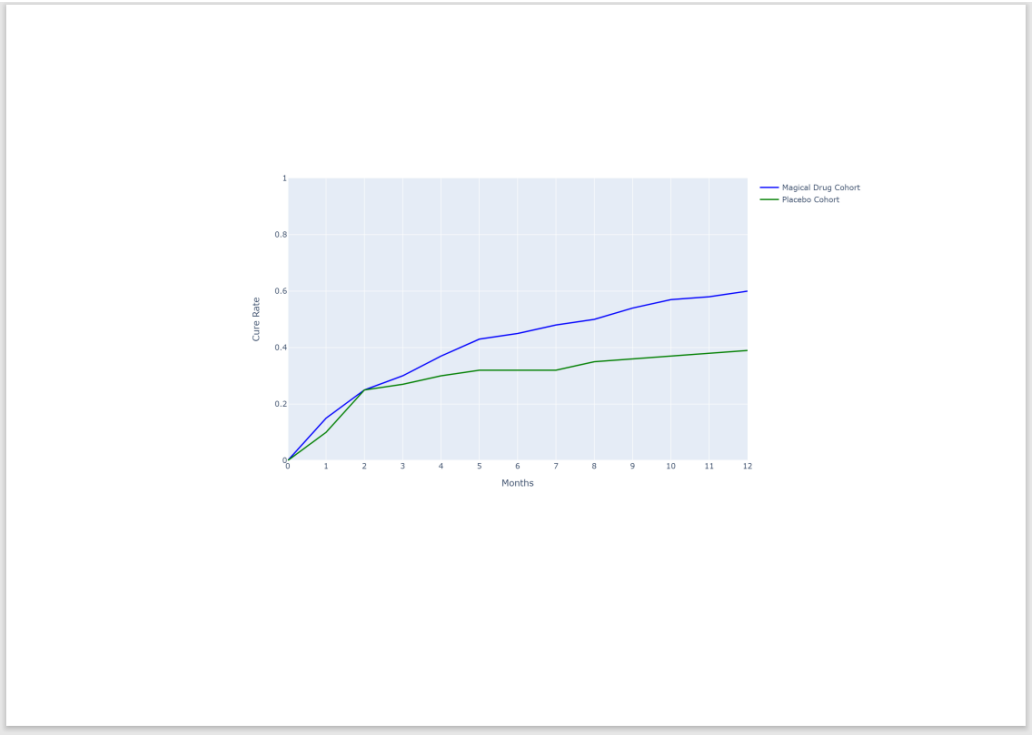

Figure28

The only thing left is to overlay the two PDFs, resulting in a professional-looking PDF file. To accomplish this, we utilized the PyPDF2 library, as shown in the code below. In the code, we overlay the pages one by one and generate a new PDF file from them (see in Figure29).

```
def overlay_pdf_files(file1, file2, output_file):
    pdf1 = PyPDF2.PdfReader (file1)
    pdf2 = PyPDF2.PdfReader(file2)
    output = PyPDF2.PdfWriter ()

    # Ensure both PDF files have the same number of pages
    if len(pdf1.pages) != len(pdf2.pages):
        raise ValueError("Both PDF files must have the same number of pages.")

    for page_num in range( len(pdf1.pages) ):
        page1 = pdf1.pages[page_num]
        page2 = pdf2.pages[page_num]

        page1.merge_page (page2)
        output.add_page (page1)

    with open(output_file, "wb") as result:
        output.write(result)

# Example usage
overlay_pdf_files("trunk.pdf", "example_flowable.pdf", "report.pdf")
```

```
! open report.pdf
```

Figure29

The result of the overlay can be observed in Figure30. It is worth noting that at this stage, we have not yet included pagination or other relevant information, such as the program's last run date or its name. These enhancements will be addressed in subsequent stages. Nevertheless, we are pleased to have successfully demonstrated proof-of-concept examples, which have motivated us to explore further and work towards creating an audit-ready environment capable of producing and reproducing Python outputs.

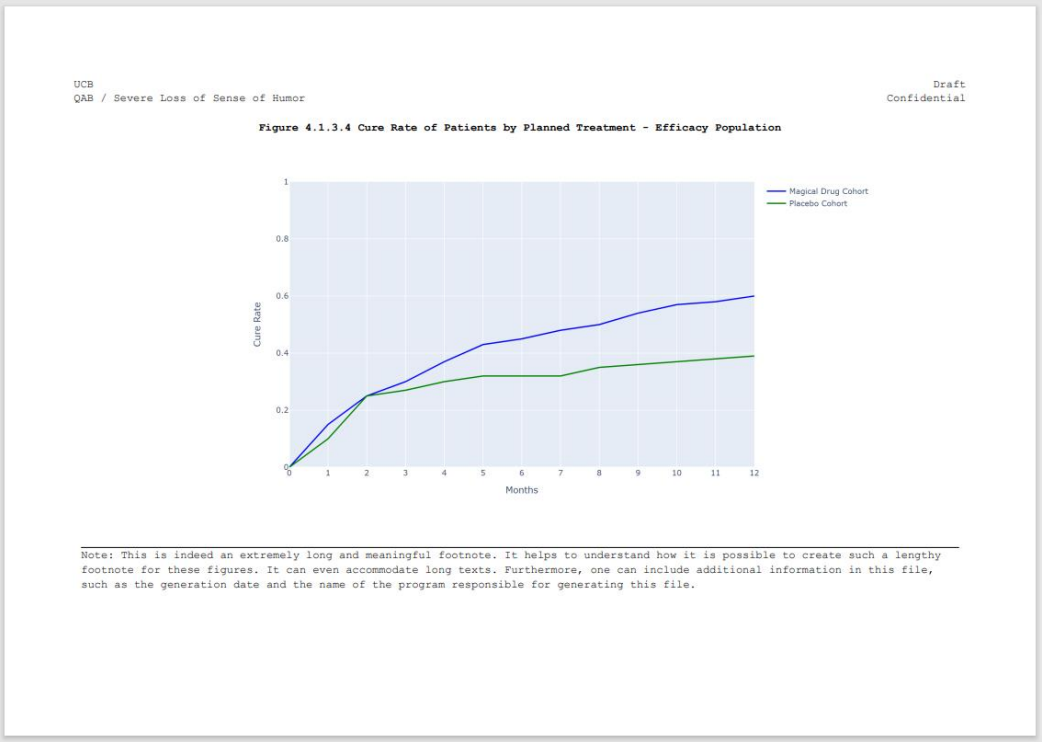

Figure30

HOW TO HANDLE PACKAGE UPDATES?

This is certainly not an easy question. One approach is to use Anaconda Navigator, where you can set up your own environments. By doing so, you can ensure that the packages used for development are not updated, thus maintaining compatibility with the syntax. Another option is to freeze environments using the "pip freeze" command and save the environment specifications to a text file. This preserves the versions of the different packages used. Currently, we are collaborating with our IT department to create containerized environments. This will allow the programs to run consistently and produce the same results, regardless of any recent library updates.

CONCLUSION

Using both SAS, R/RStudio, and Python programming languages together has many benefits. It promotes collaboration and allows us to make the most of the strengths of each language. It's clear that using a mix of languages in a multilanguage environment, where SAS, R/RStudio, and Python can work together, and having a flexible approach to language choice, will play a significant role in the pharmaceutical industry and statistical programming. By embracing this approach, we can effectively leverage the power of these languages and stay ahead in this dynamic field.
It is clear for the authors Python will play an important role in our field, it will help more and more with tool/application/ dashboards.

REFERENCES

(1) Michael Stackhouse, Covance, Inc. Why SAS Programmers Should Learn Python Too, PhUSE US Connect 2018, Paper TT02.
Why SAS Programmers Should Learn Python Too (lexjansen.com)
(2) Michael Stackhouse, Covance, Inc, Ensuring Programming Integrity with Python: Dynamic Code Plagiarism Detection, PhUSE US Connect 2019, Paper TT04.
Ensuring Programming Integrity with Python: Dynamic Code Plagiarism Detection (lexjansen.com)
(3) Hengwei Liu, Daichi Sankyo, Inc. Python for Visualization is Solid Tumor Studies, PhUSE US Connect 2020, Paper DV01.
SESUG 2020 Paper Title (lexjansen.com)
(4) Linghui Zang, A Duet of SAS and Python Generating XML File for Results Disclosure to ClinicalTRials.gov and EudraCT, PhUSE US Connect, 2022, Presentation ET09.
PRE_ET09.pdf (lexjansen.com)
(5) Sucharita Pilli, Novo Nordisk, Circus of Python in a Real Word Evidence, PhUSE US Connect, 2022
PRE_ET02.pdf (lexjansen.com)
(6) What is XSL? (w3.org)

(7) Liling Wei, Kathy Chen, Pharmacyclics Inc, SAS® vs Tableau: Creating Adverse Event/ Concomitant Medication Time Line Plot, PharmaSUG, 2017, Paper DV05
SAS vs Tableau: Creating Adverse Event/ Concomitant Medication Time Line Plot (lexjansen.com)
(8) Sanjay Matange, SAS Institute Inc, Patient Profile Graphs Using SAS, PharmaSUG, 2013, Paper MS14-SAS
PharmaSUG-2013-MS14-SAS.pdf (lexjansen.com)
(9) Amos Shu, Endo Pharmaceuticals, Qiunghua (Kathy) Chen, Exelixis Inc, Techniques of Preparing Datasets for Visualizing Clinical Adverse Events, PharmaSUG, 2014, Paper DG01.
PharmaSUG-2014-DG01.pdf (lexjansen.com)
(10) Plotly: The front end for ML and data science models
(11) pandas documentation — pandas 2.1.1 documentation (pydata.org)

ACKNOWLEDGMENTS

The authors would like to sincerely thank the management of UCB, particularly Alberto Montironi, Cindy Stroupe, Lari Dingman, Brian Mabe, Phyllis Smetana, and our former head of Programming, Andy York, for their invaluable support. Their recognition of the potential of Python has been instrumental in the success of this project.

Furthermore, Andras Kasa would like to express his deep appreciation to his father for introducing him to programming at a young age. This early exposure to programming has had a profound impact on his life and professional journey.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:
Andras Kasa
UCB Biosciences GmbH
Alfred-Nobel-Str. 10
40789 Monheim, Germany
Email: andras.kasa@ucb.com
Matt Davies
Email: mmdavies@hotmail.co.uk
13 Church Street, Aberkenfig,
Bridgend, Cardiff, CF32 9AU