

From Novice to Intermediate in (Approximately) Sixty Minutes: I. Functions, Programming, and Processing – the Power to Choose AnnMaria De Mars, University of Southern California, Los Angeles, CA

ABSTRACT

This presentation assumes that you are either a) a novice programmer or b) someone who mentors novice programmers. The differences that move someone out of the novice category boil down to design choices. We will dissect programs from novice programmers and modify these to illustrate more effective and efficient choices.

Experienced programmers have a wider variety of function options in their toolkit. For example, a researcher wants to categorize people who have ANY positive response to five questions on raising taxes, “Would you vote to raise taxes if ... the state budget isn’t balanced?” “Would you raise taxes if ... the option was to cut social services?” and so on.

There are a variety of solutions, but a single statement that does exactly what the researcher desires is :

```
Taxes = MAX(of q1 - q5) ;
```

Experienced programmers have greater knowledge about how SAS processes data and a wider selection of programming options. Examples discussed include issues in merging datasets, handling date data, and efficient processing with arrays and do-loops.

Perhaps the most important sign of a knowledgeable programmer is the recognition of the need to keep learning and developing new options. The final example of intermediate programming, use of %include, is a first step toward more advanced programming techniques with SAS macros.

[Note: No one is allowed to feel bad for having made any of the mistakes used in the examples in this session. Everyone will admit to having made the exact same errors at one time, except for a very few people. Those people are probably lying. Try to avoid having coffee with them. They are a bad influence.]

INTRODUCTION

When you fill out that conference form and it asks your level of SAS[®] experience, what do you put? What exactly ARE the points that distinguish the novice between not-so-novice? In short, what are the “newbie” mistakes that once you stop making them (or make them very seldom) you are no longer revealed as a novice? What are the features that, when you see them, consciously or not, you think to yourself that ‘This person knows what he or she is doing’ ?

Before we proceed any further, two disclaimers are needed.

Disclaimer #1: No one is allowed to feel bad for having made these mistakes. Everyone in this room will admit to having made the exact same errors at one time, except for a very few people. Those very few people are probably lying. Try to avoid having coffee with them. They are a bad influence.

Disclaimer #2: Anyone who claims to know “ALL” of SAS, be it programming, all uses of all SAS products or all of statistics, is very likely clinically insane. The functions, procedures and logic used here are not the only way to solve the problems presented. In some cases, these are not the most advanced way. That is the often the exact point.

This presentation assumes that you are either a) a novice programmer or b) someone whose career involves developing and mentoring novice programmers. In addressing the differences that move someone out of the novice category it all boils down to design choices. Throughout the presentation, we will dissect programs from novice programmers and modify these to illustrate more effective and efficient choices. More experienced programmers can choose among methods because they know more than one way to accomplish a task. This knowledge includes functions, formats and programming statements – knowing the right words to type. It also includes understanding how SAS processes data - how data are stored internally and how statements are executed by SAS. While experienced programmers may differ wildly in specific preferences, expertise and strengths one common characteristic is a recognition of how much more they don't know and of the need to continually expand and refine their knowledge.

Knowing the statements and functions of programming language doesn't make you a good programmer any more than knowing the English language makes you Hemingway. Being a great writer requires an extensive vocabulary, but more than that, it requires you to know how to put those words together.

To begin to move away from being a novice programmer, let's consider both the “vocabulary” you need to know and some different ways to put that knowledge to use.

MOVING FROM NOVICE TO INTERMEDIATE 1: KNOW A VARIETY OF FUNCTIONS

Neil Howard (2001, p.1) gives a succinct definition,

“A function returns a value from a computation or system manipulation that requires zero or more arguments.”

For example, a researcher wants to categorize people who have any positive response to five questions on raising taxes, “Would you vote to raise taxes if ... the state budget isn’t balanced?” “Would you raise taxes if ... the option was to cut social services?” and so on.

A novice response would be:

```
If q1 = 1 then taxes = 1 ;
  Else If q2 = 1 then taxes = 1 ;
  Else If q3 = 1 then taxes = 1 ;
  Else If q4 = 1 then taxes = 1 ;
  Else If q5 = 1 then taxes = 1 ;
  Else taxes = 0 ;
```

A better choice would be to use the SUM function:

```
If sum(of q1 - q5) > 0 then taxes = 1 ;
  Else if sum(of q1 - q5) = 0 then taxes = 0 ;
```

There are a variety of ways, some better some worse. A single statement that does exactly what we want is :

```
Taxes = max(of q1 - q5) ;
```

One reason the last two solutions are better than the first is that they account for missing values. In the first case, a person who did not answer any of the questions would be counted as a zero as in “No, they would not vote to raise taxes under any circumstances. In the second example, when all questions are left blank, that is, missing, neither the IF condition nor the ELSE IF condition will be true, so nothing will happen with the new variable taxes, and it will end up as a missing value. In the last example, when all of the arguments to the MAX function are missing, the result is a missing value.

Here’s another example of choosing among functions:

```
AvgQtr = (SalesJan + SalesFeb + SalesMar)/ 3 ;
```

Reveals a novice – *maybe*. Why would you calculate a mean when SAS already has a MEAN function?

```
AvgQtr = sum(SalesJan,SalesFeb, SalesMar)/3 ;
```

May actually be worse. The SUM function only returns a missing value if the data for all of the arguments are missing. So, if you have \$10,000 in sales for March, \$8,000 in sales for February and your sales data for January are missing, the above statement returns an average monthly sales value for the quarter of \$6,000. This is only correct if January sales were zero, which, unless you just opened the store is probably not the case and almost certainly not useful.

```
AvgQtr = mean(SalesJan,SalesFeb, SalesMar) ;
```

... may be what you want. If your January sales were missing, this would give you an average of \$9,000, which is the average for the months on which you have data. If that’s what you want, stop here. If you only want the average for records for which you have complete data, then you would be fine with the first option, without any functions. The answer to why you would calculate a mean when SAS already has a mean function might be that you want the variable to equal missing unless you have all of the data.

Let’s say you don’t want an all or none approach. You may have twenty questions about an issue and you don’t want to throw out the whole questionnaire every time someone skips a question. On the other hand, you don’t want to include people who only answered one of the twenty questions and treat them the same as those who had answered every item. So, in consultation with your research team, you select a cut-off, say five items out of twenty, and you use this statement with the NMISS function:

```
If nmiss(of q1 - q20) < 5 then AvgRank = MEAN(of q1 - q20) ;
```

If you are missing no more than 20% of the data (four questions out of twenty), this statement will return the mean ranking, otherwise the value of AvgRank is missing.

AVOIDING COMMON ERRORS WITH FUNCTIONS: IT DEPENDS ON THE MEANING OF 'OF'

Experienced programmers avoid common pitfalls because, well, because they have fallen into those more than once. One common feature of many mathematical functions is the meaning of 'OF', 'TO' and '-'. You might think that all of the following statements would produce the same results.

```
sum(of q1 - q4) ;
sum(q1 - q4) ;
sum(q1, q2 to q4) ;
sum(q1, of q2 - q4) ;
```

You thought wrong. In fact, if the values for the four variables are 1, 2, 3, 4 then the first example gives the correct answer: 10. The second example gives the answer -3, which is the value of q4 subtracted from q1. The third statement returns the value 5, which is the sum of q1, q2 and q4. You'll also have a new variable in your dataset named "To" which is uninitialized. The fourth example, does give you the correct answer of 10. In this case it is pretty silly to code it that way but, as you will see later in this paper, that syntax can come in handy.

Whenever you are using mathematical function, be it MAX, MEAN, SUM, or others, always be aware of what the meaning of OF is.

MOVING FROM NOVICE TO INTERMEDIATE 2: KNOW OPTIONS FOR HANDLING DATES

Unless you lead a charmed life, you will find yourself using a data set that includes dates. Understanding how SAS stores dates, the differences between formats and informat, and some simple, ubiquitous date functions like TODAY() will make your programming life infinitely easier. Informat is how data are read. Format is how data are written. There is also a third aspect of dates, which is how data are stored. Maybe you've never thought about this. Take this program, that was run on August 5, for example,

```
Data in.example ;
  Infile readstuff ;
  Input @1 birthday mmdyy8. @10 Age 11-12 name $ 13-34 ;
  Sasday = bod ;
  Age2 = (today() - bod) /365.25 ;
  Compage = round(age,1) ;
  Compage2 = int(age) ;
  Format birthday monyy7. ;
```

So, your input dataset looks like this:

```
08/12/08 1 Eva Maria Ortiz
01/04/98 12 Julia De Mars
07/17/92 18 Bob
```

When you run a PRINT procedure, it looks like this:

Bod	Age	Name	sasday	age2	compage	compage2
AUG2008	2	Eva Maria Ortiz	17756	1.9795	2	1
JAN1998	12	Julia De Mars	13883	12.5832	13	12
JUL1992	18	Bob	11886	18.0507	18	18

Now, the date was read in as 8/12/08 and written out as AUG2008. Here is your first pop quiz question: Is the date stored as:

- a) AUG2008
- b) 08/12/08
- c) something else

If you answered c you win – well, nothing - but it was the correct answer anyway.

Let's break this down step by step. You probably know that SAS stores dates as a NUMBER. This is the number of days between the date and January 1, 1960. If you didn't before, you do now. Just nod sagely as if you knew it all along. When you use the informat mmddyy8., SAS is instructed to read in the information and set it to a date value which is a number of days. When you execute the statement

```
Sasday = bod ;
```

It sets the new variable, Sasday, to be equal to the value of bod which to SAS is actually 17756 for the first person. Something else you'll note here, an assignment statement assigns the value to the new variable, but not the format.

Next we have three of the most incredibly useful functions all in a row.

TODAY() - this function has no argument . It returns the value of today's date as a SAS date value. I use this all the time.

Want to select out only records of sales in the last 30 days? Use the statement:

```
if sale_date = today() - 31 ;
```

Want to know how long someone has survived since an event? Calculate with this statement:

```
Survival = today() - diagnosedate ;
```

Of course, it's a useful function in cases like the program above, (remember the program above?) when you want to know the age.

```
Age2 = (today() - bod) / 365.25 ;
```

The statement above will give you the age in years. Of course, as you can see from the output of the PROC PRINT, it gives you a figure like 1.9795. That probably isn't what you want. Also, don't forget the parentheses around the numerator because if your code reads

```
Age2 = today() - bod/365.25 ;
```

You are going to get results telling you that your patients are something like 18,372.75 years old. No wonder they're not feeling well. By the way, those results are most likely incorrect, but no error will show up in your log. A common false assumption of very new programmers is that just because there is no error message in the log, the program ran without errors and the output is correct. This is yet another one of life's assumptions that we painfully lose early on in life. (Other assumptions we have when are young include: that those people around when you win a gold medal are your real friends, that he really will call you tomorrow, that it's her real phone number and that it's not you, it's me. All of those are false, too. Sorry.)

To get age in years as an integer, which is how adults usually express it, you could use the ROUND function.

```
Compage = round(age,1) ;
```

[Actually, Round(age,1) and round(age) will give you the exact same result because the default if a second argument is not specified is to round the nearest integer, but I like to spell things out, bearing Eagleson's law in mind, "Any code of your own that you haven't looked at for six or more months, might as well have been written by someone else."]

Rounding may be what you want to do if you're dealing with prices or some other metric, however, people generally want their actual age and not the age rounded, so in this case you probably want a different function.

So, this gives us our final function in this example, the INT function. This function returns the integer part of a number.

```
Compage2 = INT(age) ;
```

TAKE-AWAY POINTS ON FUNCTIONS

My point, and contrary to appearances, I do have one, is that it's not knowing a particular function, such as MAX that distinguishes a novice from a not-so-newbie, but rather being familiar with a range of SAS functions. Here is a good rule of thumb, given that the first "S" in SAS originally stood for "statistical" it is a pretty good bet that any statistic you can imagine already has a pre-written SAS function. If it is a common statistical or mathematical function, like finding the mode or rounding to the nearest integer, it is a certainty that a function already exists.

We've covered some mathematical and date functions while I've deliberately left out character functions. This is not because these aren't important. On the contrary, there is an entire session devoted just to character functions, so I omitted that section here to avoid redundancy.

The other point I really want to stress is that simply knowing the names of functions, being able to spit back a list of the 190 SAS functions, or however many it is up to now is not what makes you a good programmer. It is the application of those. As I said in the introduction, it is the design choices you make that start to show that you are no longer a novice. Rob Meekings commented on my blog once that he thought a key feature of a good programmer is being able to explain and defend the design choices he or she made. I agree.

For more information on functions, check the references at the end of this paper. I particularly liked Neil Howard's because he asked the important question, "Why is there a MEAN function and not a NICE function?"

MOVING FROM NOVICE TO INTERMEDIATE 3: UNDERSTAND SAS PROCESSING

Experienced programmers have fewer errors because they know how SAS processes data. For example, when two or more datasets that have variables of the same name are merged in a Data step, the value for the last data set is retained. Inexplicably, SAS will make a note in your log about multiple records with the same ID variable but will not make a note about overlapping variable names.

Let's take a common example, you need to merge two datasets that have data on the same subjects. Each dataset has variables named id, age, gender, site and testscore. Your data set named pretest has pretest scores and your data set named posttest has posttest scores. All very simple and logical.

You merge the two datasets together.
What have you done?

```
Proc sort data = libref.pretest ;
  By id ;
Proc sort data = libref.posttest ;
  By id ;
Data alltests ;
  Merge libref,pretest libref,posttest ;
  By id ;
```

You've just written over the testscore from the pretest dataset. In fact, what you have in this case is just the posttest dataset with a new name.

One option to fix this problem is to rename variables in a data step. A better option is to rename the variables in the proc sort step. [There are at least two other options.]

Try this:

```
Proc sort data = libref,pretest out = pre (rename = (testscore = pretest)) ;
  By id ;
Proc sort data = libref,posttest out= post (rename = (testscore = posttest));
  By id ;
Data libref.alltests ;
  Merge pre post ;
  By id ;
```

[For even more uses than sorting, see Kelsey Bassett (2006) SUGI 31 paper on "The SORT Procedure: Beyond the Basics".]

Another point above I would highly recommend is when you are doing anything other than simply sorting a dataset to output it to a temporary dataset. It's just a habit. If you are renaming variables, it's no big deal, but if you are dropping

variables or selecting them using the WHERE clause and there is an error (hey, it could happen), you have just written over the old dataset with another dataset of the same name that has only half the records or variables. If you are really, really, absolutely, sure that you do not now and never will want those other variables or records, then that might be okay. However, I have found the distance between now and never to be a very long time during which people are likely to say, "Remember that analysis you did for the people over 65? Well, we would like you to re-run it for people aged 40-64."

An even greater concern though is how SAS processes data in the event of errors and what SAS interprets as an error. If SAS encounters what it interprets to be an error, it will not replace your dataset. If, for example, I write the following, to pull out those who are over 18 years old (which is $18 * 365.25 = 6574.5$)

```
Proc sort data = libref.pretest ;
  By age3 ;
  Where age3 > 6574 ;
```

Now, if you have mistyped the variable name and there really is no variable age3, then your log will give you this helpful message

```
ERROR: variable age3 is not on file libref.pretest
```

Your file will not be replaced and you are free to swear, fix the code to be the correct variable name and go merrily on. However, let's say that you type the variable name correctly as age2, but it is not, as you have expected, age in days. Rather, it is age in years, as illustrated above. Then SAS goes ahead and executes the statements and tells you in a note on your SAS log:

```
NOTE: Input dataset is empty. There were 0 observations read from dataset libref.pretest where age2 > 6574.
```

```
NOTE: The dataset libref.pretest has 0 observations and seven variables.
```

I'm not sure what you are going to do after you swear, but I am sure it will be annoyingly inconvenient.

MOVING FROM NOVICE TO INTERMEDIATE 4: WRITE EFFICIENT, READABLE CODE

Mystery novels should be figured out; code should be read. I forget who said that but it is a profound statement. If your code takes a Ph.D. and assistance from Sherlock Holmes to decipher, it does not prove your brilliance. What it does prove we won't discuss here, but it isn't good.

Writing code that is easily read and easily maintained is just one of the reasons for using ARRAY statements and do-loops. This bit of programming logic will become one of the most versatile pieces in your SAS arsenal. Here is just one common example; you have twenty questions that have been coded from 1 = strongly disagree to 5 = strongly agree. However, ten of those are positively-worded

Question 2: "I think Squeezex Waffles are the greatest thing since sliced bread."

The other ten are negatively worded,

Question 1: "I would rather eat maple syrup poured over newspaper than Squeezex Waffles."

One really inefficient way to fix this would be with 50 IF statements or 10 IF statements and 40 ELSE statements. Or ... you could do this with four statements. This is shown below, with the addition of a comment and label statement that also makes your code, and, eventually, your output, more readable.

```
/* Recodes negatively-worded items */

ARRAY fixthis{10} q1 q4 q5- q7 q9 q17-q20 ;
Array fixed{10} rev1 = rev10 ;
DO I = 1 TO 10 ;
  fixed{i} = 6 - fixthis{i} ;
END ;
TOTAL = sum(of rev1 - rev10,q2,q3,q8,of q10 - q16) ;

LABEL
```

```
rev1 = "Prefers Squeezex to Eating Newspaper" ;
```

Let's break each of these statements down:

```
/* Comments */
```

Nothing between the beginning `/*` and the ending `*/` is executed by SAS. Here is where you explain your code.

The format of an ARRAY statement is :

```
ARRAY arrayname{arraydimension} variables-in-array ;
```

The array name can be any valid SAS name. Don't make it the same as the name of a function, SAS statement or any variable in the dataset, or you will be sorry. Note the brackets are the "curly" ones. The array dimension is the number of variables in the array. The variables must be either all numeric or all character. You cannot mix variable types in an array. Two final points:

1. If you define a character array, your ARRAY statement must have a \$, e.g.

```
ARRAY stuff {4} $ s1 - s4 ;
```

2. If you don't know the number of variables in an array, for example, you have an array of all numeric variables and you don't feel like going through the dataset and counting them, you can use a *

```
ARRAY numvars {*} _numeric_ ;
```

This will create an array that includes all of the numeric variables in your dataset and the dimension of the array will be set as however many numeric variables happen to be in your dataset. As with every topic discussed here, there are a great variety more options and uses for arrays. One paper I recommend for a more in-depth introduction is *Off and Running with Arrays* by Keelan (2006).

Next, the simplest DO statement, takes the form

```
DO indexvariable = startvalue TO endvalue ;
```

The index variable can be given any valid SAS name. The commands between the DO statement and END statement will be executed beginning with the start value of the index variable and ending at the end value. So, as is commonly the case, when the start value = 1 and the end value = the number of items in the array, the statements will be performed for every variable in the array.

PROBLEMS WITH DO LOOPS YOU SHOULD AVOID

If you don't know how many variables are in an array, for example, when you create an array with all numeric variables and you don't specify the dimension of the array? Use the DIM function, as in:

```
do i = 1 TO dim(numvars) ;
```

This is interpreted as "Do the following statements beginning at the number 1 and ending with however many variables the array happens to have."

Every DO must have a matching END statement. This is one of those errors, like forgetting a semi-colon or forgetting the closing quote, that catches everybody forever. It just happens less often as you gain experience. One really good habit to get into is indenting all the statements in a do-loop that follow the DO statement. You can then see if you forgot the END statement just by looking at your program, as you know it should look like this:

```
Do statement ;  
    Thing to do ;  
    Other thing to do ;  
End ;
```

For more on DO loops, including DO OVER, DO BY and DO UNTIL, you may want to check some papers from other regional and national SAS conferences. The paper by Waller (2009) is a good place to start.

A PIECE OF ADVICE ON RE-CODING VARIABLES

Why create two arrays? Wouldn't it be more efficient to simply recode into the same variables? You could do that but

I would advise against it. There is simply too much possibility for error. Later on, someone else can't remember if the variables were reverse coded or not and runs a similar program again, thus reversing your reverse coding. Or, for some reason, there is an error and your variables don't get reverse-coded. These are the types of errors that sometimes don't get caught for weeks until the internal reliability coefficients, correlates of the total score or some other statistic turns out to be far different from what was expected. When the error gets back-tracked to your program it can be, well, embarrassing.

DESIGN CHOICES: WHEN YOU MIGHT NOT BOTHER WITH AN ARRAY

First, let's assume that all you care about is the total score. Second, let's assume you have no missing data. Web-based surveys that require an answer before the submit button is active force respondents to provide complete data. Research projects where interviewees are required to make one of the choices on each question are another case where no missing data should occur. (I have worked on projects like this. Whether that is good policy or not is a question for the market researchers to argue about.) If those assumptions are met, that there is no missing data and that you don't care about separate item responses, you could simply use two statements:

```
NegativeRev = 60 - ( sum(q1,q4,q5,q6,q7,q9,of q17 - q20) ) ;
Positive = NegativeRev + sum(q2,q3,q8,of q10 - q16) ;
```

NOVICE TO INTERMEDIATE IV: LOOK PROFESSIONAL ! (& DON'T MESS WITH THINGS THAT DON'T BELONG TO YOU)

Even though what will impress most other programmers the most is how your code looks, a far greater number of people will see your output. SAS makes it so easy to create publication-quality documents that plain text like this makes you look like, well, like a novice.

The SAS System 23:40 Wednesday, August 4, 2010 1

Obs	t_day	Aban_ Calls	Calls_ offered	Abd_ Call_Pct	A_speed	Ab_time	A_time	weekday
1	2010MAR	1	19	0.05263	5	1	101	3
2	2010MAR	3	24	0.12500	5	17	118	4
3	2010APR	0	11	0.00000	4	0	50	5
4	2010APR	1	17	0.05882	5	5	74	6
5	2010APR	2	36	0.05556	4	3	157	7

Let's fix this thing.

```
Libname in "E:\wuss2010" ;

Ods html file = "e:\wuss2010\print1.html" style = meadow ;
Title "Call Center Statistics" ;
Title2 "Service Department" ;
Footnote "Because someone always has to ask for a footnote" ;
proc print data = in.dboardsfinal split = " " ;
  id t_day ;
  var Aban_calls -- abd_call_pct A_speed -- weekday ;
  Where (Month(t_day) = 4 & Year(t_day) = 2010) and area in (111,112,113) ;
  label aban_calls = "Unanswered Calls"
        calls_offered = "Total Calls"
        Abd_call_pct = '%Calls Unanswered'
        weekday = "Day of Week" ;
  format t_day mmdyy8. abd_call_pct percent10.0 weekday downname. ;
run;
ods html close ;
run;
```

Call Center Statistics Service Department

Date	Unanswered Calls	Total Calls	%Calls Unanswered	Average Speed Answered	Average Abandoned Time	Average Call Time	Day of Week
04/01/10	1	6	17%	9	29	231	Wednesday
04/02/10	0	6	0%	7	0	151	Thursday
04/05/10	0	10	0%	8	0	141	Sunday
04/06/10	1	5	20%	6	137	170	Monday
04/07/10	0	16	0%	5	0	245	Tuesday

Because someone always has to ask for a footnote

There, now doesn't that look better! Let's see how this is done, one statement at a time.

```
ods html file = "   Filename " ;
```

This statement outputs to an html file. You can also output to a pdf, or rtf file. The "style=" is optional. While creating your own style is more of an advanced area, there are dozens of styles installed with SAS. I picked meadow as the style for this report simply because I liked the colors. One of my fellow team members pointed out, after numerous meetings and several layers of approval that the table was not done in the university colors and it should actually have some red in it. True story: I solved this problem by threatening to kill him if he brought it up during a meeting. In retrospect, I could have just changed the statement to say

```
ods html file = "   Filename " style = brick;
```

Oh well, hindsight is always 20-20.

At the end of the procedures you want output to the file make sure you include a RUN statement and an ODS statement that matches the first one.

If you have ODS HTML FILE = "filename.html" ;

You should have these statements after your last procedure you want output to that file:

```
run ;
ods html close ;
run;
```

YOUR MILEAGE MAY VARY: PROBLEMS NOT TO HAVE WITH ODS

In earlier releases, if you did not have an ODS CLOSE statement to match your ODS FILE statement you would get an error. Lately, with SAS 9.2 I have tried leaving the ODS CLOSE out, opening a new ODS file without closing the old one and even mis-matching the statements and having an ODS HTML FILE = and an ODS RTF CLOSE statement. I tried this both using Windows and Unix versions and no matter what I did my ODS output file came out perfectly. This proves both that SAS 9.2 is much more "idiot-proof" and that I definitely need to find a hobby.)

```
title "whatever you want on the first line" ;
```

The TITLE statement replaces that lovely line about The SAS System and the data and time at which the job was run, as if you were just dying to know, with whatever you put between the quotes. TITLE2 will write a second line of text.

The FOOTNOTE statement prints text at the bottom of the page. You can have up to ten lines of titles and ten lines of footnotes, which should be enough for anybody.

```
proc print data = datasetname split = " " ;
```

The split= option in PROC PRINT wraps the text in your label to the next line, splitting the label at whatever is specified between the quotes, in this case, a blank space.

```
id t_day ;
```

The ID statement replaces the observation number in the first column with the variable you specified.

```
var Aban_calls -- abd_call_pct A_speed -- weekday ;
```

VAR selects only certain variables to print. While a single dash will select variables that are named in numeric order, such as q1 – q20, a double-dash will select variables in the order they are on the dataset, from the first variable in the list to the last. This isn't always the greatest idea because it too often turns out the order is not what you expected. In this case, just doing a PROC PRINT, no great harm can result and any errors would be obvious; you wouldn't see the columns you expected on the print out.

```
Where ( Month(t_day) = 4 & Year(t_day) = 2010) and area in (111,112,113) ;
```

You probably know that you cannot use an IF statement in a procedure, only in a DATA step. To print only those records from April, 2010 and only the areas in a specific department, you could create a new dataset and select only the records to meet your criteria, but that would be inefficient. Learn from the voice of experience, Sky Masterson, who, in Guys and Dolls said, and I quote:

One of these days in your travels, a guy is going to show you a brand-new deck of cards on which the seal is not yet broken. Then this guy is going to offer to bet you that he can make the jack of spades jump out of this brand-new deck of cards and squirt cider in your ear. But, son, do not accept this bet, because as sure as you stand there, you're going to wind up with an ear full of cider.

The same is true of data sets. The second you delete those observations you are absolutely sure you will never need again, the president of your company is going to ask for a report on March of 2003. As a general rule, your time to recreate a dataset is worth far more than a bit of disk space. (On the same note, any data set you might accidentally delete that is large enough for you to panic is probably backed up every night and someone in your organization can restore it for you.)

Unlike the IF statement, WHERE can be used in a PROC step, thus leaving your original data intact. The WHERE statement only uses those observations that meet the conditions in the statement. You can, as in this example, include functions in your WHERE statement. The MONTH function returns the value from 1 to 12 of the month part of a date. YEAR returns the year as a four-digit number. In this example, the department of interest has three different subgroups with the variable name "area" and the WHERE statement selects records from those three areas that are from April, 2010.

```
label aban_calls = "Unanswered Calls"  
      calls_offered = "Total Calls"  
      Abd_call_pct = = '%Calls Unanswered'  
      weekday = "Day of Week" ;
```

The LABEL statement provides a label for use during that PROC step. If a different label is given than what is stored in the data set, this statement overrides that label for this procedure only. If, as in this example, you specify labels for only four of your variables, then the specified labels will be used for those four, and for the other variables in your VAR and ID statements, the labels stored with those variables will be used.

LABEL DETAILS

1. If you are extremely perceptive you may have noticed in the statement above the %Calls Unanswered is shown in single quotes. If you include in double quotes a % followed by something that could be a valid SAS name, it will be interpreted as an attempt to call a macro. If no such macro exists, it is harmless and you will simply get a warning in your SAS log that states,

"Warning: Apparent invocation of macro Calls could not be resolved".

However, if there was a macro named Calls it would be invoked, with undesired results. Macros within single quotes are not resolved. It's sloppy to have warnings show up in your log and makes clients nervous. Try to avoid it.

2. According to the SAS documentation you are supposed to include label in your PROC PRINT statement as in :

```
Proc print data=datasetname label ;
```

" If you omit LABEL, PROC PRINT uses the variable's name as the column heading even if the PROC PRINT step contains a LABEL statement. " (SAS Institute, 2009).

Well, that's not 100% true. If you use the LABEL option or the SPLIT = option in your PROC PRINT statement, either one will work.

```
format t_day mmdyy8. abd_call_pct percent10.0 weekday downame. ;
```

The FORMAT statement doesn't change the way your data are stored, only how they are printed. You can change the format for a specific procedure without changing it in the dataset. As you can see, the FORMAT statement in the print procedure overrides the format stored in the dataset. Although the format in the data set is MONYYYY, the format used to print the date variable is mmdyy8., as specified. Just as it is a pretty safe assumption that any commonly used function is going to already be available pre-written in SAS, probably any common format you can imagine, such as percent, already exists. Even many that aren't so common exist, such as day of the week name for the numbers 1 through 7. There have been times when I have wondered seriously if managers imagine new formats just to test me, "This is great but can we get it in a format like this – 01 April 2010?" Before writing your own formats peruse the SAS documentation for the formats that already exist.

NOVICE TO INTERMEDIATE V: PROC FORMAT TO CREATE YOUR OWN FORMATS (& MORE)

This may be the most versatile procedure in your SAS toolkit, so it's a good idea to learn PROC FORMAT early on in your career. A common use of the FORMAT procedure, but far from the only one, is to create your own formats. Let's suppose you know there is no format for what you need. For example, your company has site offices stored by number, but you really want your reports not to show Site= 1 but rather, site = "Anchorage,AK" . All you require is a PROC FORMAT statement and one or more VALUE statements.

```
Proc Format ;
  Value site
    1 = "Anchorage,AK"
    2 = "Boise,ID"
    3 = "Chicago,IL"
    4 = "Denver,CO"
    5 = "Las Vegas, NV" ;
  Value $mgr
    "JNF" = "Jacob N. Flores"
    "HHN" = "Hayward H. Nishioka"
    "JEP" = "James E. Pedro"
    "CAM" = "Crespine A. Mojica" ;
```

You can have as many VALUE statements as you would like. The format is simple, the value stored in the data set on the left side of the = sign and the value you want shown on the right. Enclose text in quotes. You can use formats to convert numbers to text, text to different text, text to numbers or numbers to different numbers. Format names can be any valid SAS name. A character format, which is one that has text as input (that is, what is to the left of the = sign), needs to be preceded by a \$ both when you create it and when you call it.

To reference your formats, just use a FORMAT statement:

```
Format sales dollar9.1 storesite site. manager $mgr. ;
```

You can use both SAS formats and your user-written formats in the same statement. You can use the FORMAT statement in both DATA step and PROC steps. Let's consider the choice you make here because it has some major implications.

In a PROC step, the format is used but your data are unchanged. In a DATA step, the format information is stored with the data.

Be careful in creating a dataset with user-written formats. There are two problems that may arise. The first involves you. If you use this data set later, you may get error messages saying that the format cannot be found. One solution to this problem is to store your format permanently.

One choice is to create permanent formats and that solves the problem for you. However, your second problem can arise when you share this file with a colleague who does not have the same format library as you.

One solution to the errors from a dataset with user-written formats is to include this OPTIONS statement.

```
Options nofmterr ;
```

SAS will simply ignore the format errors and act as if those variables did not have a format.

A second option is to include a SAS program with the PROC FORMAT code with data sets that use user-written formats. The Inter-university Consortium for Political and Social Research (ICPSR) is one well-known repository of SAS data sets that uses this option. There is a third option

NOVICE TO INTERMEDIATE V: BEYOND DATA AND PROC - %INCLUDE

If you thought PROC FORMAT was cool allowing you to write your own formats, you'll really love the SAS macro language, a major extension of Base SAS that allows you to write a program to write programs. Learning to write macros is a big move away from being a novice. There are a couple of baby steps you can take to get started in that direction. The first is the use of %INCLUDE. I usually introduce this to new programmers first because it is relatively easy.

%INCLUDE is a gentle way to get a novice introduced to the concept of SAS macros, which can be a bit intimidating.

When you %INCLUDE something, unlike with a macro, you don't need to change any of the variables, statements or functions. Debugging is a snap because you can run the code exactly as is, copy it into another dataset once it is bug free and then run it. Yet, it leads you into the idea of having code from somewhere other than within your program run over and over.

So, what does %INCLUDE do?

It includes programming statements from another file outside of your program. There are two reasons to use this very frequently. One is to save copying and pasting the same code over and over. A common example is acknowledgements of funding, legal disclaimers, legends or other text that might always go at the bottom of every page on a website, such as:

```
Footnote1 "SAS and all other SAS Institute Inc. product or service names are
registered trademarks or trademarks of" ;
Footnote2 "SAS Institute Inc. in the USA and other countries. " ;
Footnote3 "Other brand and product names are trademarks of their respective
companies. " ;
Footnote4 "This research supported through USDA Grant #2005 - 123456 " ;
```

If you have 10 lines of this, you can copy and paste it every time you write a new program, or you can simply save the footnotes to a file and when ever you need these use a line something like this:

```
***** This is the text for footnotes ***** ;
%include "c:\myfiles\mysasfiles\footers.sas " ;
```

You really don't need those ten lines of footnotes getting in your way every time you're trying to read the program and de-bug it or document it. A major advantage over copying and pasting the same lines in every program is that if you get a new grant, or SAS gets bought by IBM, you don't have to go find every program and change that code. I can just change it in the footers.sas file and I'm done.

A second common use for %include is , again, to make the code more readable and get rid of distractions. Having just said that ICPSR includes PROC FORMAT code with many of the SAS data sets available on its site let me add that I find that somewhat annoying and I use %INCLUDE to handle the formats.

You may get a program from ICSPR or another source that has hundreds of lines of Label and Format statements (this happens to me all the time).

```
proc format;
value statnum 1='(1) Alabama'
              2='(2) Arizona'
              3='(3) Arkansas'
```

And a hundred more lines of the same ...

Followed by :

```
label
  rec_bh = 'bh:record type'
  statnum = 'numeric state code'
  ori = 'originating agency identifier' ;
format state statnum.  division divisn. ;
```

And a hundred more lines ...

Moving the Proc format, labels and formats cuts the length of the program in this example that I had received from 387 lines to 160. There are several reasons I may want to do this. The first is debugging. Before I am sure I read in the code exactly the way I want it, the formats are pretty irrelevant. Often, for whatever reason, the file is not in the exact format I expected. I don't want to slog through a lot of useless formats and labels before the data are actually read in. When I do have a permanent data set, I may want to use those formats but not have to see hundreds of extra lines in my code.

Important point: %INCLUDE statements are executed as they are read. When SAS hits a %INCLUDE statement it is as if the code as copied into your program. Think about this. What this means is that I had to move the FORMAT procedure into one file and the LABEL and FORMAT statements into another file.

My program now looks like this:

```
***** Proc Format to create user-written formats ***** ;
%include "c:\myfiles\mysasfile\crimefmt.sas" ;

Data libref.crimes ;
  Infile  datasetname ;
  Input variables ;
  <bunch of other statements >

***** Labels and Formats ***** ;
  %include "c:\myfiles\mysasfile\labelfmt.sas" ;
run;
```

Note that if I have the labels and formats after the run statement, it will give me an error.

CONCLUSION – TYING IT ALL TOGETHER

Does moving from being a novice to a not-so-novice programmer mean knowing everything there is to know about PROC TABULATE? Well, yes and no. It would be hard to say someone who knew everything about any facet of SAS, from ODS to SQL, was a novice. However, how useful is that person when you need to perform calculations, format your data differently, merge files or write more efficient, readable code? In the long-term, whether going deep or going wide is better is a debatable topic. For a beginning programmer, though, my recommendation is to spread

out and increase your knowledge across a range of topics from programming to presentation to new procedures.

There are a lot of possible next steps. One is to learn about all of the resources for learning SAS, such as the mailing list SAS-L, sasCommunity.org, SAS publications, of course, and some of the many blogs by both SAS Institute employees and SAS users. A second direction is to become familiar with more SAS products, such as SAS Enterprise Guide, Enterprise Miner, and smaller features like the Power and Sample Size application. Both of those directions show that you have become interested enough to search for new knowledge and that is always a great sign.

The biggest step forward, though, I think is – PLAY! As you learn more about SAS statements, functions, procedures and products you'll invariably like some more than others. For example, my brain tries to crawl out of my skull to escape being melted by boredom whenever I have to look at PROC SQL code. You may love SQL (unlikely as that may be). The more time you spend working on different projects, the more you'll learn and the more you'll discover what really interests you. I don't have a profound thought to end on except this. Remember the movie, Pleasantville, where his mother said

"...I had the right house. I had the right car. I had the right life."

And her son answered,

"There is no right house. There is no right car."

Learning SAS (or any programming language) is very much like that. There is no right choice. There are a lot of choices. The more you learn, the greater the number of choices you get.

REFERENCES

Kelsey Basset, B. (2006). The SORT Procedure: Beyond the Basics <http://www2.sas.com/proceedings/sugi31/030-31.pdf>

Howard, N. (1999) An introduction to SAS functions.
<http://www2.sas.com/proceedings/sugi24/Begtutor/p57-24.pdf>

Keelan, S. (2002) Off and running with arrays. <http://www2.sas.com/proceedings/sugi27/p066-27.pdf>

SAS Institute (2009). Base SAS 9.2 Procedures Guide.
<http://support.sas.com/documentation/cdl/en/proc/61895/HTML/default/viewer.htm#/documentation/cdl/en/proc/61895/HTML/default/a002291718.htm>

Waller, J. L. (2009). How to use arrays and DO –loops: Do I DO OVER or Do I DO i ?
<http://support.sas.com/resources/papers/proceedings09/155-2009.pdf>

CONTACT INFORMATION

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

Name: AnnMaria De Mars
Enterprise: The Julia Group
Address: 2111 7th St #8
City, State ZIP: Santa Monica, CA 90405
Work Phone: (310) 717-9089
Fax: (310) 396-0785
E-mail: annmaria@thejuliagroup.com
Web: <http://www.thejuliagroup.com>

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.