Paper PP07

Stuck in a Rut?

Prakash Mistry, PHASTAR, London, UK
Jordan Bristow, PHASTAR, London, UK

ABSTRACT

In standard programming practices, one can occasionally become stuck over seemingly trivial issues. This can cause a slowdown in programming efficiency and cause bottlenecks across a study that must be overcome before progressing forward. The tendency for some programmers to favor a particular logical approach can exacerbate this issue. This paper will provide a wide selection of possible solutions to common problems encountered while programming in a general environment and aim to inspire new ways of thinking, thus avoiding becoming 'Stuck in a rut'.

INTRODUCTION

When programming code, occasionally we can encounter challenges that we will have not faced before which we need to overcome in order to progress forward within the study/project. The aim would be to try and solve these issues as quickly as possible, and in the most efficient way so that they don't slowdown programming. However, sometimes these challenges will cause us to become 'Stuck in a rut', either by favoring an approach we have been using previously or by causing us to become stuck altogether. It is important to remember that the best approach to solving these problems is to talk to other programmers and get their advice, which is one of the golden rules employed by PHASTAR. They may have encountered similar challenges and will have potential solutions to help with solving these issues. It is also important to keep an open mind as there may be many ways in overcoming these hurdles. Some of these solutions may seems obvious to some people but will provide a completely new logical paradigm to others.

The poster will provide a selection of different coding techniques which will present logical aspects on how to approach coding challenges. Examples will be provided on the instances where these techniques can be implemented with the hopes of providing the reader with inspiration on how to clear other programming hurdles that may have seemed insurmountable previously.

COUNT FOR NON-EXISTENT VALUES

Occasionally a row is needed for a certain table, figure or listing (TFL) that includes a count (N) value, but the value searched for does not exist in the data. This can cause issues when trying to create the output. There are a multitude of ways around this issue, but one novel method presented here takes advantage of the way SAS processes data, and side-steps around the issue of non-existence by pre-processing based on current data and allowing for correct processing if present at a later date, thus ensuring a dynamic approach. The essential idea is to work based on what you do have and to use this to your advantage to create a logic process within the SAS framework to process what you don't.

Below we demonstrate a simplistic example of this method using a standard SAS dataset, which has been pre-filtered to only include Male subjects for demonstration purposes. In the final output we want a count of both Male and Female; this can be achieved through a logic process. We create a new variable called "count" and set this to be 1 if the searched for value is true and 0 otherwise. We then output a dataset where the count variable is summated with desired labelling. If the value searched for does not exist in the dataset then we now have a count value return of 0, a numeric result that will also update if the data updates further.

```
data test;
  set sashelp.class;
  where sex = "M";
run;

%macro count_miss(dsin=,    /*Specify dataset to be input*/
                  dsout=,   /*Specify dataset of count to be output*/
                  var=,     /*Specify variable of interest*/
                  value=    /*Specify value to be counted*/
                  );

data dsin2;
  set &dsin.;
  if &var. = "&value." then count = 1;
  else count=0;
run;

proc sql;
  create table &dsout. as
  select sum(count) as count_&var.&value.
  from dsin2;
quit;

%mend count_miss;

%count_miss(dsin=test, dsout=count_male, var=sex, value=M);
```

	NAME	SEX	AGE	HEIGHT	WEIGHT	COUNT
1	Alfred	M	14	69	112.5	1
2	Henry	M	14	63.5	102.5	1
3	James	M	12	57.3	83	1
4	Jeffrey	M	13	62.5	84	1
5	John	M	12	59	99.5	1
6	Philip	M	16	72	150	1
7	Robert	M	12	64.8	128	1
8	Ronald	M	15	67	133	1
9	Thomas	M	11	57.5	85	1
10	William	M	15	66.5	112	1

	COUNT_SEXM
1	10

```
%count_miss(dsin=test, dsout=count_female, var=sex, value=F);
```

	NAME	SEX	AGE	HEIGHT	WEIGHT	COUNT
1	Alfred	M	14	69	112.5	0
2	Henry	M	14	63.5	102.5	0
3	James	M	12	57.3	83	0
4	Jeffrey	M	13	62.5	84	0
5	John	M	12	59	99.5	0
6	Philip	M	16	72	150	0
7	Robert	M	12	64.8	128	0
8	Ronald	M	15	67	133	0
9	Thomas	M	11	57.5	85	0
10	William	M	15	66.5	112	0

	COUNT_SEXF
1	0

This code could be further developed for more complex cases where multiple values could be searched for and counted, within a single call of the macro.

REMOVING UNWANTED OBSERVATIONS

Usually when removing data, we would use datastep filtering. This approach can be made considerably more difficult depending on the complexity of the filters we need to implement and the state of the data that we have. Sometimes the simpler way of filtering would be to create a dataset containing the data we want to remove from the parent dataset, then merging this dataset, using the in= option, with the parent dataset and using the condition “If a and not b”.

This can be substantially easier and quicker to work with than processing using multiple datasteps and further filtering applied with sorting.

PRESERVING ORDER

Having constructed a dataset exactly the way we desire, we can come across issues with duplicate results that have come from unforeseen processes within the code. This can create difficulties in finding the root cause and, depending on the amount of data being used, can become very time-consuming to resolve. A method of removing these issues can be found by applying a new variable (e.g. ord) and setting it to be equal to _n_. This essentially fixes the current order of the dataset.

A nodup can then be applied to the data and re-sorted according to the values contained within the data. One further sort by ord will then return the original desired sort order with any previously troublesome duplicates also removed.

```
data test;
infile datalines delimiter='/';
input value $;
datalines;
D
N
N
A
;
run;
```

	VALUE
1	D
2	N
3	N
4	A

```
data test2;
  set test;
  ord= _n_;
run;
```

	VALUE	ORD
1	D	1
2	N	2
3	N	3
4	A	4

```
proc sort data = test2 out = test3 nodupkey;
  by value;
run;
```

	VALUE	ORD
1	A	4
2	D	1
3	N	2

```
proc sort data = test3 out = test4;
  by ord;
run;
```

	VALUE	ORD
1	D	1
2	N	2
3	A	4

Further applications of this can be found when trying preserve the order when we have numeric data present in character format, for example percentages. If in the example above, we also had percentages for each value and the dataset has already been sorted by decreasing frequency and we tried the nodup sort we would be left with a completely different order to what we require. If we tried sorting by the character percentage, we could get the data back in the original order but sometimes we would get a new order entirely.

	VALUE	PERCENT
1	D	10%
2	N	7%
3	N	7%
4	A	1%

```
proc sort data = test2 out = test3 nodupkey;
  by value;
run;
```

	VALUE	PERCENT
1	A	1%
2	D	10%
3	N	7%

```
proc sort data = test3 out = test5;
  by descending percent;
run;
```

	VALUE	PERCENT
1	N	7%
2	D	10%
3	A	1%

LATE-IN-THE-DAY ADDITIONS

Situations can occur when programming SDTMs and ADaMs, of a variable that will need to be present within the dataset, but the predecessor data relating to this variable does not exist, either because the dataset in question is not present (e.g. SUPP dataset does not exist) or because the predecessor variable that will feed into this new variable is not yet available within the dataset. This predecessor data could either consist of a dataset or just a variable. We could set the variable to missing at the start to get around this issue and just update the code when the data relating to these variables becomes present. However, this will require constantly checking the data transfers to see if the data is available and then updating the program to account for this.

By using the dictionary datasets or SAS help views, we can program the code for these variables right at the start to account for these situations. We then only need program it once, and not have to constantly check the data as the program is doing this for you.

These Dictionary/SAS help datasets contain the SAS session metadata. This information is created at initialization and updated automatically as changes are made to tables. The dictionary datasets can only be accessed by PROC SQL.

The datasets that we can use are:

- DICTIONARY.MEMBERS (or SASHELP.VMEMBER) – Contains library information
- DICTIONARY.TABLES (or SASHELP.VTABLE) – Contains dataset information
- DICTIONARY.COLUMNS (or SASHELP.VCOLUMN) – Contains variables information
- DICTIONARY.CATALOGS (or SASHELP.VCATALG) – Contains catalogue information
- DICTIONARY.OPTIONS (or SASHELP.VOPTION) – Contains current settings for SAS system options
- DICTIONARY.MACROS (or SASHELP.VMACRO) – Contains macro variable information
- DICTIONARY.TITLES (or SASHELP.VTITLE) – Contains Title Information

If the variable that needs to be programmed stems for a predecessor dataset then using the tables dictionary dataset is the best option for use. If it is a predecessor variable, then the columns dataset should be used. Filtering when calling these two datasets should be based on the libname variable, which contains the library that the dataset or variable exists in, and the memname variable, which contains the name of the dataset.

```
/*Checking the dictionary datasets to see if SUPPAE exists in the SDTM folder*/
proc sql;
  create table exist as
  select *
  from dictionary.tables
  where libname = "SDTM" and memname = "SUPPAE";

  select count(*)
  into: nobs
  from exist;
quit;

/*If the SUPPAE dataset does exist then do the processing for variable which comes
from the SUPPAE dataset*/
%if &nobs ne 0 %then %do;

data suppae;
  set sdtm.suppae;

  aeseq = input(idvarval, best.);

run

proc sort data = suppae;
  by usubjid aeseq;
run;

proc transpose data = suppae out = suppae2;
  by usubjid aeseq;
  var qval;
  id qnam;
  id label qlabel;
run;

data ae2;
  merge ae
        suppae2;
  by usubjid aeseq;
run;

%end;

/*If the SUPPAE dataset does not exist then set the variable as missing*/
%else %do;

data ae2;
  set ae;

/*Example variable that is not present in SUPPAE that needs to be added to the ADaM
dataset which will set as missing*/
  aetrtem = "";

run;

%end;
```

CALL EXECUTE

Generally, macros are executed in open code as simple calls. But, by using the CALL EXECUTE function we can execute macros within a data step. The benefits of using CALL EXECUTE are that we can execute macros conditionally as well as use existing values within the dataset as values in the macro call. Below are examples where call execute can be used to good effect.

EXAMPLE 1

The call execute function can be used when creating outputs that are dependent on different populations. Normally, you would create the code needed for this output (i.e. code to create frequency counts) and put this into a macro where the population being used is a macro variable. You would then call this macro multiple times for all the populations you want to use to create the output that is needed.
By using the call execute function along with the dictionary datasets that we have already discussed we can run this macro only once.

```
/*Macro that counts number of subjects which have population flag as "Y"*/
%macro pop (pop=);

proc freq data = adam.adsl (where=(&pop.="Y"));
  tables trta / out = &pop._1;
run;

proc transpose data = &pop._1 out = &pop._2;
  id trta;
  var count;
run;

%mend pop;



/*Using the dictionary datasets to create a dataset that contains all the
population flags in ADSL*/
proc sql;
  create table flags as
  select name
  from dictionary.columns
  where libname = "ADAM" and memname = "ADSL"
     and index(name, "FL");
quit;
```

	NAME
1	FASFL
2	RANDFL
3	SAFFL
4	ITTFL
5	PPROTFL

```
/*Using call execute to call the pop macro using the flags dataset*/
data _null_;
  set flags;
  CALL EXECUTE ("%pop(pop="||strip(name)");");
run;
```

EXAMPLE 2

This function is also a very good tool to use where there is conditional processing attached to the creation of an output or dataset. As the call execute function is part of a data step, we can use some of filters that would normally be implemented, such as an 'if' statement or a 'where' clause. An example of this is when a demographic table needs to be created and needs to be repeated on a subset of subjects that have specific medical conditions (e.g. History of Diabetes, Heart Condition,…), with the condition attached that we can only print the tables if there is a subject present with that condition.

```
/*Using the dictionary dataset to determine the number of medical history flags in
the ADSL dataset*/
proc sql noprint;
  create table medhist as
  select name
  from dictionary.columns
  where libname = "ADAM" and memname = "ADSL"
  and index(name, "MH");

  select max(input(substr(name, 5), best.))
  into: max
  from medhist;
quit;
```

	NAME
1	MH01FL
2	MH02FL
3	MH03FL
4	MH04FL

```
data medhist1;
  set medhist;

  ord = input(substr(name, 5), best.);

run;

/*Determine whether there are any subjects flagged by each of the medical history
flags*/
%do i= 1 %to &max.;

proc sql noprint;
  create table mhfl as
  select distinct mhfl&i.
  from adam.adsl
  where mhfl&i.="Y";

  select count(*)
  into: nobs
  from mhfl;
quit;




/*If there is at least one subject then flag the medical history flags*/
```

```
%if &nobs ne 0 %then %do;

data medhist1;
  set medhist1;

  if name = "MHFL&i." then occur = "Y";

run;

%end;
%end;
```

	NAME	OCCUR
1	MH01FL	Y
2	MH02FL	
3	MH03FL	Y
4	MH04FL	Y

```
/*Using call execute with conditional processing to run the demographic macro only
on the medical history flags which has subjects flagged*/
data _null_;
  set medhist1 (where=(occur="Y"));
  call execute ('%nrstr(%demographics(flag='||strip(name)||' ,ord='||strip(put(ord,
2.0))||'));');
run;
```

CONCLUSION

In conclusion, the best practice to take when programming various pieces of code is to always remain open to the possibility that there might be many different methods for coding the same results. We should take some time to look through each of these methods and pick the one that is best suited to the task at hand; ensuring that we are using the most efficient methods and that we don't become 'Stuck in a rut'.

CONTACT INFORMATION

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

Prakash Mistry
PHASTAR
Unit 2, 2A Bollo Lane
London, W4 5LE
Email: prakash.mistry@phastar.com
Web: https://phastar.com/

Jordan Bristow
PHASTAR
Unit 2, 2A Bollo Lane
London, W4 5LE
Email: jordan.bristow@phastar.com
Web: https://phastar.com/