PhUSE EU Connect 2019

Paper TT05

An Introduction to Python: The SAS Programmers Guide

Bradley Harris, GlaxoSmithKline, Uxbridge, UK

ABSTRACT

SAS® has been the preferred programming language of the pharmaceutical industry for the last 25+ years. Until recently, SAS has enjoyed unrivalled tenure as the universally accepted language across the industry, but now there are at least two other viable candidates that could represent a realistic alternative: one is Python™, the other is R. The rigorous level of standards and regulations within the industry, along with perceptions of what will be accepted by regulatory agencies, has typically led to slow adoption of new tools and technology. If we are to move forward, as with all things, we need to explore new and possibly more effective ways of working.

As such in this paper I will explore the Python programming language, a highly versatile language that can be used for anything from simple data manipulation and website development, to machine learning and complex visualizations. This paper will cover my own journey as a SAS programmer exploring the vast landscape of Python Programming. I will cover ways to perform several key SAS programming tasks within Python, important Python packages (primarily NumPy and Pandas), the different Python programming environments available, as well as thoughts on how we can begin to implement this in our day to day roles as programmers within the pharmaceutical industry.

INTRODUCTION

The objective of this paper is not to scrutinise the use of SAS, it is an extremely powerful, tried and tested language that has served the pharmaceutical industry for many years. Rather this paper is looking to explore several aspects of the Python programming language, providing learning resource to any SAS programmers interested in learning the language like myself, and to review the plausibility of future use of Python within the industry.

Through the course of this paper I will look to provide a comprehensive introduction to the Python programming language for the complete beginner. I will start from the very beginning with the tools we will need to educate ourselves, and by the end hopefully leaving you with the expertise to begin producing datasets through various data manipulation techniques, and through different statistical procedures. My aim is to explore only what is useful to the complete beginner, as such I will try to steer away from overly technical information surrounding the language and its implementation.

AN OVERVIEW OF PYTHON

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. It's high-level built in data structures, combined with dynamic typing and dynamic binding make it a very attractive programming language for many tasks. Python has a simple and easy to learn syntax which emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages which promotes program modularity, and code re-use. I will refer to several such packages within this paper. Translation: Python is a language that makes life easy for the programmer, it is simple to program in, and a lot of tasks are made more efficient by its intelligent interpreter.

The Python interpreter and its extensive base library are available free of cost.

Off the shelf, Python does not offer much to the programmer of the pharmaceutical industry, however the option to import further code libraries (we could compare this to needing to install a module like PROC SQL before using it in SAS) transforms it into a powerful tool for data manipulation/analysis and visualization. I will make use of the following:

- **Pandas –** Provides high performance, easy-to-use data structures and data analysis tools, including easy methods of importing and exporting data files.
- **datetime –** This is a module contained in the standard Python library, with several useful functions for working with date and time data.

Technical Specifications

Throughout this paper there will be multiple snippets of code presented, utilising both base Python and several additional packages. Listed below are the packages used and their versions:

- Python (including datetime module) v3.7.3
- Pandas v0.24.2

OTHER PACKAGES OF INTEREST

Although I will not cover these packages, they are worth mentioning as they offer functionality that could be of use.

- **NumPy** – 'Numerical Python' this is the core library for scientific computing. It contains a powerful n-dimensional array object. It also has uses in linear algebra and random number operations. I see some use for this in the simulation / random generation of data. However, the mathematical operations that it provides, are not so useful with the datasets we work with. Especially when compared to the functionality pandas offers. For further documentation see **Ref. 1.**
- **MatPlotLib** – 2D plotting library, which can produce publication quality figures in a variety of hardcopy formats and interactive environments across platforms. For further documentation see **Ref. 2.**
- **Scikit-Learn** – Useful module for getting started with machine learning in Python. Contains simple and efficient tools for data mining and data analysis. For further documentation see **Ref. 3.**

GETTING STARTED WITH PYTHON

Before we can start programming in Python, we need to setup an environment to do so. There are several good options for a programmer taking their first steps with Python. The example programming environments listed below are all freely available. Some require downloading and installing, whereas some are browser notebooks, allowing the creation of Python programmers in a solely web browser-based platform.

- Spyder
- VB Code
- JupyterLab
- Jupyter Notebook

I recommend downloading and installing Anaconda (the download can be found easily on the internet or see **Ref. 4.**). This is a package management system, it acts as a central location/launcher in which you can access some very useful environments such as Jupyter and Spyder mentioned above. It also helps to simplify the installation of further Python programming packages through its user interface. If you aim to use one of these environments without Anaconda, or any other environment (there are a large number more than the few I have focused on here), refer to the individual documentation for each for guidance on installing modules.

With one of these environments set-up we can then begin exploring Python programming.

COMPLETE BASICS OF PYTHON PROGRAMMING

Before we begin looking at the programming techniques that will really be of value to a pharmaceutical industry programmer, it is necessary to review some of the core elements that any programmer needs to know when starting to program in a new language.

ARITHMETIC OPERATORS

These are the basic operators that give us arithmetic functionality. They operate in the normal mathematical way as they do in SAS. The arithmetic operators available are addition (**+**), subtraction (**-**), multiplication (*****), division (**/**), modulus (**%**), exponent (******) and floor division (**//**).

For example, if the variable *a* holds 10 and the variable *b* holds 20 then *a+b = 30* and *a*b = 200* etc.

ASSIGNMENT OPERATORS

These are a selection of operators that facilitate the assignment of calculated values into new variables. The most basic is the assignment operator itself (**=**), it is worth mentioning though that unlike SAS, Python has a separate operator for comparing if values of two variables are equal (**==** covered in comparison operators). There also a number of assignment operators that are useful in iterative scenarios, in fact there is one for each arithmetic operator: add AND (**+=**), subtract AND (**-=**), multiply AND (***=**), divide AND (**/=**), modulus AND (**%=**), exponent AND (****=**) and floor division AND (**//=**). These work by performing he arithmetic operation on the left and right operand and then assigning it to the right operand.

For example, *c*+= *a* is equivalent to *c = c+a* and *c***=*a* is equivalent to *c = c**a*.

COMPARISON OPERATORS

These operators are used for performing arithmetic comparisons of variables and constants. They compare given operands and return either true or false. The comparison operators are: equal to (**==**), not equal to (**!= or <>**), greater than (**>**), less than (**<**), greater than or equal to (**>=**) and less than or equal to (**<=**).

For example, if the variable *a* holds 10 and the variable *b* holds 20 then *a == b* is false but *a != b* is true.

LOGICAL OPERATORS

These operators are used for performing the logical operations. They are used in conjunction with a number of other operations statements and return a value of true or false. The operators are: AND, OR and NOT.

For example, if the variable *a* holds 10 and the variable *b* holds 20 then *a == 20 AND b == 20* is false, but *a == 20 OR b == 20* is true. Also *NOT(a == 20 AND b == 20)* is true.

OTHER OPERATORS

There are several other operators available in Python, such as bitwise operators which need no exploration here. We also have access to membership operators ***in*** and ***not in*** which operate similarly to SAS returning true or false based on whether a variable value is present within a specified series. **For a full explanation of all operators described in the previous sections see Ref. 5.**

IMPORTING CODE LIBRARIES

To make use of additional code libraries that we have installed, we must import them at the beginning of our Python programs. For example:

```
import pandas as pd
from datetime import datetime
```

We use the ***import*** statement followed by the name of the code library we wish to import. If we wish we can also define a nickname for our library to save us time when we call it. This is done by following the import statement with ***as*** and the short hand name we wish to use. We use the ***from*** statement when we only wish to import a certain module from a library.

COMMENTING CODE

Commenting code is fundamental in any programming language, to ensure readability, re-usability of code, and to provide detail in the case of regulatory review or audit. In Python we can add multi-line comments to serve as documentation for others reading our code, and single line comments to explain small snippets of code.

```
''' Multi line comments are declared like this '''
# Single line comments are declared like this
```

MOVING INTO PYTHON FOR SAS PROGRAMMERS

IMPORTING DATA

The first thing any programmer would wish to do when creating an analysis dataset, or an output is to set up their data libraries and import the datasets to be worked with. Pandas offers an extremely easy way to import any data that we wish to work with. It offers ***'read_xxx'*** and ***'to_xxx'*** functions that allow us to read in and write to several different data objects. **For the full list see Ref. 6.** See the below example which illustrates how we can create working datasets from permanent SAS datasets.

```
#Create variable to store the file paths
sdtm = 'C:/dummypath/SDTM'
adam = 'C:/dummypath/ADAM'

#Create the working datasets
ae = pd.read_sas(sdtm+'/ae.sas7bdat', format = 'sas7bdat', encoding="ISO-8859-1")
suppae = pd.read_sas(sdtm+'/suppae.sas7bdat', format = 'sas7bdat', encoding="ISO-8859-1")
```

Here, like setting a libname in SAS, we initialise a variable with a string containing the file path of our desired data. We then utilise the ***read_sas*** function (when using a function outside of the base Python library we must reference the package we are using, here we write ***pd.read_sas*** to reference pandas) to import the ae and supp ae SDTM datasets. Again, in a similar fashion to SAS we can use our stored file path and concatenate it (one of the simplest methods of concatenation in Python is to literally add the strings together as seen in the example) with the filename of the dataset that we wish to import. The ***read_sas*** function requires us to specify a couple more parameters to ensure we get the data as we expect it. Firstly, we must specify the format, the function can handle both sas7bdat and xport files. Finally, we must specify the encoding, if we do not the text data will be stored as raw bytes (this makes it impossible for us to work with the text fields in our datasets).

One challenge faced importing SAS datasets is that Python cannot interpret any formats etc coming from the SAS dataset. As such all numeric variables are stored as float. This becomes an issue when we consider the various code variables used in CDISC standard datasets. They will be represented for example as 12345678.0. As such we need a way to convert these to integer. We can either do this in conjunction with another pandas statement (as the output of each pandas function is a data frame we can link multiple functions together using dots), or as a standalone step using the pandas function ***'astype'***.

```
#Convert variables to integer straight after reading in the data
suppae = pd.read_sas(sdtm+'/suppae.sas7bdat', format = 'sas7bdat', encoding="ISO-8859-1")\
                                    .astype({'IDVARVAL': int}).astype({'SUBJID': int})

#Convert variables to integer in a seperate step
suppae[['IDVARVAL', 'SUBJID']] = suppae[['IDVARVAL', 'SUBJID']].astype(int)
```

Note that backslash is used to allow us to continue a code statement on the next line.

For the full pandas user guide see Ref. 7. The guide is comprehensive and covers all available features within the pandas package.

COLUMN OPERATIONS

To perform any sort of derivation we will need to perform a number of different operations on the columns in our dataset. Any of the operators mentioned earlier in this paper can be used, however for operations other than arithmetic ones we will need to use custom functions, which will be covered later. So, for now we will look at performing some arithmetic functions on different variables/columns in our datasets.

We can specify a column from our dataset in the following ways:

```
#First way, can only specify one column this way
ae.aestdtc
#Second way, can specify one or multiple columns this way if required
ae['AESTDTC']
ae[['AESTDTC', 'AEENDTC']]
```

We can then use the columns the same way we would for any SAS dataset variables. For example, deriving the duration of an ae.

```
#import python datetime module for working with datetime data
from datetime import datetime
#convert character datetime variables to datetime
#fromisoformat can only handle strings, as a column is a series we use
#a simple loop to process each observation individually
ae3['AESTDTTM'] = [datetime.fromisoformat(x) for x in ae3['AESTDTC']]
ae3['AEENDTTM'] = [datetime.fromisoformat(x) for x in ae3['AEENDTC']]
#Now create the seperate time and date variables
ae3['AESTDT'] = [x.date() for x in ae3['AESTDTTM']]
ae3['AEENDT'] = [x.date() for x in ae3['AEENDTTM']]
#perform the derivation of the duration from the two new datetime variables
ae3['AEDUR'] = (ae3['AEENDT'] - ae3['AESTDT']).dt.days +1
```

Utilising the ***datetime*** module (extremely useful code module from the Python standard library for working with datetime variables) is necessary here to convert the character SDTM datetime variables to numeric datetime variables. Along with the pandas ***dt.days*** function we are able to calculate the duration of the ae in in days as a column of integer type. We can perform any arithmetic operations on the columns of our dataset as required.

SUBSETTING DATA, CONTROLLING COLUMNS AND SORTING

Another common task that a SAS programmer would wish to achieve, is to control the contents of a dataset. Through dropping (or keeping) columns, subset a dataset on given criteria, sorting the data or any combination of the three. Pandas offers us simple ways to accomplish all these actions.

```
#dropping unneeded variables
ae3 = ae3.drop(['AESTDTC', 'AEENDTC'], axis=1)
#keeping required variables
safetypop = ae3[['USUBJID','SAFFL', 'AESEQ']]
#subsetting data
safetypop = safetypop[safetypop.SAFFL=='Y'] #Method1
safetypop = safetypop.query('SAFFL == "Y"') #Method2
#sorting the data by descending usubjid
safetypop = safetypop.sort_values('USUBJID', ascending=False)
```

Dropping columns – We can use the pandas function ***drop*** (it is worth briefly mentioning here, as Python is an object oriented language when we created our working datasets earlier using ***pd.read_sas*** certain pandas objects are associated with that dataset, as such we are not required to call pandas again to use methods such as ***drop***), we can specify one or more variables to drop from the dataset (the ***axis*** parameter is present in a number of different functions ***axis=1*** tells the function to work on the columns instead of rows).

Keeping columns – We can simply create a new dataset, specifying the columns we wish to keep in the same way we saw in the column operations section.
Sub-setting the data – This can be done in two ways. The first by specifying the rows to select through the same method we select variables to be displayed, however this method limits us to simple subsets only (no logical operators) and requires us to call our dataset twice. A simpler way is the second method, which is to use pandas ***query*** function, which functions almost identically to a where statement in SAS. With the query function we can define subsets with multiple criteria using logical operators.
Sorting the data – This can be done using the pandas function ***sort_values***, we can sort on any number of variables that we require. The ***ascending*** can be used to specify whether to sort in ascending order or descending order, if we specify multiple variables, we need to specify a series for the parameter value. For example:

```
safetypop = safetypop.sort_values(['USUBJID', 'AESEQ', 'SAFFL'], ascending=[1,0,1])
```

This will result in our dataset being sorted by ascending USUBJID, descending AESEQ and ascending SAFFL.

APPENDING AND MERGING

Another commonly required technique is the combining of two or more datasets. Either by appending/concatenating them or merging them together. Pandas offers a number of functions for the appending of data of which two proved to be most useful.

```
#append seperated age group data
all_ages1 = age1.append([age2, age3, age4, age5], ignore_index=True)
all_ages2 = pd.concat([age1, age2, age3, age4, age5], ignore_index=True)
```

We can use both the ***append*** and ***concat*** functions from pandas to accomplish the task, here though I prefer ***append*** when appending just two datasets as it can be done directly on to the first dataset. Whereas for ***concat*** all datasets to be appended are named within the function which feels slightly cleaner when combining multiple datasets. I specified ***ignore_index=True*** so that the index of each observations remains unique.

Moving on to merging, this is extremely simple using pandas. Like SQL it has the advantage over SAS that no sorting is required prior to the merge, and also, we can merge on columns that have different names between datasets. Another useful thing is that no data is ever overwritten. If datasets contain duplicate columns both will be kept, and each will be appended with a different letter to specify which dataset it originated from! For a simple example consider merging common variables from the ADSL dataset.

```
#Merge on the adsl variables
ae3 = pd.merge(ae2, adsl, how='left', on='USUBJID')
```

The syntax is very straight forward, the first parameter is our ‘left’ dataset, and the second is our right dataset. We can then specify the type of join that we wish which is again very similar to SQL. Finally, we specify the variable to be our merge key. Now for a more complex example, we can consider the joining of the supplemental data on to its corresponding SDTM domain, where we first need to transpose the supplemental data.

```
#Transpose the suppae dataset
suppae_t = suppae.pivot_table(index=['USUBJID','IDVARVAL'], columns='QNAM', values='QVAL',\
                              aggfunc=lambda x: ' '.join(x))

#Merge the ae and supp ae
ae2 = pd.merge(ae, suppae_t.reset_index(level=['USUBJID', 'IDVARVAL']).astype({'IDVARVAL': int}),\
               how='left', left_on=['USUBJID', 'AESEQ'], right_on=['USUBJID', 'IDVARVAL'])
```

First, we transpose the data using pandas ***pivot_table***, this functions similarly to SAS***.*** Where ***index*** is the by variables, ***columns*** are the id variables and ***values*** are the var variables in an equivalent ***PROC TRANSPOSE***. A couple of caveats here though. Firstly, we are required to specify an aggregate function, this tells ***pivot_table*** what to do when there is more than one observation per combination of the index variables. Second is that the variables specified in ***index*** will be converted to indexes, and so if we wish to continue using them in our program, we will need to convert that back to data columns. We do that in the next step using the ***reset_index*** function, which simply returns the specified index to a data column.

In the above example we have the ae SDTM domain as our ‘left’ dataset, then as our ‘right’ dataset we have the transposed suppae, where we have reset USUBJID and IDVARVAL back to data columns and converted IDVARVAL into integer form using ***astype*** as seen earlier in this paper. In this example, we have used two levels of merge key, and you can see that we have specified them separately for each dataset using ***left_on*** and ***right_on***. So here USUBJID is common, and then AESEQ will be merged with IDVARVAL. This will give us our ae data with correctly merged supplemental data.

For the full documentation on pandas ***merge*** see **Ref. 8.**
For a useful guide on merging in Python see **Ref. 9.**

CREATING CUSTOM FUNCTIONS

One thing that we are not able to do directly in Python is to apply logic to the columns of our datasets. For example, if I wanted to create a category variable, we cannot use the required if statements directly with our dataset. To apply logic in this way, to each individual observation in a column, we need to create a simple user defined function to help us. These functions share a lot of similarities with SAS macros, however I will not go into detail in this paper. If you wish to read more about custom functions, please see **Ref. 10.**

For a simple example, consider creating a code variable. Here I create code variable AESEVN from the decode AESEV.

```
#Create custom function to create code variable
#AESEVN for var AESEV
def sev_num(dset): #define the custom logic function
    if dset['AESEV'] == 'MILD':
        value = 1
    else:
        value = 2
    return value

#create the new var by applying the logic function
#to the appropriate variable
ae3['AESEVN'] = ae3.apply(sev_num, axis=1)
```

The required logic is defined within the function, we then use the pandas ***apply*** function to create our new code variable. ***Apply*** allows us to perform the logic on every row of our dataset, Python will not do so automatically like we would see with SAS. The ***apply*** function passes the name of our dataset (here ae3) into the ***dset*** parameter of the custom function ***sev_num***. The function returns value which is assigned to the current row of the new AESEVN column of ae3.

For a more complicated example consider deriving code and decode variables for weight groupings. Here I create WGR1 and WGR1N based on the subject's weight.

```
#Create custom function to create weight group
#code and decode variables
def cr8_weight(dset): #define the custom logic function
    if dset['WEIGHTSC'] < 25:
        wgr1 = '<25kg'
        wgr1n = 1
    elif 25 <= dset['WEIGHTSC'] < 50:
        wgr1 = '25-<50kg'
        wgr1 = 2
    elif 50 <= dset['WEIGHTSC'] < 75:
        wgr1 = '50-<75kg'
        wgr1n = 3
    else:
        wgr1 = '>=75kg'
        wgr1n = 4
    return wgr1, wgr1n

#we have to return our two values in one string so create a temp
#column for now
ae3['TEMP'] = ae3.apply(cr8_weight, axis=1).astype(str)

#now seperate into two columns in a temp dataset
#we cannot do this directly in our working dataset
temp = pd.DataFrame([x[1:-1].split(', ') for x in ae3['TEMP']], columns = ['t1', 't2'])
#now save the new columns into our working dataset
ae3['WGR1'] = [x[1:-1] for x in temp['t1']]
ae3['WGR1N'] = temp['t2'].astype(int)
#now finally drop the temp variable
ae3 = ae3.drop(['TEMP'], axis=1)
```

This time we look to create two variables within our function. The issue faced is that we can only return one value from the function, so here we return both values as a tuple and then create the two separate variables with some processing after the function. First, I create a temporary column in the dataset ae3 which contains a string of the values of WGR1 and WGR1N separated by a comma and in parentheses, this is done by applying the ***cr8_weight*** custom function. Next, I create a temporary dataset containing two columns of the individual values of WGR1 and WGR1N (note ***x[1:-1]*** removes the parentheses from the string). Finally, I assign the two temporary columns into our ae3 dataset as our required WGR1N and WGR1 columns (note we use ***x[1:-1]*** again to remove the quotes from the string), and then drop the temporary variable.

SUMMARISING DATA

Another task regularly performed, is the summarising of data. Primarily descriptive summary statistics, and frequency counts. Pandas offers us a very straightforward way to perform these tasks, using the ***groupby*** function, and various functions for producing statistics. To produce a standard set of summary statistics we can use ***describe,*** for example we could look at the statistics for the weight values within each weight group.

```
#summary statistics by weight group
#WEIGHTSC
summary_stats2 = ae3.drop_duplicates('USUBJID').groupby(['WGR1N', 'WGR1'])['WEIGHTSC'].describe()
```

I had to include the ***drop_duplicates*** function to remove duplicate values here as I had calculated the weight groups in the AE data only, and would have been counting each subjects weight multiple time otherwise (I could have derived them in ADSL to avoid this, but it's a good way to show the similarity between ***drop_duplicates*** and a PROC SORT with nodupkey in SAS). We specify the variables we wish to use as the different levels for our summary statistics in the ***groupby*** function, then use the square brackets to keep only the variables we wish to perform the calculations on. Finally, we use the describe function, and in this example, we are left with a dataset with the weight groups as the index and then columns with the following summary statistics: n, mean, std, min, Q1, median, Q3 and max. We could then use the ***reset_index*** function used earlier in this paper, if needed, to return the weight group variables to columns for further use.

We can also calculate singular statistics, which can be useful in different derivations.

```
#Individual statistics
mean_only = ae3.drop_duplicates('USUBJID').groupby(['WGR1N', 'WGR1'])['AAGE'].mean()
std_only = ae3.drop_duplicates('USUBJID').groupby(['WGR1N', 'WGR1'])['AAGE'].std()
min_only = ae3.drop_duplicates('USUBJID').groupby(['WGR1N', 'WGR1'])['AAGE'].min()
max_only = ae3.drop_duplicates('USUBJID').groupby(['WGR1N', 'WGR1'])['AAGE'].max()
median_only = ae3.drop_duplicates('USUBJID').groupby(['WGR1N', 'WGR1'])['AAGE'].median()
```

Here are a few standard examples, but pandas does offer calculations for more complex statistics too.

Now for an example of frequency counts, we can easily modify the variables we use in our ***drop_duplicates*** and ***groupby*** calls to produce some AE frequency counts.

```
#frequency counts
freq_count = ae3.drop_duplicates(['USUBJID', 'TRT01AN', 'TRT01A', 'AEBODSYS', 'AEDECOD'])\
                .groupby(['TRT01AN', 'TRT01A', 'AEBODSYS', 'AEDECOD'])['USUBJID'].count()
```

By modifying the variables in our function calls I have used the ***count*** function on the USUBJID variable to get unique subject AE counts.

For a useful guide on producing descriptive statistics with pandas please see **Ref. 11.**

EXPORTING DATA

The one shortfall I experienced in my time learning Python, was the lack of a reliable way to export data to an appropriate format. Ideally, we would need to be able to output our analysis data into XPORT format. And although there were Python modules available for this, after numerous hours of testing with them I could not find a way to export the data correctly. Unfortunately, this is the one functionality pandas does not offer (although not to say it won't be available in the future).

We can at least easily output a CSV file with our data. This would give us the option to do our programming in Python still. Although it would also mean we still would have a dependency on SAS despite our programming in Python, as we would need to do some amount of programming in SAS to import the CSV, define the appropriate metadata for our dataset and to export it correctly.

USEFUL RESOURCES

One of the great things about Python is that due to its wide availability, there are many users out there, posting their experiences, code etc. on to the internet. Which means there is an abundance of material out there to assist us. Throughout my journey I have found several useful pages, and guides which I have referenced throughout this paper. As such my first suggested port of call when facing any issue would be to turn to the internet. However, I also wanted to provide some final useful resources here to help.

It is very useful having several important things summarised in a small amount of space, so that we can just refer to one guide rather than having to search for many different things. **Ref. 12** is a 10 minute to pandas guide which covers many of the complete basics of pandas. **Ref. 13** is a pandas cheat sheet that covers a wide range of the functions available in pandas within just a couple of pages.

Finally, for those looking delve further into the world of Python for data analysis I suggest the O'Reilly ® '*Python Data Science Handbook'.* This is the most comprehensive guide to working with datasets in Python that I have found whilst working on this project. **Ref. 14** has links to read the full version as an online e-book, as well as several resources to

accompany it. Especially helpful are the Jupyter Notebooks which contain executable versions of every code demonstration in the book!

CONCLUSION

It has been a very interesting experience teaching myself Python for this project. Using Python, and in particular pandas, I feel as though I can accomplish most if not all tasks a programmer would wish to perform in relation to data manipulation and more specifically the creation of analysis datasets. SAS is a built for purpose programming language and so is unsurprisingly good at what it does, however I was impressed with how easily pandas handled some of the programming covered in this paper. As an example, the production of summary statistics, I felt this was much simpler to accomplish with pandas than writing a PROC SUMMARY is in SAS.

Based on my findings there could be room to argue in Pythons favour as a programming language to be utilised heavily within the pharmaceutical industry. The only drawback currently being the lack of ability to create SAS datasets or XPORT files, which means the process would still be dependent on SAS. However, in future it could be possible to see that functionality made available.

For now, I could still see Python being utilised for visualizations. If I were to take this research further, I would like to expose myself to the MatPlotLib library fully. With the ability to read in SAS data using pandas, and to produce high quality visualizations both static and interactive in several different formats. Python could help us to advance the way we think about reporting clinical trials.

REFERENCES

1. https://docs.scipy.org/doc/numpy/user/
2. https://matplotlib.org/3.1.1/users/index.html
3. https://scikit-learn.org/stable/user_guide.html
4. https://www.anaconda.com/distribution/
5. https://www.tutorialspoint.com/Python/Python_basic_operators
6. https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html#io-sas-reader
7. https://pandas.pydata.org/pandas-docs/stable/user_guide/
8. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.merge.html
9. https://www.shanelynn.ie/merge-join-dataframes-Python-pandas-index-1/
10. https://www.tutorialspoint.com/Python/Python_functions.htm
11. https://www.marsja.se/pandas-Python-descriptive-statistics/
12. https://pandas.pydata.org/pandas-docs/stable/getting_started/10min.html#min
13. https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf
14. https://github.com/jakevdp/PythonDataScienceHandbook

CONTACT INFORMATION

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

Bradley Harris
GlaxoSmithKline
Stockley Park West
Uxbridge
Middlesex
UB11 1BT
Email: Bradley.x.harris@gsk.com
Web: www.gsk.com