PharmaSUG China 2023 - Paper AP-144

Automatically Setting Column Width and Pagination using Python

Tao Zhang, Phastar Inc.

ABSTRACT

In producing TLFs, adjusting column width and pagination to produce visually good outputs is a tedious and time-consuming task. SAS programmers often spend a lot of time on these tasks even for a listing. When the data changes, there is a possibility of encountering issues with column width and pagination, requiring further adjustments. This article introduces a method for automating column width and pagination using Python. The method utilizes ReportLab library of Python to measure the space required for displaying text in a specific style. Based on these measurements, we can then set appropriate column width and pagination and produce the final PDF report. The approach provides a streamlined and efficient alternative to SAS PROC REPORT. With a simple SAS macro, it can be easily integrated in existing TLF creation processes.

INTRODUCTION

The ODS RTF/PDF destination and PROC REPORT provide a wide range of functionalities that enable SAS programmers to create various high-quality tables and listings. To ensure that the output presents data clearly and aesthetically, we often engage in a process of continuous trial and error to find the appropriate column width allocation and pagination logic. The fundamental reason for this is our lack of knowledge regarding the space required for displaying text in a specific style. In the case of ODS RTF, text wrapping and pagination are even determined by MS Word when rendering the output document. This iterative adjustment method can be tedious and time-consuming. Additionally, our programs often need to be rerun after future data updates to generate new output, and the previously set column width and pagination settings may no longer be suitable due to changes in data. This can increase the maintenance cost of the program. However, if we can calculate the space required for displaying text, we can utilize this information to set appropriate column width and pagination, thereby achieving fully data-driven layout of report. Much time and effort can be saved to focus on other programming and validation checks.

Python is a general-purpose programming language with a rich ecosystem of software libraries many of which find applications in the field of data science. ReportLab, an open-source software library based on Python, has the capability to generate complex PDF documents. It implements precise measurements of the space required for displaying text. In this article, we introduce a method for automatically setting column width and pagination based on ReportLab. With a simple SAS macro to invoke Python program, this method can be applied to our daily TLF processes and can also be utilized in the report module of TLF generation tools.

HOW TO MEASURE THE SPACE REQUIRED FOR DISPLAYING TEXT

The space required for displaying text is determined by many factors, such as font attributes (font face, font size, bold, italics) and paragraph properties (line spacing, indentation, line breaks). Some methods try to estimate the space based on the number of characters in the text. This approach can work well for fonts where each character takes up the same amount of space, known as monospaced fonts. However, this method can be less accurate for proportional fonts, where different characters can take up different amounts of space. An example of a commonly used proportional font is Times New Roman. Furthermore, various styles are applied to the text in our reports. This makes estimating space based on character count even less precise.

In ReportLab, when generating PDF documents, paragraphs are the fundamental building blocks. These paragraphs can be placed within tables, and the space (width and height) of both paragraphs and tables can be accurately measured in points (1 inch = 72 points). The following pieces of Python code demonstrate how to measure the space required for displaying a paragraph. The complete Python script can be found in Appendix.

We first create a paragraph with the text "The brown fox jumps over the lazy dog." It is wrapped within a space of 7200x7200, intentionally chosen to avoid line breaks. This allows us to measure the width required to display the paragraph without line breaks, which is found to be 160.518. Figure 1 shows what the paragraph looks like when displayed in a PDF file. We put it in a table cell with the width of 166.518 (160.518 plus 2*3 points as cell padding).

```
# paragraph style
pstyle = ParagraphStyle('test')
pstyle.fontName = 'Times New Roman'
pstyle.fontSize = 10
pstyle.leading = 12

# create a paragraph without line breaks
para1 = Paragraph('The brown fox jumps over the lazy dog.', style=pstyle)
height1 = para1.wrap(7200, 7200)[1]
widths1 = para1.getActualLineWidths0()
print('Height of the paragraph without line breaks: ', height1)
print('Width(s) of line(s) in the paragraph without line breaks: ',
widths1)
```

```
Outputs:
Height of the paragraph without line breaks:  12
Width(s) of line(s) in the paragraph without line breaks: [160.517578125]
```

Output 1. Measurement of the first paragraph

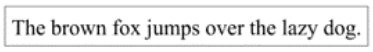

Figure 1. The first paragraph displayed

If we do not give the paragraph enough space, ReportLab will insert line breaks according to the given space. In below code, we create the second paragraph with the same text but wrap it within a space of 100x7200, since the text requires 160.518(see Output 1) points to display without line break. 100 points is not enough, so we get two lines instead. We can obtain the widths of the two lines, according to Output 2, they are 86.377 and 71.641 respectively. We put it in a table cell with the width of 92.377 (86.377 plus 2*3 points as cell padding). The paragraph is shown in Figure 2.

```
# create a paragraph with line breaks
para2 = Paragraph('The brown fox jumps over the lazy dog.', style=pstyle)
height2 = para2.wrap(100, 7200)[1]
widths2 = para2.getActualLineWidths0()
print('Height of the paragraph with line breaks: ', height2)
print('Width(s) of line(s) in the paragraph with line breaks: ', widths2)
```

```
Outputs:
Height of the paragraph with line breaks:  24
Width(s) of line(s) in the paragraph with line breaks: [86.376953125,
71.640625]
```

Output 2. Measurement of the second paragraph

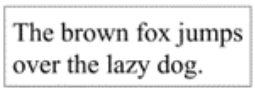

Figure 2. The second paragraph displayed

Sometimes, we may want to insert line breaks or apply other paragraph styles within the text. ReportLab uses a basic set of XML tags to provide markup. The most basic of these are bold (...), italic (...) and underline (<u>...</u>). A break (
) tag is also allowed. The following example demonstrates the measurement and display of text with bold and line break tags.

```
# create a paragraph with intra-paragraph markup
para3 = Paragraph('The <b>brown</b> fox<br/>jumps over the lazy dog.',
style=pstyle)
height3 = para3.wrap(7200, 7200)[1]
widths3 = para3.getActualLineWidths0()
print('Height of the paragraph with intra-paragraph markup: ', height3)
print('Width(s) of line(s) in the paragraph with intra-paragraph markup: ',
widths3)
```

```
Outputs:
Height of the paragraph with intra-paragraph markup:  24
Width(s) of line(s) in the paragraph with intra-paragraph markup:
[61.66015625, 98.5888671875]
```

Output 3. Measurement of the third paragraph

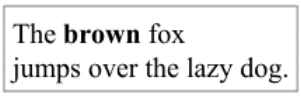

Figure 3. A paragraph with line break tag

AUTOMATICALLY SETTING COLUMN WIDTH AND PAGINATION

In the previous section, we demonstrated how to measure the space required for text display. We can apply these measurements to data that needs to be displayed in tables and listings. Based on these measurements, we can design strategies to set column widths and pagination. An example of this is provided below.

Step 1: Measure the Original Widths of All Data Points and Column Headers

Figure 4 is a sample of 5 records from our test data. Figure 5, on the other hand, shows the column headers. We can apply the method we discussed in the previous section to calculate the width of all data points and column headers.

Table 1 provides a summary of these calculated widths. It includes rows that represent the percentiles of the width of data points, ranging from the minimum to the maximum. The row labeled as "_max" represents the maximum value among the "max" row and the "header" row. The column labeled as "_total" represents the total width of each row.

	col1	col2	col3	col4	col5	col6	col7	col8	col9	col10
0	E1301002	On Treatment	47/F/Asian	Blood and lymphatic system disorders/ Anaemia/ anemia	9/ 35	2	Y	DNC	Y	RECOVERED/RESOLVED WITH SEQUELAE
1	E1301002	On Treatment	47/F/Asian	Metabolism and nutrition disorders/ Hyperuricaemia / hyperuricemia	12/ 59	1	N	DNC	N	RECOVERED/RESOLVED
2	E1301002	On Treatment	47/F/Asian	Investigations/ Blood creatinine increased/ Creatinine increased	16/ 6	1	Y	DNC	Y	RECOVERED/RESOLVED
3	E1301002	On Treatment	47/F/Asian	Investigations/ Blood urea increased/ Increased urea	16/ 6	1	Y	DNC	Y	RECOVERED/RESOLVED
4	E1301002	On Treatment	47/F/Asian	Investigations/ Blood alkaline phosphatase increased/ Alkaline phosphatase is elevated	21/	1	Y	DNC	Y	NOT RECOVERED/NOT RESOLVED
5	E1301002	On Treatment	47/F/Asian	Metabolism and nutrition disorders/ Hypoalbuminaemia/ hypoalbuminemia	21/ 23	1	N	DNC	N	RECOVERED/RESOLVED

Figure 4. Test data

name	label
col1	Subject identifier
col2	Treatment period
col3	Age/Sex/Race
col4	Body system/ Dictionary-derived term [a]/ Reported term for the adverse event
col5	Study day of start of adverse event [b]/ Duration of adverse event (days)
col6	CTCAE grade
col7	Serious event
col8	Action taken
col9	Causality
col10	Outcome

Figure 5. Column headers for the test data

	col1	col2	col3	col4	col5	col6	col7	col8	col9	col10	_total
min	47.108	8.500	50.448	22.660	8.778	8.500	13.222	19.892	13.222	38.222	230.552
5%	47.108	61.811	50.448	103.247	13.778	8.500	13.222	20.443	13.222	121.000	452.779
10%	47.108	61.811	50.448	105.399	18.778	11.000	13.222	27.113	13.222	121.000	469.102
15%	47.108	61.811	50.448	111.259	18.778	11.000	13.222	27.113	13.222	121.000	474.961
20%	47.108	61.811	50.448	111.259	18.778	11.000	13.222	27.113	13.222	121.000	474.961
25%	47.108	61.811	50.448	111.259	21.278	11.000	13.222	27.113	13.222	121.000	477.461
30%	47.108	61.811	50.448	111.259	21.278	11.000	13.222	27.113	13.222	121.000	477.461
35%	47.108	61.811	50.448	114.306	21.278	11.000	13.222	27.113	13.222	121.000	480.508
40%	47.108	61.811	50.448	117.962	23.778	11.000	13.222	27.113	13.222	121.000	486.664
45%	47.108	61.811	50.448	118.734	26.278	11.000	13.222	27.113	13.222	121.000	489.937
50%	47.108	61.811	50.448	137.890	26.278	11.000	13.222	27.113	13.222	121.000	509.092
55%	47.108	61.811	50.448	142.055	26.278	11.000	13.222	27.113	13.222	121.000	513.257
60%	47.108	61.811	53.778	149.594	26.278	11.000	13.222	27.113	13.222	121.000	524.126
65%	47.108	61.811	53.778	149.594	26.278	11.000	13.222	27.113	13.222	121.000	524.126

	col1	col2	col3	col4	col5	col6	col7	col8	col9	col10	_total
70%	47.108	61.811	53.778	149.594	26.278	11.000	13.222	27.113	13.222	167.104	570.229
75%	47.108	61.811	53.778	153.437	31.278	11.000	13.222	27.113	13.222	167.104	579.072
80%	47.108	61.811	53.778	162.094	31.278	11.000	13.222	27.113	13.222	167.104	587.729
85%	47.108	61.811	53.778	162.318	31.278	11.000	13.222	27.113	13.222	167.104	587.954
90%	47.108	61.811	53.778	168.979	31.278	11.000	13.222	27.113	13.222	167.104	594.614
95%	47.108	63.749	53.778	199.003	36.278	11.000	13.222	27.113	13.222	167.104	631.577
max	47.108	63.749	53.778	223.998	36.278	11.000	13.222	29.882	13.222	203.759	695.996
header	42.650	75.136	63.202	149.550	108.178	38.778	36.000	33.217	43.773	42.655	633.140
_max	47.108	75.136	63.202	223.998	108.178	38.778	36.000	33.217	43.773	203.759	873.149

Table 1. Summary of widths

Step 2: Determine Column Widths for the Final Report

Let's assume we're creating a PDF file on an 11-inch wide (792 points) and 8.5-inch high (612 points) paper. After setting both the left and right margins to 3/8 inches (27 points), we're left with 738 points of paper width to fill with columns. Here's a strategy to distribute this width among the columns:

Scenario A: If the total width of the "_max" row is less than the available page width, we can display all data points and column headers without any unexpected line breaks. The remaining width can be evenly distributed among the columns to fully utilize the page width.

Scenario B: If the total width of the "_max" row is more than the available page width, we can't display all data points and column headers without unexpected line breaks. A good approach is to use the first row with a total width less than the available page width, starting from the "max" row and going to the "min" row. If there's any remaining space, we can distribute it to the columns with wider headers (i.e., columns where the header row width is greater than the used percentile), proportionate to the width of the header.

Scenario C: If the total width of the "min" row is still more than the available page width, we can use the numbers in the "min" row and subtract a certain amount from each column to fit all the data on the page. This amount can be calculated as the excess width distributed among each column, proportionate to the width of the "min" row.

In all these scenarios, we can fix the width of some columns (i.e., using the "_max" row values directly) and calculate the widths for the remaining columns using the strategies outlined above.

Our test data matches the above Scenario B, if we fix the width of col1, col5, col6, col7, col8 and col9 and distribute the width of other columns. According to the rules, we set the column width based on the numbers in red in Table 3, the sum of the numbers is 693.237, the remaining width on the page will be allocated to the wider headers of col2 and col3, the final column width allocation is shown in Table 2.

col1	col2	col3	col4	col5	col6	col7	col8	col9	col10
47.108	86.123	74.229	149.594	108.178	38.778	36.000	33.217	43.773	121.000

Table 2. Width allocation for the test data

Step 3: Pagination

After setting the column width, we measure the height required for the titles and footnotes. This allows us to calculate the height available for displaying data. We then paginate based on the height required for each record. Finally, we output the result. The first page of the final output is shown in Figure 6.

Sponsor Name | Confidential | Page 1 of 102

Study Code: xxxxxx | Data Cut-off: yyyy-mm-dd

Listing 16.2.7.1

Adverse events (Safety analysis set)

Subject identifier	Treatment period	Age/Sex/Race	Body system/ Dictionary-derived term [a]/ Reported term for the adverse event	Study day of start of adverse event [b]/ Duration of adverse event (days)	CTCAE grade	Serious event	Action taken	Causality	Outcome
E1301002	On Treatment	47/F/Asian	Blood and lymphatic system disorders/ Anaemia/ anemia	9/ 35	2	Y	DNC	Y	RECOVERED/RESOLVED WITH SEQUELAE
E1301002	On Treatment	47/F/Asian	Metabolism and nutrition disorders/ Hyperuricaemia/ hyperuricemia	12/ 59	1	N	DNC	N	RECOVERED/RESOLVED
E1301002	On Treatment	47/F/Asian	Investigations/ Blood creatinine increased/ Creatinine increased	16/ 6	1	Y	DNC	Y	RECOVERED/RESOLVED
E1301002	On Treatment	47/F/Asian	Investigations/ Blood urea increased/ Increased urea	16/ 6	1	Y	DNC	Y	RECOVERED/RESOLVED
E1301002	On Treatment	47/F/Asian	Investigations/ Blood alkaline phosphatase increased/ Alkaline phosphatase is elevated	21/	1	Y	DNC	Y	NOT RECOVERED/NOT RESOLVED
E1301002	On Treatment	47/F/Asian	Metabolism and nutrition disorders/ Hypoalbuminaemia/ hypoalbuminemia	21/ 23	1	N	DNC	N	RECOVERED/RESOLVED

[a] MedDRA version 25.1

[b] Onset day is given in relation to first dose. A negative onset day indicates that the AE started before first dose.

Program: path\to\program\progname.sas | Executed: yyyy-mm-ddThh:mm:ss

Figure 6. First page of the final output

INTEGRATION OF SAS AND PYTHON

Based on the strategy in the previous section. We can develop a Python program to automatically generate PDF reports and invoke the Python program in SAS code. The Python program in this case can be used as an alternative to ODS and PROC REPORT. The following is a SAS macro, used for invoking the Python program and passing data to it.

```
%macro pyreport(
    items,
    file=
);
%let workpath = %sysfunc(pathname(work));
%put &workpath.;

%sysexec %str(python ..\main.py "&items." "&file." "&workpath.");
%mend;
```

The following is the SAS program which produced the report shown by Figure 6, we prepare data and pass to Python module to generate the report.

```
data ae100;
  length col1-col7 $200;
  set adam.adae;
```

```
    col1=strip(scan(usubjid, 2, '/'));
    col2=propcase(aphase);
    col3=strip(put(age,best.))||'/'||strip(sex)||'/'||strip(propcase(race))
;
    col4=strip(aebodsys)||'/<br/>'||strip(aedecod)||'/<br/>'||strip(aeterm)
;
    col5=put(aestdy,4.)||'/'||put(adurn,4.);
    col6=strip(put(atoxgrn,best.));
    col7=arel;
    col8=aacn;
    col9=arel;
    col10=aeout;

    label
    col1 = "Subject<br/>identifier"
    col2 = "Treatment period"
    col3 = "Age/Sex/Race"
    col4 = "Body system/<br/>Dictionary-derived term [a]/<br/>Reported term
for the adverse event"
    col5 = "Study day of<br/>start of adverse event [b]/<br/>Duration
of<br/>adverse event (days)"
    col6 = "CTCAE<br/>grade"
    col7 = "Serious<br/>event"
    col8 = "Action<br/>taken"
    col9 = "Causality"
    col10 = "Outcome"
    ;

run;

*Sort the outpt dataset;
proc sort data=ae100 out=body(keep=usubjid trtsdt astdt aedecod col:);
  by usubjid trtsdt astdt aedecod;
run;

*configuration data for body data;
proc sql noprint;
    create table config as
          select name, label
          from dictionary.columns
          where libname = 'WORK' and memname = 'BODY';
    ;
quit;
data config;
    set config;
    if _n_ = 1 then do;
          cols = 'col1 col2 col3 col4 col5 col6 col7 col8 col9 col10';
          fixcols = 'col1 col5 col6 col7 col8 col9';
    end;
run;

%tf_solution();

%pyreport(header title body:config footnote footer,
          file=.\ae100.pdf
)*;
```

CONCLUSION

This article introduces a method for automating column width and pagination using Python ReportLab library. The Python-based approach for generating reports from SAS datasets provides a streamlined and efficient alternative to SAS PROC REPORT. It offers automatic column width adjustment and pagination. With a simple SAS macro, it can be easily integrated in existing TLF creation processes, making report generation more user-friendly. This approach can be further enhanced and customized according to the specific needs of the users. For example, instead of producing PDF file directly from Python side, we can also use the calculated width and pagination settings as the input of PROC REPORT, though handling of control words like
 will be needed.

REFERENCES

Website ReportLab Docs available at https://docs.reportlab.com/.

ACKNOWLEDGMENTS

The author would like to thank Yong Cao and Lu Zhang for their great advice.

CONTACT INFORMATION

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

Tao Zhang
Phastar Inc.
tao.zhang@phastar.com
www.phastar.com

APPENDIX

```
from reportlab.pdfgen.canvas import Canvas
from reportlab.platypus import Paragraph, Table, TableStyle
from reportlab.lib.styles import ParagraphStyle
from reportlab.lib.units import inch
from reportlab.lib import colors
from reportlab.pdfbase.pdfmetrics import registerFont, registerFontFamily
from reportlab.pdfbase.ttfonts import TTFont

# register font
registerFont(TTFont('Times New Roman', 'times.ttf'))
registerFont(TTFont('Times New Roman Bold', 'timesbd.ttf'))
registerFont(TTFont('Times New Roman Italic', 'timesi.ttf'))
registerFont(TTFont('Times New Roman Bold Italic', 'timesbi.ttf'))
registerFontFamily('Times New Roman', normal='Times New Roman', bold='Times
New Roman Bold', italic='Times New Roman Italic', boldItalic='Times New
Roman Bold Italic')

# paragraph style
pstyle = ParagraphStyle('test')
pstyle.fontName = 'Times New Roman'
pstyle.fontSize = 10
pstyle.leading = 12

# table style
padding = 3
tstyle = TableStyle([
```

```
    ('GRID', (0,0), (-1,-1), 0.5, colors.gray),
    ('VALIGN', (0,0), (-1,-1), 'TOP'),
    ('LEFTPADDING', (0,0), (-1,-1), padding),
    ('RIGHTPADDING', (0,0), (-1,-1), padding),
    ('TOPPADDING', (0,0), (-1,-1), padding),
    ('BOTTOMPADDING', (0,0), (-1,-1), padding),
])

# create a paragraph without line breaks
para1 = Paragraph('The brown fox jumps over the lazy dog.', style=pstyle)
height1 = para1.wrap(7200, 7200)[1]
widths1 = para1.getActualLineWidths0()
print('Height of the paragraph without line breaks: ', height1)
print('Width(s) of line(s) in the paragraph without line breaks: ',
widths1)

# create a paragraph with line breaks
para2 = Paragraph('The brown fox jumps over the lazy dog.', style=pstyle)
height2 = para2.wrap(100, 7200)[1]
widths2 = para2.getActualLineWidths0()
print('Height of the paragraph with line breaks: ', height2)
print('Width(s) of line(s) in the paragraph with line breaks: ', widths2)

# create two table cells with the measured widths
table1 = Table([[para1]], colWidths=max(widths1)+2*padding, style=tstyle)
t1_space = table1.wrap(7200, 7200)
print('The space of table1: ', t1_space)

table2 = Table([[para2]], colWidths=max(widths2)+2*padding, style=tstyle)
t2_space = table2.wrap(7200, 7200)
print('The space of table2: ', t2_space)

# create a paragraph with intra-paragraph markup
para3 = Paragraph('The <b>brown</b> fox<br/>jumps over the lazy dog.',
style=pstyle)
height3 = para3.wrap(7200, 7200)[1]
widths3 = para3.getActualLineWidths0()
print('Height of the paragraph with intra-paragraph markup: ', height3)
print('Width(s) of line(s) in the paragraph with intra-paragraph markup: ',
widths3)

table3 = Table([[para3]], colWidths=max(widths3)+2*padding, style=tstyle)
t3_space = table3.wrap(7200, 7200)
print('The space of table3: ', t3_space)


# create a pdf file and draw the tables
canvas = Canvas(r'.\para_space.pdf', pagesize=(11*inch, 8.5*inch))
table1.drawOn(canvas, 100, 100)
table2.drawOn(canvas, 300, 100)
table3.drawOn(canvas, 300, 300)
canvas.save()
```