TOP 20 WAYS TO OPTIMIZE YOUR SAS CODE

Handy Tips for the Savvy Programmer

CUSTOMER LOYALTY TEAM • Support You Can Count On

SAS PROGRAMMING BEST PRACTICES

AGENDA

- Create Readable Code
- Basic Coding Recommendations
- Developing Code

CREATE READABLE CODE

TIPS FOR CREATING CODE THAT YOU AND YOUR CO-WORKERS WILL FIND EASY TO READ AND UNDERSTAND.

1. COMMENT, COMMENT, COMMENT! WHICH COMMENT FORMAT TO USE?

Method 1:

```
/* create summary report*/
proc means data=new;
   more statements here;
run;
```

Method 2:

```
*create summary report;
proc means data=old;
   more statements here;
run;
```

Note: Method 1 may also be helpful when developing and debugging code.

1. COMMENT, COMMENT, COMMENT!

Method 1:

```
/*
data new;
   set old;
   more statements;
run;
*/
proc means data=new;
   more statements here;
run;
```

Efficiency consideration: every submission of the DATA step re-creates the SAS data.

2. USE RECOMMENDED FORMATTING FOR SAS CODE

Not this:

```
data new; set old; run;
proc means data=new;
var newvar; class year;
run;
```

Do this:

```
Data new;
  set old;
Run;

proc means data=new;
  var newvar;
  class year;
run;
```

3. USE DESCRIPTIVE NAMES

Do this:

```
data salaryinfo2012;
  set salaryinfo2011;
  newsalary=oldsalary+increase;
run;
```

Not this:

```
data new;
  set old;
  z=x+y;
run;
```

4. USE UNDERSCORES OR CAMEL CASE TO CREATE DESCRIPTIVE NAMES

Camel Case

```
data salaryInfo2015;
  set salaryInfo2014;
  newSalary=
      oldSalary+increase;
run;
```

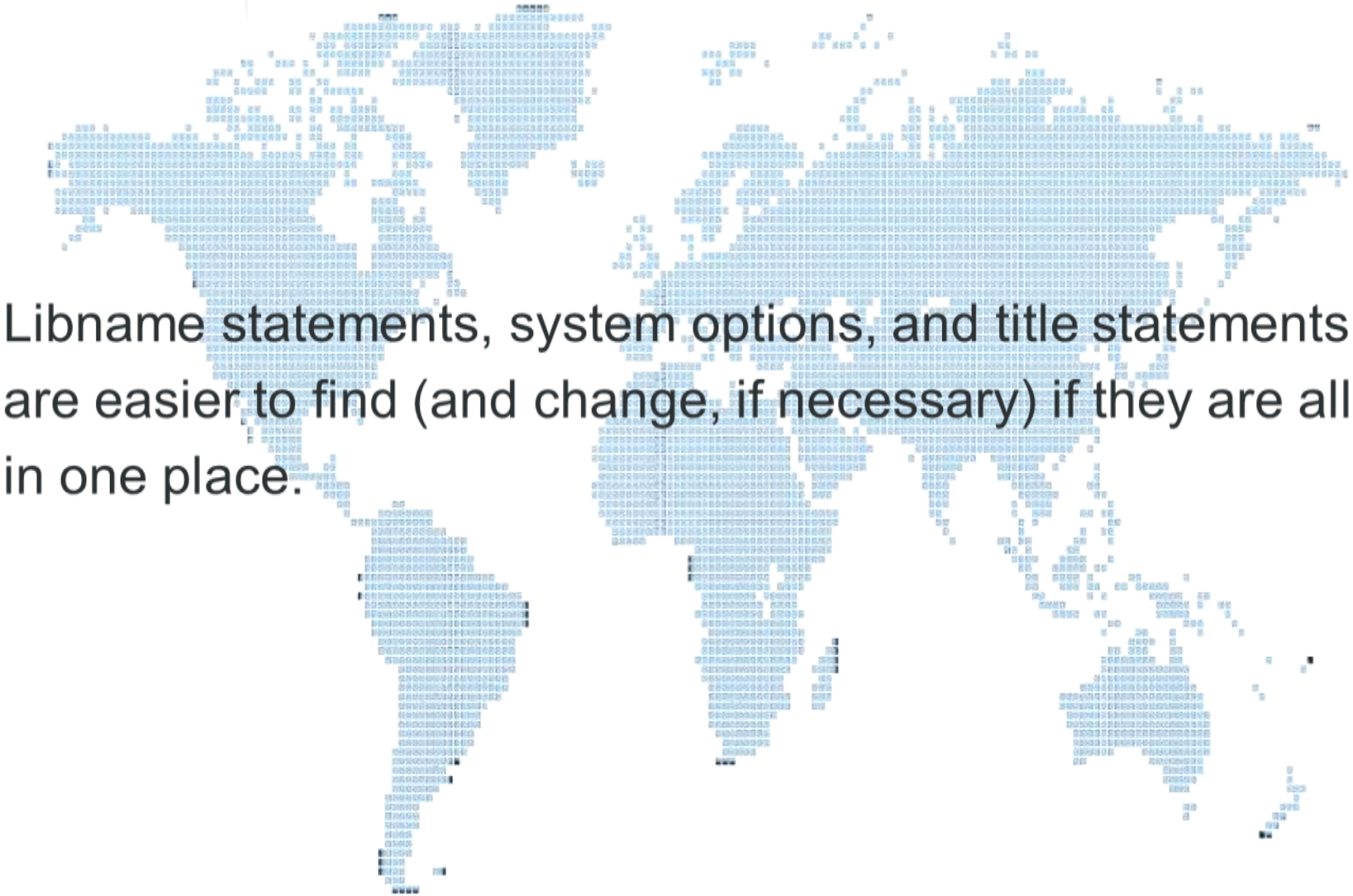
Underscores

```
data salary_info2015;
  set salary_info2014;
  new_salary=
     old_salary+increase;
run;
```

SAS names:

- Can be 32 characters long.
- Must start with a letter or underscore, continuing with numbers, letters or underscores.
- Can be uppercase, lowercase or mixed case.
- Are not case sensitive.

5. PUT ALL "GLOBAL" STATEMENTS AT THE BEGINNING OF YOUR CODE



Libname statements, system options, and title statements are easier to find (and change, if necessary) if they are all in one place.

BASIC CODING RECOMMENDATIONS

BASIC CODING RECOMMENDATIONS TO INCREASE THE EFFICIENCY OF YOUR SAS PROGRAMS.

6. MINIMIZE THE NUMBER OF TIMES YOU READ YOUR DATA

Do this:

```
data a b c;
  set old;
  if condition then
     output a;
  else if condition then
     output b;
  else if condition then
     output c;
run;
```

Not this:

```
data a;
    set old;
    [more code]
run;
data b;
    set old;
    [more code]
run;
data c;
    set old;
    [more code]
run;
```

6. MINIMIZE THE NUMBER OF TIMES YOU READ YOUR DATA

Do this:

```
proc freq data = sashelp.shoes;
table region / list out=region_freq1;
table region*product/ list out=region_freq2;
table region*stores / list out=region_freq3;
run;
```

Not only use for data processing, also use for procedures

Not this:

```
proc freq data = sashelp.shoes;
table region / list out=region_freq1;
run;

proc freq data = sashelp.shoes;
Table region*product/ list out=region_freq2;
run;

proc freq data = sashelp.shoes;
table region*stores / list out=region_freq3;
run;
```

7. LIMIT THE NUMBER OF TIMES YOU SORT YOUR DATA

```
data new;
     infile 'rawdata.dat';
     input ID $ 1-4 name $ 5-25 salary 26-35;
run;

proc sort data=new out=new_sorted presorted;
    by ID;
run;
```

If you think the incoming data is already sorted, use the **presorted** option on your SORT statement; the sort order will be verified.

7. LIMIT THE NUMBER OF TIMES YOU SORT YOUR DATA

When creating an SQL view, avoid including an ORDER BY clause in the view, as the data will need to be sorted every time the view is used.

```
proc sql;
   create view sql.new as
      select *
         from sql.old
         order by firstvar;

proc print data=sql.new;
Run;
```

The PROC PRINT or any other procedure/Data step that uses the view will execute the stored SQL query, including the ORDER BY.

7. OVERVIEW OF SQL VIEWS

A PROC SQL view contains a stored query that is executed when you use the view in a SAS procedure or DATA step. Views are useful for the following reasons:

- Often save space, because a view is frequently quite small compared with the data that it accesses
- Shield sensitive or confidential columns from users while enabling the same users to view other columns in the same table
- Ensure that input data sets are always current, because data is derived from tables at execution time
- Hid complex joins or queries from users

7. WHEN IS A SORT REQUIRED

Requires sorting

- DATA step with SET or MERGE and BY statements
- By statement in PROC MEANS, PROC FREQ, etc.

Does not require sorting

- PROC SQL joins
- CLASS statements in PROC MEANS, PROC FREQ, etc

8. USE IF-THEN-ELSE INSTEAD OF IF-IF-IF

Do this:

```
data new;
  set old;
  if condition then
     some action;
  else if condition then
     some other action;
  else if condition then
     some other action;
run;
```

Not this:

```
data new;
  set old;
  if condition then
     some action;
  if condition then
     some other action;
  if condition then
     some other action;
run;
```

Please note that this general recommendation relates to conditions that are mutually exclusive. If the conditions are not mutually exclusive, then further consideration is in order to see whether IF THEN ELSE is appropriate.

9. ORDER IF THEN CONDITIONS IN DESCENDING ORDER OF PROBABILITY

```
data new;
  set old;
  if condition occurring most often then
     some action;
  else if condition then
     some other action;
  else if condition then
     some other action;
run;
```

10. SELECT ONLY THE COLUMNS YOU NEED WHEN WORKING WITH SAS DATA

Do This:

```
data new;
   set old (drop=category
      type value ...);
   more statements here;
run;
```

Not This:

```
data new;
   set old;
   more statements here;
run;
```

Variations:

- Use the keep= option if you need to keep less variables than you need to drop!
- Use both keep= and drop= options to control variables on both the incoming and outgoing sides!
- Keep= and drop= options can be used in PROC steps, too!

11. SELECT ONLY THE ROWS YOU NEED WHEN WORKING WITH SAS DATA

Do this:

```
data new;
   infile ‘old.dat’;
   if city=‘CLEVELAND';
   more statements here;
run;
```

Not this:

```
data new;
   infile ‘old.dat’;
   more statements here;
run;
```

12. CONSIDER THE POSITION OF THE SUBSETTING IF

Do this:

```
data new;
   infile ‘old.dat’;
   if city=‘CLEVELAND';
   more statements here;
run;
```

Not this:

```
data new;
   infile ‘old.dat’;
   more statements here;
   if city=‘CLEVELAND';
run;
```

Subset as soon as you have all necessary values in order to prevent unnecessary creation of variables and additional processing.

13. IF YOU ARE READING SAS DATA, USE WHERE INSTEAD OF SUBSETTING IF

Try this:

```
data new;
   set old;
   where condition;
   more statements here;
run;
```

Instead of this:

```
data new;
   set old;
   if condition;
   more statements here;
run;
```

WHERE is a pre-processor. It subsets data before it is loaded into the Program Data Vector (PFV).

Added efficiency: when using SAS/Access engines, SAS attempts to send the WHERE clause to the RDBMS for evaluation rather than to SAS;
With the IF statement, SAS must do the processing.

13. IF YOU'RE GOING TO RUN A PROCEDURE ON THE DATA, USE THE "WHERE" STATEMENT IN THE PROCEDURE

Instead of this:

```
data new;
   set old;
   where condition;
   more statements here;
run;
proc means data=new;
   more statements here;
Run;
```

Try this:

```
proc means data=old;
   where condition;
   more statements here;
run;
```

13. WHERE VS IF

WHERE and IF processing are not always 'interchangeable'

- IF processing must be used with:
  - Accessing raw data using INPUT statements
  - With Automatic Variables, e.g. first.variable, last.variable, _N_, etc.
  - Using newly created variables in the same DATA Step
  - In combination with data set options such as OBS =, POINT = , FIRSTOBS =
  - To conditionally execute a statement

13. WHERE VS IF

WHERE and IF processing are not always 'interchangeable'

- WHERE processing must be used to:
  - Utilize special operators such as LIKE or CONTAINS
  - Filter rows for input to SAS Procedures
  - Trigger use of indexes*, if available
  - When sub-setting as data set option
  - When sub-setting using PROC SQL

*The presence of an index column on a SAS data set does not always guarantee its use in a query

13. WHERE VS IF

- WHERE and IF processing are applied differently in MERGE operations:
- With WHERE processing the sub-setting takes place before the MERGE operation.
- With IF processing the sub-setting takes place after the MERGE operation.

Be careful!

13B. SELECT VS IF THEN ELSE

IF THEN

- When there are few conditions to check
- The values are not uniformly distributed
- The values are character or the values are discrete numeric data

SELECT

- Where there is a long series of mutually exclusive conditions
- The values are numeric and are uniformly distributed

Efficiency Considerations Using the SAS® System
EFFECTIVE USE OF SAS SELECT LANGUAGE STATEMENT

14. CONSIDER DECLARING VARIABLES AS CHARACTER WHEN THERE IS A STORAGE SAVINGS.

Consider Employee ID values similar to the following:

```
1015
2034
5543
6793
...
```

```
data new;
   input ID 1-4;
```

- ID is numeric requiring 8 bytes of storage

```
data new;
   input ID $ 1-4;
```

- ID is character requiring 4 bytes of storage

A savings of 4 bytes per observation adds up when dealing with large data!

15. USE BUILT IN FEATURES AND FUNCTIONS

Better:

```
if (upcase(a) = 'YES') then x = 1;
```

When testing for all possible combinations

Works:

```
if (a = 'YES' or
    a = 'YEs' or
    a = 'YeS' or
    a = 'yES' or
    a = 'yeS' or
    a = 'yEs' or
    a = 'Yes' or
    a = 'yes' ) then x = 1;
```

15. USE BUILT IN FEATURES AND FUNCTIONS

Better:

```
Passed = sum((Q1 = 'Y'),
     (Q2 = 'Y'),
     (Q3 = 'Y'),
     (Q4 = 'Y'),
     (Q5 = 'Y'),
     (Q6 = 'Y'),
     (Q7 = 'Y'),
     (Q8 = 'Y'),
     (Q9 = 'Y'))/9 > .5;
```

No need for temporary counter

Works:

```
Counter = 0;
If (Q1 = 'Y') then counter +1;
If (Q2 = 'Y') then counter +1;
If (Q3 = 'Y') then counter +1;
If (Q4 = 'Y') then counter +1;
If (Q5 = 'Y') then counter +1;
If (Q6 = 'Y') then counter +1;
If (Q7 = 'Y') then counter +1;
If (Q8 = 'Y') then counter +1;
If (Q9 = 'Y') then counter +1;
If ((counter/ 9) > .5) then Passed=1;
Else Passed=0;
```

15. USE BUILT IN FEATURES AND FUNCTIONS

Better:

```
My_variable = ‘PASSNFAIL’;
substr(My_variable,5,1) = ‘Y’;
```

Works:

```
My_variable = ‘PASSNFAIL’;
My_variable = substr(My_variable,1,4) ||
                    ‘Y’ ||
                    Substr(My_variable,6,4);
```

15. USE BUILT IN FEATURES AND FUNCTIONS

Types of Functions

- Character (SUBSTR, LEFT, RIGHT, UPCASE)
- Arithmetic (ABS, SUM, SQRT)
- Array (DIM)
- Date and Time (TODAY, YRDIFF, MDY, TIMEPART)
- Financial (MORT, NPV, SAVINGS
- Mathematical (LOG, EXP)
- Probability (POISSON, PROBCHI)
- Quantile, Random Number (NOMINAL, UNIFORM)
- Sample Statistics (MEAN, MIN, MAX, STD, NMISS)
- Special (LAG, PUT, INPUT)

Introduction to SAS Functions Paper
SAS 9.4 Language Reference Documentation

16. USE CEDA WISELY

- Reading SAS 9.2 or earlier data sets in SAS 9.3 results in a translation process using CEDA (cross-environment data access)
- Because the BASE engine translates the data as the data is read, multiple procedures require SAS to read and translate the data multiple times. In this way, the translation could affect system performance.
- “Convert” SAS data sets by using PROC MIGRATE or other techniques.

16. USE CEDA WISELY

If you see the following note in your log, consider using PROC MIGRATE or other techniques to convert your data to your current environment.

```
Note: Data file HEALTH.GRADES.DATA is in a format that is native to another
host, or the file encoding does not match the session encoding. Cross
Environment Data Access will be used, which might require additional CPU
resources and might reduce performance.
```

WHEN YOU ARE DEVELOPING CODE

TIPS TO SAVE TIME AND CREATE EFFICIENT CODE

17. TEST YOUR PROGRAMS WITH THE OBS= OPTION

```
data complicated_program;
  set sample_data(obs=50);
  many, many, many more statements here;
run;
```

This technique may not adequately test all conditions, but will confirm the correctness of the overall program logic – and save time and computer resources!

17. TEST YOUR PROGRAMS WITH THE PUT STATEMENT

```
data complicated_program;
  set sample_data(obs=50);
  if condition then do;
     put 'write value here' value;
     other statements to execute;
End;
run;
```

This technique allows you to test certain coding logic to determine if conditions are met as well as variable values.

18. BENCHMARK PRODUCTION JOBS

Recommendations for benchmarking include:

- Benchmark your programs in separate SAS sessions
- Run each program multiple times and average the performance statistics.
- Use realistic data for tests.
- Elapsed time should not be used for benchmarking.

19. USE MACRO VARIABLES TO SIMPLIFY MAINTENANCE

- Use macro variables to reduce the number of changes that have to be applied manually when code needs to be updated.

This works:

```
Libname file_01 ‘c:\SGF_2015\project_dir\input_data’;

Libname file_02 ‘c:\SGF_2015\project_dir\output_data’;

Filename in_file1 ‘c:\SGF_2015\project_dir\my_text.txt’;
```

This requires less Maintenance

```
%let My_Dir = c:\SGF_2015\project_dir;

Libname file_01 “&My_dir.\input_data“;

Libname file_02 “&My_dir.\output_data“;

Filename in_file1 “&My_dir.\my_test.txt“;
```

Note the Double Quote will cause the macro variable to be resolved, and one change is applied to all three commands.

19. USE MACRO VARIABLES TO SIMPLIFY MAINTENANCE

- Use macro variables to reduce the number of changes that have to be applied manually when code needs to be updated.

This works:

```
Data array_test;
Array counters {15} var_01-var_15;

Do I = 1 to 15;
      Counters(i) = 0;
End;
```

Three changes needed
if array size changes

This requires less maintenance

```
%let max_size = 15;

Data array_test;

Array counters {&max_size} var_01 var_&max_size;

Do I = 1 to &max_size;
      Counters(i) = 0;
End;
```

Only one change here

19. USE MACRO VARIABLES TO SIMPLIFY MAINTENANCE

- Use macro variables to reduce the number of changes that have to be applied manually when code needs to be updated.

This works:

```
Data myfile.Monthly_Data_for_2015_feb;
   Set myfile.Monthly_Data_for_2015_Jan;
Run;
```

Most people code the dates into file names

This requires less maintenance

```
%let old_month = 2015_Jan;
%let new_month = 2015_Feb;

Data myfile.Monthly_Data_for_&new_month;
    Set myfile.Monthly_Data_for_&old_month;
Run;
```

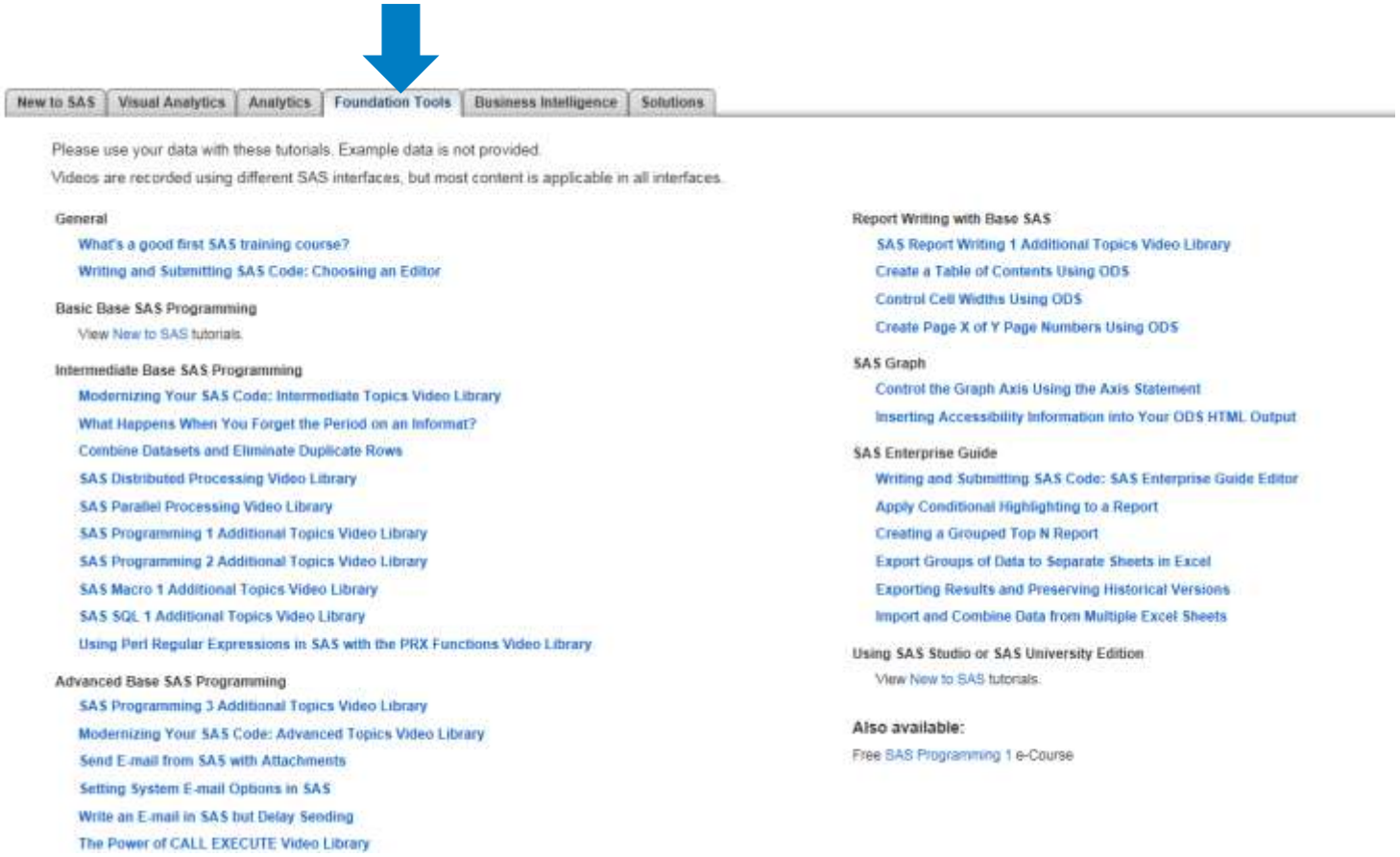
Using macro variables allows for one change at the top of the program

20. MAKE THINGS EASIER FOR YOURSELF: EFFICIENCY ALSO MEANS WORKING SMARTER!

- Be “GREEN” – save code and reuse it later!
- Collaborate with your co-workers to share tips and suggestions
- Meet regularly to share ideas
- Some ways SAS code fosters reusability:
  - Macro library
  - Stored processes
  - User-written functions and procedures.

RESOURCES ONLINE

- SAS Tutorials on Programming
- YouTube Video on SAS Programming (2-hour)



RESOURCES | SAS GLOBAL FORUM PAPERS

- **Leave Your Bad Code Behind: 50 Ways to Make Your SAS® Code Execute More Efficiently** William E Benjamin Jr, Owl Computer Consultancy, LLC
- **SAS® Shorts: Valuable Tips for Everyday Programming** Jeff McCartney and Raymond Hu, Social and Scientific Systems, Inc., Bethesda, MD
- **Productivity Tips for SAS® Enterprise Guide® Users** Jennifer First and Steven First, Systems Seminar Consultants, Madison, WI, United States
- **Tips and Techniques for the SAS® Programmer** Helen Carey, Carey Consulting, Kaneohe, HI, Ginger Carey, Carey Consulting, Kaneohe, HI

TO SEARCH PAST PAPERS CLICK HERE

YOUTUBE VIDEOS

VIDEOS FOR TOPICS A-Z

- SAS Software YouTube Channel
  - http://www.youtube.com/user/SASsoftware?feature=watch

Connect with me:
LinkedIn: https://www.linkedin.com/in/melodierush
Twitter: @Melodie_Rush

QUESTIONS?

Thank you for your time and attention!

sas | THE POWER TO KNOW.

CUSTOMER LOYALTY TEAM • Support You Can Count On