Executing SAS® Functions with the %SYSFUNC Macro Function

Transcript

Executing SAS® Functions with the %SYSFUNC Macro Function Transcript was developed by Lise Cragen. Additional contributions were made by Michelle Buchecker, Lynn Mackay, Kathy Passarella, and Russ Tyndall. Editing and production support was provided by the Curriculum Development and Support Department.

Executing SAS® Functions with the %SYSFUNC Macro Function Transcript

Book code E1626, course code RLSPSYSF, prepared date 21Dec2009. RLSPSYSF_001

ISBN 978-1-60764-374-6

Table of Contents

Lecture Description

This e-lecture describes how to use the %SYSFUNC macro function to manipulate text in the SAS macro facility.

To learn more…

For information on other courses in the curriculum, contact the SAS Education Division at 1-800-333-7660, or send e-mail to training@sas.com. You can also find this information on the Web at support.sas.com/training/ as well as in the Training Course Catalog.

For a list of other SAS books that relate to the topics covered in this Course Notes, USA customers can contact our SAS Publishing Department at 1-800-727-3228 or send e-mail to sasbook@sas.com. Customers outside the USA, please contact your local SAS office.

Also, see the Publications Catalog on the Web at support.sas.com/pubs for a complete list of books and a convenient order form.

Prerequisites

Before listening to this e-lecture, you should be familiar with the SAS macro facility. This includes knowing how to create macro variables using %LET statements, the CALL SYMPUTX routine, and the INTO clause in the SQL procedure. It also includes the ability to generate SAS log messages, apply macro functions, create and invoke macro programs, and use conditional macro logic. You should also have experience using SAS language functions.

Accessibility Tips

If you are using a screen reader, such as Freedom Scientific's JAWS, you may want to configure your punctuation settings so that characters used in code samples (comma, ampersand, semicolon, percent) are announced. Typically, the screen reader default for the character & is to read "and." For clarity in code samples, you may want to configure your screen reader to read & as "ampersand." In addition, depending on your verbosity options, the character & might be omitted. The same is true for some commas before a code variable. To confirm code lines, you may choose to read some lines character by character. When testing this scenario with Adobe Acrobat Reader 9.1 and JAWS 10, ampersands before SAS macro names were announced only when in character-reading mode.

Executing SAS Functions with the %SYSFUNC Macro Function

Executing SAS® Functions with the %SYSFUNC Macro Function

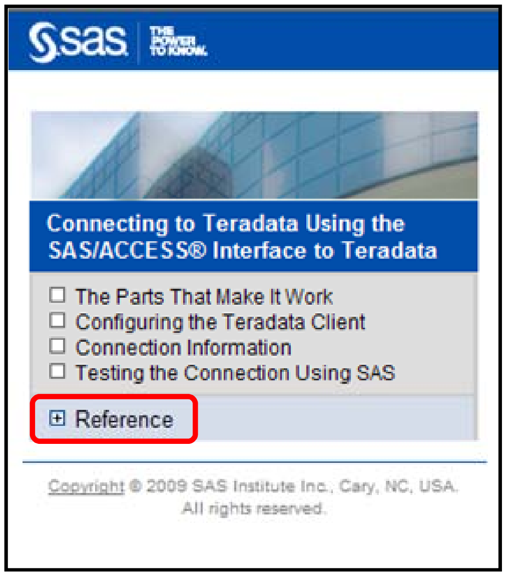
THE POWER TO KNOW®

Welcome to this SAS e-Lecture on *Executing SAS® Functions with the %SYSFUNC Macro Function*. My name is Lise and I'm an instructor with SAS Education.

Reference Materials

To access the transcript for this lecture:

1. Go to the table of contents on the left side of the viewer.
2. Select **Reference**.
3. Select **Transcript**.


sas THE POWER TO KNOW.
Connecting to Teradata Using the SAS/ACCESS® Interface to Teradata
☐ The Parts That Make It Work
☐ Configuring the Teradata Client
☐ Connection Information
☐ Testing the Connection Using SAS
⊞ Reference
Copyright © 2009 SAS Institute Inc., Cary, NC, USA. All rights reserved.

2

Before we begin the lecture, let me mention that we have included a transcript so that you can print all of the technical information provided in this lecture. To access the transcript, select **Reference** and then **Transcript** in the table of contents on the left side of the viewer, as in the example shown here. You can print this transcript now for use when viewing the lecture or print it later to keep as a reference. Also, note that Appendix A in the transcript contains copies of the programs used for the demonstrations in this lecture.

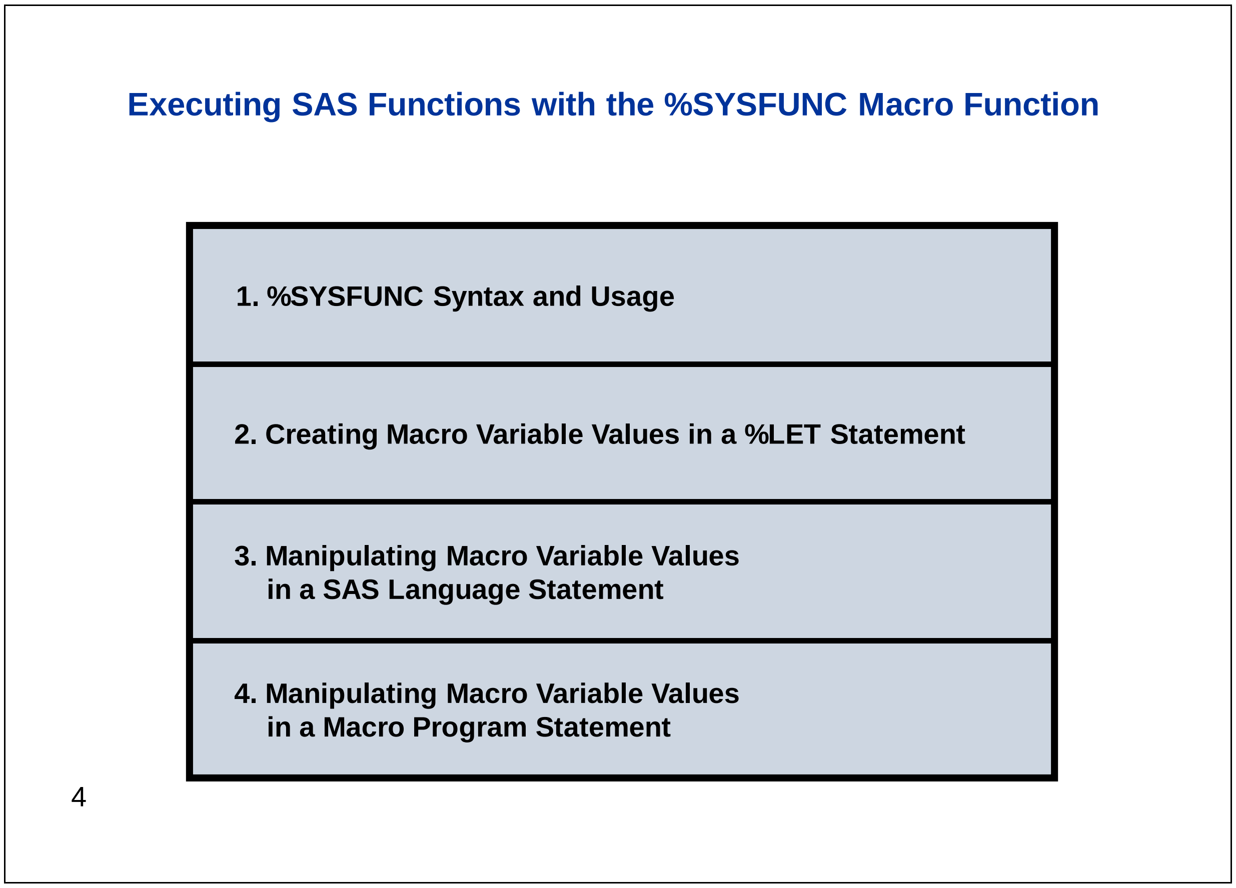
Navigation Help

For information on how to navigate this lecture:

1. Go to the upper right corner of the browser.
2. Select **Help**.

3

If you need help with the navigation of this lecture, please select **Help** in the upper right corner of the browser.

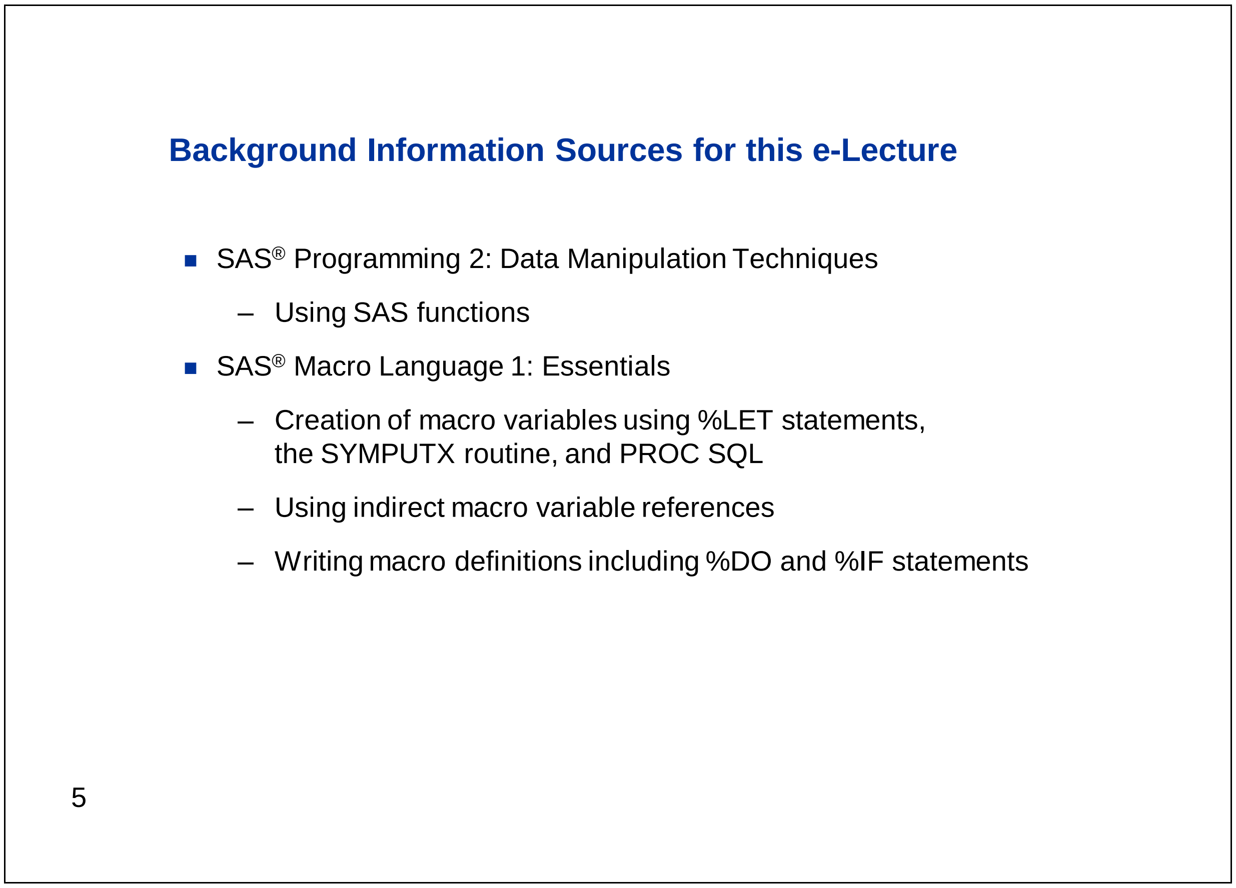


Executing SAS Functions with the %SYSFUNC Macro Function

1. %SYSFUNC Syntax and Usage
2. Creating Macro Variable Values in a %LET Statement
3. Manipulating Macro Variable Values in a SAS Language Statement
4. Manipulating Macro Variable Values in a Macro Program Statement

4

Let's start with an overview of the topics we'll be covering in this lecture. In the first section, we'll begin with a discussion of macro function concepts in general. Then we'll discuss the specifics of %SYSFUNC macro function in terms of syntax and usage. In Sections 2, 3, and 4, we'll focus on using %SYSFUNC in three different programming contexts.



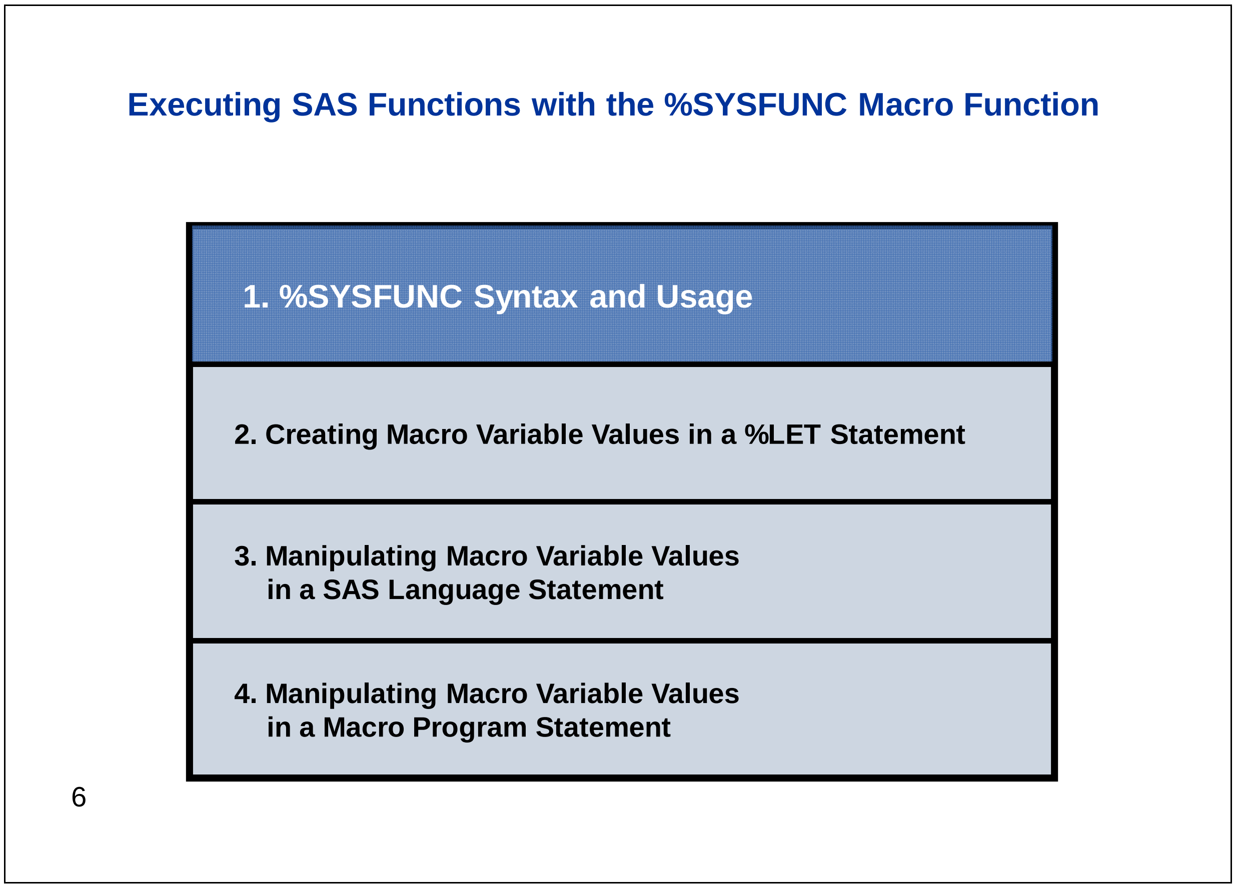
Background Information Sources for this e-Lecture

- SAS® Programming 2: Data Manipulation Techniques
  - Using SAS functions
- SAS® Macro Language 1: Essentials
  - Creation of macro variables using %LET statements, the SYMPUTX routine, and PROC SQL
  - Using indirect macro variable references
  - Writing macro definitions including %DO and %IF statements

5

For this e-lecture, I'll assume that you have quite a bit of SAS programming and macro programming knowledge and experience. First, I'll assume that you know how to use SAS language functions. I'll expect that you know how to create macro variables using %LET statements, the SYMPUTX routine in the DATA step, and the INTO clause in PROC SQL, and that you know how to use indirect macro variable references. Finally, I'll assume that you're familiar with how to write macro definitions including both %DO and %IF statements. The use of SAS functions is covered in detail in the SAS® Programming 2 course. All of the macro topics that I mentioned are addressed in the SAS® Macro Language 1 course.

1. %SYSFUNC Syntax and Usage



Executing SAS Functions with the %SYSFUNC Macro Function

1. %SYSFUNC Syntax and Usage

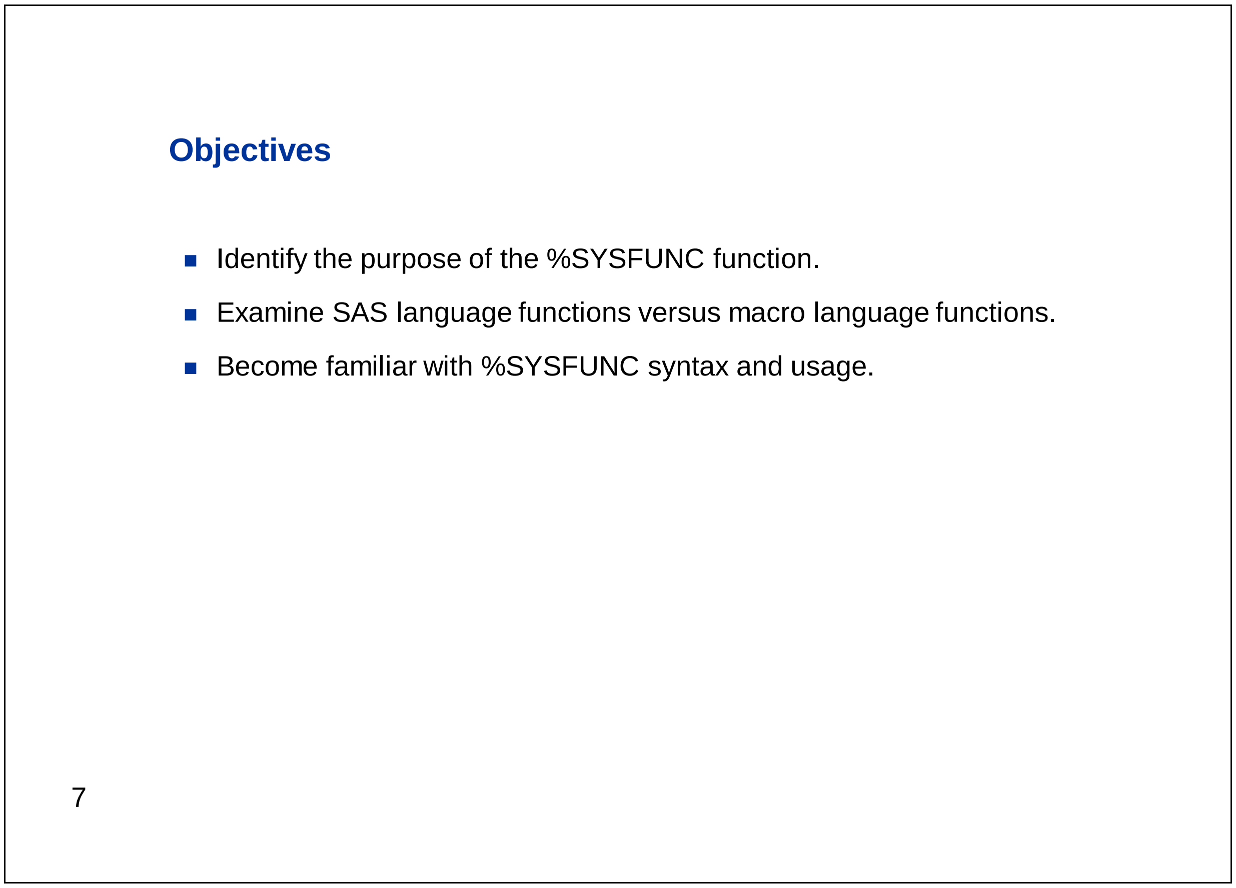
2. Creating Macro Variable Values in a %LET Statement

3. Manipulating Macro Variable Values
   in a SAS Language Statement

4. Manipulating Macro Variable Values
   in a Macro Program Statement

6

So let's begin with our first section, an introduction to %SYSFUNC.

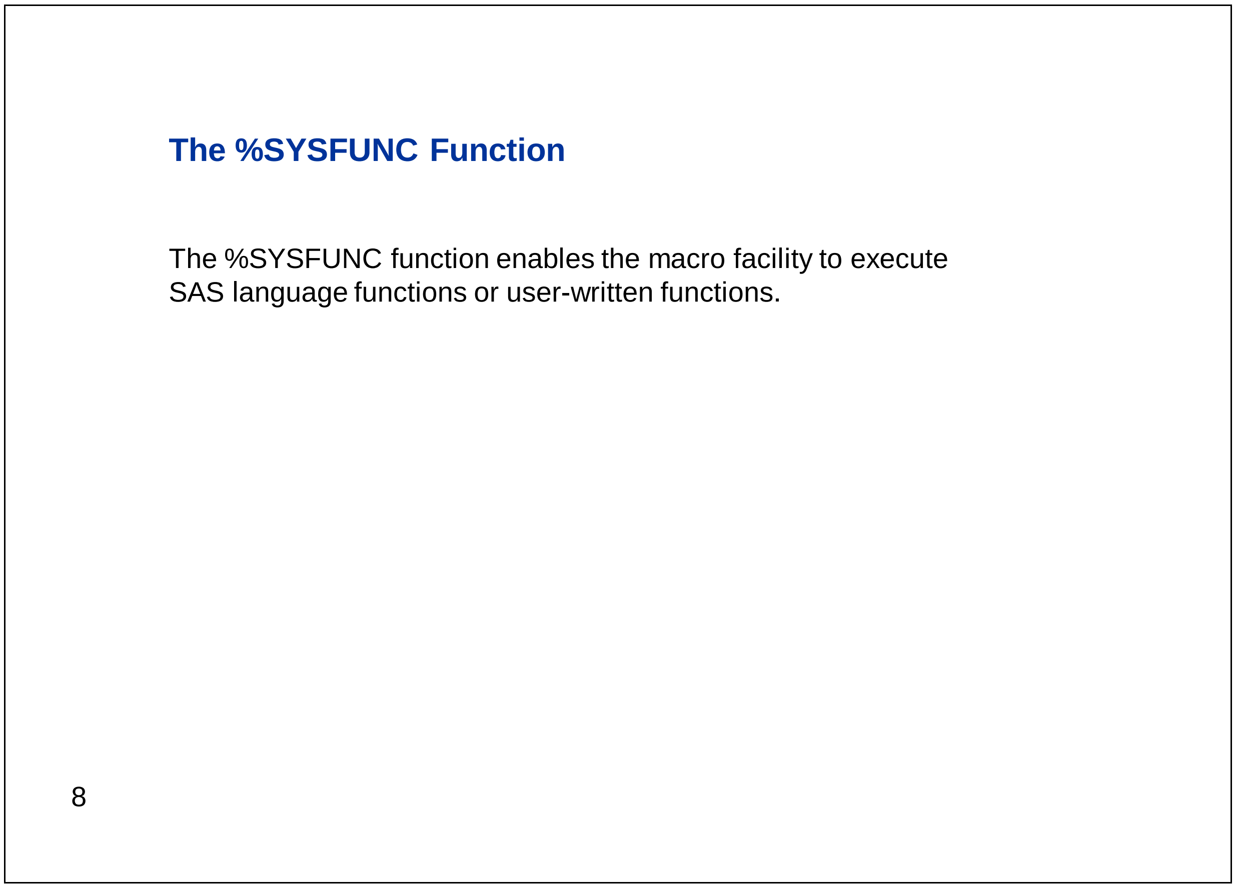


Objectives

- Identify the purpose of the %SYSFUNC function.
- Examine SAS language functions versus macro language functions.
- Become familiar with %SYSFUNC syntax and usage.

7

In this first section, we'll begin by identifying the purpose of %SYSFUNC. Then we'll have a general discussion of SAS language functions versus macro functions in general. Finally we'll cover some of the details of the syntax and usage of %SYSFUNC.

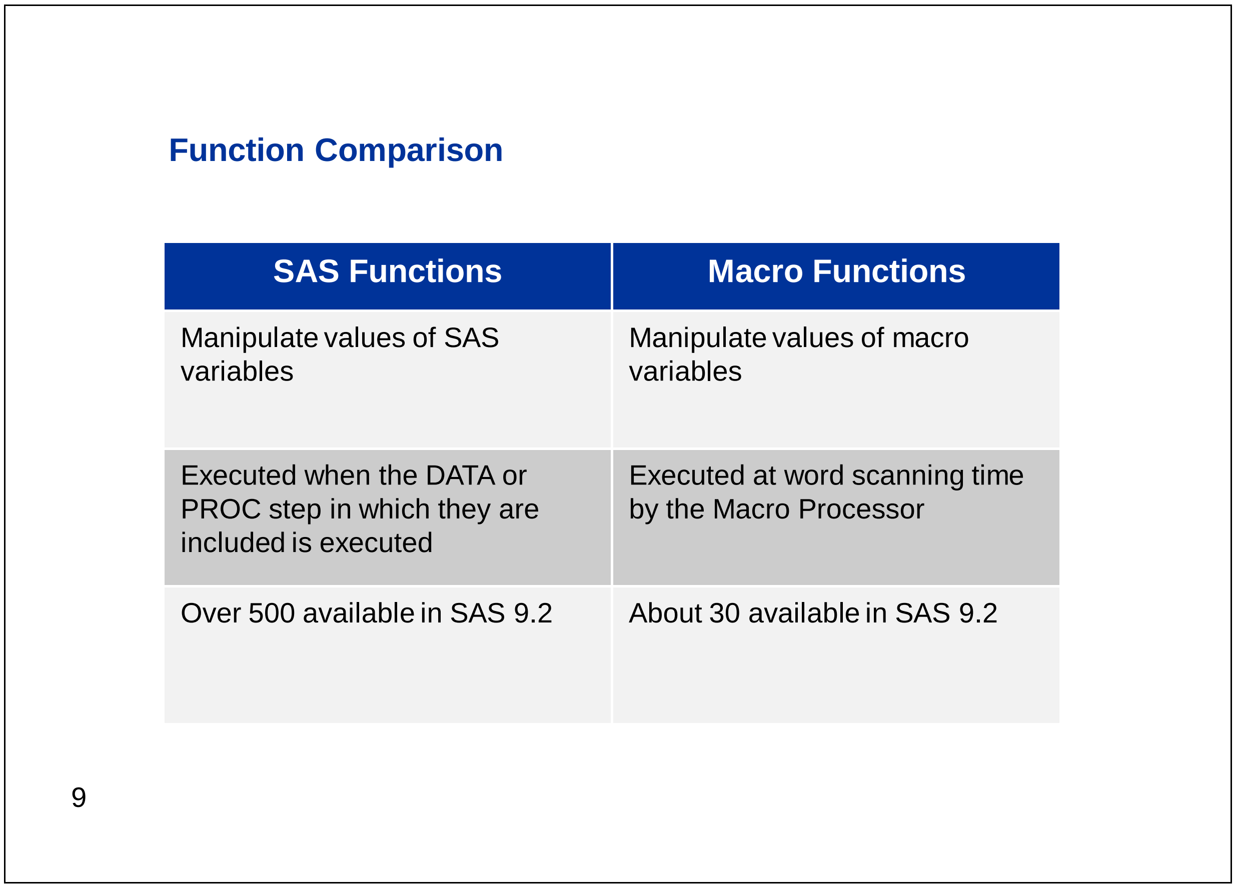


The %SYSFUNC Function

The %SYSFUNC function enables the macro facility to execute SAS language functions or user-written functions.

8

What is the purpose of %SYSFUNC? Well, it allows you to take advantage of SAS functions from within the macro facility. Those functions can be functions supplied by SAS, or they can be functions that you write. In order to understand why that’s so valuable, it’s helpful to review some concepts regarding SAS functions versus macro functions.

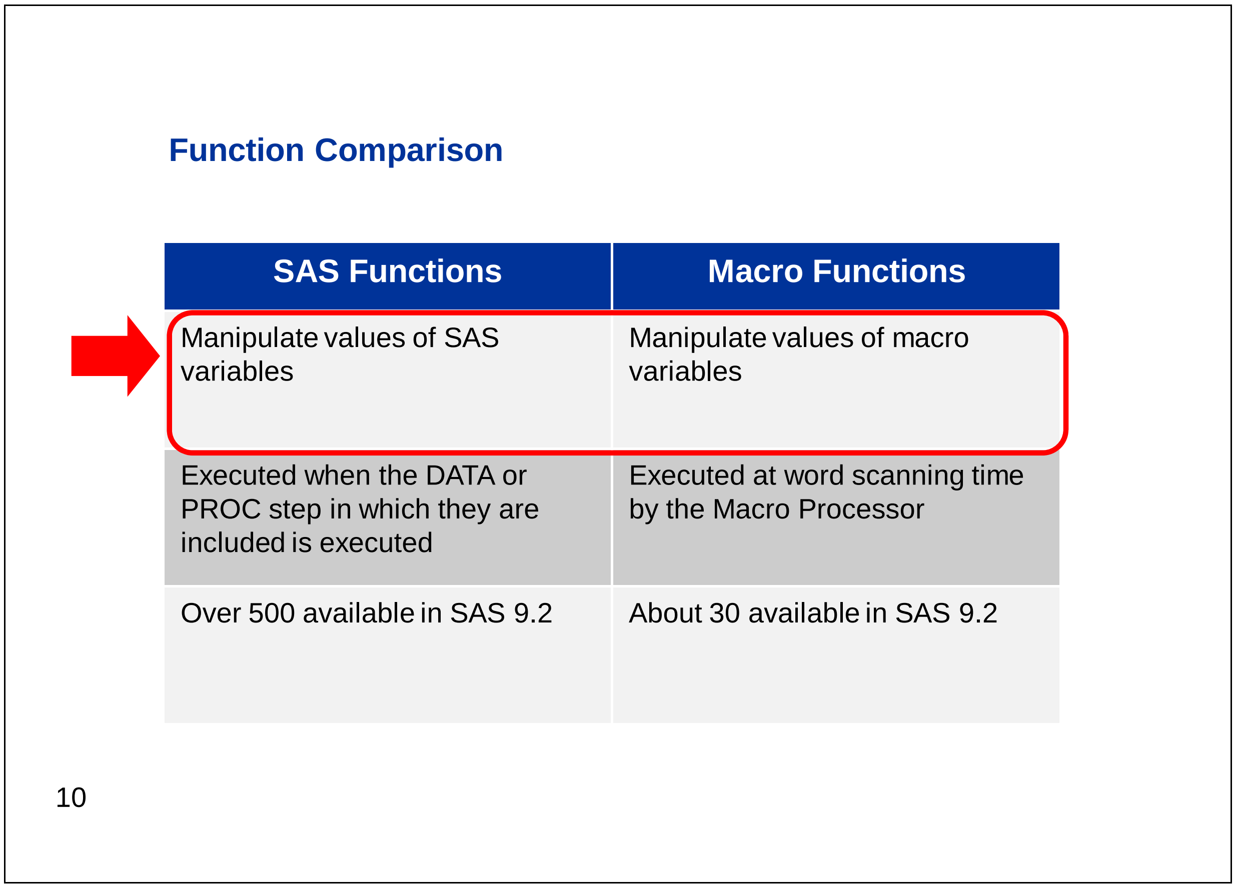


Function Comparison

SAS Functions	Macro Functions
Manipulate values of SAS variables	Manipulate values of macro variables
Executed when the DATA or PROC step in which they are included is executed	Executed at word scanning time by the Macro Processor
Over 500 available in SAS 9.2	About 30 available in SAS 9.2

9

This table highlights some of the key differences between SAS functions and macro functions. Let’s go through these differences one by one.

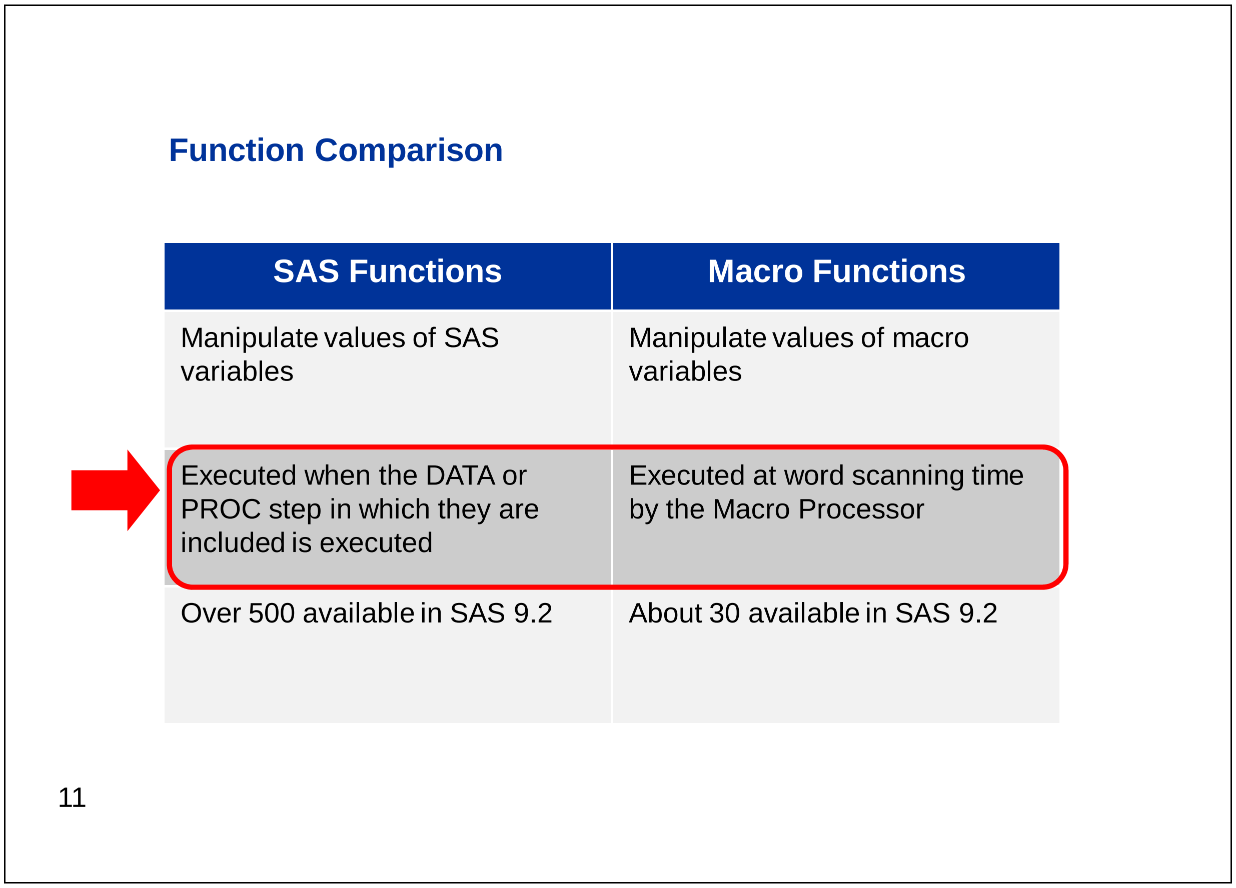


Function Comparison

SAS Functions	Macro Functions
Manipulate values of SAS variables	Manipulate values of macro variables
Executed when the DATA or PROC step in which they are included is executed	Executed at word scanning time by the Macro Processor
Over 500 available in SAS 9.2	About 30 available in SAS 9.2

10

First of all, SAS functions are used to create or manipulate SAS variable values. Macro functions, on the other hand, are typically used to manipulate values of macro variables.

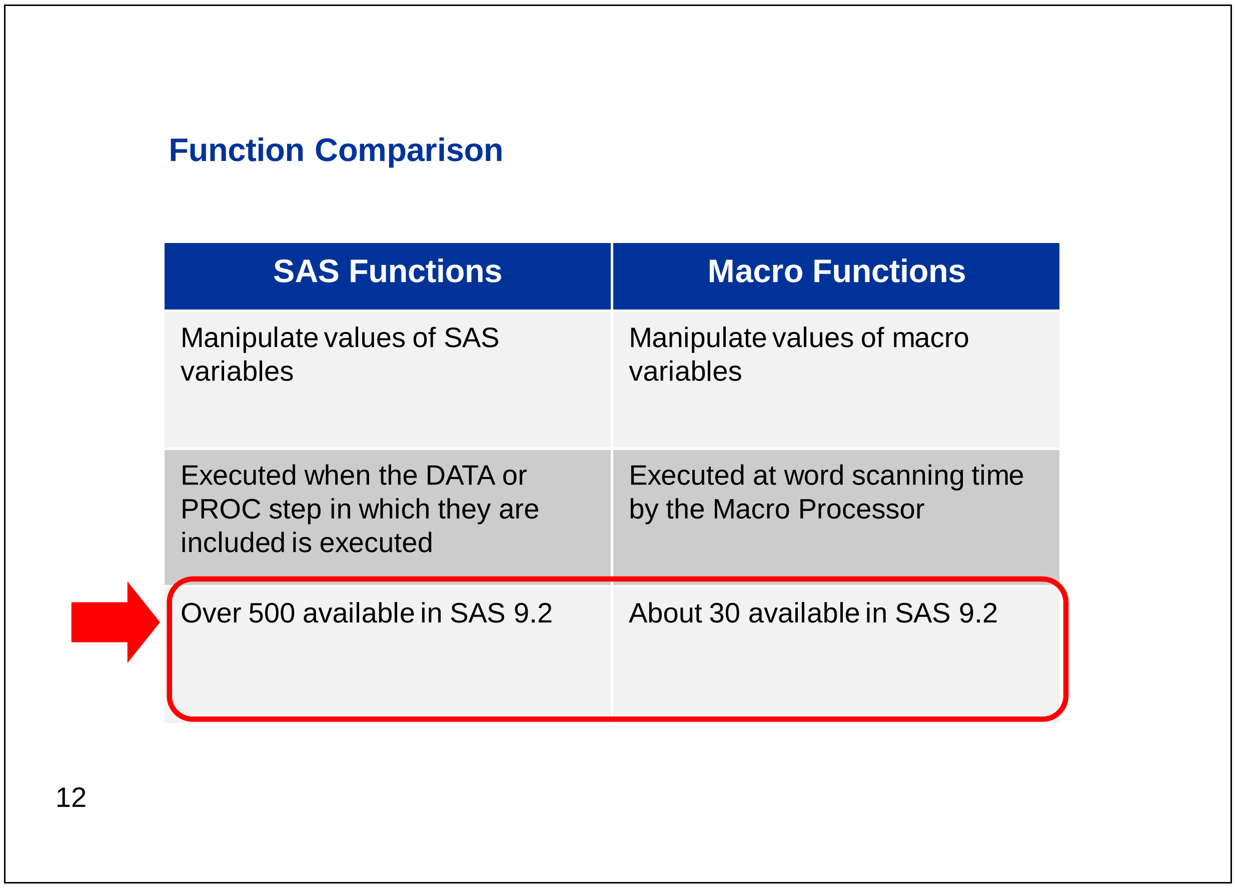


Function Comparison

SAS Functions	Macro Functions
Manipulate values of SAS variables	Manipulate values of macro variables
Executed when the DATA or PROC step in which they are included is executed	Executed at word scanning time by the Macro Processor
Over 500 available in SAS 9.2	About 30 available in SAS 9.2

11

A key difference between SAS functions and macro functions has to do with timing. SAS functions are executed when the DATA or PROC step in which they are included is executed. Macro functions are processed earlier, at word-scanning time. If your program has a step that includes both a SAS language function and a macro function, the macro function happens first. We’ll review that in a little more detail shortly.

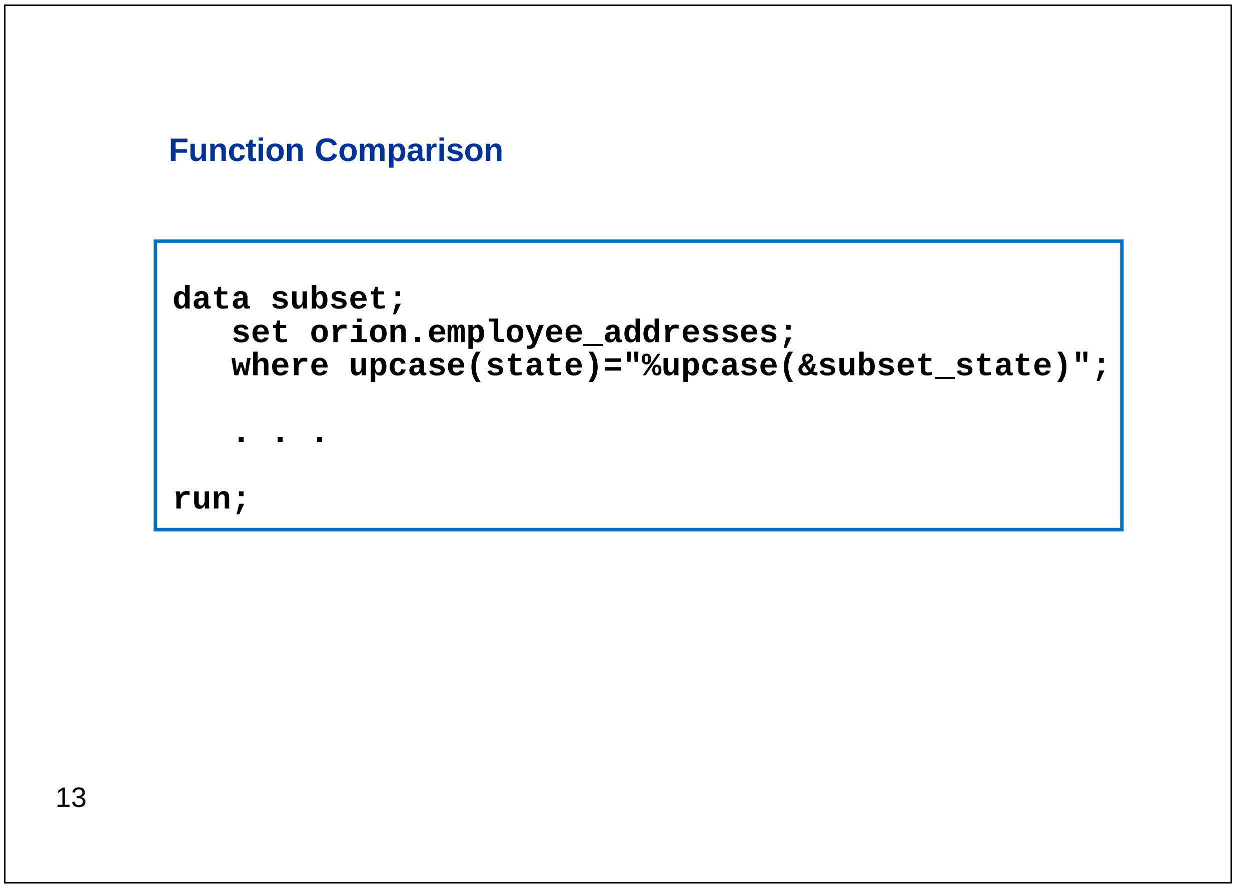


Function Comparison

SAS Functions	Macro Functions
Manipulate values of SAS variables	Manipulate values of macro variables
Executed when the DATA or PROC step in which they are included is executed	Executed at word scanning time by the Macro Processor
Over 500 available in SAS 9.2	About 30 available in SAS 9.2

12

The last difference here is the number of functions available. In SAS 9.2, there are over 500 SAS functions included, and you can also add to those by creating your own functions. In contrast, there are about 30 macro functions. Now, these 30 macro functions include several that mimic the functionality of a similar SAS language function. You’ll probably find that the available macro functions address many of the key capabilities that you need when manipulating macro variables, but probably not all.

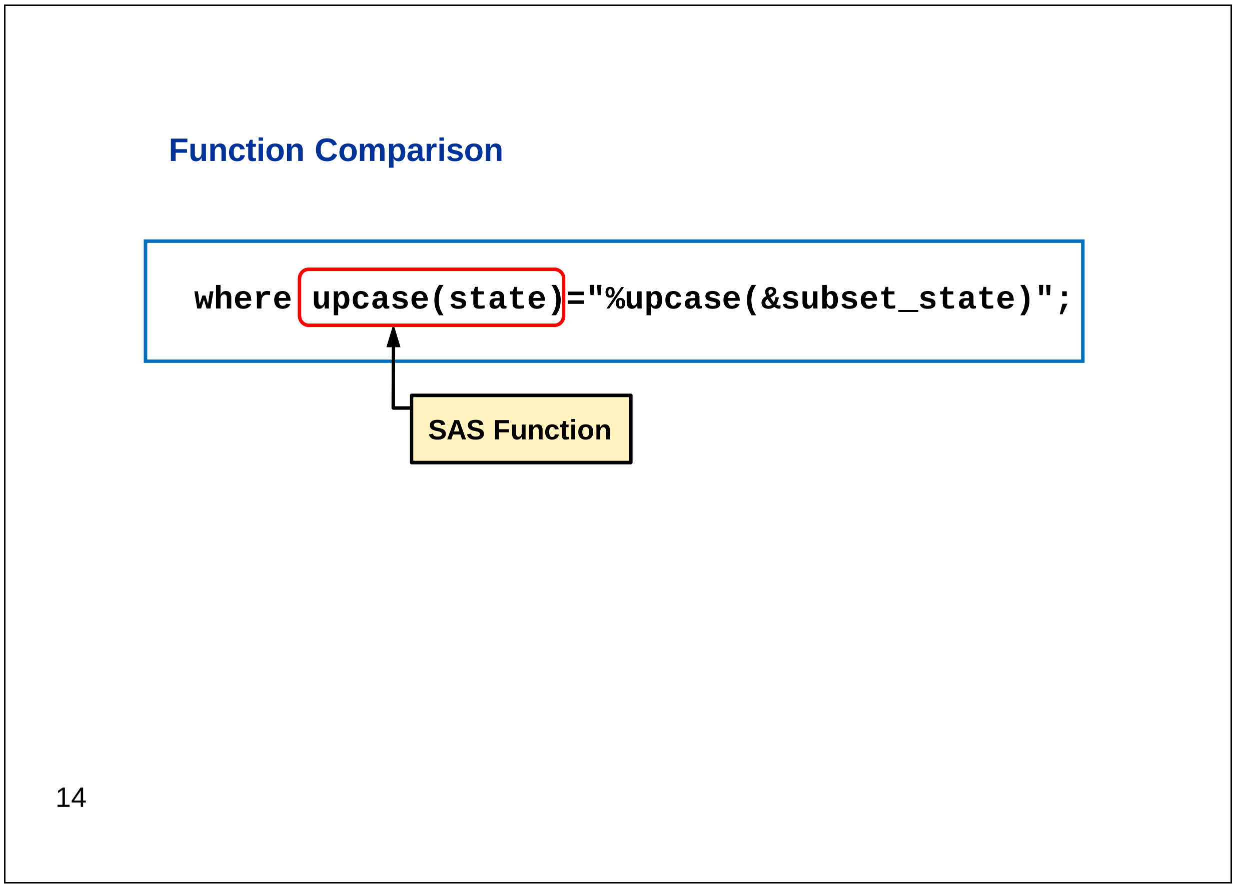


Function Comparison

```
data subset;
   set orion.employee_addresses;
   where upcase(state)="%upcase(&subset_state)";

   . . .

run;
```

13

Let’s take a quick look at the differences between SAS language functions and macro functions in action. Here we have a DATA step in which the WHERE statement includes both a SAS language function and a macro function. Let’s talk about the differences.

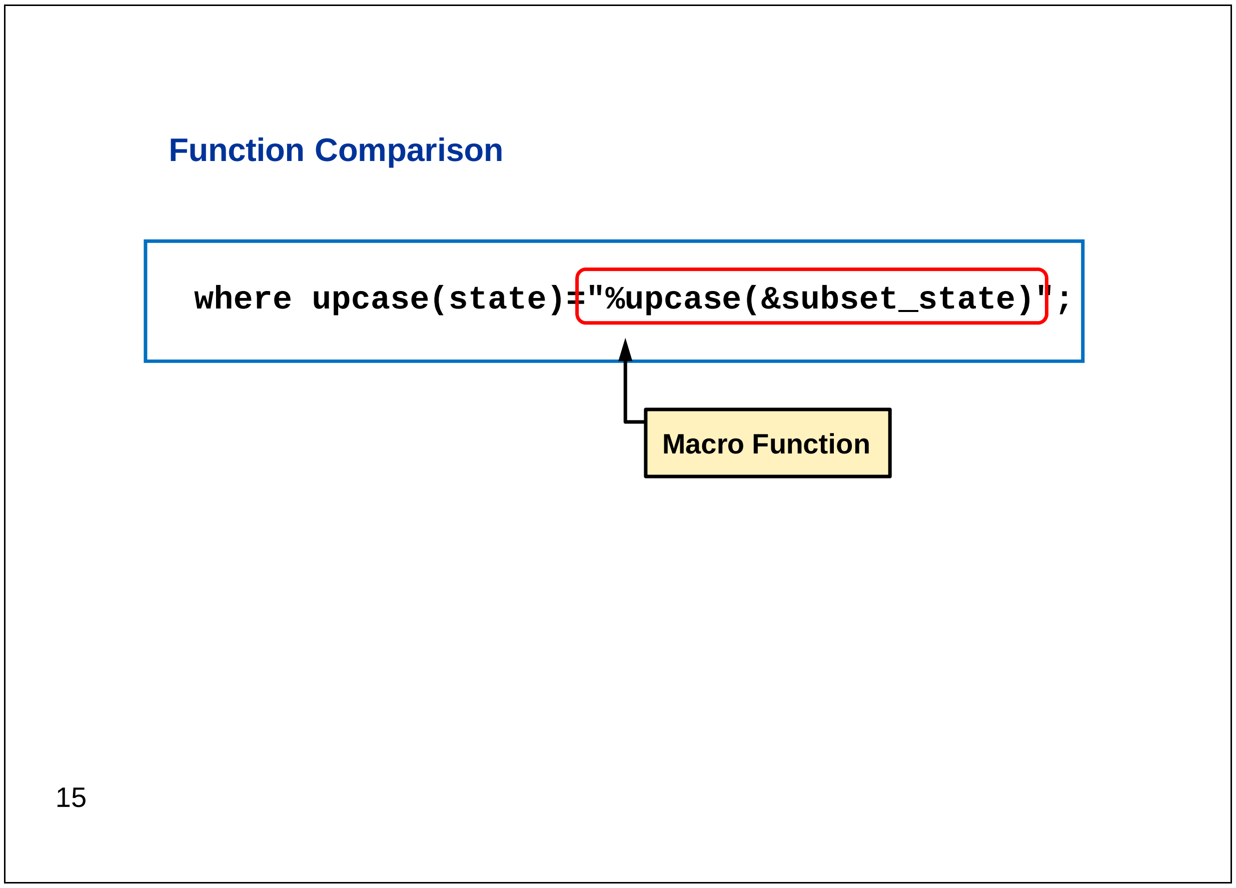


Function Comparison

```
where upcase(state)="%upcase(&subset_state)";
```

SAS Function
14

Let’s drill into the WHERE statement. On the left side of the expression, we’re using the SAS language function UPCASE. The argument to this function is the SAS variable **`state`**.



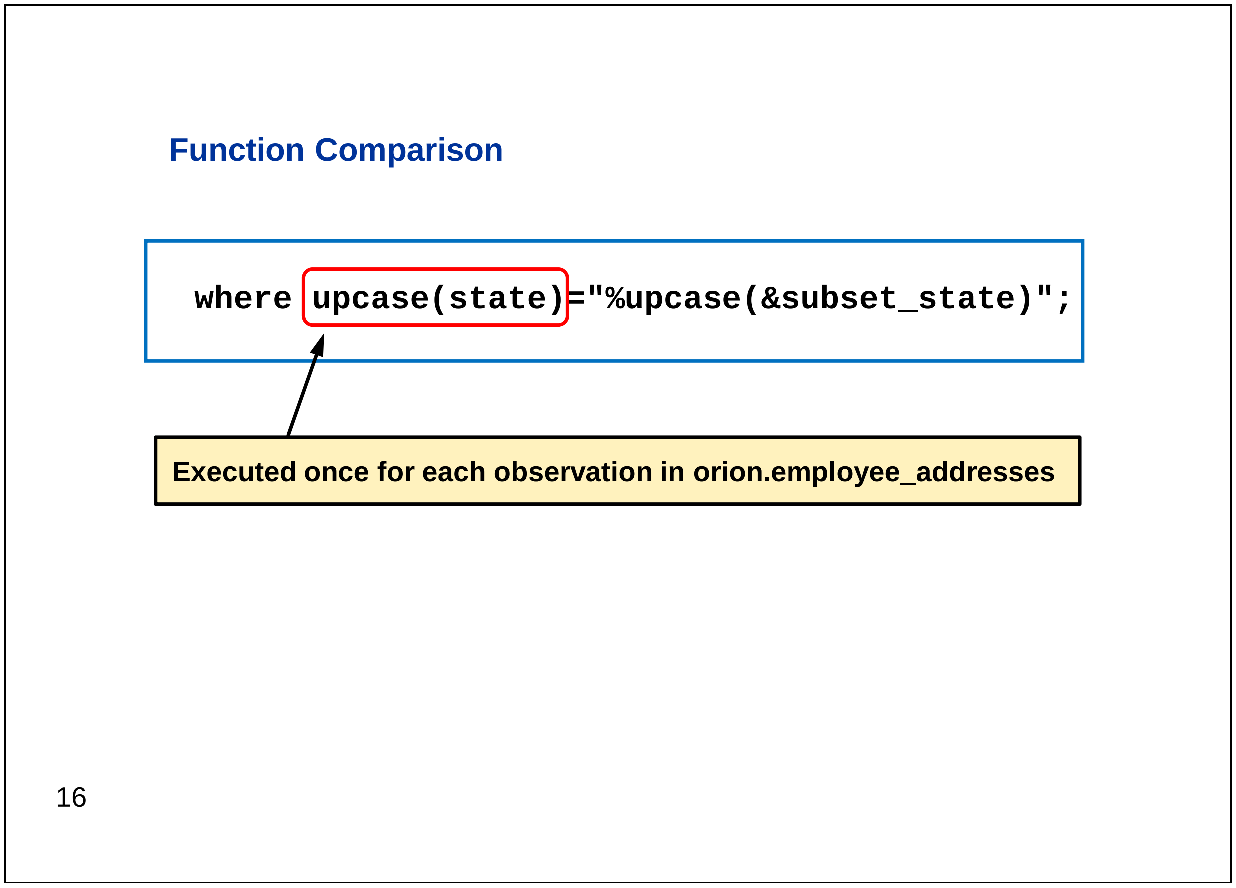
Function Comparison

```
where upcase(state)="%upcase(&subset_state)";
```

Macro Function

15

On the right side of the expression, we’re using the macro function %UPCASE. Its argument is the macro variable &SUBSET_STATE.



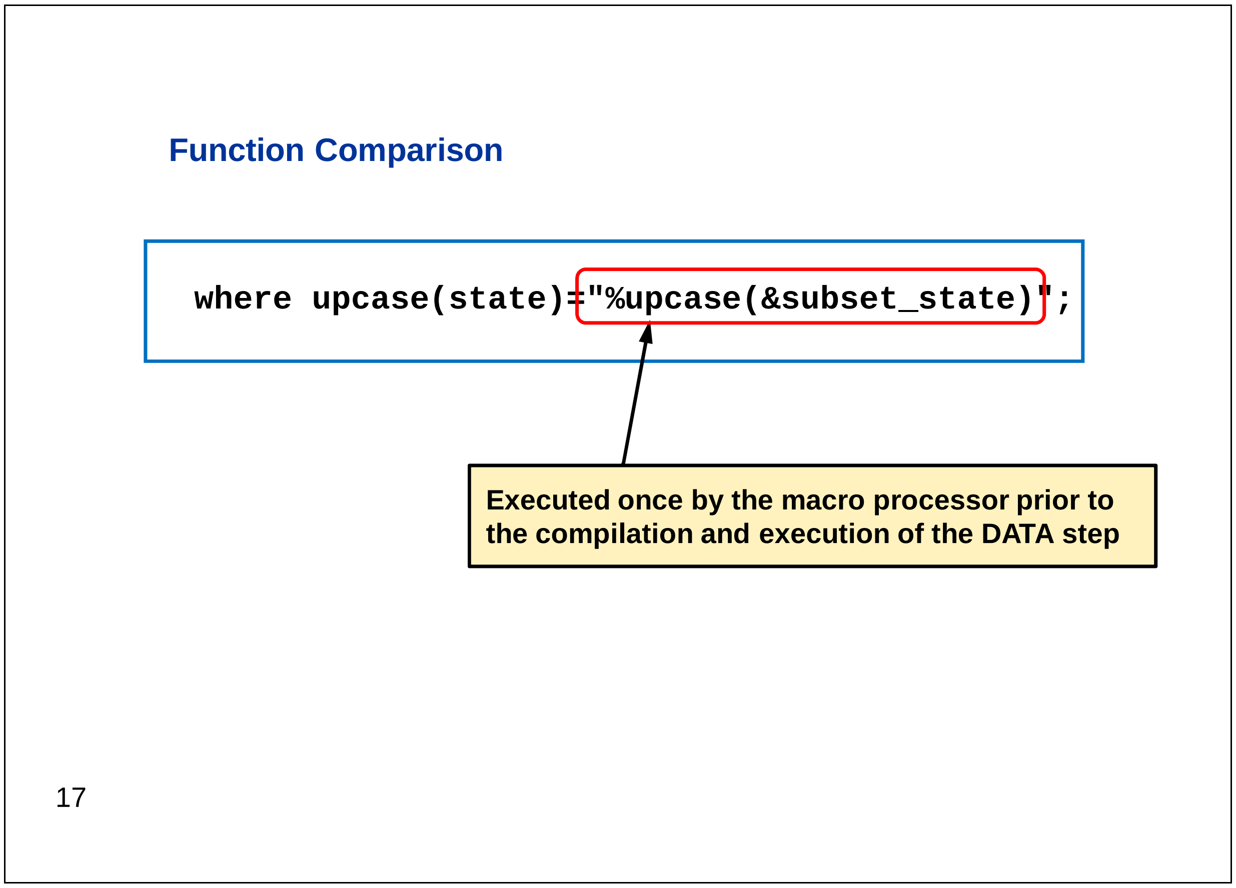
Function Comparison

```
where upcase(state)="%upcase(&subset_state)";
```

Executed once for each observation in orion.employee_addresses

16

The SAS language function will be executed when the WHERE statement is executed. It will be processed once for each observation in the source data set to determine whether that observation will be read.

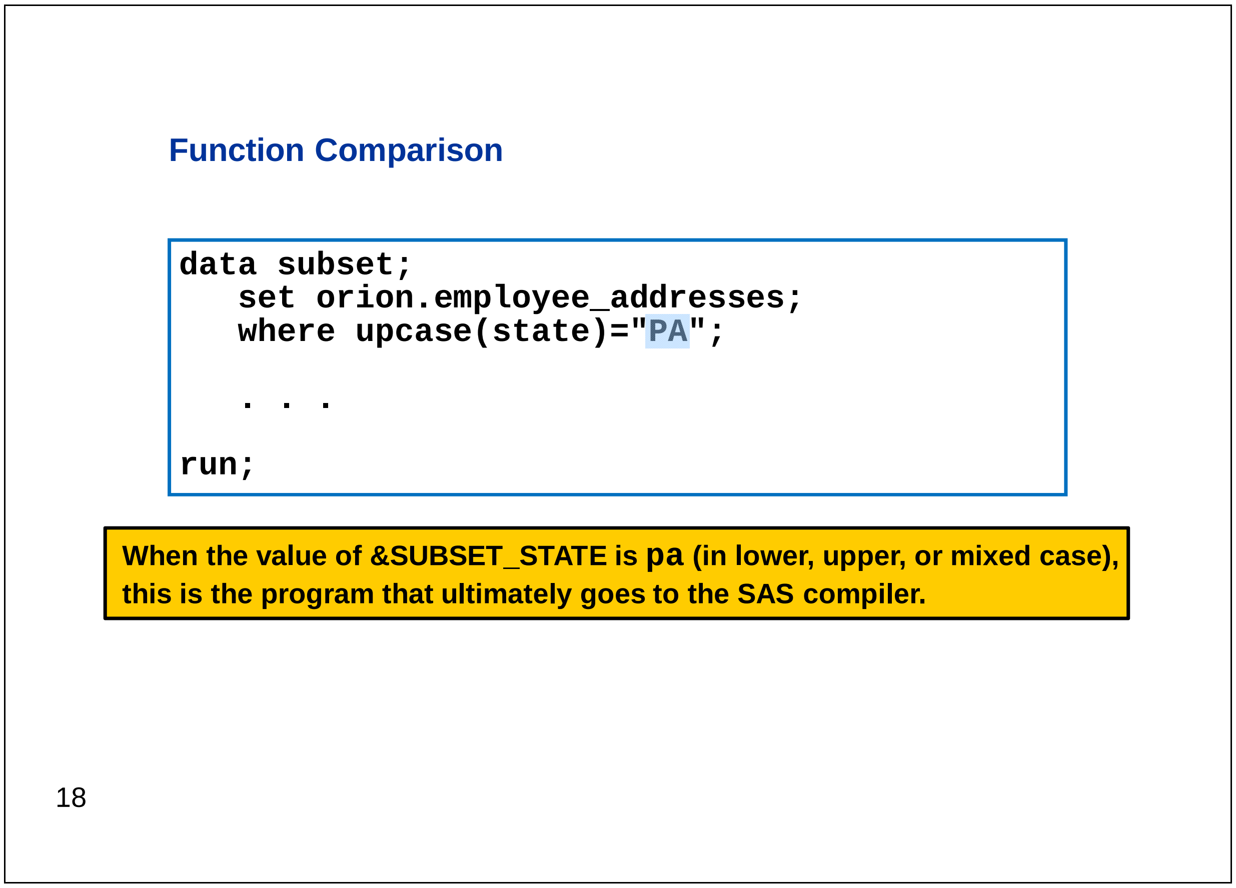


Function Comparison

```
where upcase(state)="%upcase(&subset_state)";
```

Executed once by the macro processor prior to the compilation and execution of the DATA step
17

The macro function will be executed by the macro processor at word-scanning time. This happens before the compilation and execution of the DATA step. It will be executed only one time. Suppose the value of the macro variable &SUBSET_STATE is `pa`, the code for Pennsylvania, in lowercase. When the program is submitted, the word scanner sends the %UPCASE function to be processed by the macro processor. The macro processor converts the value of &SUBSET_STATE to uppercase and returns it to the input stack.



Function Comparison

```
data subset;
   set orion.employee_addresses;
   where upcase(state)="PA";

   . . .

run;
```

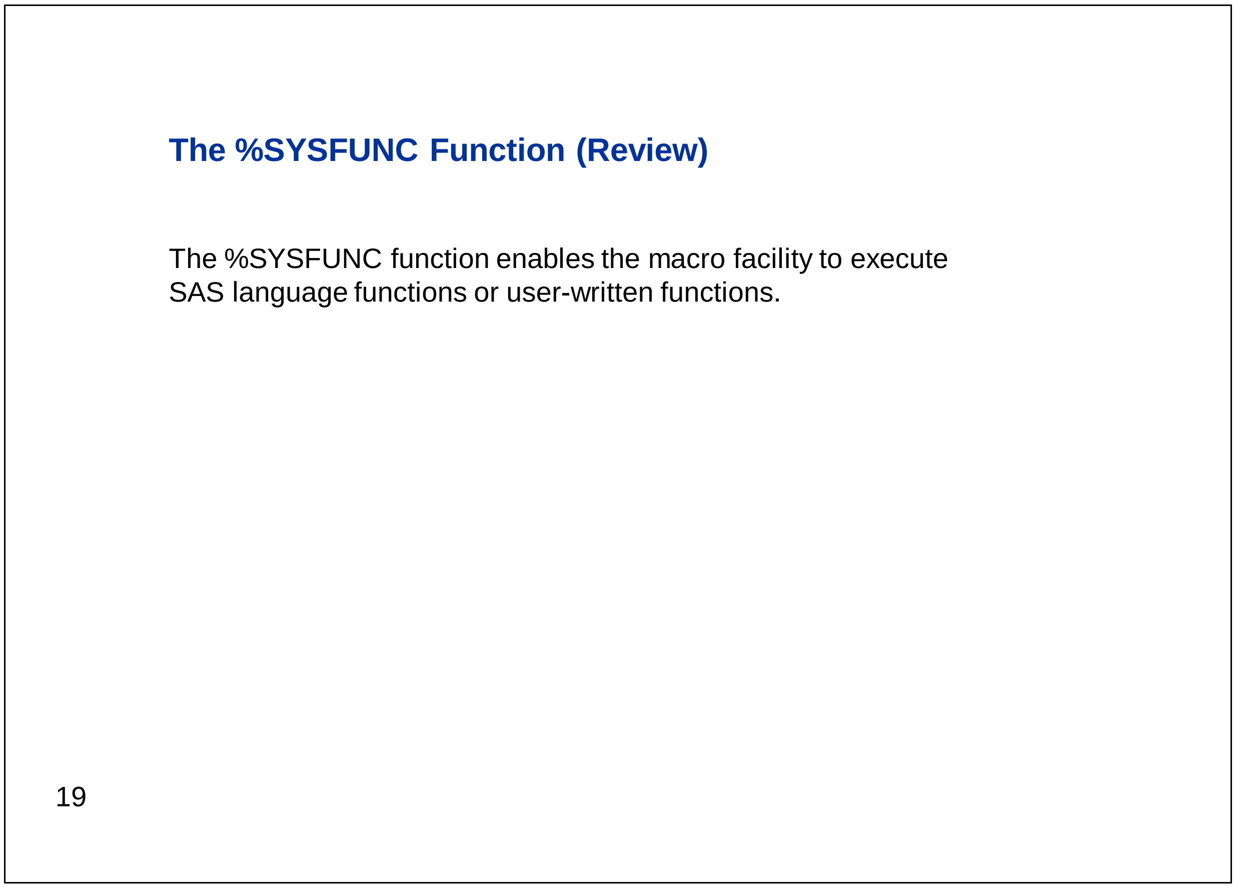
When the value of &SUBSET_STATE is pa (in lower, upper, or mixed case), this is the program that ultimately goes to the SAS compiler.

18

This is the program that ultimately goes to the SAS compiler. The right side of the WHERE expression is only a character constant, that is, the state code for Pennsylvania in uppercase.

Now in this case, it happens that both the SAS and macro function are doing the same task, specifically converting to uppercase, but that's not the point of this example. We could use a different SAS function or a different macro function (such as %SYSFUNC) in an expression like this one. The timing would be the same. Macro functions are executed prior to compilation, and DATA step functions are executed during DATA step execution.

Also, converting to uppercase is clearly not a situation where we need %SYSFUNC because we already have a macro function that handles this task. %UPCASE is one of the macro functions that I referred to that mimic the capabilities of a SAS language function. As we proceed we'll see several cases where %SYSFUNC is needed.

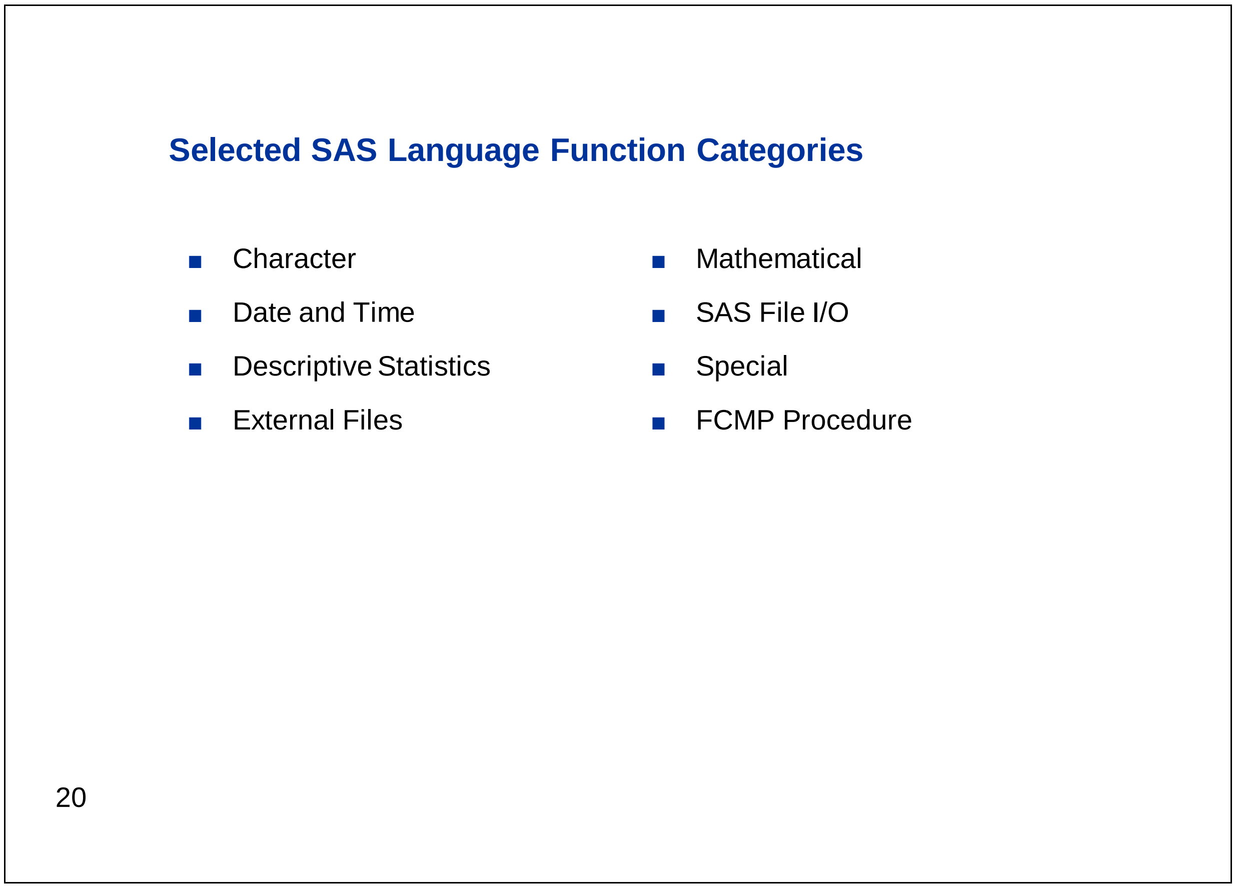


The %SYSFUNC Function (Review)

The %SYSFUNC function enables the macro facility to execute SAS language functions or user-written functions.

19

So let’s get back to %SYSFUNC. We already highlighted the fact that there are many more SAS functions than macro functions, and you can create your own functions to supplement those provided by SAS. So, %SYSFUNC gives you access to a very broad range of capabilities to manipulate macro variable values.



Selected SAS Language Function Categories

- Character
- Date and Time
- Descriptive Statistics
- External Files
- Mathematical
- SAS File I/O
- Special
- FCMP Procedure

20

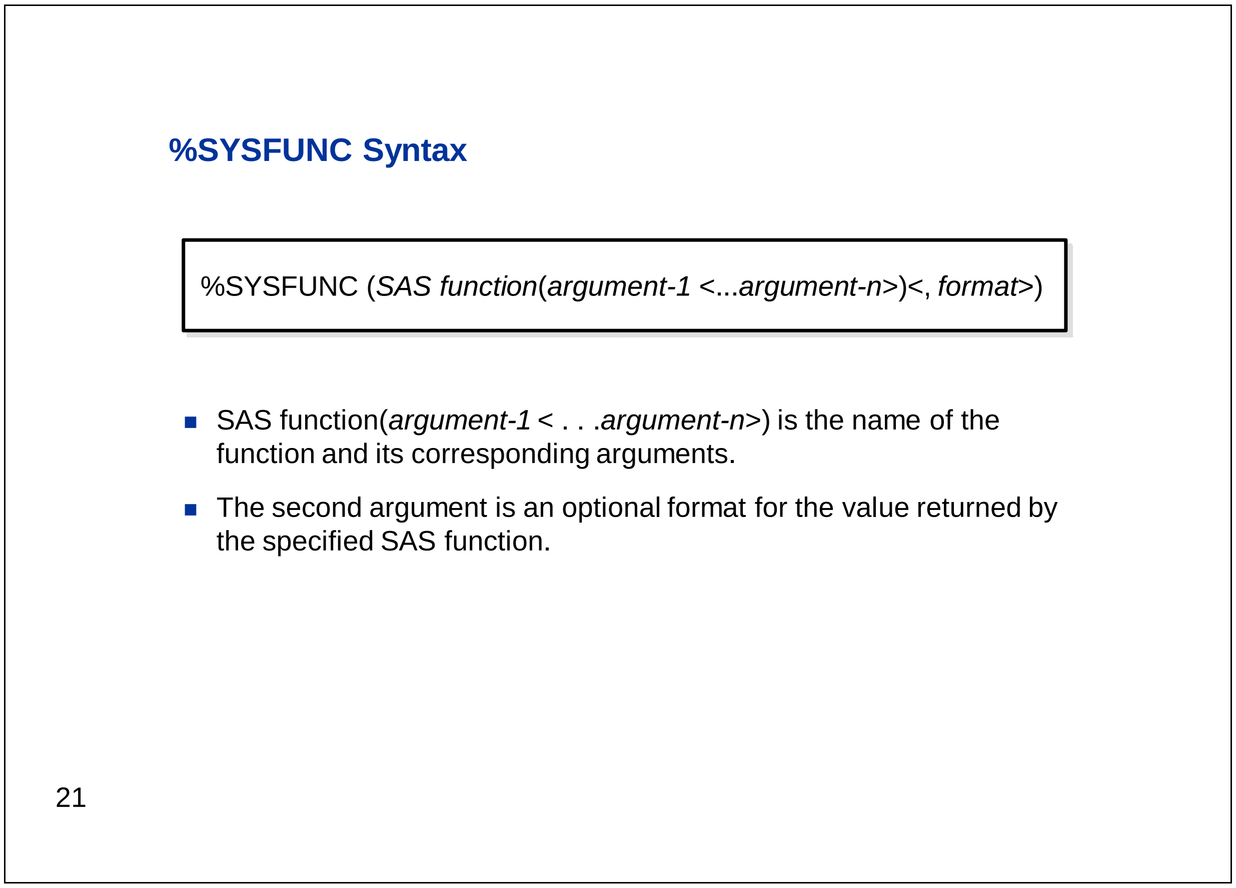
We talked about the number of SAS language functions available. Let's talk about some of the categories of functions. Now this is not a full list of all function categories, just some that you might be most likely to use with %SYSFUNC.

The first category shown here is Character. There are a great many character functions available in SAS. You're very likely to find these handy to manipulate text in the macro facility.

You might use Date and Time functions with %SYSFUNC to create or manipulate macro variables that contain a calendar date or clock time.

There are External File functions that you can use to return information about or manipulate external files such as a spreadsheet. You might use these functions with %SYSFUNC in a program that accesses these files.

I've mentioned a couple of times that you can use %SYSFUNC to access functions that you've written yourself. The FCMP procedure, which is new to SAS 9.2, provides the capability to write your own functions.



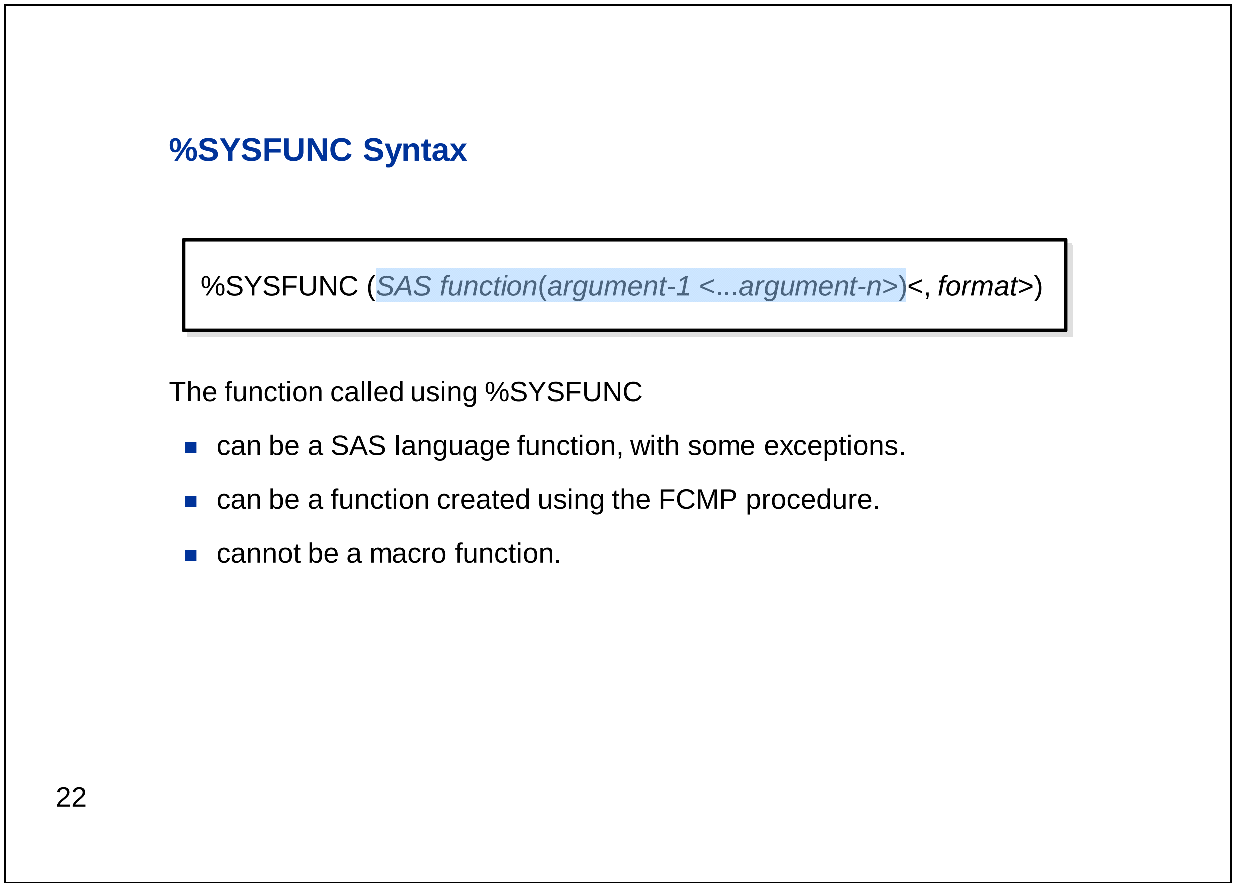
%SYSFUNC Syntax

```
%SYSFUNC (SAS function(argument-1 <...argument-n>)<, format>)
```

- SAS function(*argument-1* < . . .*argument-n*>) is the name of the function and its corresponding arguments.
- The second argument is an optional format for the value returned by the specified SAS function.

21

Let’s take a look at the syntax of %SYSFUNC. There are only two arguments. The first is the SAS function that you want to access, specified with its own arguments as appropriate. The second argument to %SYSFUNC, which is optional, specifies a format to be applied to the value calculated by the function before it is returned to the macro facility.



%SYSFUNC Syntax

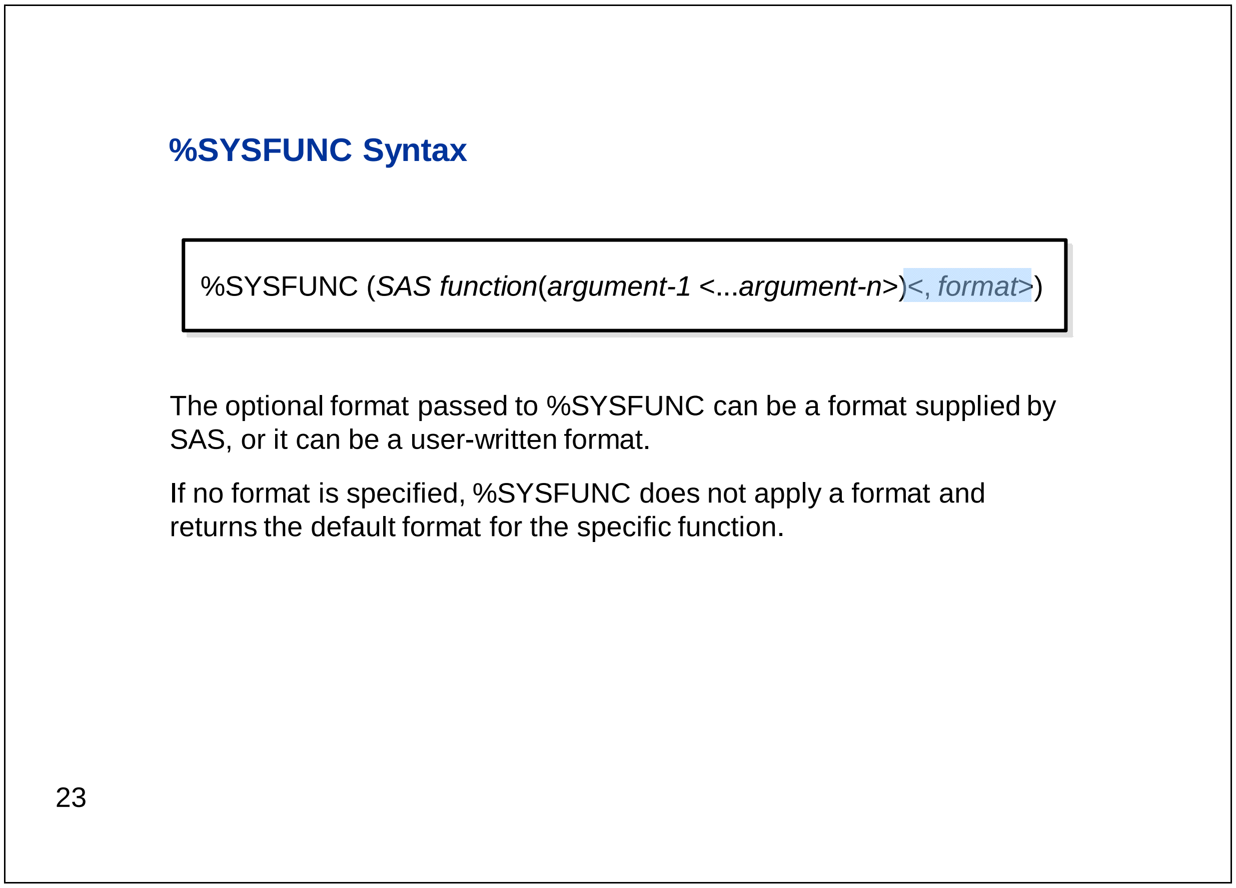
%SYSFUNC (*SAS function*(*argument-1* <*...argument-n*>)<, *format*>)

The function called using %SYSFUNC

- can be a SAS language function, with some exceptions.
- can be a function created using the FCMP procedure.
- cannot be a macro function.

22

The function that you specify as the first argument can be a SAS function or it can be a function created using PROC FCMP. You can't specify a macro function here. Also, there are some SAS language functions that are not accessible using %SYSFUNC. For example, you can't use array functions. Arrays are valid only in the context of the DATA step, so they're not available here. There's a full list of the SAS 9.2 functions that can't be used with %SYSFUNC in the handout for this e-lecture.



%SYSFUNC Syntax

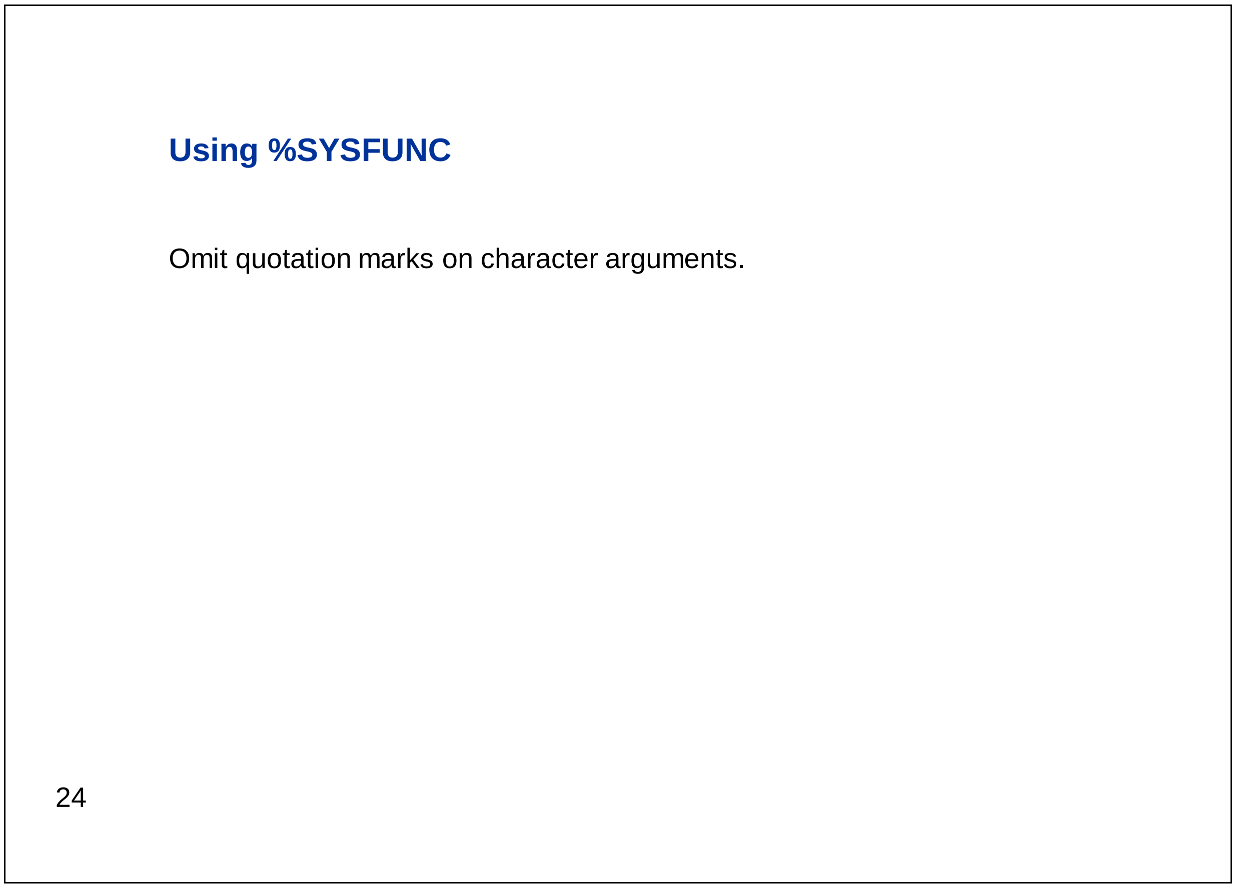
```
%SYSFUNC (SAS function(argument-1 <...argument-n>)<, format>)
```

The optional format passed to %SYSFUNC can be a format supplied by SAS, or it can be a user-written format.

If no format is specified, %SYSFUNC does not apply a format and returns the default format for the specific function.

23

The second argument to %SYSFUNC is a format, and it can be either a SAS format or one that you’ve written. It is optional, though in many cases you will want to specify a format. If you don’t, %SYSFUNC returns the value in the default format for the specific function.

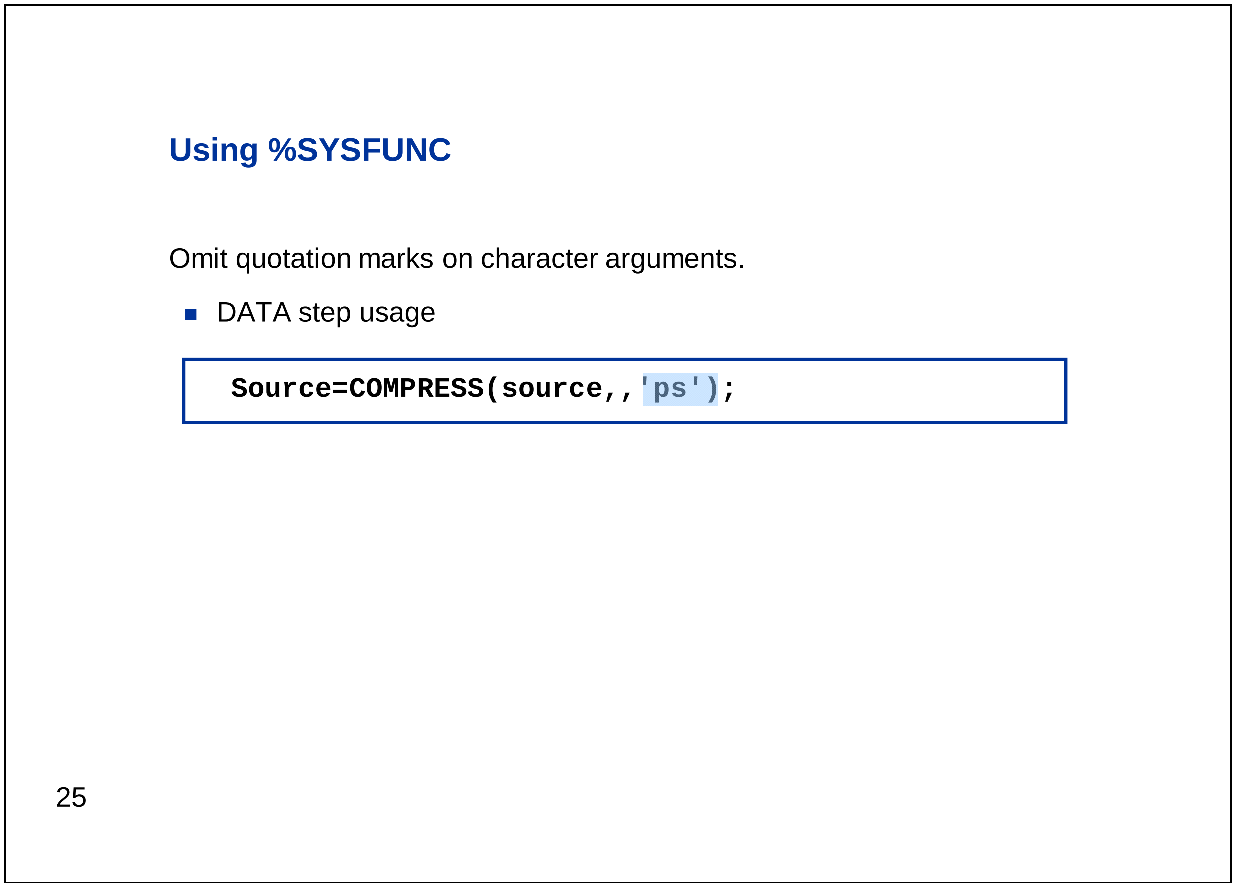


Using %SYSFUNC

Omit quotation marks on character arguments.

24

So we’ve seen the basic syntax of %SYSFUNC. Let’s talk about some of the considerations to remember when using it. The first issue to discuss is the use of quotation marks for character arguments. You can use %SYSFUNC with a function whose arguments would be quoted if it were used in a DATA step. However, because %SYSFUNC is a macro function, character arguments to the function being called are **not** enclosed in quotation marks.



Using %SYSFUNC

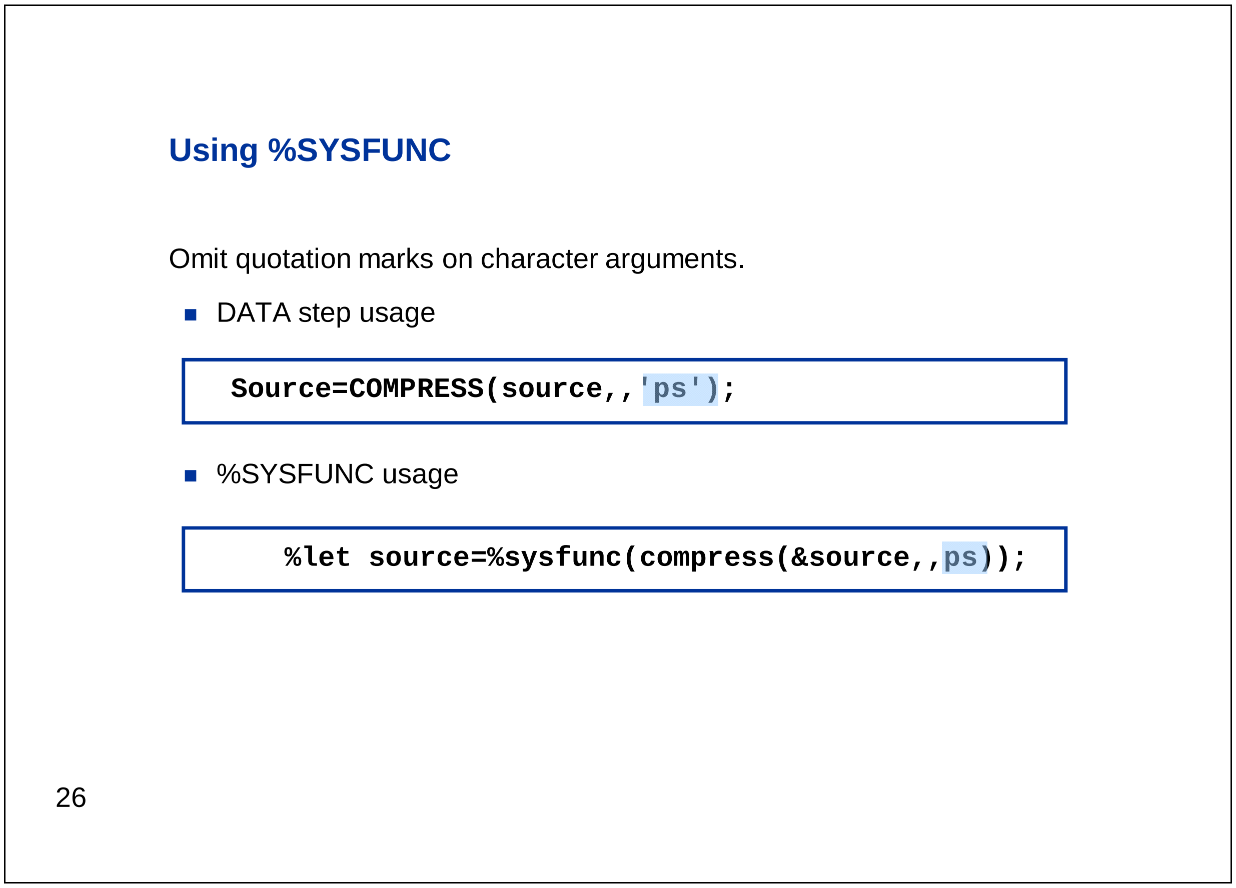
Omit quotation marks on character arguments.

- DATA step usage

```
Source=COMPRESS(source,,'ps');
```

25

Let’s look at the way character constant arguments are specified in the context of a DATA step. Here we’re using the COMPRESS function in an assignment statement. If you haven’t used it before, this function removes specified characters from the string that’s named in the first argument. You can list specific individual characters to remove as the second argument to the function. You can also use various modifiers in the third argument to remove categories of characters. In this instance, we’re using the function with the modifiers ”p,” which removes punctuation marks, and ”s,” which removes space characters such as spaces and tabs. In this situation, the list of modifiers is a character constant and must be quoted.



Using %SYSFUNC

Omit quotation marks on character arguments.

- DATA step usage

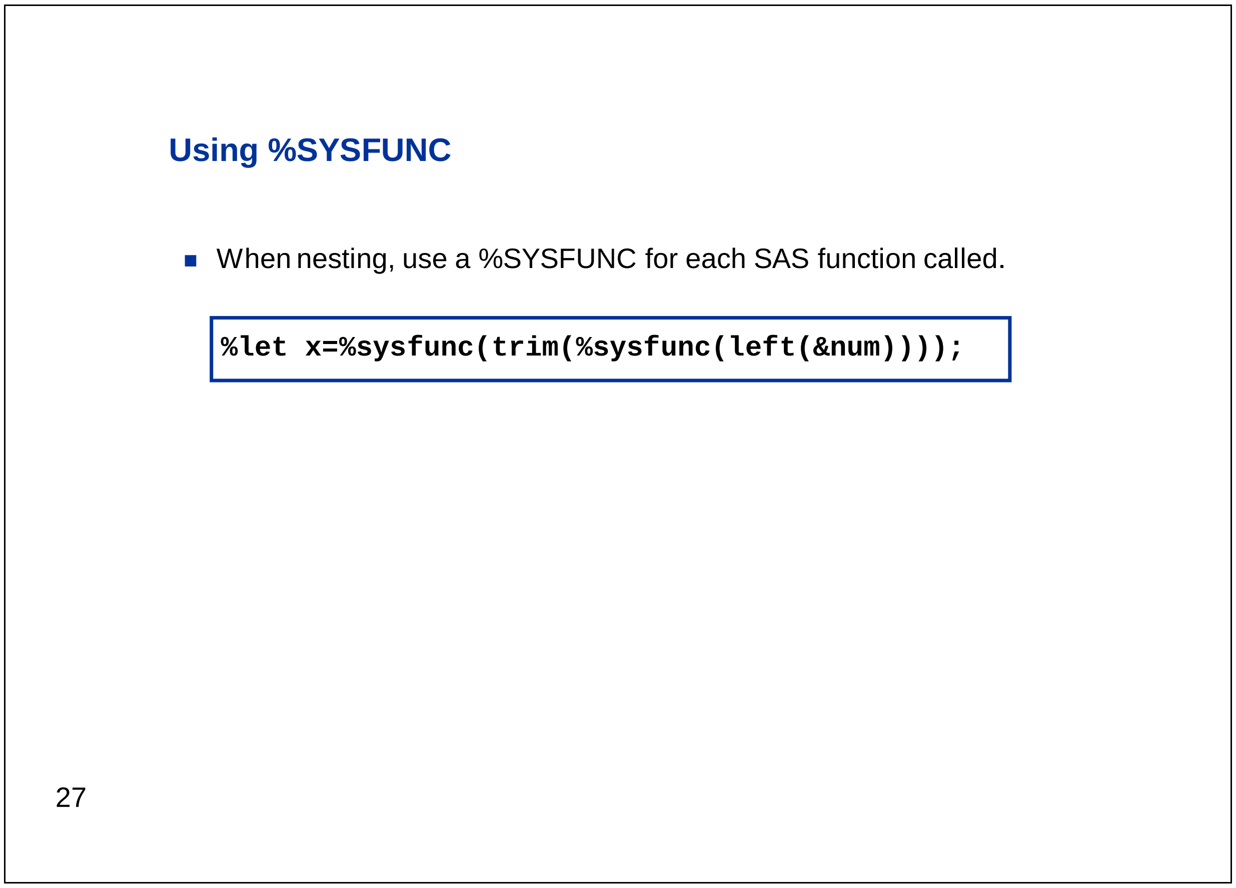
```
Source=COMPRESS(source,,'ps');
```

- %SYSFUNC usage

```
    %let source=%sysfunc(compress(&source,,ps));
```

26

Now let’s switch to the context of the macro facility. Here we’re using %SYSFUNC to access the COMPRESS function in a %LET statement. We’re using COMPRESS with the same modifiers, but here quotation marks are not used. Remember that the macro facility is a text processor, so there is no need to use quotation marks to differentiate text from variable names or mnemonics as is the case in the DATA step. Including them will likely cause a WARNING message in the SAS log at best, and might cause errors or unintended results.



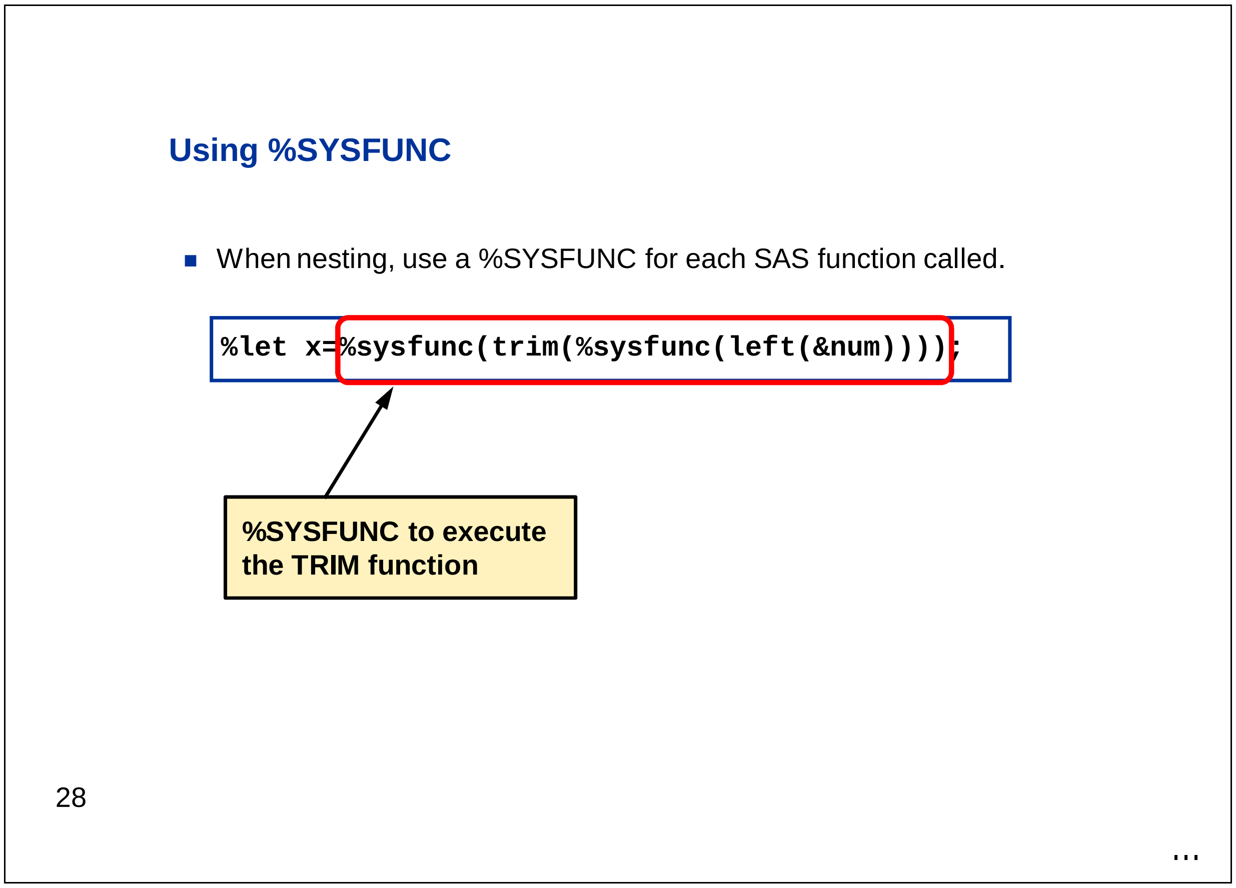
Using %SYSFUNC

- When nesting, use a %SYSFUNC for each SAS function called.

```
%let x=%sysfunc(trim(%sysfunc(left(&num))));
```

27

You might need to nest calls to multiple SAS functions. For example, here we’re passing the result of the LEFT function to the TRIM function. In this situation, you can’t use a single %SYSFUNC call. You need one for each SAS function to be invoked.



Using %SYSFUNC

- When nesting, use a %SYSFUNC for each SAS function called.

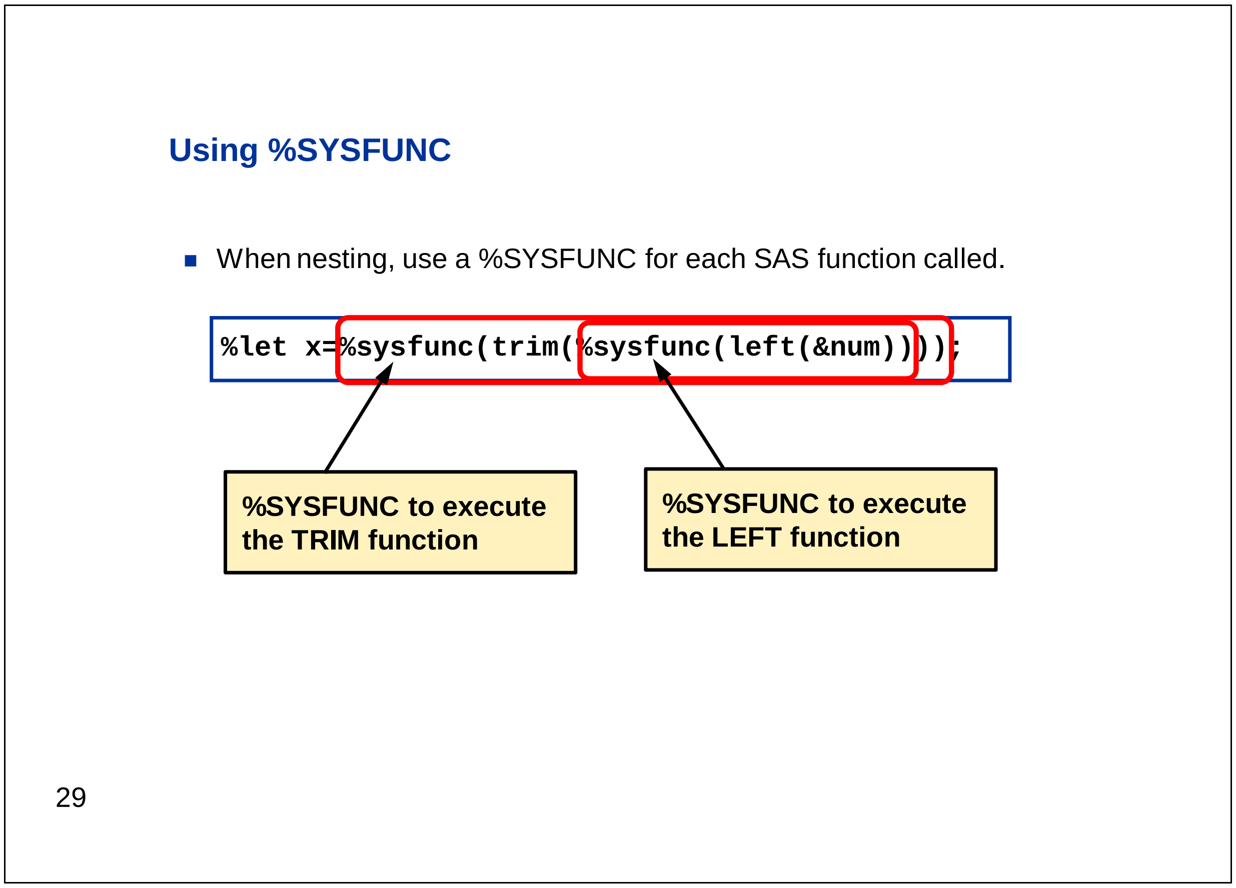
```
%let x=%sysfunc(trim(%sysfunc(left(&num))));
```

%SYSFUNC to execute the TRIM function

28

...

So here we see the %SYSFUNC that is used to access the TRIM function.



Using %SYSFUNC

- When nesting, use a %SYSFUNC for each SAS function called.

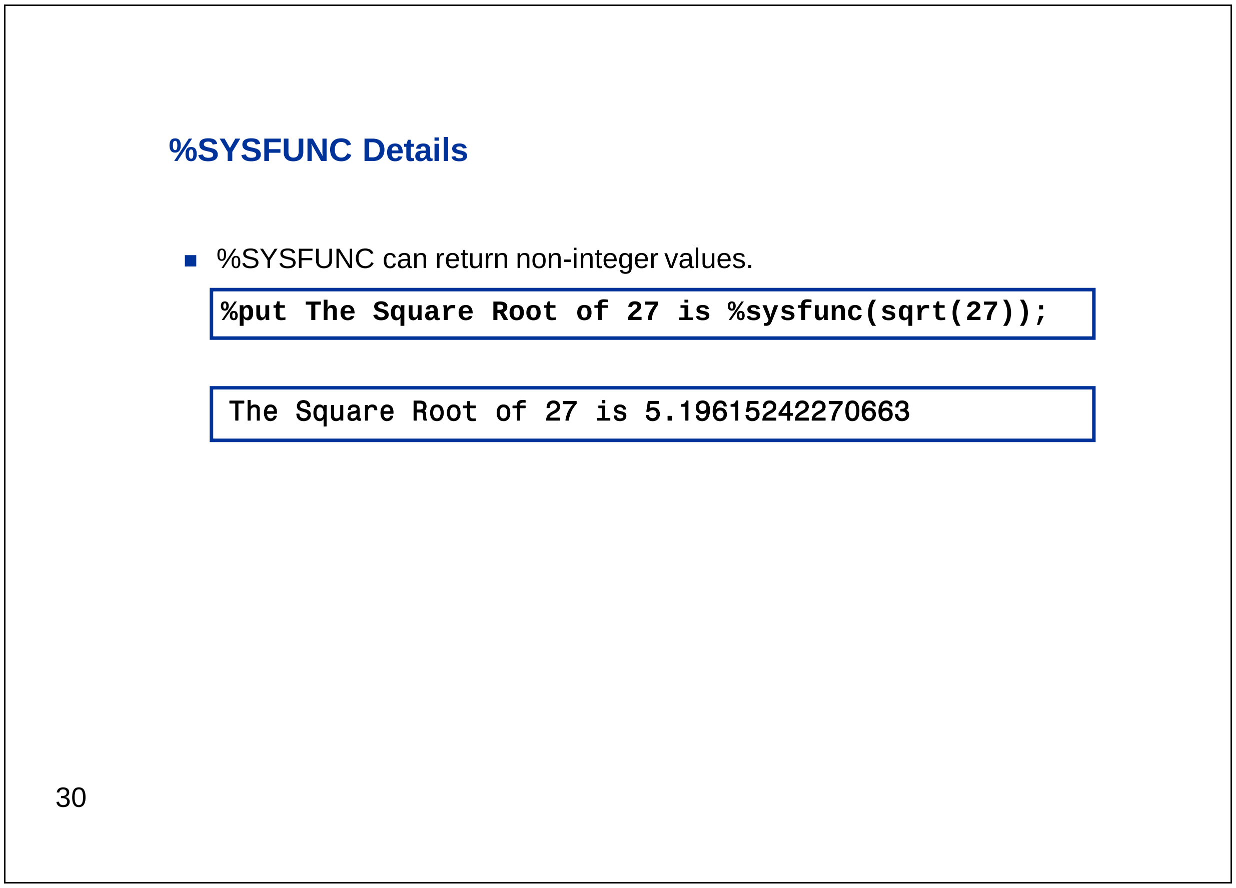
```
%let x=%sysfunc(trim(%sysfunc(left(&num))));
```

%SYSFUNC to execute the TRIM function

%SYSFUNC to execute the LEFT function

29

Nested within that is a second use of %SYSFUNC to access the LEFT function. So here we have two SAS functions to call, and therefore we need two references to %SYSFUNC. Notice that the innermost function, in this case LEFT, is called first and the results are passed to the TRIM function.



%SYSFUNC Details

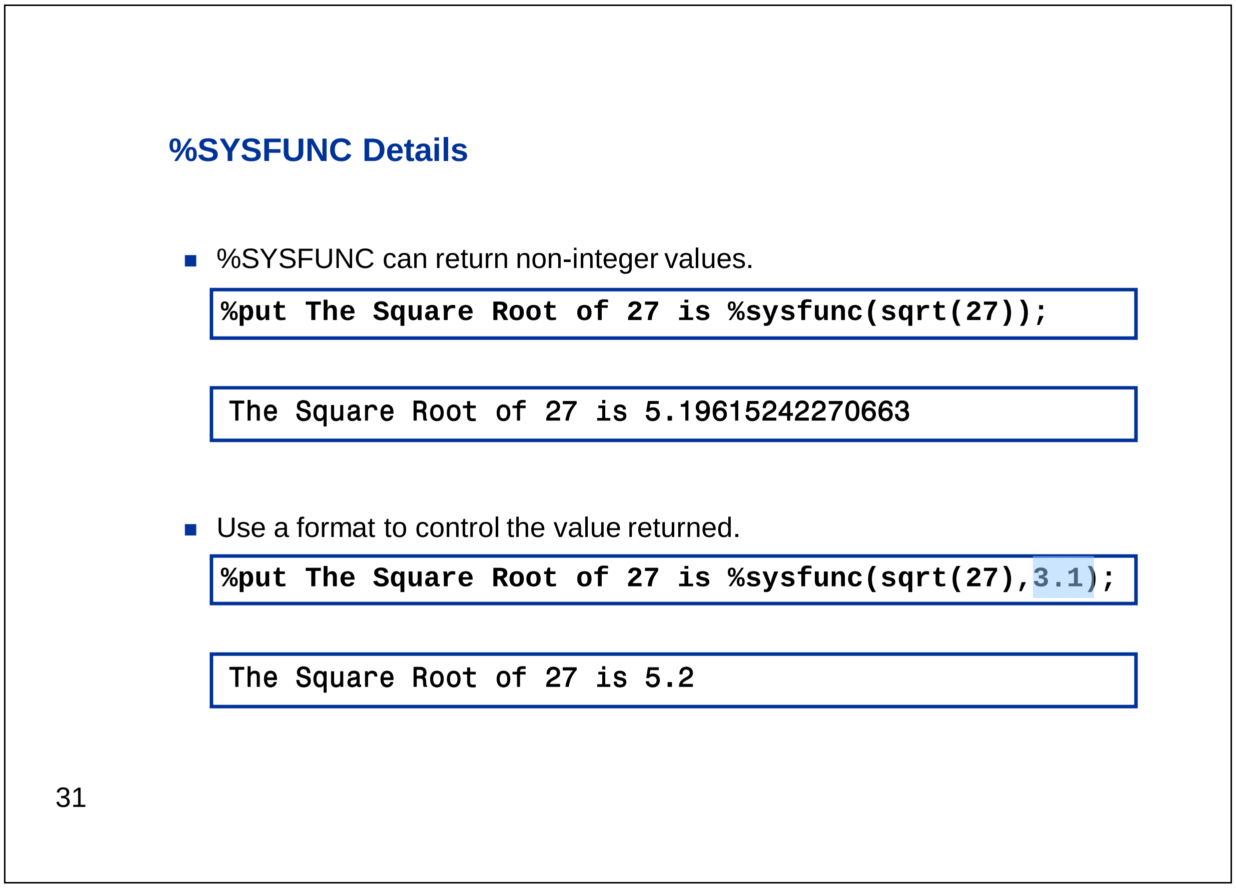
- %SYSFUNC can return non-integer values.

```
%put The Square Root of 27 is %sysfunc(sqrt(27));
```

```
The Square Root of 27 is 5.19615242270663
```

30

You might have observed that the macro facility will perform mathematical operations in some situations, but those operations typically return integer values only. However, in the case of %SYSFUNC, a floating point number will be returned if the function being called supports floating point numbers. Here we’re calculating the square root of a number, and we can see that the default value returned by the SQRT function is showing many places to the right of the decimal point. Now of course, to the macro facility this is only text, not an actual floating point number. However, we’re not limited to an integer representation only.



%SYSFUNC Details

- %SYSFUNC can return non-integer values.

```
%put The Square Root of 27 is %sysfunc(sqrt(27));
```

```
The Square Root of 27 is 5.19615242270663
```

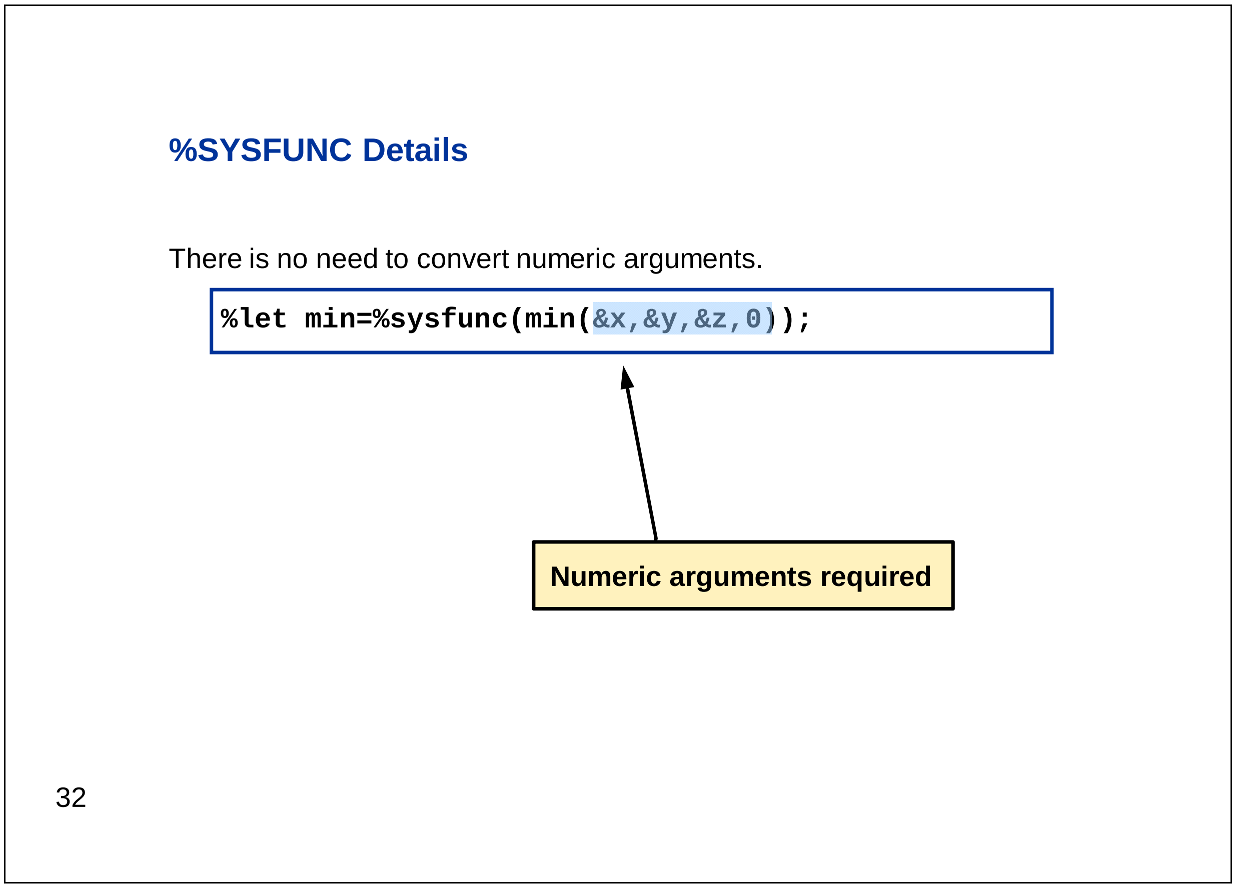
- Use a format to control the value returned.

```
%put The Square Root of 27 is %sysfunc(sqrt(27),3.1);
```

```
The Square Root of 27 is 5.2
```

31

This example illustrates a case where you might want to take advantage of the optional second argument to %SYSFUNC, that is, the format. In the second case shown on this slide, we’re using a format to control the number of decimal places returned.



%SYSFUNC Details

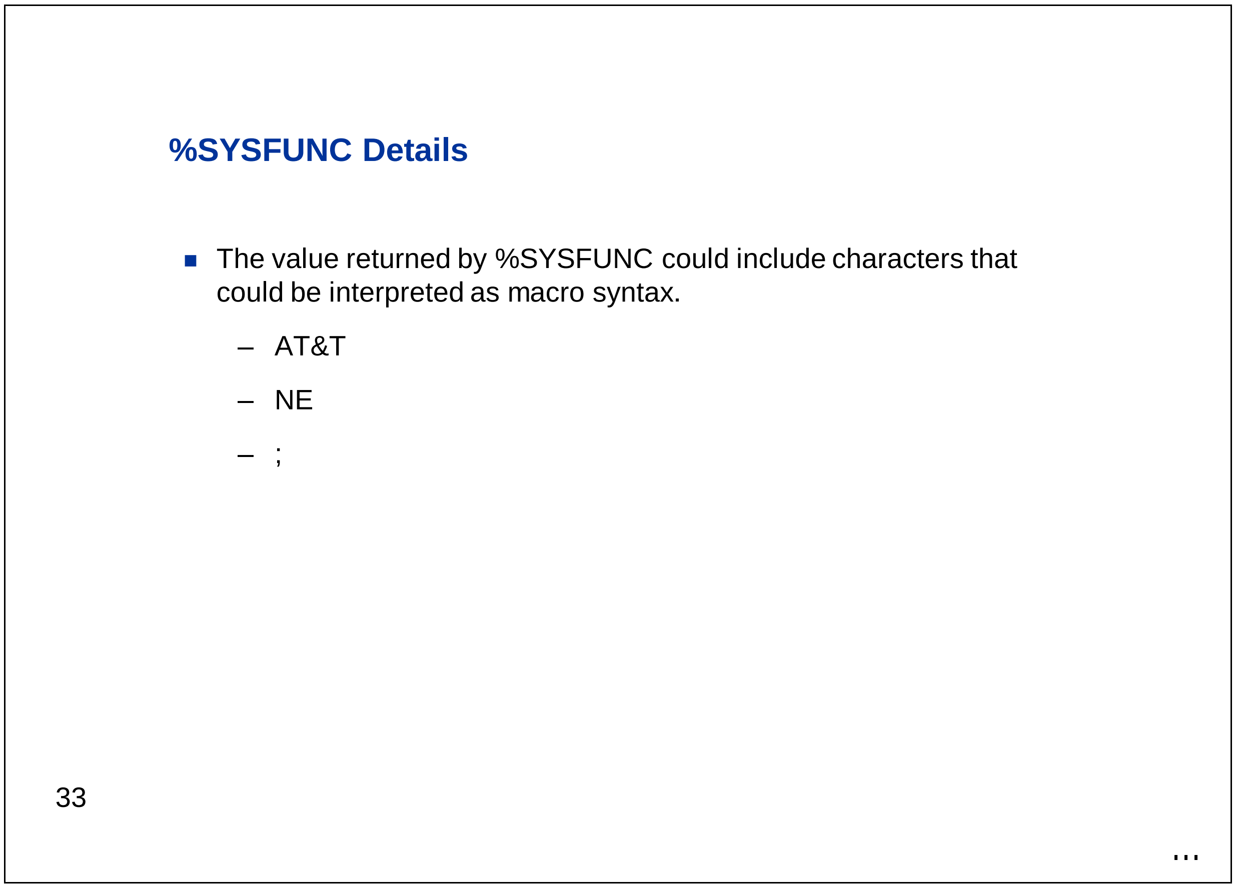
There is no need to convert numeric arguments.

```
%let min=%sysfunc(min(&x,&y,&z,0));
```

Numeric arguments required

32

If you’re using a function with macro variable references in place of numeric arguments, you might wonder if you need to do some kind of conversion. After all, macro variable values are only text. No conversion is necessary. When a function called by %SYSFUNC requires a numeric argument, as with the MIN function shown here, the macro facility passes the argument to the function as a numeric value.



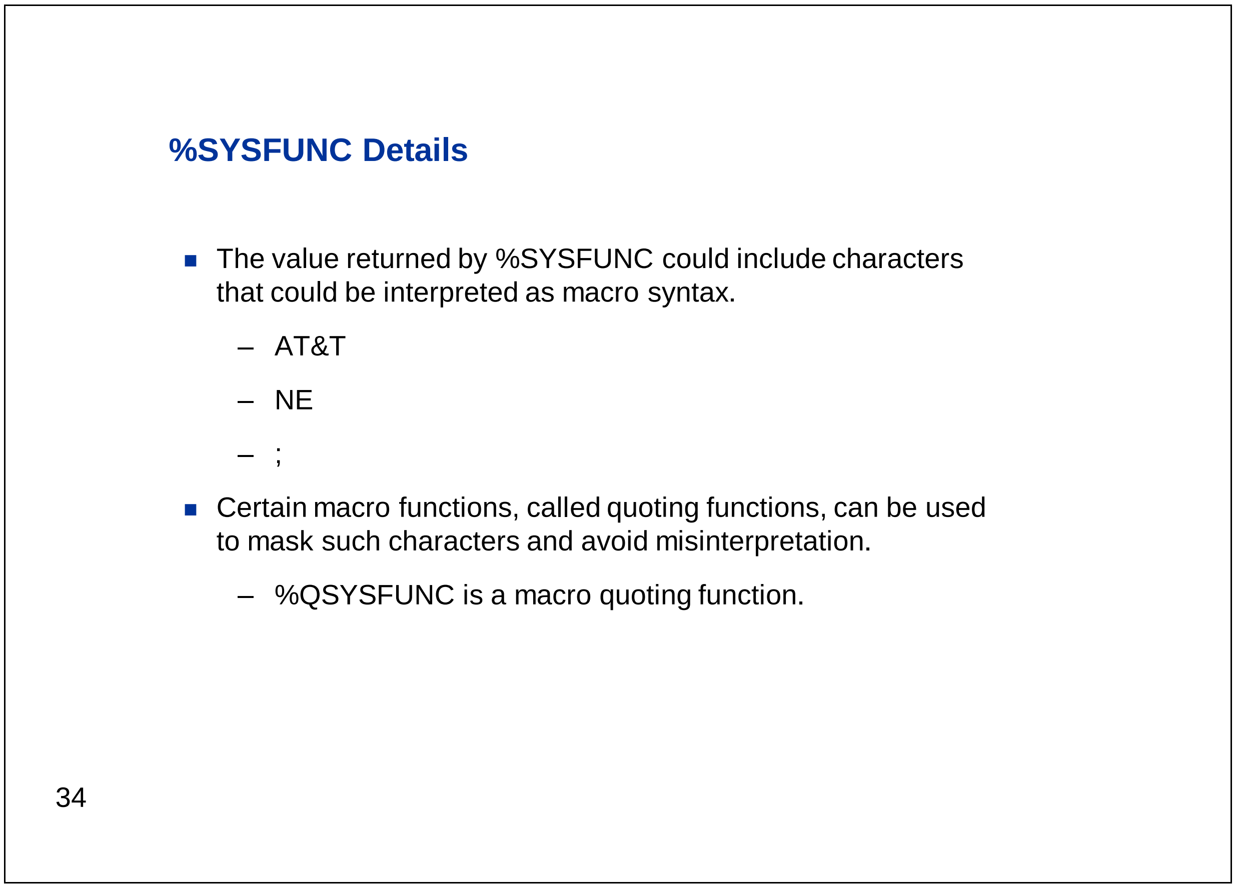
%SYSFUNC Details

- The value returned by %SYSFUNC could include characters that could be interpreted as macro syntax.
  - AT&T
  - NE
  - ;

33

...

In some situations, the value that is returned by %SYSFUNC can contain characters that could be interpreted as macro syntax. Let's look at a few examples. The text **`AT&T`** might refer to a company name. However, it could also be seen as text followed by a macro variable reference that the macro processor would attempt to resolve. The characters **`NE`** could be the abbreviation for the state of Nebraska. In some situations this text might be interpreted as the Not Equal to operator. A semicolon might be meant to be only text, but it would very likely be seen as the end of a statement. In all of these cases, these characters could cause unintended behavior in a program if they are intended as text rather than macro syntax. There are many more examples of characters that can cause these kinds of problems.

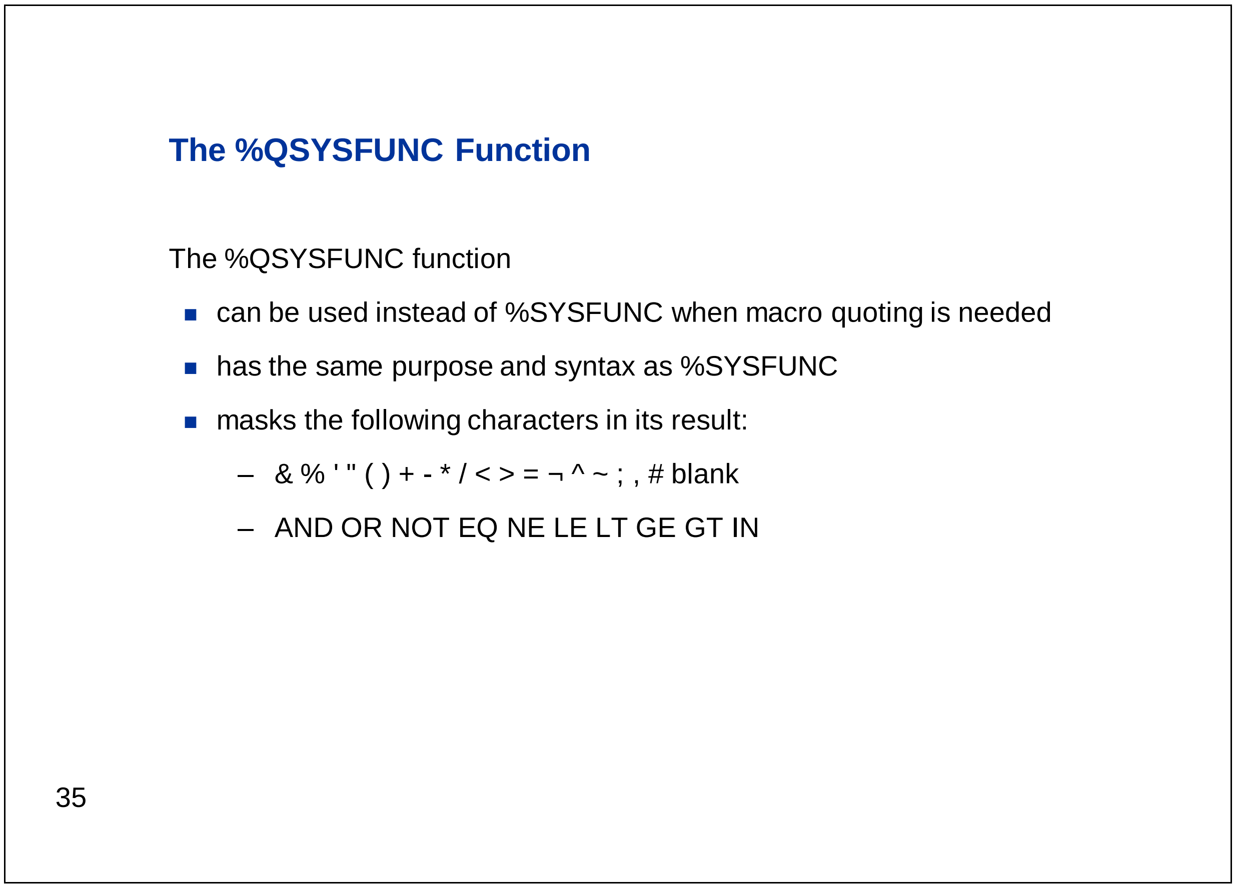


%SYSFUNC Details

- The value returned by %SYSFUNC could include characters that could be interpreted as macro syntax.
  - AT&T
  - NE
  - ;
- Certain macro functions, called quoting functions, can be used to mask such characters and avoid misinterpretation.
  - %QSYSFUNC is a macro quoting function.

34

These issues can occur in a variety of situations. There’s a class of macro functions, called *quoting functions*, that you can use to handle these kinds of problems. You can use a quoting function when you want certain characters that could be seen as macro syntax to be masked. Then, those characters are only seen as text. There are many different quoting functions available. One of them is %QSYSFUNC.



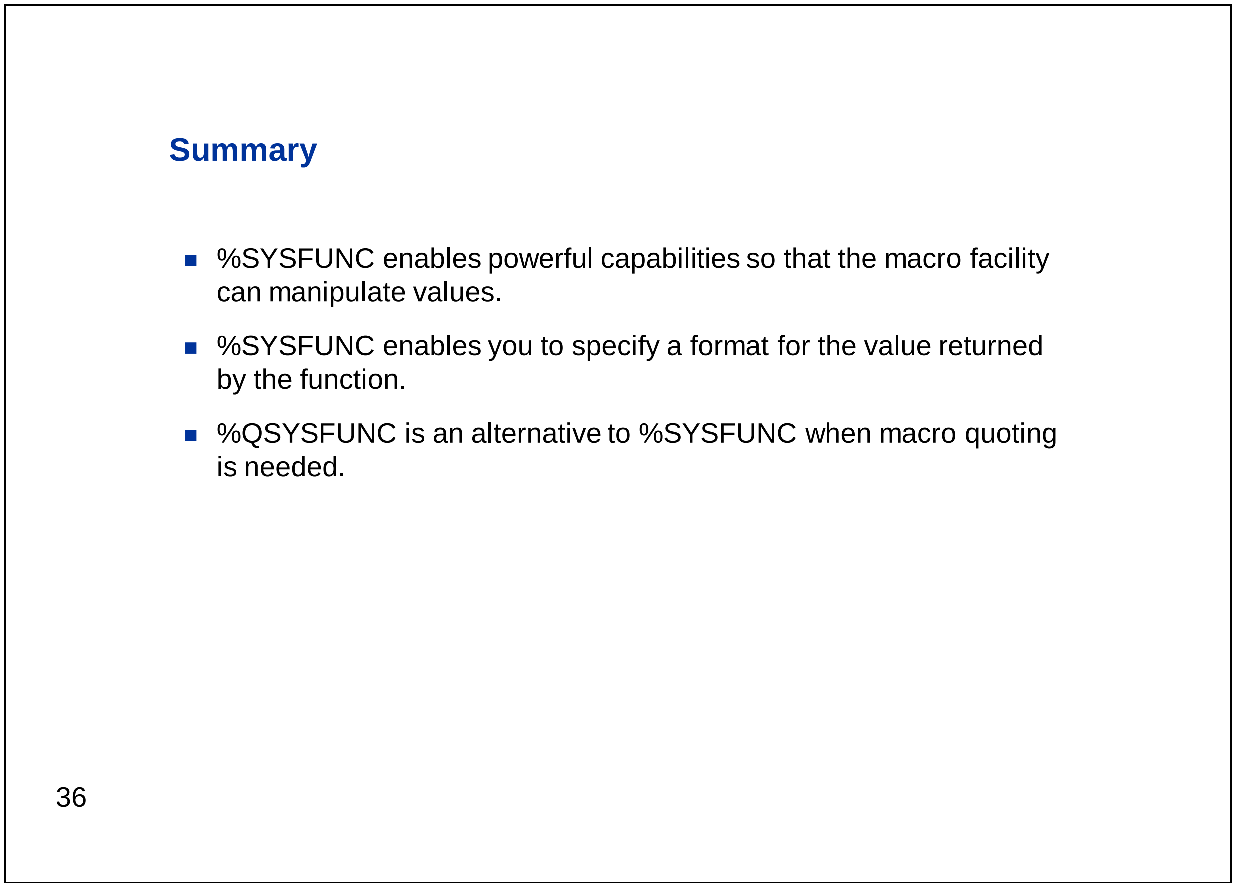
The %QSYSFUNC Function

The %QSYSFUNC function

- can be used instead of %SYSFUNC when macro quoting is needed
- has the same purpose and syntax as %SYSFUNC
- masks the following characters in its result:
  - & % ' " ( ) + - * / < > = ¬ ^ ~ ; , # blank
  - AND OR NOT EQ NE LE LT GE GT IN

35

%QSYSFUNC is an alternative to %SYSFUNC when you need to quote the result returned by the function. It does the same thing that %SYSFUNC does, and has the same syntax, but it masks the characters shown here in its result. Macro quoting is a significant topic on its own, and we don’t have time to discuss it further in this lecture. If you want to know more about macro quoting functions, they are discussed in detail in the SAS Macro Language 2 course. There’s also an e-lecture that covers this topic. I’ll mention that again at the end of this lecture. Of course you can also investigate the section on macro quoting that’s in *SAS 9.2 Macro Language: Reference*.



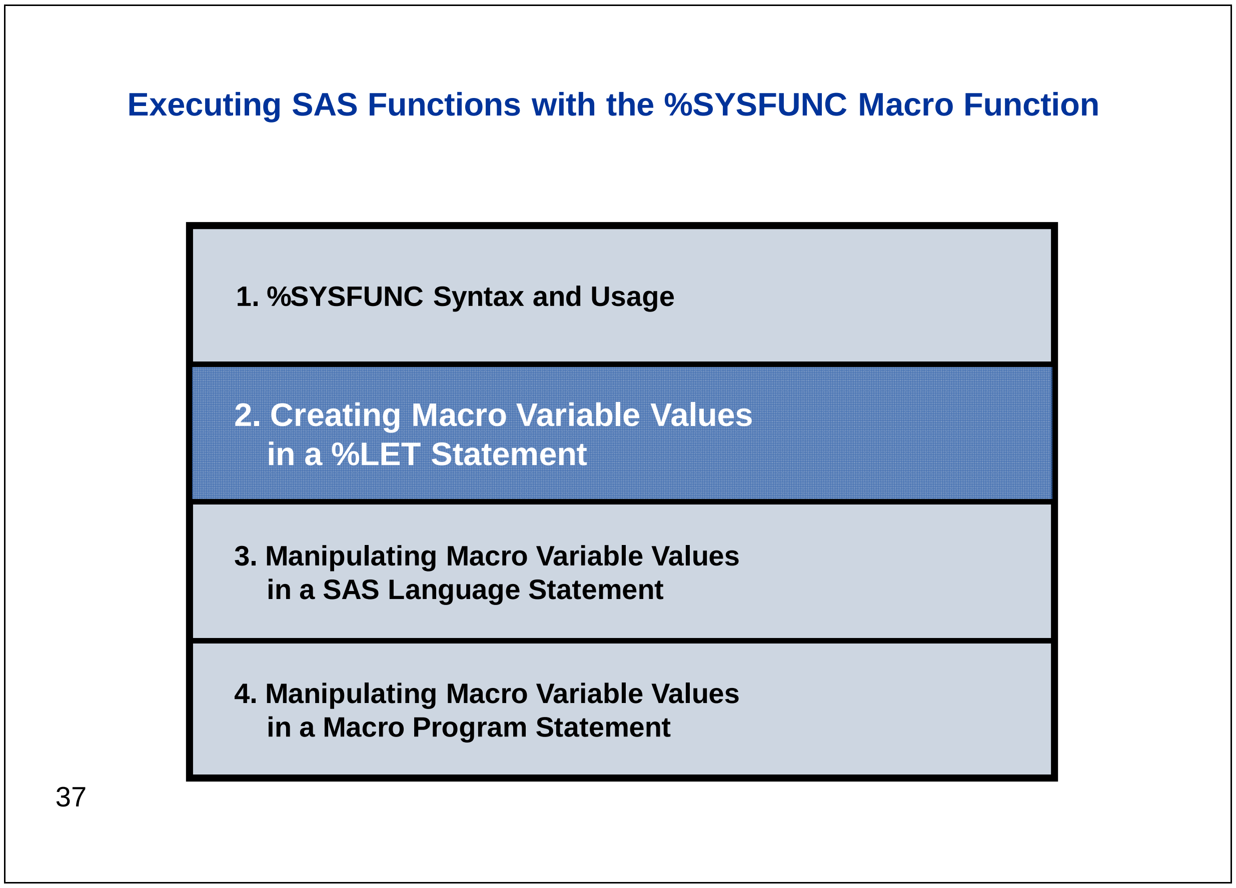
Summary

- %SYSFUNC enables powerful capabilities so that the macro facility can manipulate values.
- %SYSFUNC enables you to specify a format for the value returned by the function.
- %QSYSFUNC is an alternative to %SYSFUNC when macro quoting is needed.

36

To summarize, %SYSFUNC opens up all kinds of capabilities for you to manipulate macro variable values. You can specify a format with %SYSFUNC to control the value returned. %QSYSFUNC is an alternative to %SYSFUNC when you need to mask certain characters in the results.

2. Creating Macro Variable Values in a %LET Statement

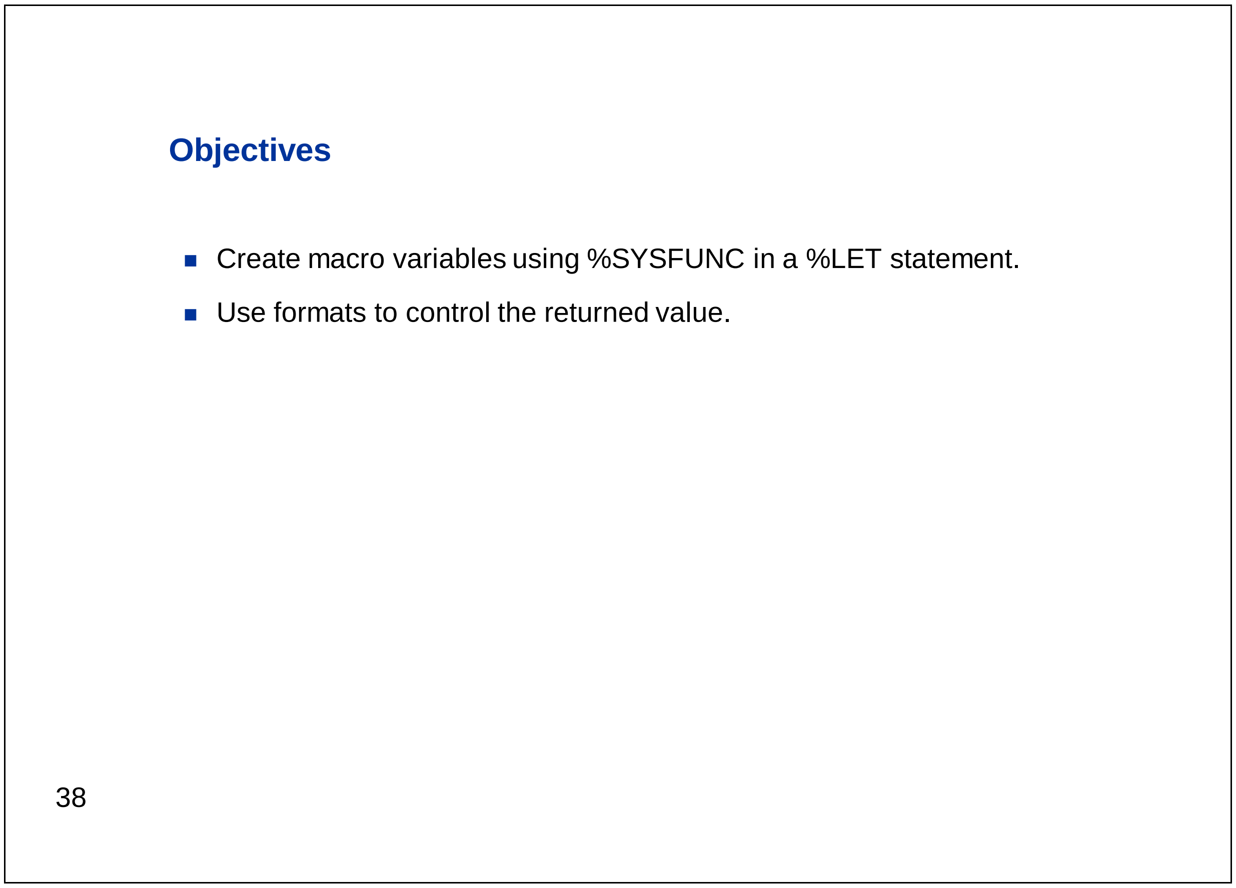


Executing SAS Functions with the %SYSFUNC Macro Function

1. %SYSFUNC Syntax and Usage
2. Creating Macro Variable Values in a %LET Statement
3. Manipulating Macro Variable Values in a SAS Language Statement
4. Manipulating Macro Variable Values in a Macro Program Statement

37

Well, we've talked a bit about what %SYSFUNC does and how to use it. Now let's see it in action.



Objectives

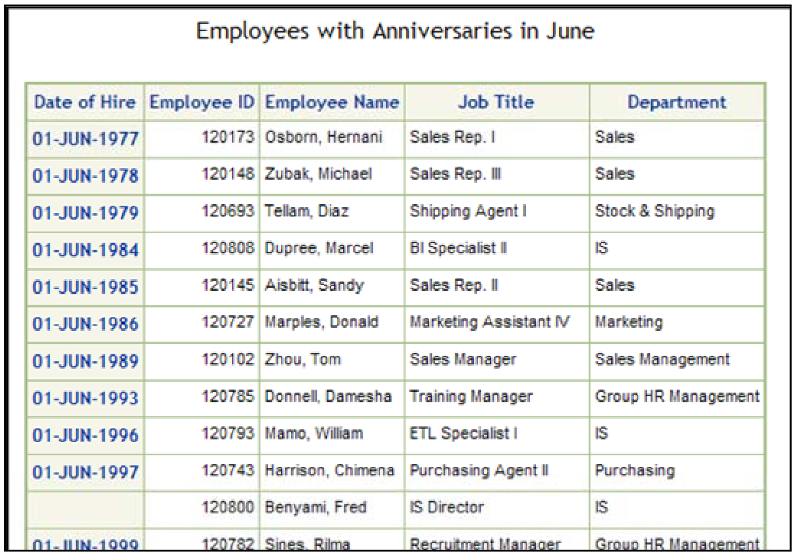
- Create macro variables using %SYSFUNC in a %LET statement.
- Use formats to control the returned value.

38

In this section, we’re going to use %SYSFUNC to create values for macro variables in a %LET statement. We’re also going to take advantage of the ability to apply a format to the returned value.

Business Scenario

A report is run on a monthly basis to list employees with anniversary dates in the current month.

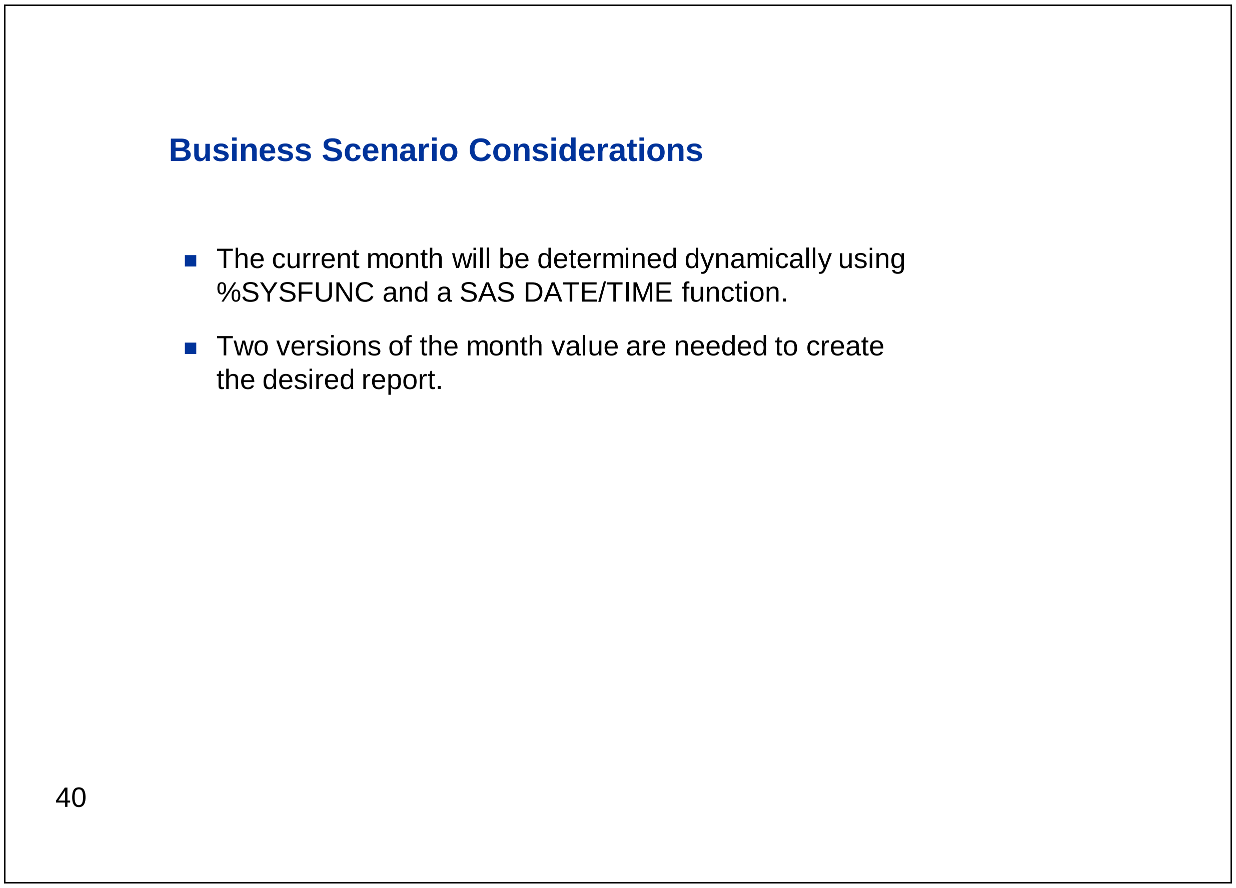


Employees with Anniversaries in June

Date of Hire	Employee ID	Employee Name	Job Title	Department
01-JUN-1977	120173	Osborn, Hernani	Sales Rep. I	Sales
01-JUN-1978	120148	Zubak, Michael	Sales Rep. III	Sales
01-JUN-1979	120693	Tellam, Diaz	Shipping Agent I	Stock & Shipping
01-JUN-1984	120808	Dupree, Marcel	BI Specialist II	IS
01-JUN-1985	120145	Aisbitt, Sandy	Sales Rep. II	Sales
01-JUN-1986	120727	Marples, Donald	Marketing Assistant IV	Marketing
01-JUN-1989	120102	Zhou, Tom	Sales Manager	Sales Management
01-JUN-1993	120785	Donnell, Damesha	Training Manager	Group HR Management
01-JUN-1996	120793	Mamo, William	ETL Specialist I	IS
01-JUN-1997	120743	Harrison, Chimena	Purchasing Agent II	Purchasing
	120800	Benyami, Fred	IS Director	IS
01-JUN-1999	120782	Sines, Rilma	Recruitment Manager	Group HR Management

39

Here’s the problem that we need to solve. We need to create a report that lists the employees with anniversary dates in the current month. Here we see a version of the report that was run in the month of June, so it shows employees who were hired during that month of the year. The title indicates that the report displays employees with anniversaries in June, and although only a portion of the report is shown here, it includes only employees hired in June.

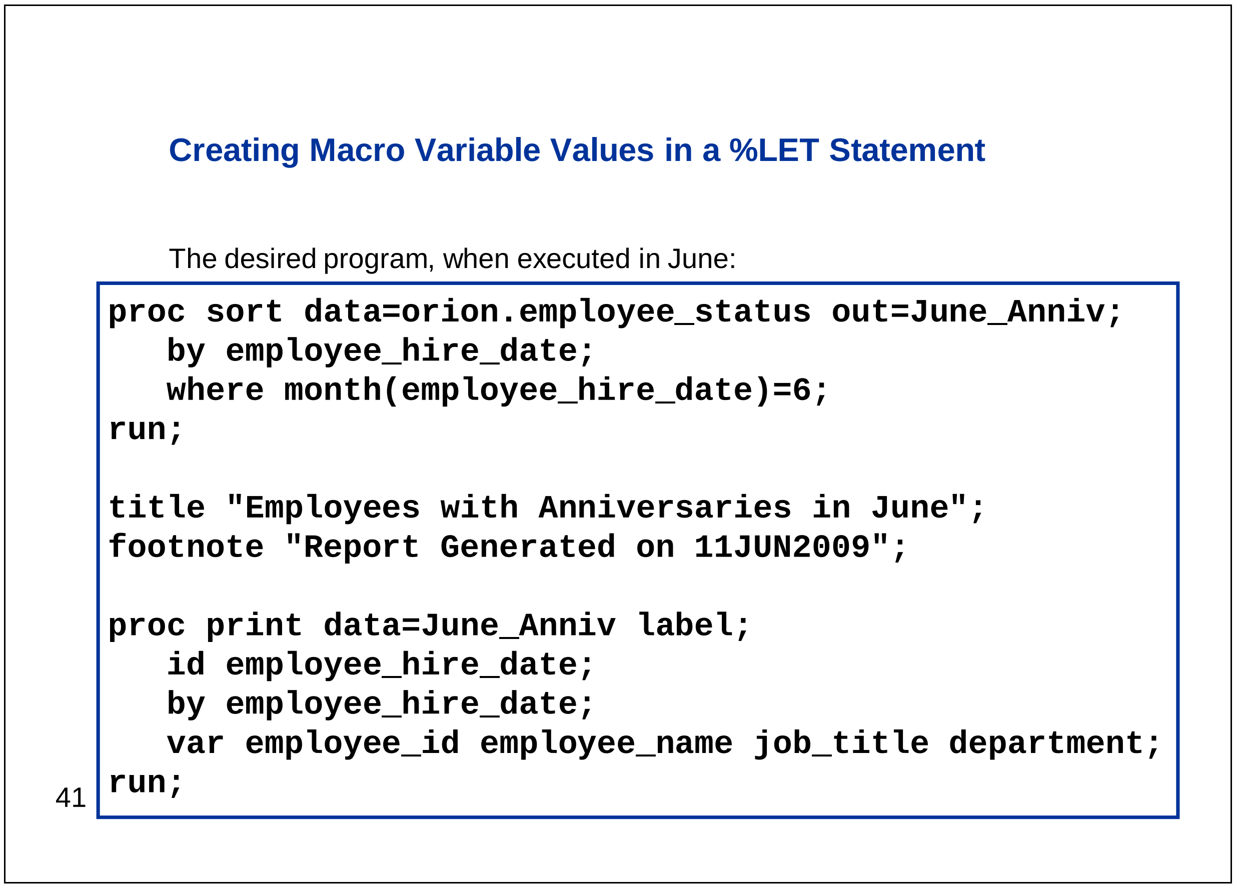


Business Scenario Considerations

- The current month will be determined dynamically using %SYSFUNC and a SAS DATE/TIME function.
- Two versions of the month value are needed to create the desired report.

40

We want the program that creates the report to be dynamic and not require editing each month in order to get the current month’s report. We’re going to have the macro facility determine the appropriate month depending on when we submit the program. We can do that using %SYSFUNC to access a SAS DATE/TIME function. For this program, we’ll need two different versions of the month value in order to create the report that we want.



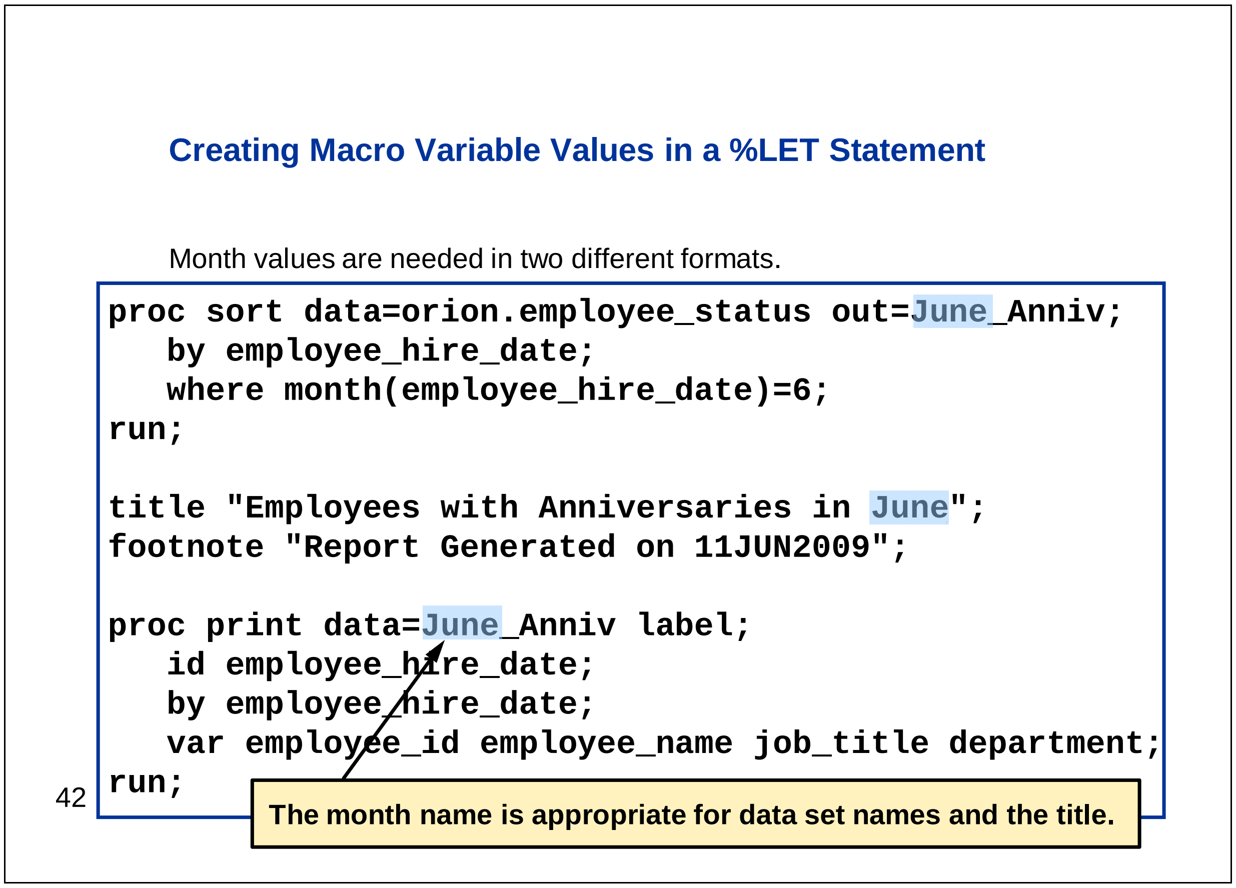
Creating Macro Variable Values in a %LET Statement

The desired program, when executed in June:

```
proc sort data=orion.employee_status out=June_Anniv;
   by employee_hire_date;
   where month(employee_hire_date)=6;
run;

title "Employees with Anniversaries in June";
footnote "Report Generated on 11JUN2009";

proc print data=June_Anniv label;
   id employee_hire_date;
   by employee_hire_date;
   var employee_id employee_name job_title department;
run;
```

41

This slide shows the final program that we want to be executed after the macro facility has done its job filling in the appropriate month values. This version of the program was generated in June.



Creating Macro Variable Values in a %LET Statement

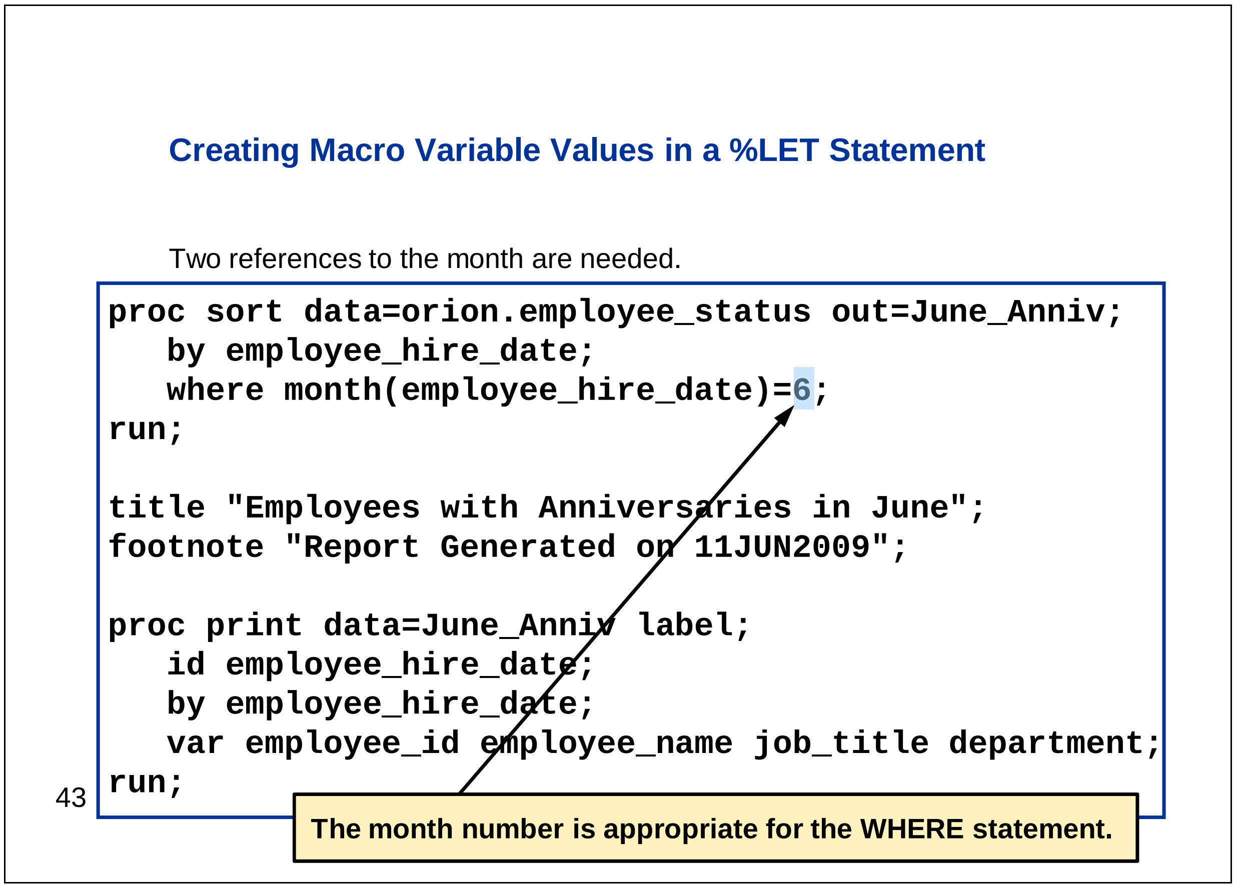
Month values are needed in two different formats.

```
proc sort data=orion.employee_status out=June_Anniv;
   by employee_hire_date;
   where month(employee_hire_date)=6;
run;

title "Employees with Anniversaries in June";
footnote "Report Generated on 11JUN2009";

proc print data=June_Anniv label;
   id employee_hire_date;
   by employee_hire_date;
   var employee_id employee_name job_title department;
run;
```

The month name is appropriate for data set names and the title.

42

If we look at the code, we'll see the month is referenced in two different ways. First, we're using the month name. The month name is used to name a data set containing employees with anniveraries in the current month and also to specify the current month in the report title.



Creating Macro Variable Values in a %LET Statement

Two references to the month are needed.

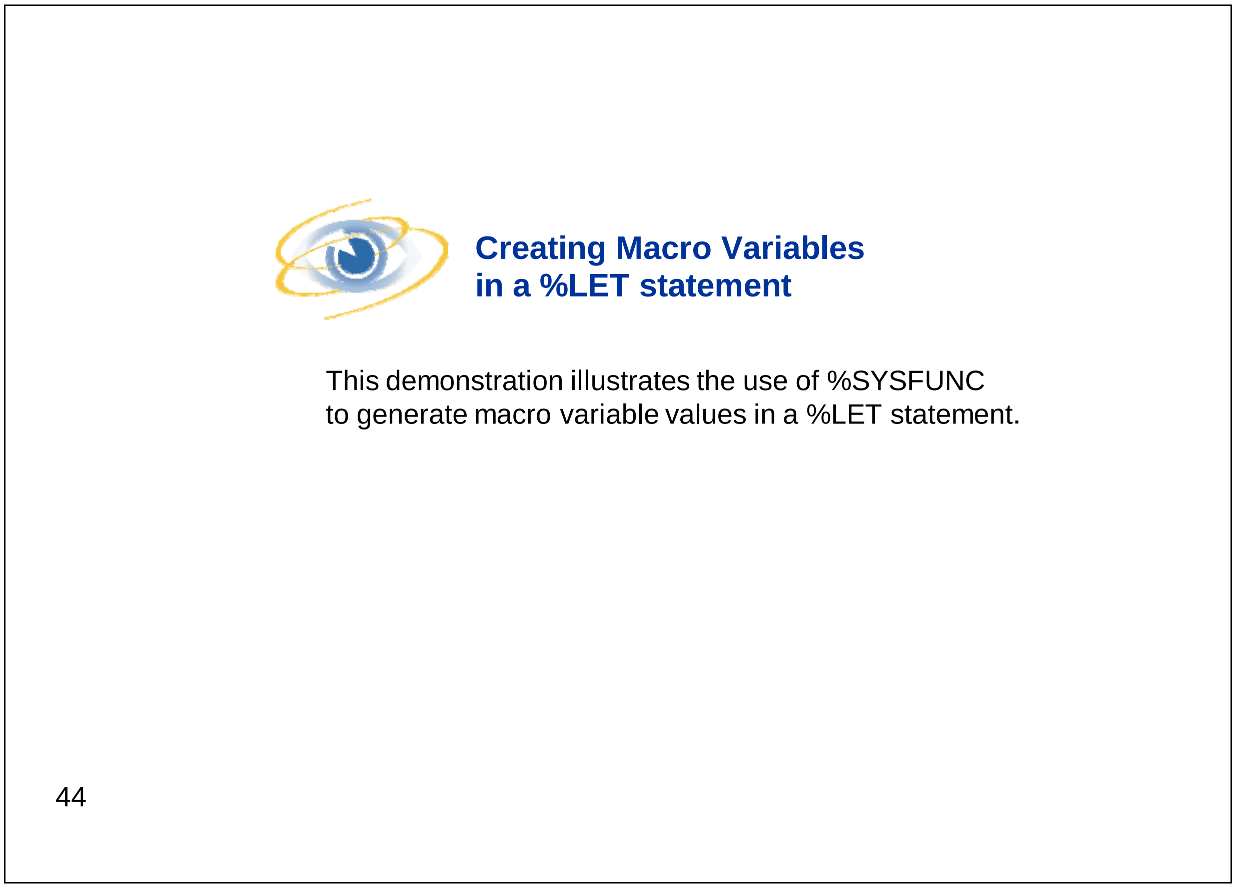
```
proc sort data=orion.employee_status out=June_Anniv;
   by employee_hire_date;
   where month(employee_hire_date)=6;
run;

title "Employees with Anniversaries in June";
footnote "Report Generated on 11JUN2009";

proc print data=June_Anniv label;
   id employee_hire_date;
   by employee_hire_date;
   var employee_id employee_name job_title department;
run;
```

43

The month number is appropriate for the WHERE statement.

Secondly, we're using the month number in the WHERE statement. We need a value to compare to the employee's month of hire, which is calculated here by the MONTH function. The MONTH function returns a numeric value from 1 to 12, so we need a numeric value for this comparison.

So this is the final program that we want the macro facility to generate for us in June.



Creating Macro Variables in a %LET statement

This demonstration illustrates the use of %SYSFUNC to generate macro variable values in a %LET statement.

44

Let’s switch over to a SAS session and take a look at the program that we’ll submit to get that result.

So here’s the program. At the top of the program we have an OPTIONS statement. Notice that we’re turning on the SYMBOLGEN option so that we’ll see messages about macro variable resolution in the log. Then we have two %LET statements creating the two macro variables that we’ll use to reference the current month. They’re both calling the TODAY function, but they’re specifying different formats to apply to the returned value. The first %LET, which is creating a macro variable named MONTHNAME, is using the MONNAME. format. That format returns the name of the month such as January, February, and so on. Because we haven’t specified a width for the format, we’ll get the default of 9, which accommodates the longest possible month name. The second %LET statement is creating a macro variable named MONTHNUM and it is using the format MONTH. The MONTH. format returns a number from 1 to 12.

Next, we have the SAS code to create the desired results. First is a PROC SORT step. We’re using this step both to sort the data, because we want to see the employees in order by hire date on the final report, and also to subset for the current month. We’re referencing the &MONTHNAME macro variable in the OUT= option so that the sorted data will be named using the current month. Notice that we’re using the period delimiter following the reference to &MONTHNAME so that we append the text **`_ANNIV`** to the resolved value of month to generate the data set name. The &MONTHNUM macro variable is specified in the WHERE statement so that we’ll only get employees with anniversary dates in the current month in the output data set.

Following the PROC SORT step, there are ODS statements to route the output to an HTML file and to close the listing destination. Then we have a TITLE statement that references the &MONTHNAME macro variable so the name of the current month will appear in the report title. Notice that we're also using the automatic macro variable &SYSDATE9 in the footnote.

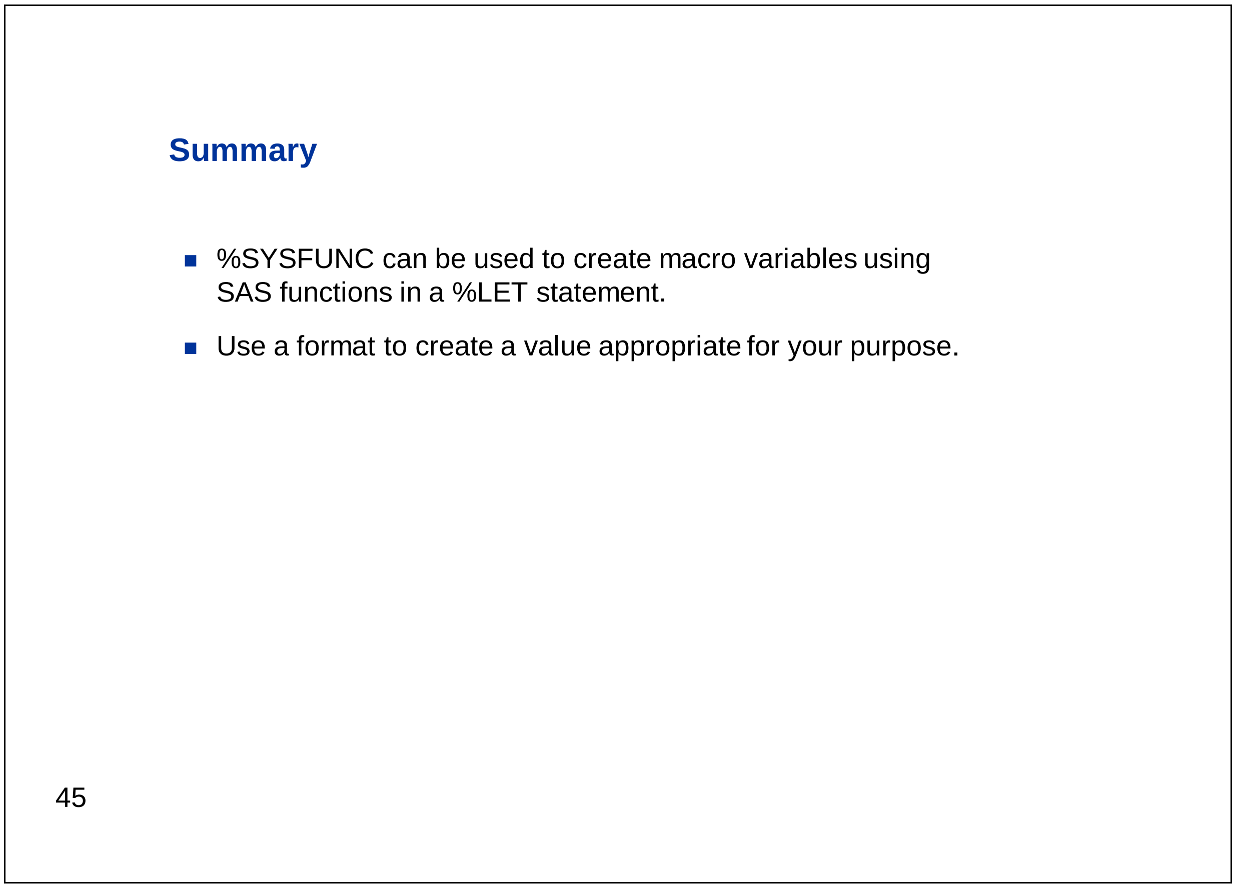
Finally, the PROC PRINT step generates the desired report. We've got a reference to the &MONTHNAME variable to build the appropriate data set name in the DATA= option in the PROC PRINT statement. Lastly, we have an OPTIONS statement to turn the SYMBOLGEN option off.

Let's go ahead and submit the report. I'm submitting this program in August, and we can see from the results that I did get a listing for employees with August anniversaries. Also notice that the month of August is referenced in the title.

Let's take a look at the log. Because we had the SYMBOLGEN option turned on, we see messages telling us how each macro variable reference resolved. So we see that &MONTHNAME resolved to August and that &MONTHNUM resolved to the digit 8. So, we got the results that we wanted.

Now, you might notice here, particularly when looking at the log, that although we had repeated references to the macro variable &MONTHNAME, we only used &MONTHNUM one time. It was only used here in the WHERE statement. We certainly can delete these macro variables from the symbol table to free that memory, but in the case of &MONTHNUM we might choose not to create a macro variable at all. Let's go back to the program. I'm going to take the reference to %SYSFUNC that was previously in the %LET statement and put it directly in the WHERE statement. I'm going to highlight what's on the right side of the equal sign in the %LET, copy it, and paste it over the reference to **`monthnum`** in the WHERE statement. Now let's rerun the PROC SORT step, and we can see that we got the same result as we did previously. So, it's not necessary in this case to create a macro variable at all. In this instance of &MONTHNAME, it makes sense to do so because we're using it several times, but you can also use %SYSFUNC without necessarily creating a macro variable. We'll see more of that in the following sections.

You might notice some blank hire dates on this report. I should probably point out that those are not missing values. HireDate was used as both a BY and ID variable on this PROC PRINT report, and therefore the HireDate is printed only for the first employee when multiple employees were hired on the same day.



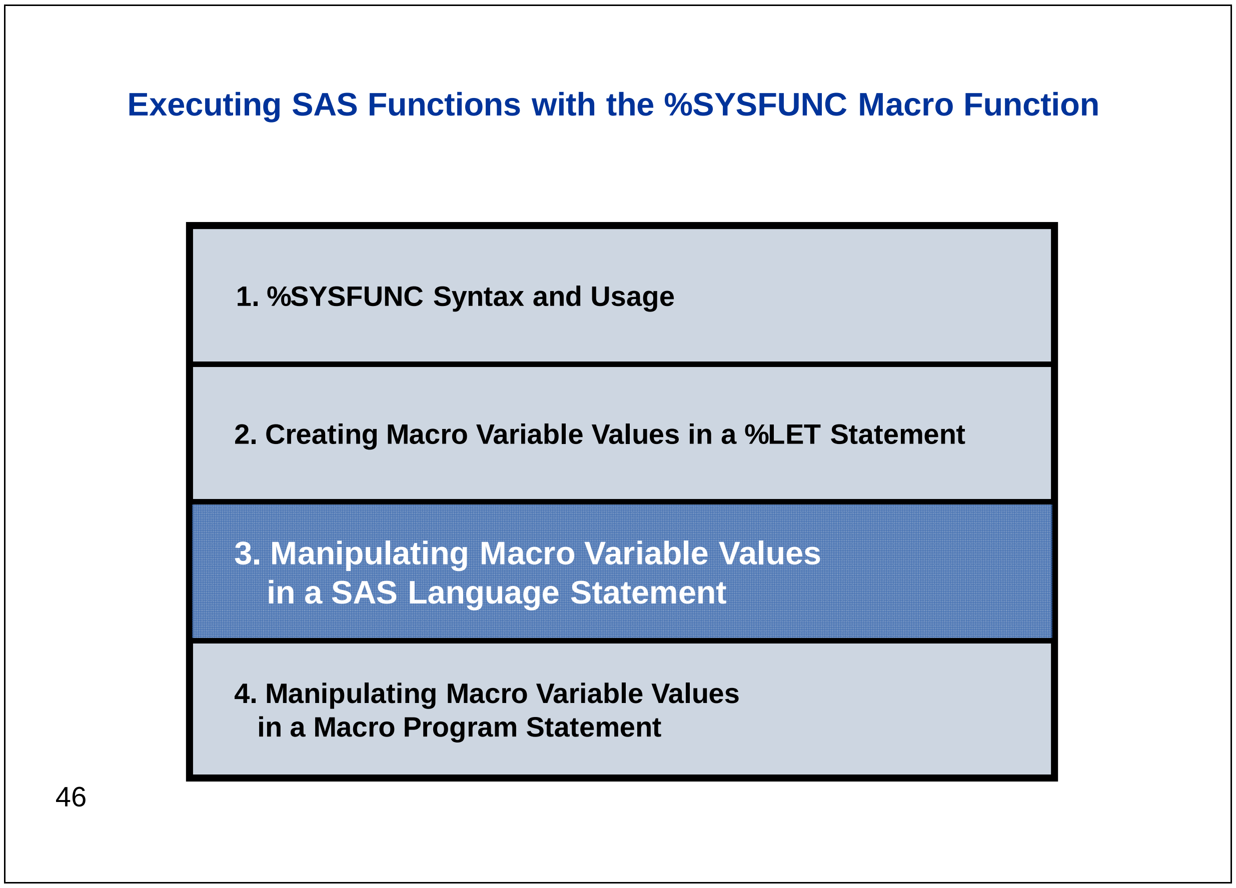
Summary

- %SYSFUNC can be used to create macro variables using SAS functions in a %LET statement.
- Use a format to create a value appropriate for your purpose.

45

So we’ve seen that you can use %SYSFUNC to create macro variable values in a %LET statement. We’ve used formats to return a value appropriate for our purpose.

3. Manipulating Macro Variable Values in a SAS Language Statement

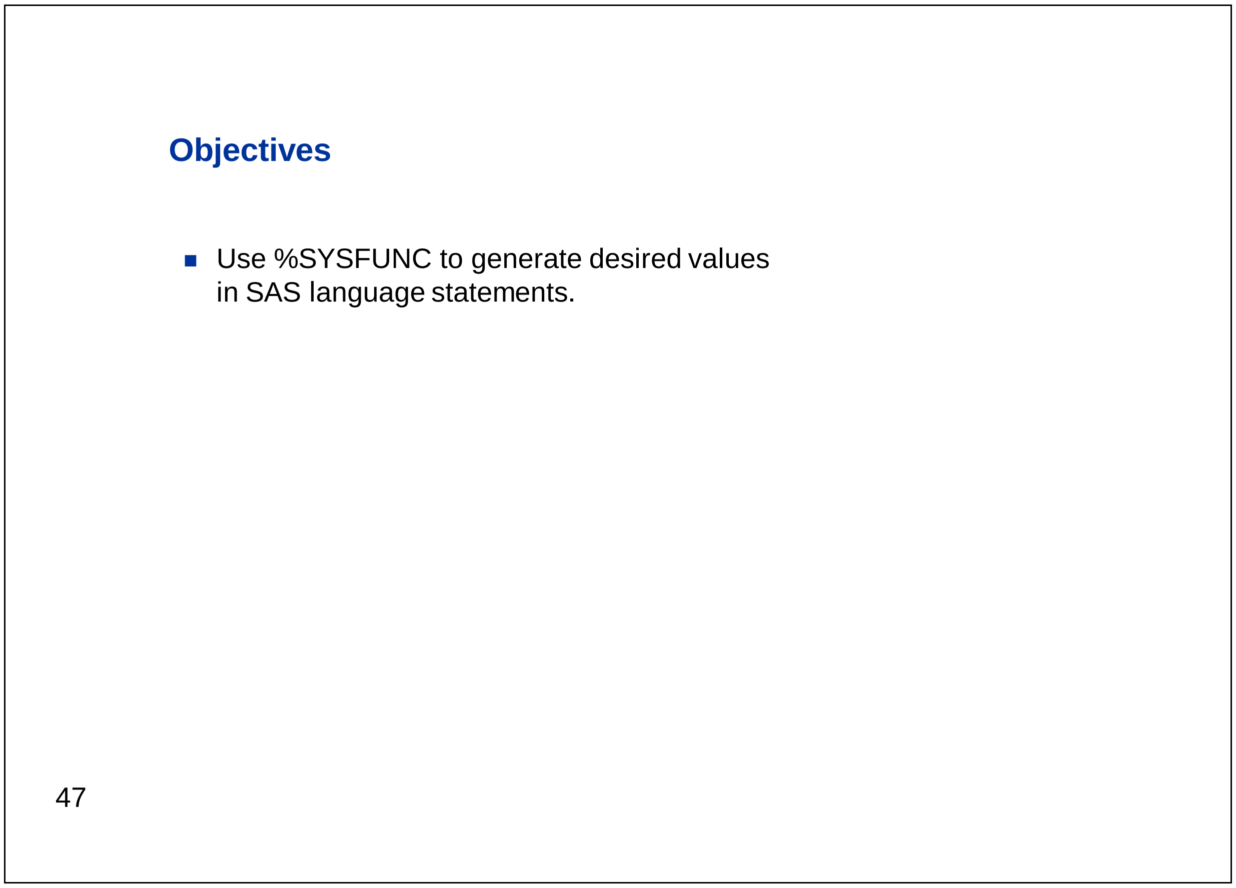


Executing SAS Functions with the %SYSFUNC Macro Function

1. %SYSFUNC Syntax and Usage
2. Creating Macro Variable Values in a %LET Statement
3. Manipulating Macro Variable Values in a SAS Language Statement
4. Manipulating Macro Variable Values in a Macro Program Statement

46

We saw, at the end of the last demo, that you can use %SYSFUNC directly in a SAS language statement. Let’s go on and look at that in more detail.

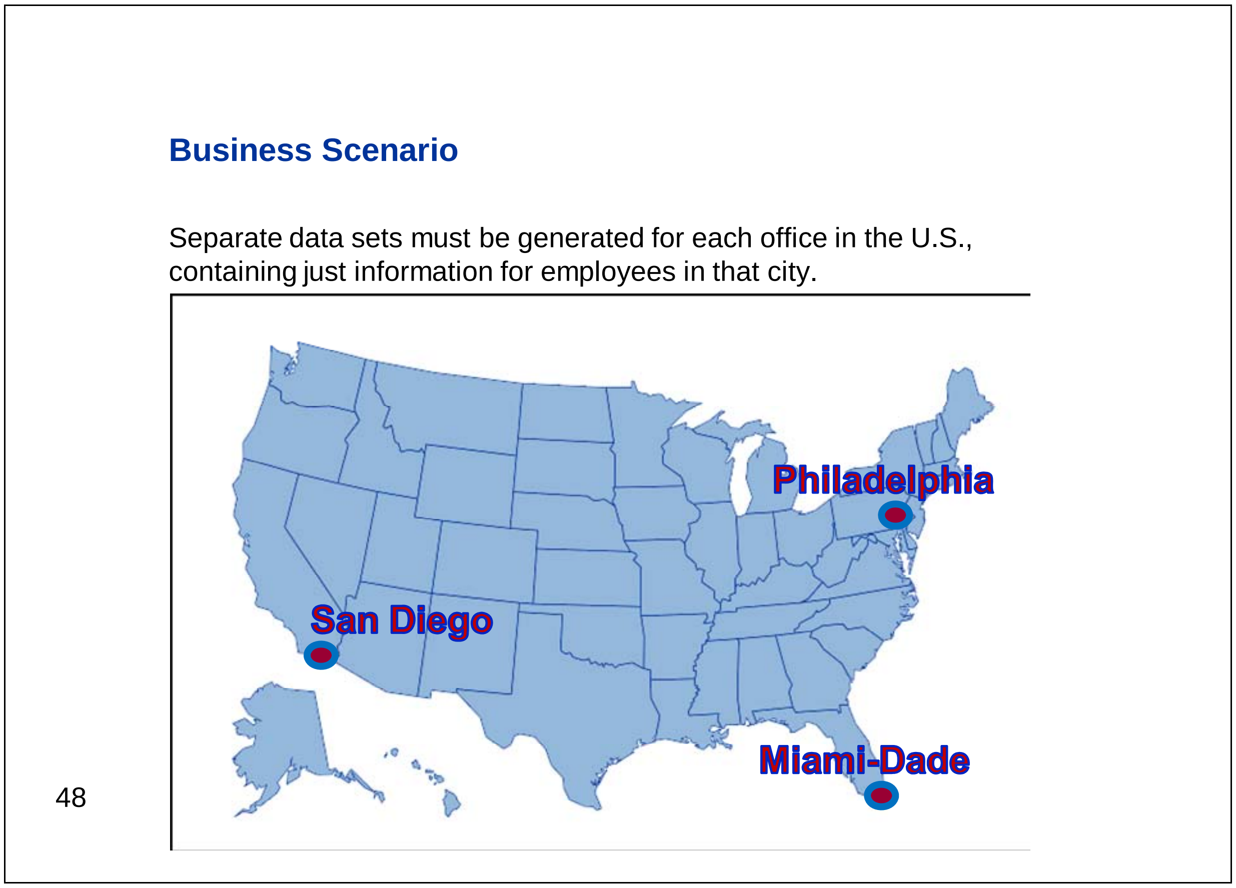


Objectives

- Use %SYSFUNC to generate desired values in SAS language statements.

47

Specifically in this section, we’re going to use %SYSFUNC to generate portions of DATA step statements. You could also use the function to generate text in a PROC step or in a global statement.

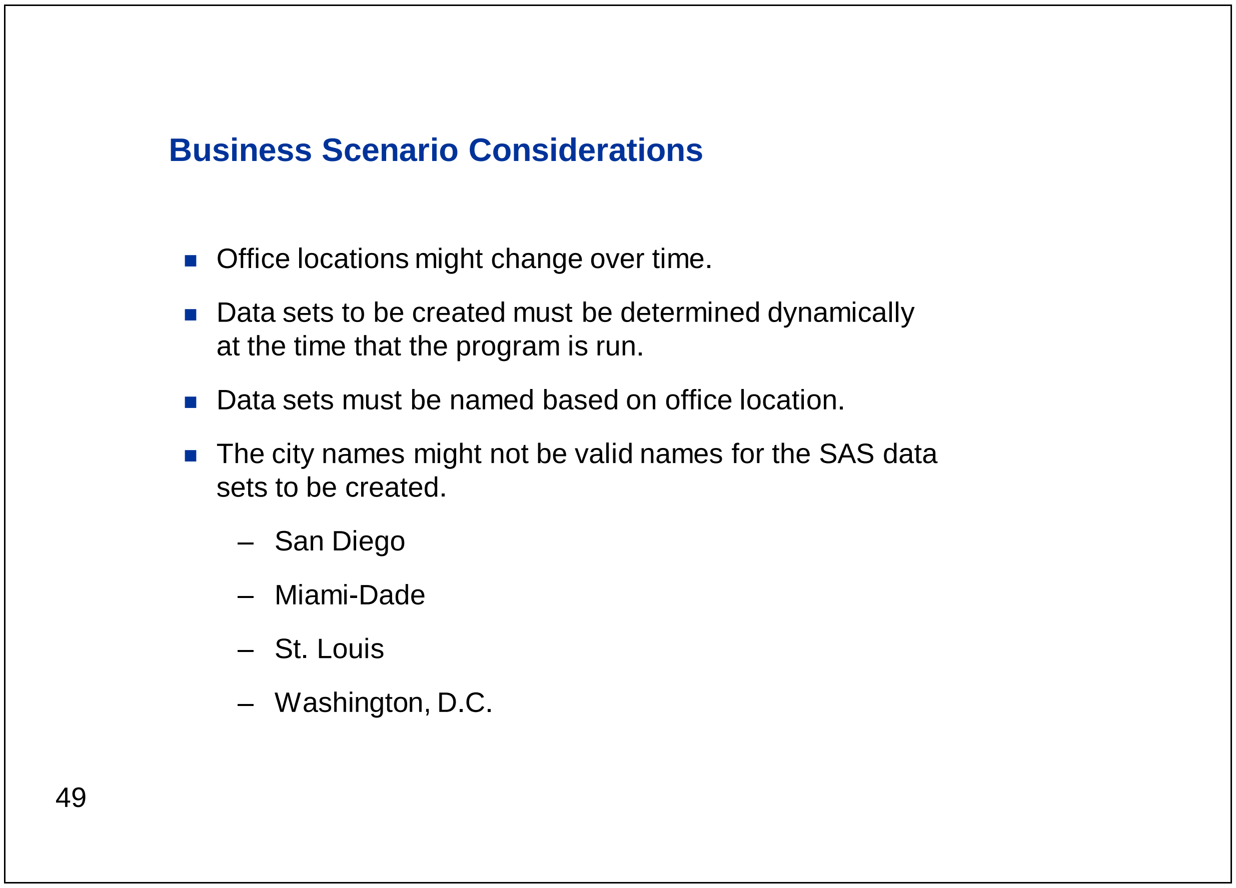


Business Scenario

Separate data sets must be generated for each office in the U.S., containing just information for employees in that city.

48

Here’s the problem that we need to address. Our company has offices and employees all over the world. For each U.S. location, we need to create a separate data set that contains only the employees for that city. So suppose we currently have offices in San Diego, Philadelphia, and Miami-Dade. We want to create three data sets: one containing the San Diego employee information, one for Philadelphia, and one for Miami-Dade.

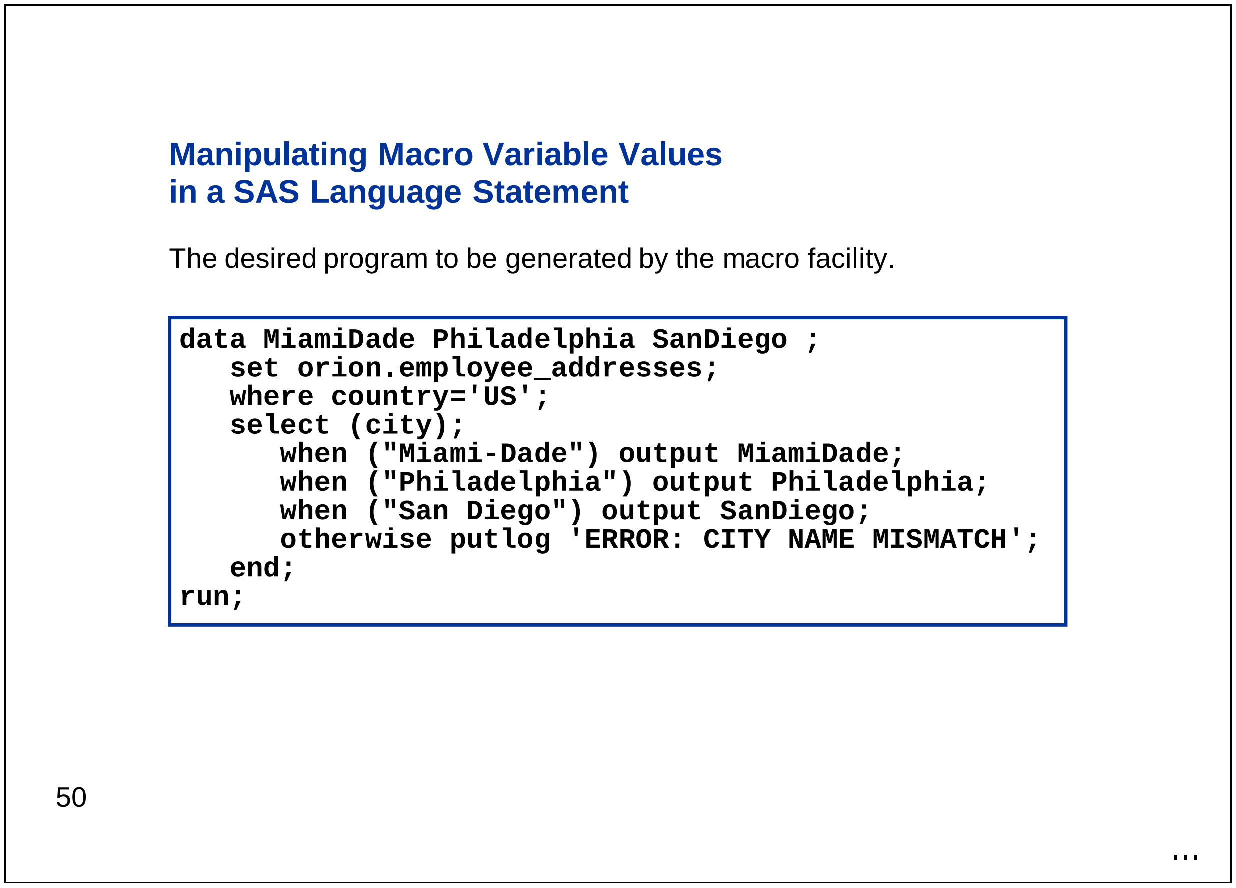


Business Scenario Considerations

- Office locations might change over time.
- Data sets to be created must be determined dynamically at the time that the program is run.
- Data sets must be named based on office location.
- The city names might not be valid names for the SAS data sets to be created.
  - San Diego
  - Miami-Dade
  - St. Louis
  - Washington, D.C.

49

Now, our office locations can change over time. We might open new offices or close existing ones. Therefore, we want to write a program that will determine the data sets to be created dynamically, based on the U.S. locations that are currently in the source data. We also want to use the name of each location to create the name for the corresponding data set. However, the city names as stored in the data might not be appropriate to use directly as data set names. For example, we could have city names such as the ones shown here. They include special characters, for example, spaces, dashes, or punctuation characters that aren’t allowed by default in the name of a SAS data set. Now not all of these examples are currently in our data, but we need to write a program that will address these kinds of values.



Manipulating Macro Variable Values in a SAS Language Statement

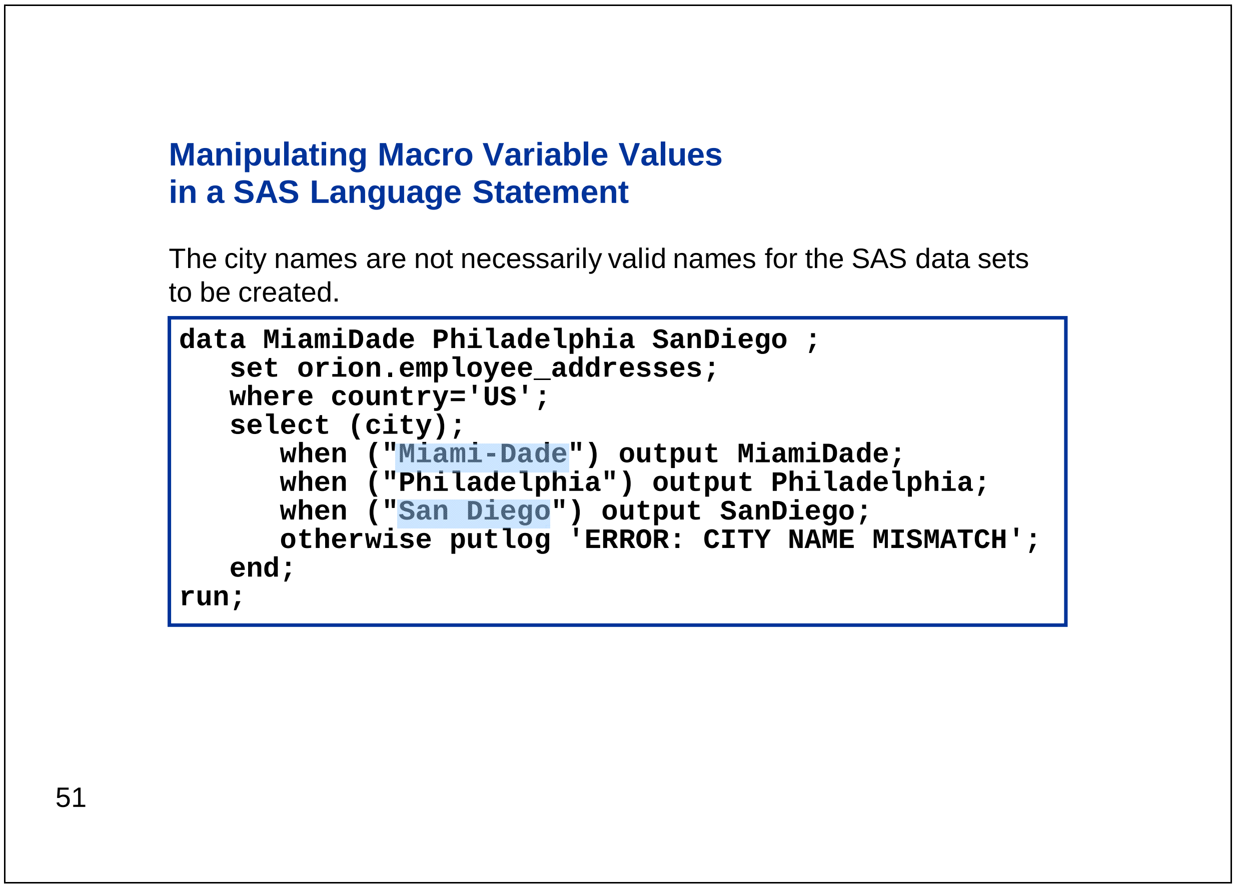
The desired program to be generated by the macro facility.

```
data MiamiDade Philadelphia SanDiego ;
   set orion.employee_addresses;
   where country='US';
   select (city);
      when ("Miami-Dade") output MiamiDade;
      when ("Philadelphia") output Philadelphia;
      when ("San Diego") output SanDiego;
      otherwise putlog 'ERROR: CITY NAME MISMATCH';
   end;
run;
```

50

...

So here’s an example of the program that we want the macro facility to generate. The source data set, **`orion.employee_addresses`**, contains information for all employees globally, so we’re subsetting it for **`country`** equal to US in this program. At the time this program was generated, there were three U.S. locations in our source data. So, this DATA step specifies three data sets in the DATA statement. Then, we’re using a SELECT statement to output the observations to the appropriate data set based on the value of the variable **`city`**.



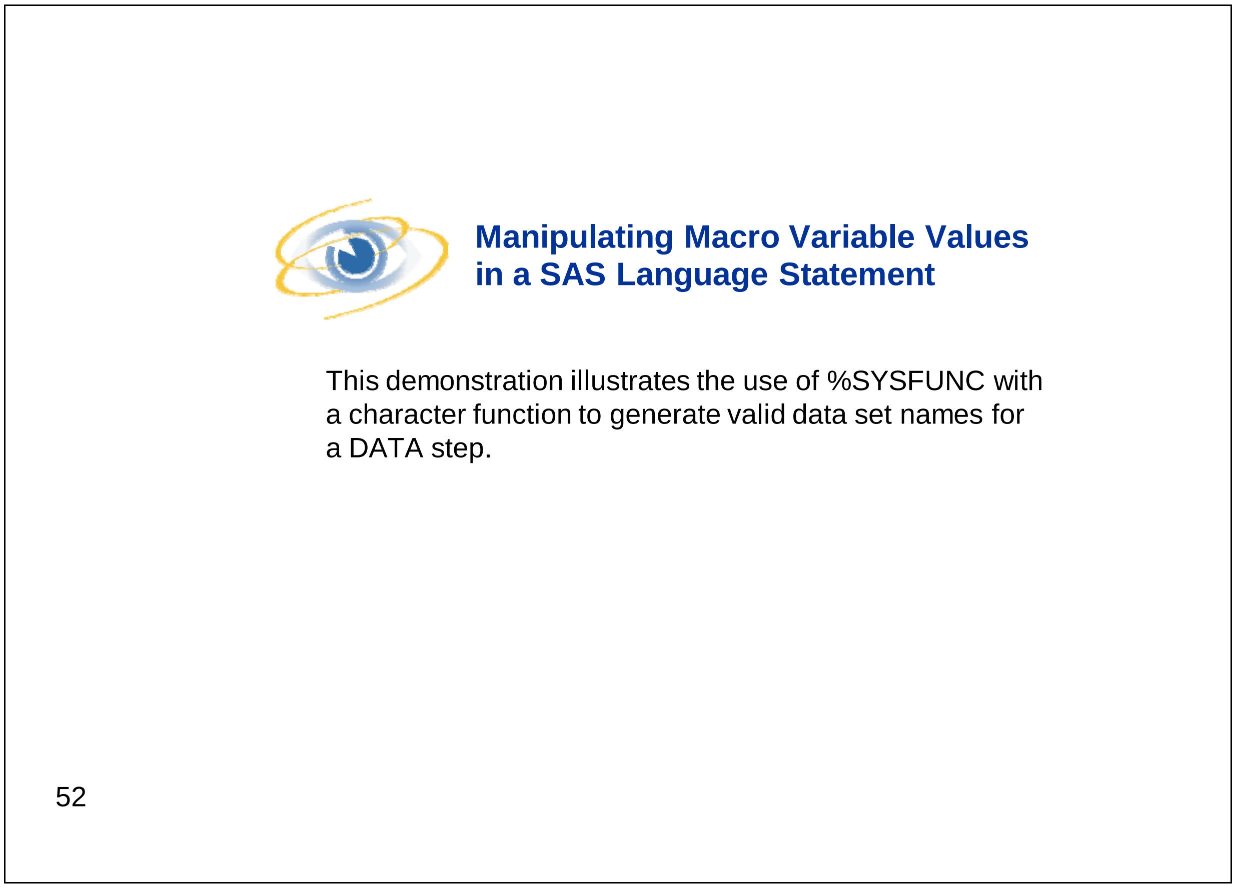
Manipulating Macro Variable Values in a SAS Language Statement

The city names are not necessarily valid names for the SAS data sets to be created.

```
data MiamiDade Philadelphia SanDiego ;
   set orion.employee_addresses;
   where country='US';
   select (city);
      when ("Miami-Dade") output MiamiDade;
      when ("Philadelphia") output Philadelphia;
      when ("San Diego") output SanDiego;
      otherwise putlog 'ERROR: CITY NAME MISMATCH';
   end;
run;
```

51

Notice that the city names currently in the data aren’t all valid for use directly as a data set name. We have a value of `Miami-Dade`, which includes a hyphen, and a value of `San Diego`, which includes a space. We want to remove these special characters in order to create a valid data set name for each location. There is no available macro function that does that, but we can achieve that result using %SYSFUNC with a SAS function.



Manipulating Macro Variable Values in a SAS Language Statement

This demonstration illustrates the use of %SYSFUNC with a character function to generate valid data set names for a DATA step.

52

Let’s go to a SAS session to look at the program that will generate this final result.

So here’s our program. Now I’m going to start with a simplified version of the program. In this simplified version, we’ve manually assigning the city names to macro variables rather than looking them up dynamically. We’re going to walk through the process of building the DATA step code. Then we’ll come back and add the part that will determine the city names from the data.

First of all, notice that the program includes a macro definition. That’s because we need to do some iterative processing with %DO statements and that can only happen inside a macro definition.

Here at the top we have a %LOCAL statement naming the macro variables that we’ll be creating here. NUMCITIES will store the total number of office locations in the data, and CITYCOUNT will be used to loop through each city in some %DO loops. CITYNAME1, CITYNAME2, and CITYNAME3 will store the names of the three cities currently in our data. We want to make sure that these variables are stored in the local table for this macro definition. That way, if we happen to have a macro variable of the same name in another table, it won’t be overwritten.

Below that, we have %LET statements that assign values to all of those macro variables except for the index variable for the loop. So we’re manually specifying that we have three cities and that the city names are Miami-Dade, Philadelphia, and San Diego. As I mentioned, we’re going to make this part dynamic in the final version of the program, but for now we’ll specify these values manually.

Now we can use that set of variables to build the appropriate DATA step code. First, we're going to write the DATA statement itself. It begins with the keyword DATA, so we've specified that here. Now we need to name the data sets that we want to create, and we're going to use the macro variables that we created to do so. We're using a %DO statement to loop through the values from one to the value of &NUMCITIES. Remember, &NUMCITIES contains the number of cities in our data. However, we can't write out the city names as is because those names might not be appropriate data set names. As we saw, they might contain blanks or punctuation characters that are not allowed by default in SAS data set names. So, we're going to use %SYSFUNC with the COMPRESS function to build valid data set names from the city names.

The COMPRESS function removes specified characters from a character string. The first argument is the string that you want to modify. In this case it's our city names. We're generating those using indirect references to the CITYNAME macro variables. The indirect reference here is made up of two ampersands, followed by the variable prefix, which is CITYNAME, followed by a reference to the index variable for the %DO loop, which is &CITYCOUNT. Remember that when this goes to the macro processor, it will see &&CITYNAME and resolve to &CITYNAME because two ampersands together resolve to one. Then it will resolve &CITYCOUNT to the value for the current iteration of the %DO loop. For example, on the first iteration it would be the digit 1. So we put those two resolved items together to get &CITYNAME1, which is then resolved to return the name of the first city in our data.

The second argument to the COMPRESS function is the specific characters that you want to remove. Here, we've omitted that argument because we're going to use the third argument instead. The third argument specifies modifiers that remove categories of characters. The modifier P specifies that punctuation marks, such as the hyphen or period, are removed. The modifier S includes blanks. Those two modifiers should take care of the characters that we identified to be removed. Notice that these modifier values P and S are not quoted here. If we were using the COMPRESS function in the DATA step, then we would need to specify these values in quotation marks. Here the quotation marks aren't necessary, and if we did include them, they could cause errors or unintended results.

Let's walk through the generation of that DATA statement. First, we have the DATA keyword. Then, the %DO loop begins. We are on iteration one, so we write out the value of the macro variable &CITYNAME1 with spaces and punctuation marks removed. When we reach the %END statement, control goes back to the top of the loop. On the second iteration we write out the value of &CITYNAME2 with spaces and punctuation removed. We repeat with &CITYNAME3, and then we looped through all the cities. Next we write out the semicolon that ends the DATA statement.

Following the DATA statement we have SET and WHERE statements. There's no macro content there. Then we have a SELECT statement. This is the conditional step that will output the observations to the appropriate data set depending on the value of the variable **`CITY`** in the data. Here, the SELECT statement names the variable that we're going to use. In this case it's **`CITY`**. Following that, we need to generate the WHEN statements that check the value of **`CITY`** and output matching observations to the appropriate data set. As before, we're going to use a %DO statement with indirect macro variable references to generate the desired code.

Each statement begins with the keyword WHEN followed by an open parenthesis. Then we want to specify the value of the **`CITY`** variable to be matched. The DATA step requires that this value appear in quotation marks, so we're specifying them. We're using double quotation marks because the quoted string will contain macro triggers that we want to resolve. Within the double quotation marks we have an indirect macro variable reference, with no use of %SYSFUNC. Here, we want to write out the value of the city as it exists in the data. Those values are stored in the CITYNAME macro variable series, so there's no need to modify them. Following the closing parenthesis we're going to specify what we want to happen when the value of **`CITY`** matches what we've written out. Well, when there's a match, we want to output to the appropriate data set. So, we complete this statement with OUTPUT and the code to generate the appropriate data set name. This is using %SYSFUNC with the COMPRESS function just as we used it in the DATA statement.

Let's walk through the creation of the series of WHEN statements. On the first iteration, we write out the WHEN statement with an opening parenthesis and a quotation mark, and then the value of the macro variable &CITYNAME1. Then we have the closing quotation mark and the parenthesis, then OUTPUT, and then the value of &CITYNAME1 with spaces and punctuation removed. Then type a semicolon to end the WHEN statement. Then we're finished with the first iteration of the %DO loop. We continue until we've looped through all the values.

Then we have OTHERWISE and END statements to finish the SELECT block and a RUN statement to end the DATA step. That's the end of the macro definition. Now technically we probably could omit the OTHERWISE statement in this situation. The OTHERWISE statement enables you to specify a statement to be executed if no WHEN condition is met. We looked up all of the values of **`CITY`** in the data when we built this code, so as long as we've written the code correctly, one of our WHEN conditions should always be met unless a new city was added between the time the program code was generated and executed. That's very unlikely, but I would say that it's a best practice to have the OTHERWISE statement there. If we did have an observation that didn't meet any of the WHEN conditions and didn't have an OTHERWISE statement, then the step would fail. Here, in the unlikely event that we have that situation, the step will run, but a customized error message will be written to the log.

Let's go ahead and run it and see what happens. We'll turn the MPRINT option on so that we can see the code that is being generated.

Now this particular program didn't create any kind of report, so there's nothing to see in the OUTPUT window. Let's go the log to see what happened. Focus on the generated DATA step code. We see the original city names in the WHEN statement. For example, we have Miami-Dade with a hyphen and San Diego with a space. The corresponding data set names, here in the DATA statement for example, have those characters removed.

Now let me switch over to the final version of the program. In this version, we're identifying the appropriate locations from the data rather than specifying them manually. First, we have a %LOCAL statement, but this time we're only naming CITYCOUNT and NUMCITIES here. We also want our CITYNAME macro variables to be stored locally, but we don't know how many of those there will be yet, so we'll have to specify that later. Following that, we've replaced the %LET statements that we had previously with a PROC SQL step that will assign values to the macro variables. We're using the first query in this PROC SQL step to count the number of unique values of **`CITY`** for the U.S. locations in the source data set and saving the result to a macro variable named &NUMCITIES.

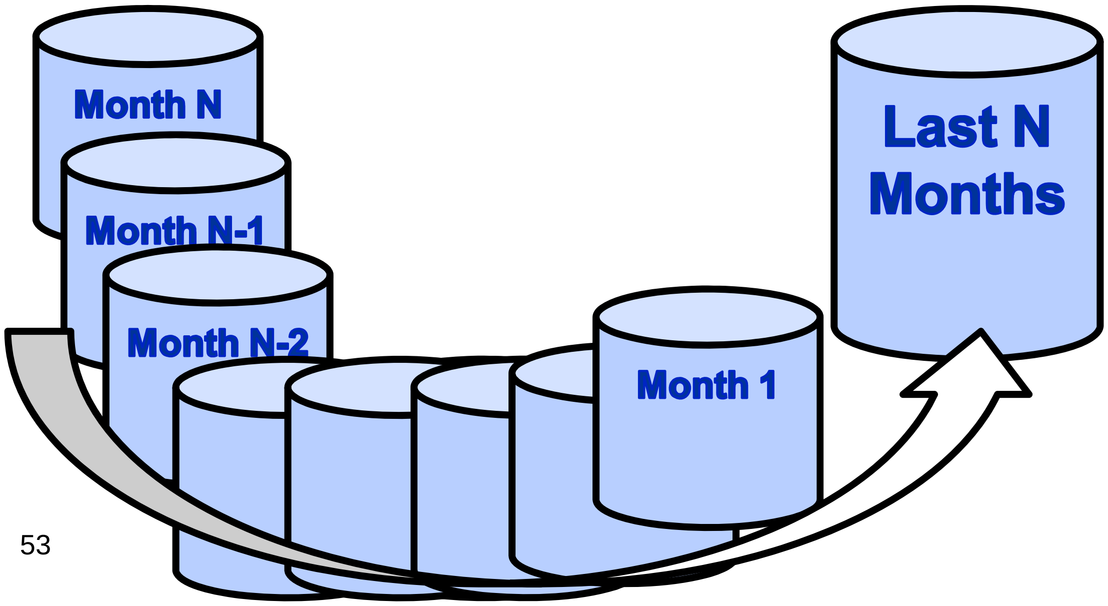
Next, we want to create a series of macro variables that contain the city name for each distinct city in the data. That's what we're doing down here. First of all, we want to store these variables in the local symbol table as before, so we're using a %DO loop to generate a %LOCAL statement for each of the macro variables that we'll be creating. We're looping from 1 up to the value of &NUMCITIES, which we calculated. Then we're writing a %LOCAL statement. The series of macro variables will be named with the prefix CITYNAME followed by a number. The number is being generated by the index variable &CITYCOUNT.

Now let’s look at the query that generates the values of these macro variables. Here we’re naming the series of macro variables in the INTO clause. Each macro variable name begins with the prefix CITYNAME followed by a number from 1 up to the value of &NUMCITIES. This is another place in this program where we’re taking advantage of %SYSFUNC. We need to make sure that the value of &NUMCITIES appears in the final code immediately following the word CITYNAME, with no spaces in between, because macro variable names cannot contain spaces. It’s possible that the value of &NUMCITIES contains leading blanks. We need to delete those or we won’t have a valid macro variable range specified in this INTO clause. We’ll get an error. There are a couple ways to handle those leading blanks, but here we’re using %SYSFUNC to access the LEFT function.

After the PROC SQL step runs, we will have the series of macro variables with the prefix CITYNAME for each value of city in the data. The rest of the program that generates the DATA step is identical to the previous version. Let’s run this version, and then take a look at the log. We get the same results as before, but now the locations are determined dynamically.

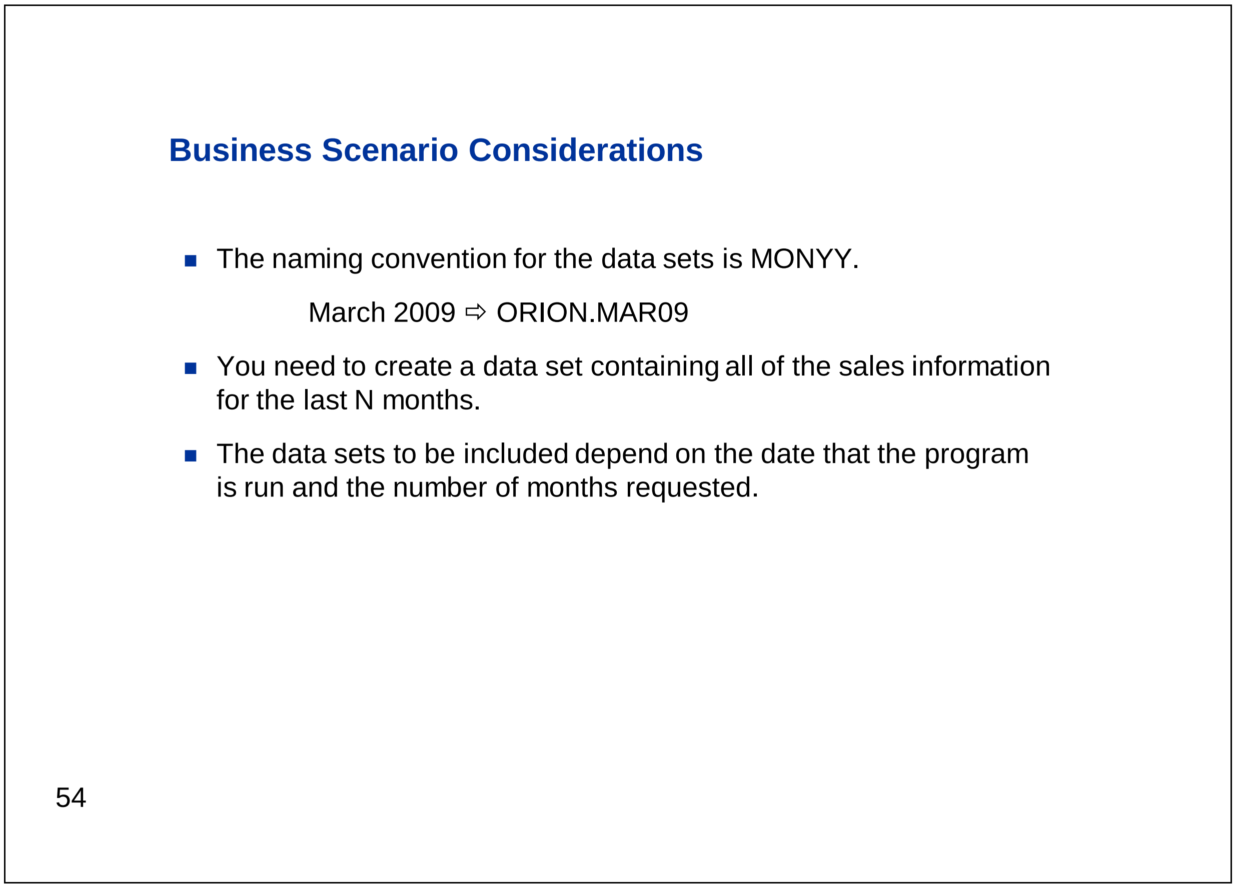
Business Scenario

Build a data set containing the last N months of historical data.



53

Let’s take a look at another example. Here we need to access historical sales data, which has been separated and stored in separate data sets by the month and year of the sale. We need to create a data set containing the last *N* months of history. The number of months requested, *N*, will vary from run to run. We’re going to dynamically build a list of the input data sets to include, based on the current date and the number of months of history that is requested.



Business Scenario Considerations

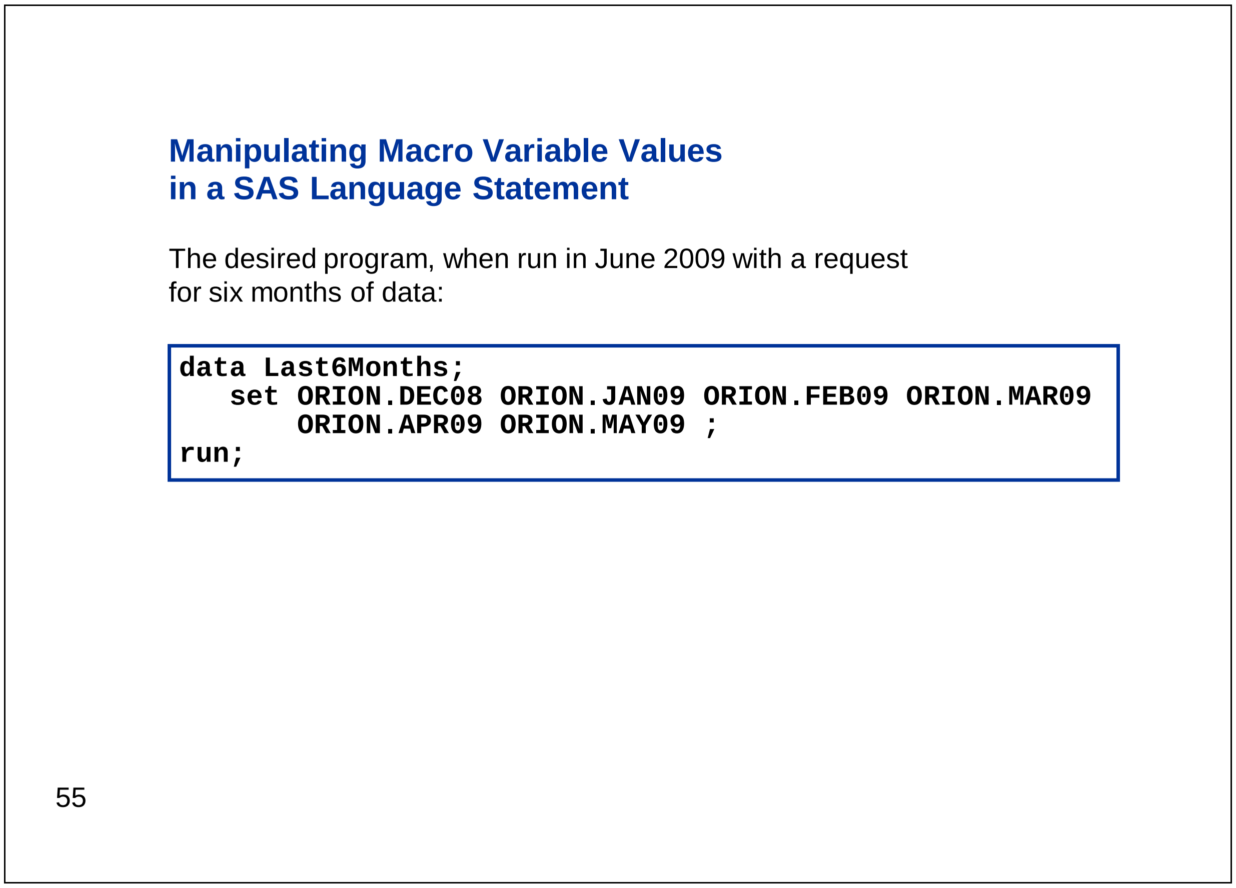
- The naming convention for the data sets is MONYY.

  March 2009 ⇨ ORION.MAR09

- You need to create a data set containing all of the sales information for the last N months.
- The data sets to be included depend on the date that the program is run and the number of months requested.

54

Let’s take a look at some of the details.The data sets that we’ll need to use as input are stored in the **`ORION`** library, and named based on the month and year. The first three characters of the data set name are the abbreviation for the month, such as JAN for January, and the last two characters are the year. We need to combine these into a data set containing sales information for the past *N* months. The specific data sets to include will depend on when the program is run and the number of months of history requested.



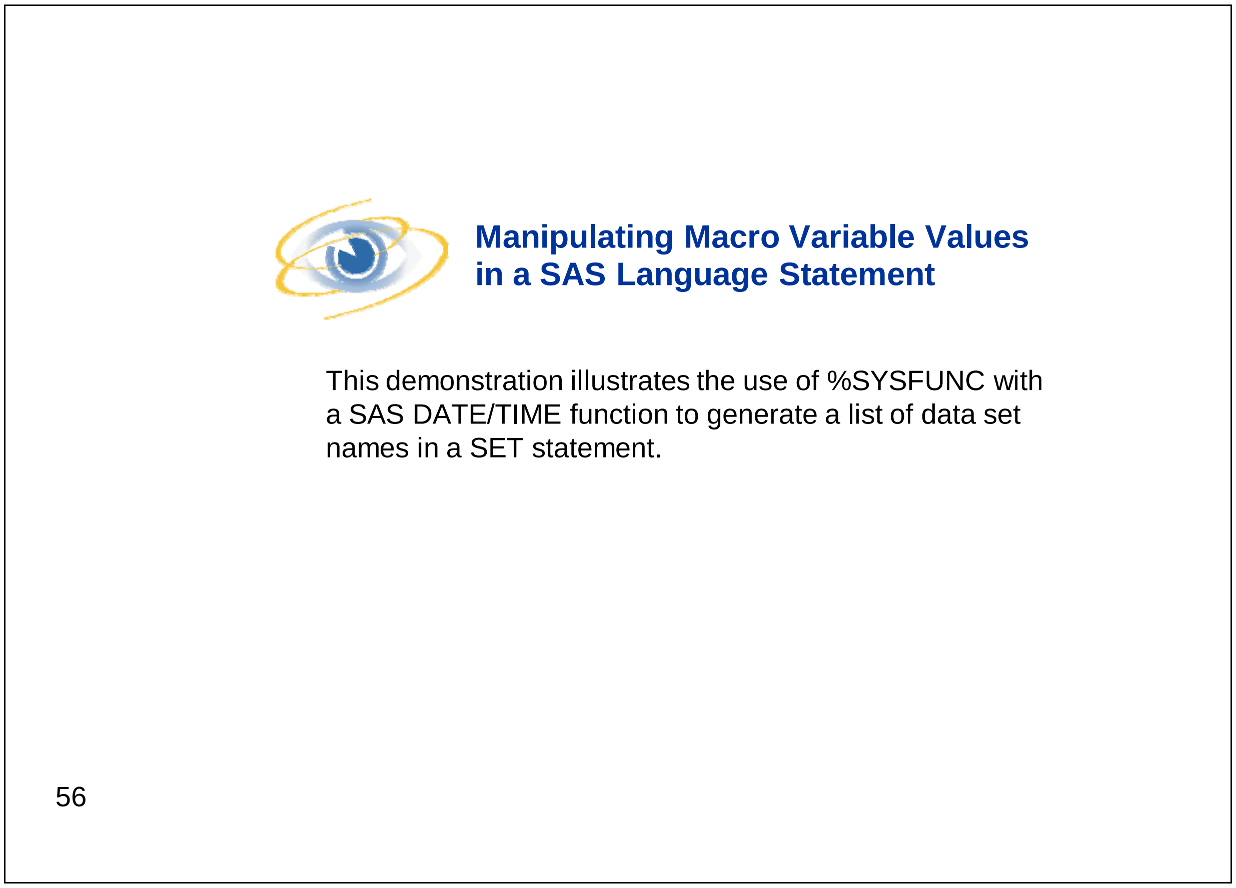
Manipulating Macro Variable Values in a SAS Language Statement

The desired program, when run in June 2009 with a request for six months of data:

```
data Last6Months;
   set ORION.DEC08 ORION.JAN09 ORION.FEB09 ORION.MAR09
       ORION.APR09 ORION.MAY09 ;
run;
```

55

Let’s take a look at the program that we want the macro facility to generate for us. This particular version of the program was run during June of 2009 and based on a request for six months of data. In this scenario, we want full months only, and the month of June could have additional sales at the time that the program was run. May was the last full month of data. Therefore, we have six monthly data sets listed in the SET statement, beginning with December of 2008 and ending with May of 2009.



Manipulating Macro Variable Values in a SAS Language Statement

This demonstration illustrates the use of %SYSFUNC with a SAS DATE/TIME function to generate a list of data set names in a SET statement.

56

I'm going to switch to a SAS session so that we can take a look at the program that generated this result.

Here's the program that dynamically generates the desired code. As in the last program, we're using a macro definition here because we need to do some iterative processing with a %DO loop. The macro definition has one parameter, which is the number of months of data that we want to retrieve. First we have a %LOCAL statement to specify that the macro variable DIFF, which we'll be using as the index variable for the %DO loop, be stored in the local symbol table. As before, that's only a precaution to prevent a macro variable of the same name in another symbol table from being overwritten.

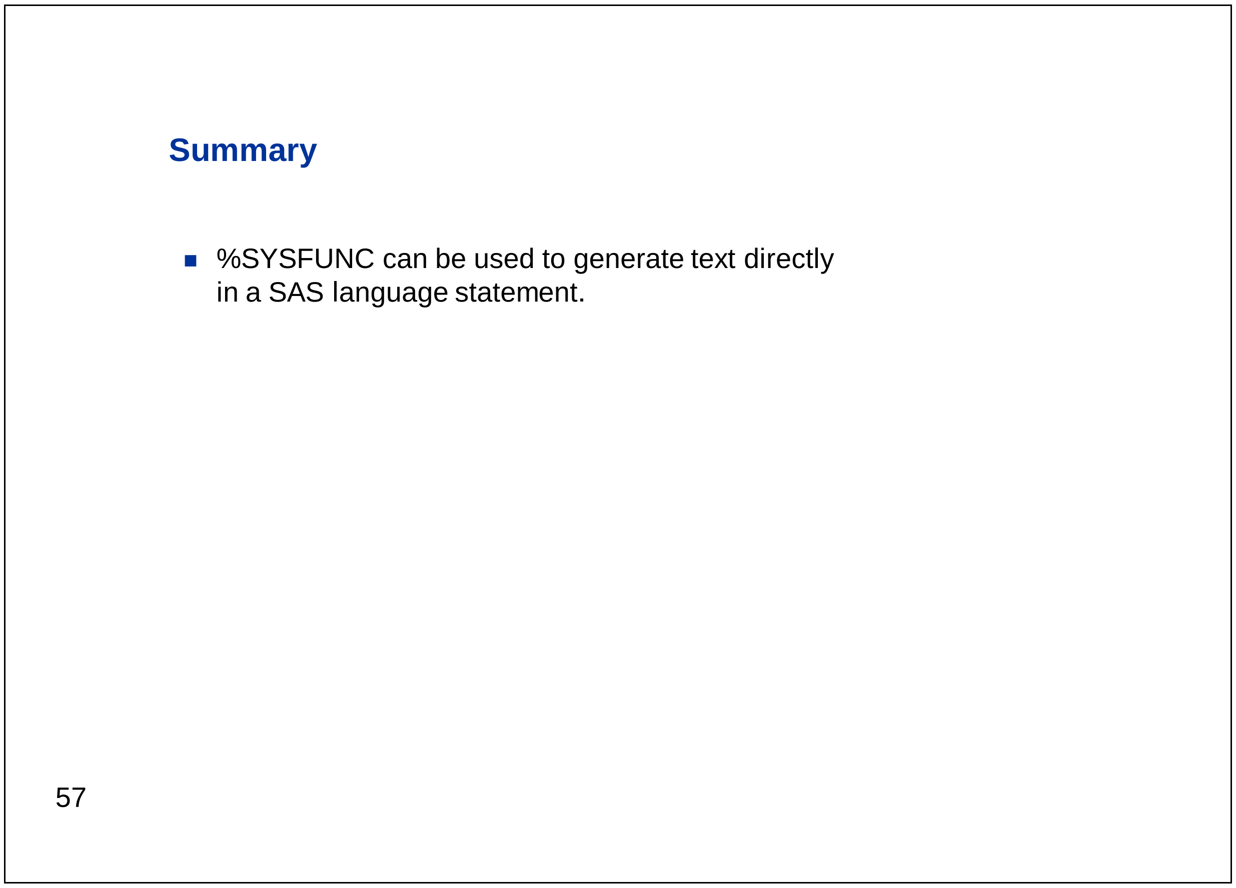
Next we've got a DATA step to build the desired data table. We're using the parameter &MONTHS in the DATA statement to include the number of months requested in the name of the data set that we're building. Then we've got the SET keyword to begin the SET statement. Now we need to generate the list of data sets to appear in the SET statement. That list depends on the current date and the number of months requested. We're going to use a %DO loop to iterate through the number of months requested, from the total number of months down to 1, iterating by -1. We could have iterated in the other direction, but we're doing it in this order so that we get the oldest month listed first in the SET statement. You can do it in either order.

Now let's drill into the text that each iteration of this %DO is generating. First we have a reference to the library that we're reading from; in this case, it's **`ORION`**. That's followed by the period delimiter that we need to separate the libref from the data set name. Then we need to generate the name of the data set for this iteration. We're using %SYSFUNC with the INTNX function to do that. INTNX is a DATE/TIME function. It increments a SAS date, time, or datetime value by a specified increment. For example, we can use it to calculate the previous month and year values based on the current month and year. The first argument to INTNX is the increment that you want to use. We're specifying months to increment through the previous *N* months. Notice that the word **`months`** is not quoted here. It would be if you were using INTNX in the DATA step, but as we already discussed, quotation marks are not appropriate with %SYSFUNC. The second argument to INTNX is the starting point. We're using the TODAY function to capture the current date, and we need another %SYSFUNC to access this second function. Finally, the third argument sets the increment amount. We're using our loop variable, which is &DIFF. We're preceding &DIFF with a negative sign because we want to generate past months. Because we're using it here, INTNX is going to return a date at the beginning of the requested interval; in this case, the interval is months. When the current date is in the month of June and the increment is -1, INTNX will return the first day of the previous month, which is May 1. That ends the call to INTNX, but we are specifying the optional argument to %SYSFUNC, which is the format that we want to apply to the results. INTNX will return a numeric SAS date value by default in this situation. We want the formatted month and year to match the data set names so we're using the MONYY. format.

Let's walk through an example. Suppose, as in the example program, we submit this code in June of 2009 and request six months of history. On the first iteration of the %DO loop, the value of &DIFF is 6. The TODAY() function returns a date in June of 2009, and based on that, the INTNX function will return the first day of the month six months previous, which will be December of 2008. We apply the MONYY. format and get **`ORION.DEC08`**, the name of the first data set that we need. On the next iteration, we get **`ORION.JAN09`**, and so on up to **`ORION.MAY09`**. Then the %DO loop is finished. Following the %END statement, we have the semicolon to end the SET statement and then the RUN statement to end the DATA step.

Let's run this program with a request for six months. Notice that I've got the MPRINT option on, so we'll see the code that the macro definition is generating in the log. Let's look at the log to see that code. We're running this program in August of 2009, and we can see here that we get six months of history beginning with February of 2009 and ending with July. Let's go back to the program and submit it again. I'll change the number of months requested from 6 to 12. Look at the log for this run. Now we get the data sets for August of 2008 through July of 2009.

I just realized that in that last submission I submitted everything in my editor window, including both the macro definition and the macro call. That wasn't necessary. I just needed to call the macro, but that's okay.



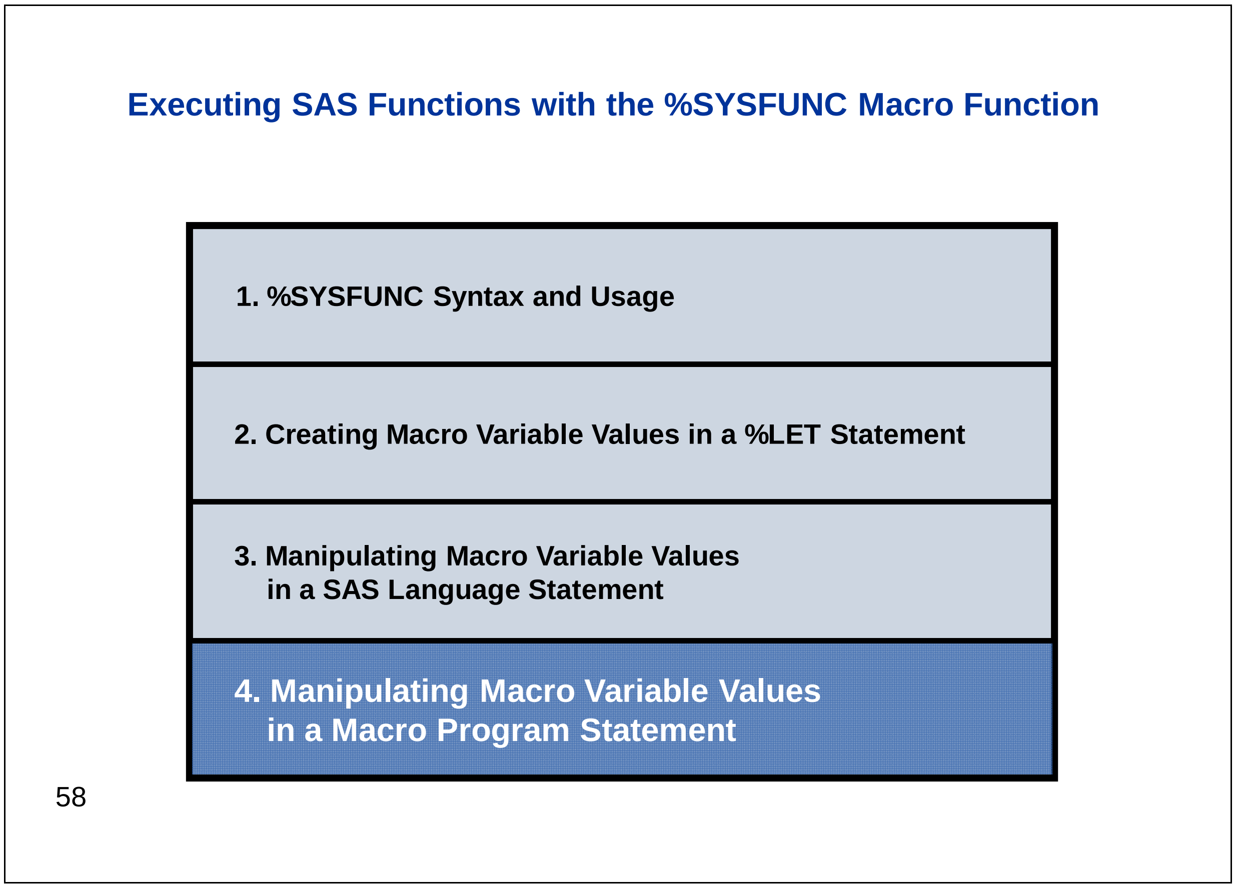
Summary

- %SYSFUNC can be used to generate text directly in a SAS language statement.

57

We saw a couple examples using %SYSFUNC to generate text directly in a SAS language statement. Specifically, we used these capabilities in the context of the DATA step, but you could use these methods with any SAS language statement.

4. Manipulating Macro Variable Values in a Macro Program Statement

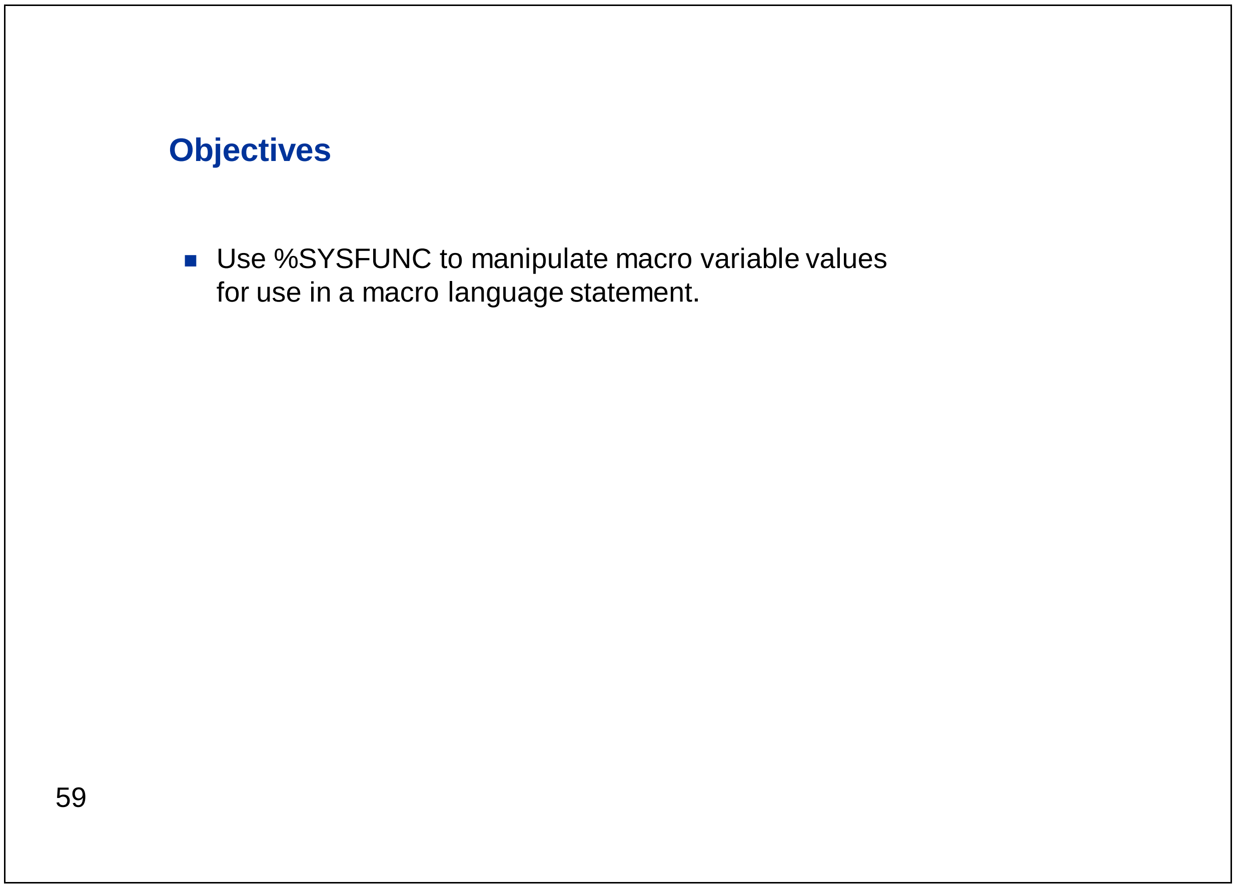


Executing SAS Functions with the %SYSFUNC Macro Function

1. %SYSFUNC Syntax and Usage
2. Creating Macro Variable Values in a %LET Statement
3. Manipulating Macro Variable Values in a SAS Language Statement
4. Manipulating Macro Variable Values in a Macro Program Statement

58

Let’s move on to the final section.

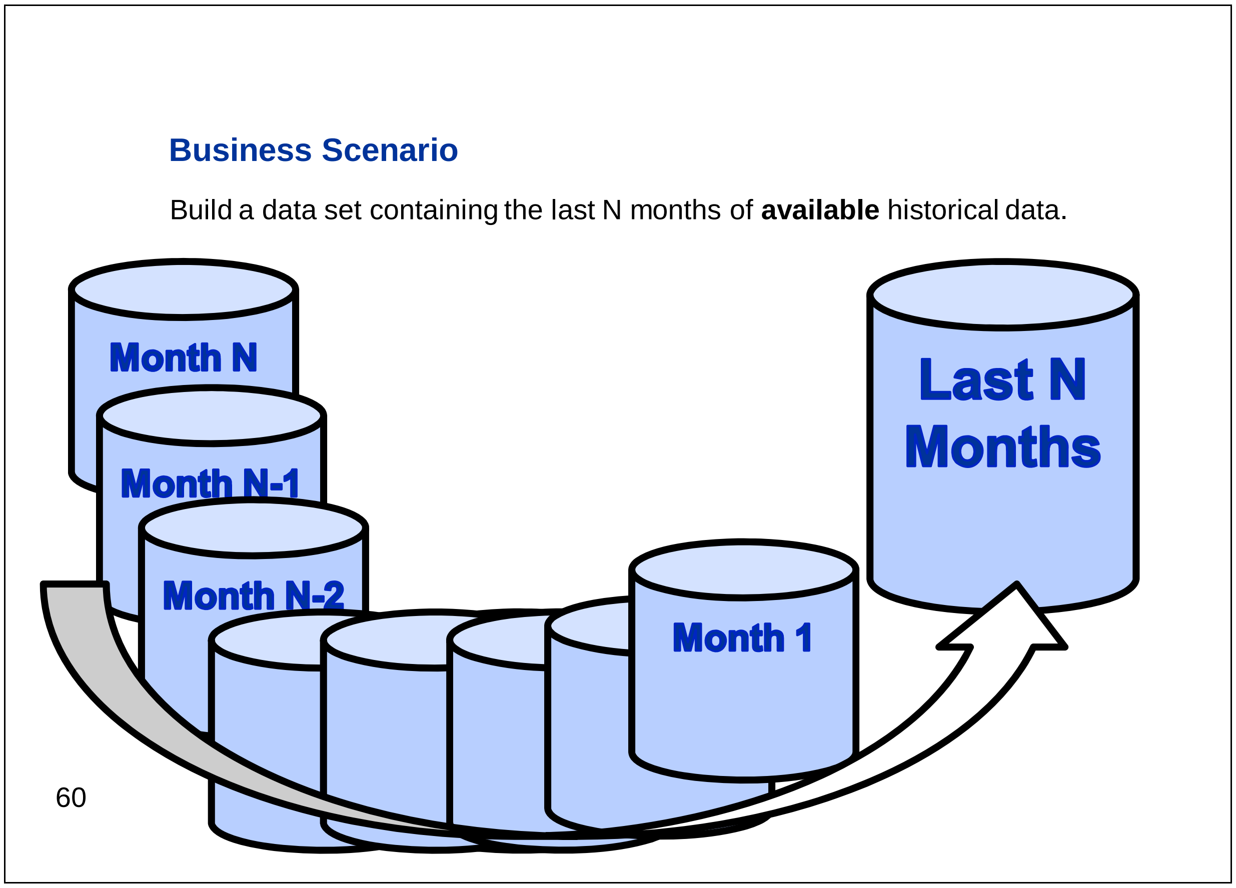


Objectives

- Use %SYSFUNC to manipulate macro variable values for use in a macro language statement.

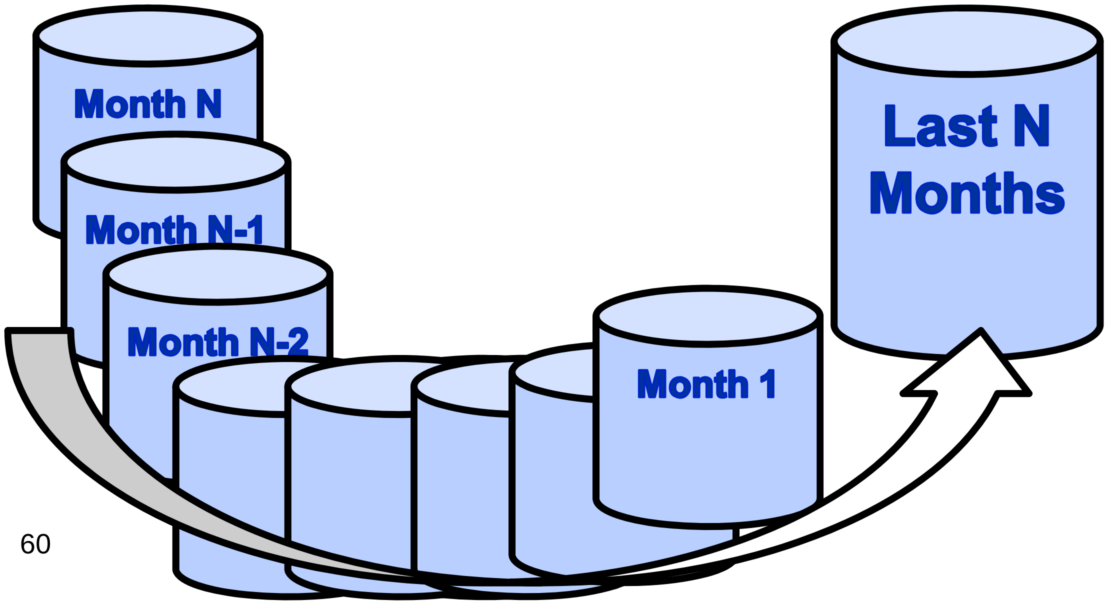
59

Here we’re going to use %SYSFUNC to manipulate values in a macro program statement.



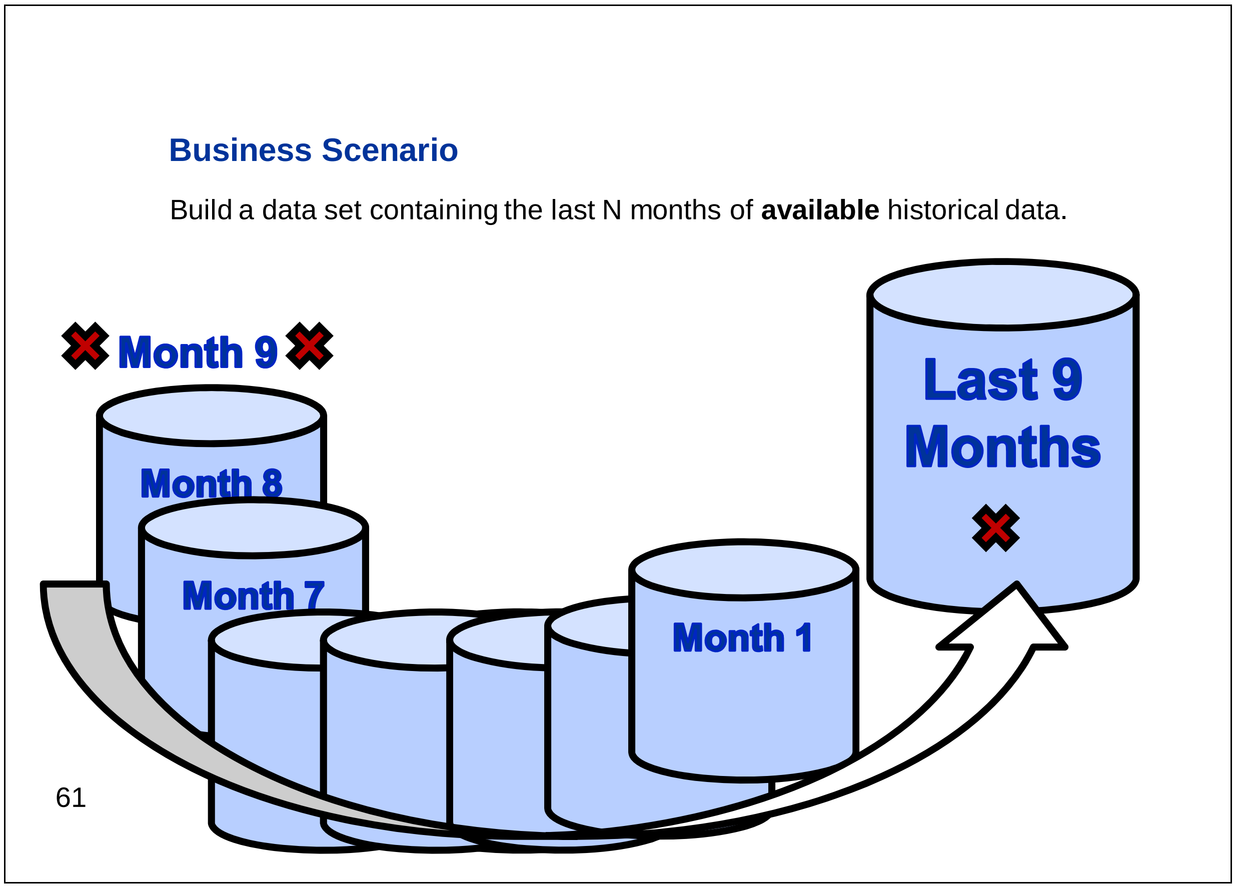
Business Scenario

Build a data set containing the last N months of **available** historical data.



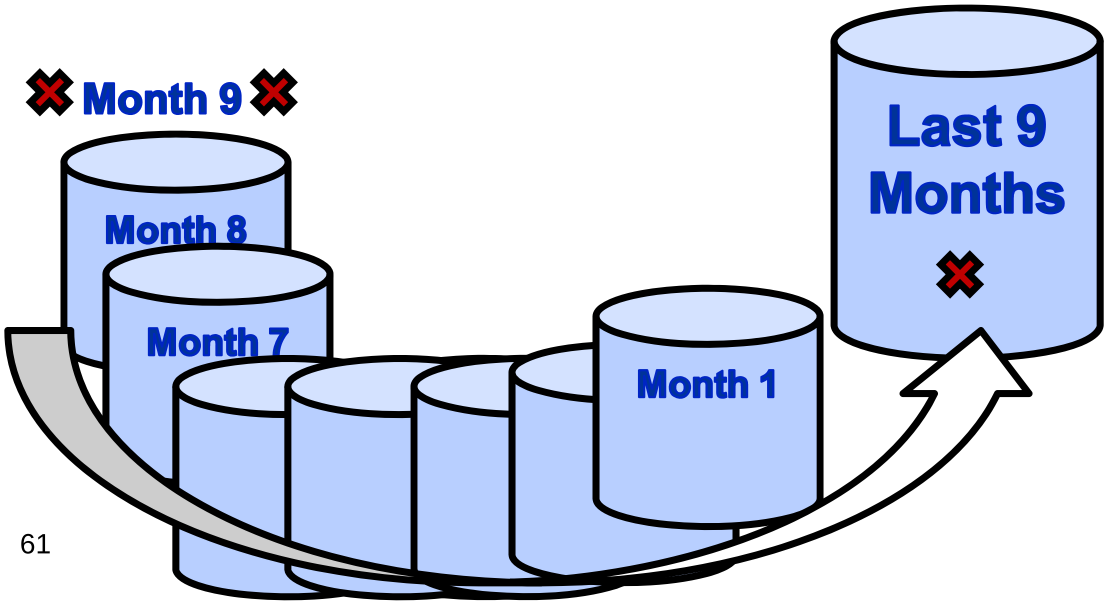
60

In the first scenario for this section, we're going to revisit our last demonstration. That was where we built a list of historical data sets based on the current date and the number of months of history requested. However, we want to make a change to the previous program. It's possible that a user could request more months of historical data than are available. If that happened with the previous program, the DATA step would fail because not all of the data sets listed in the SET statement would exist.



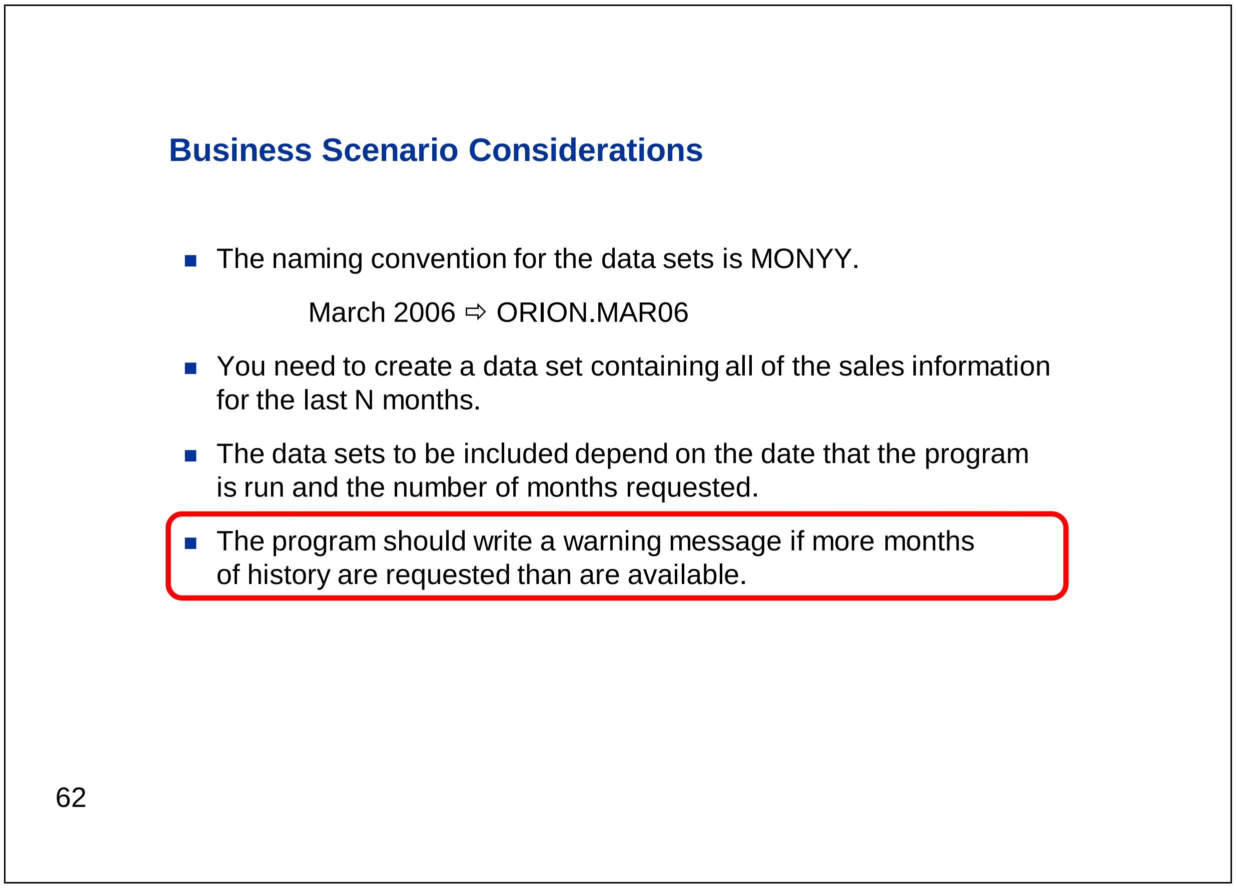
Business Scenario

Build a data set containing the last N months of **available** historical data.



61

For example, let's suppose a user requests nine months of data, but there are only eight available. The DATA step would fail because the data set for the ninth month doesn't exist.



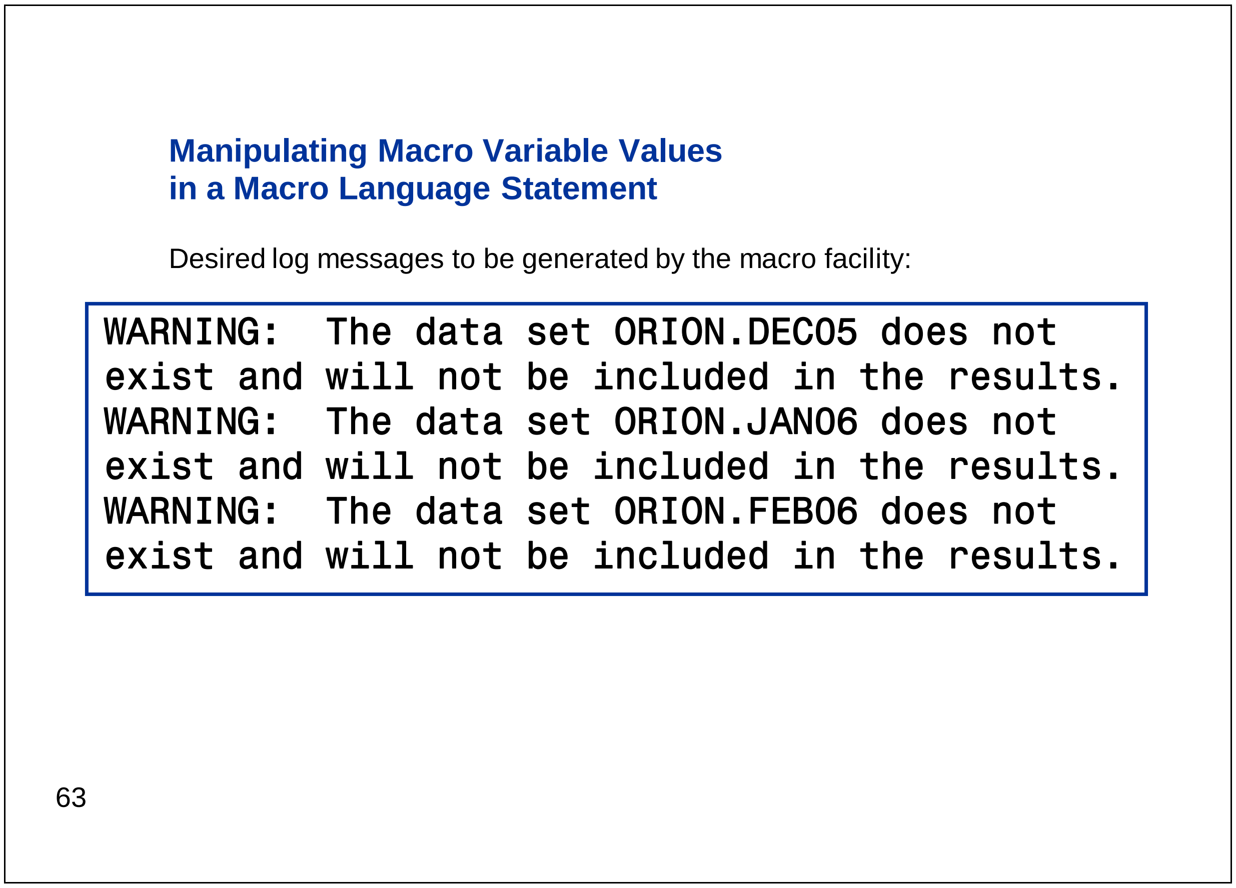
Business Scenario Considerations

- The naming convention for the data sets is MONYY.

  March 2006 ⇨ ORION.MAR06
- You need to create a data set containing all of the sales information for the last N months.
- The data sets to be included depend on the date that the program is run and the number of months requested.
- The program should write a warning message if more months of history are requested than are available.

62

Our scenario here is pretty much as before, but we're adding this final requirement. We're going to modify the previous program to verify that the data sets exist, and write a warning message to the SAS log for any that don't.



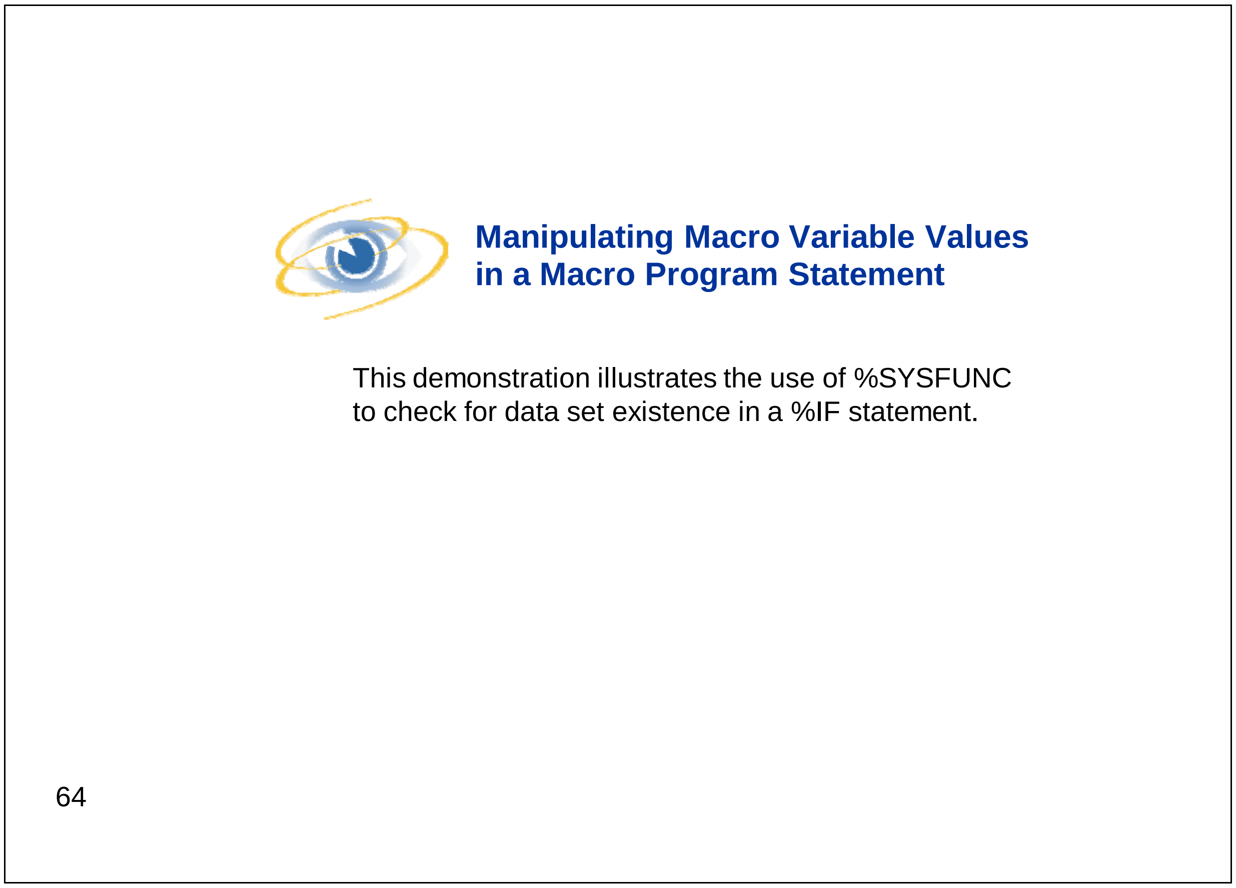
Manipulating Macro Variable Values in a Macro Language Statement

Desired log messages to be generated by the macro facility:

```
WARNING:  The data set ORION.DEC05 does not
exist and will not be included in the results.
WARNING:  The data set ORION.JAN06 does not
exist and will not be included in the results.
WARNING:  The data set ORION.FEB06 does not
exist and will not be included in the results.
```

63

We want the program to work pretty much as before. However, if more months of history are requested than exist, the data set should be created with the number of months available, and warning messages should be written to the log for each data set that isn’t found.



Manipulating Macro Variable Values in a Macro Program Statement

This demonstration illustrates the use of %SYSFUNC to check for data set existence in a %IF statement.

64

Let’s switch to SAS and look at the modified program.

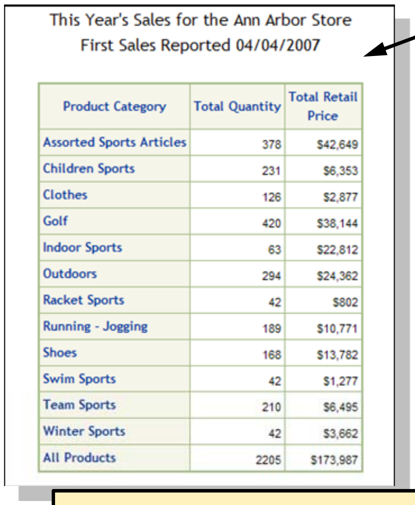
Here’s the modified program. First of all, we have a %LOCAL statement to specify that all of the macro variables created in this macro definition be stored in the local table. The macro variable DIFF will be the index variable for the %DO loop, DSNAME will be the name of the data set for each iteration of the loop, and DSLIST will be used to build a list of the valid data set names. Next comes the %DO loop, which increments in the same way as in the last program. However, what we’re doing within the %DO loop here is different. We’re not generating the list of data sets directly in the SET statement as we did before, because some of them might not exist. Instead, we’re using a %LET statement to build each data set name. We’re using %SYSFUNC with the INTNX and TODAY functions as we did before, so we won’t talk about that. The only difference is that we’re using these %SYSFUNC calls in a %LET statement. Next we see what’s new here. We have a %IF with a %SYSFUNC call. We’re using %SYSFUNC here to access the EXIST function. The argument to the EXIST function is the data set name that we built, stored in the macro variable &DSNAME. The EXIST function will return a 1 if the data set exists and a 0 if not. If it returns a value of 0, we’ll write a warning message to the log. Otherwise, we’ll append the data set name for the given iteration to the list of valid data set names contained in the macro variable &DSLIST. After we’ve iterated through all of the selected months, &DSLIST will contain the existing data sets within the number of months requested. We can use that macro variable in the SET statement in the DATA step that ends this program.

Let’s run this program requesting six months of history as we did before. If we look at the log, we see that we get the same results as before. Now let’s run it again requesting 36 months of data. This time, we see warning messages in the log because we don’t have that much history. However, the data set is still created with the history that we do have.

I just realized that in that last submission I submitted everything in my editor window, including both the macro definition and the macro call. That wasn’t necessary. I just needed to call the macro, but that’s okay.

Business Scenario

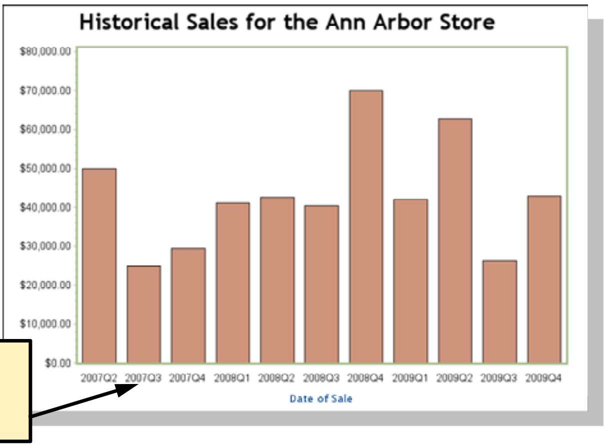
A sales report for current store locations includes tabular and graphical components.



This Year's Sales for the Ann Arbor Store
First Sales Reported 04/04/2007

Product Category	Total Quantity	Total Retail Price
Assorted Sports Articles	378	$42,649
Children Sports	231	$6,353
Clothes	126	$2,877
Golf	420	$38,144
Indoor Sports	63	$22,812
Outdoors	294	$24,362
Racket Sports	42	$802
Running - Jogging	189	$10,771
Shoes	168	$13,782
Swim Sports	42	$1,277
Team Sports	210	$6,495
Winter Sports	42	$3,662
All Products	2205	$173,987

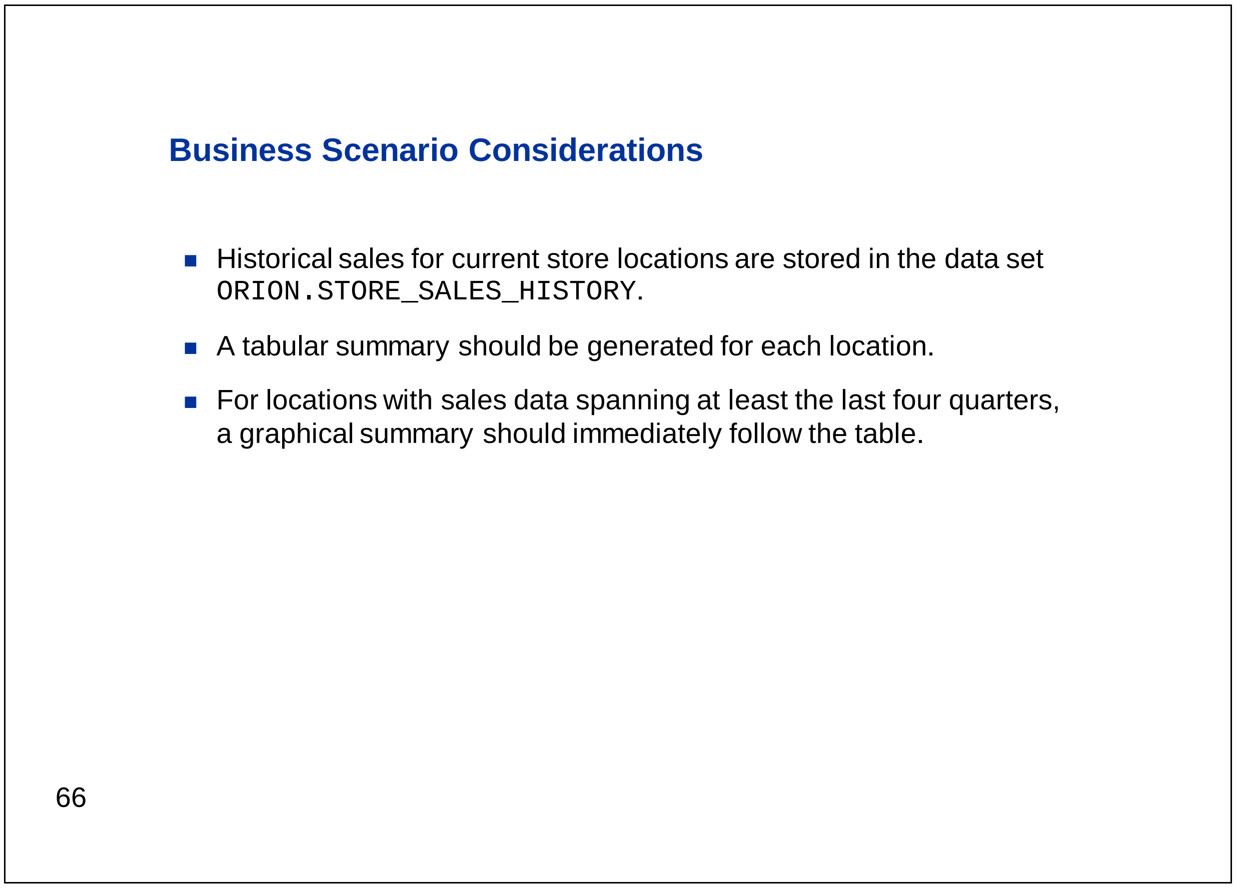
A table is included for every store location.



65

A graph is included for some but not all locations.

In the final scenario, we want to generate some SAS code conditionally. We want to create a report that includes tabular and graphical summaries of sales for the current store locations. The table is a summary of this year's sales for the particular store, while the graph is a summary of quarterly sales since the store has been open. The tabular summary will be included in the report for every store, but the graph will be included only for stores that have historical sales data spanning at least the last four quarters. This slide shows the results for the Ann Arbor store, which opened in 2007. It has more than four quarters of historical data, so it meets the requirement to create the graph.

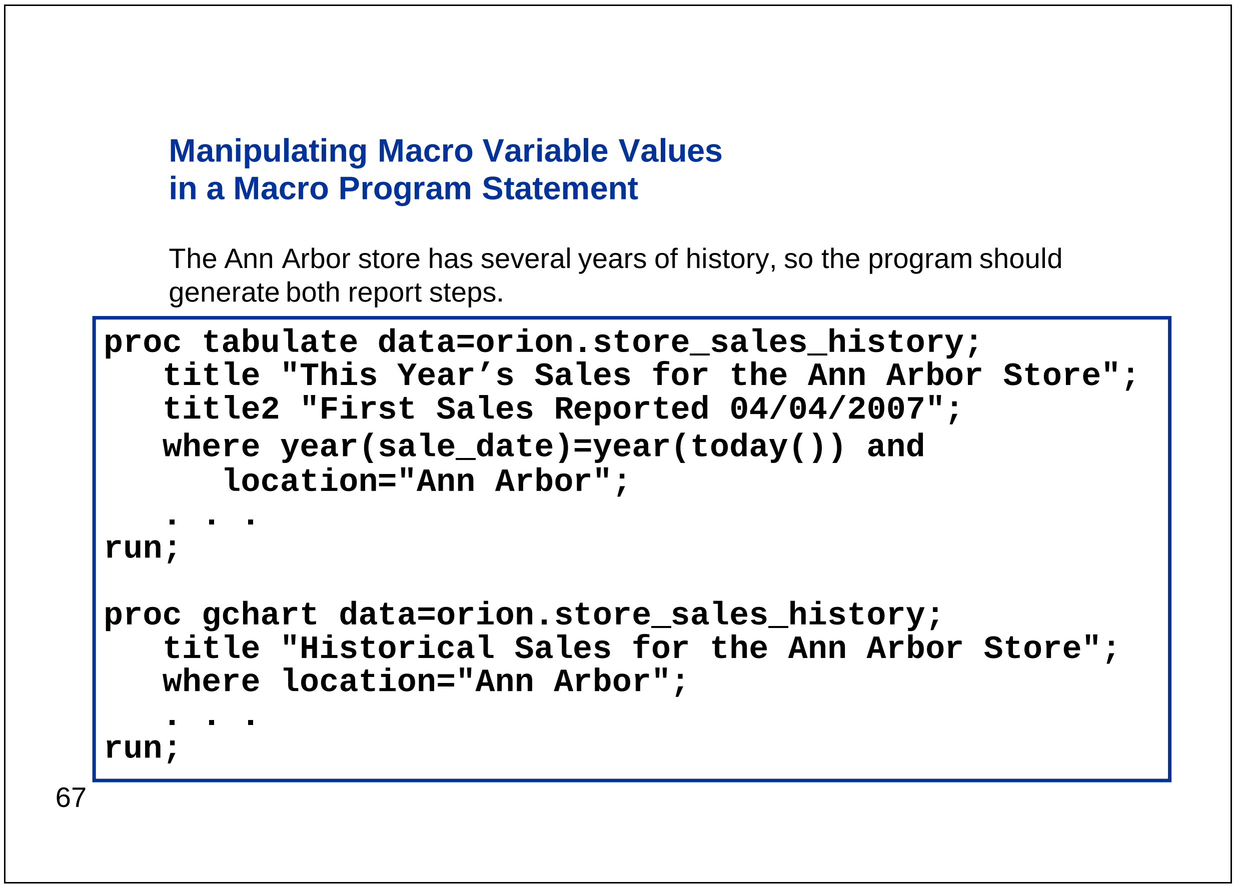


Business Scenario Considerations

- Historical sales for current store locations are stored in the data set `ORION.STORE_SALES_HISTORY`.
- A tabular summary should be generated for each location.
- For locations with sales data spanning at least the last four quarters, a graphical summary should immediately follow the table.

66

Let's take a look at some of the details. The source data for all of the stores is stored in the data set **`orion.store_sales_history`**. We want the report to contain a separate tabular summary for each store, immediately followed by the graphical summary if that store meets the four-quarter condition. So, we're going to need to loop through each store location in the data. We'll generate the code to produce the tabular summary and then, if the particular location meets the condition, we'll follow that with the code to produce the graphical summary.



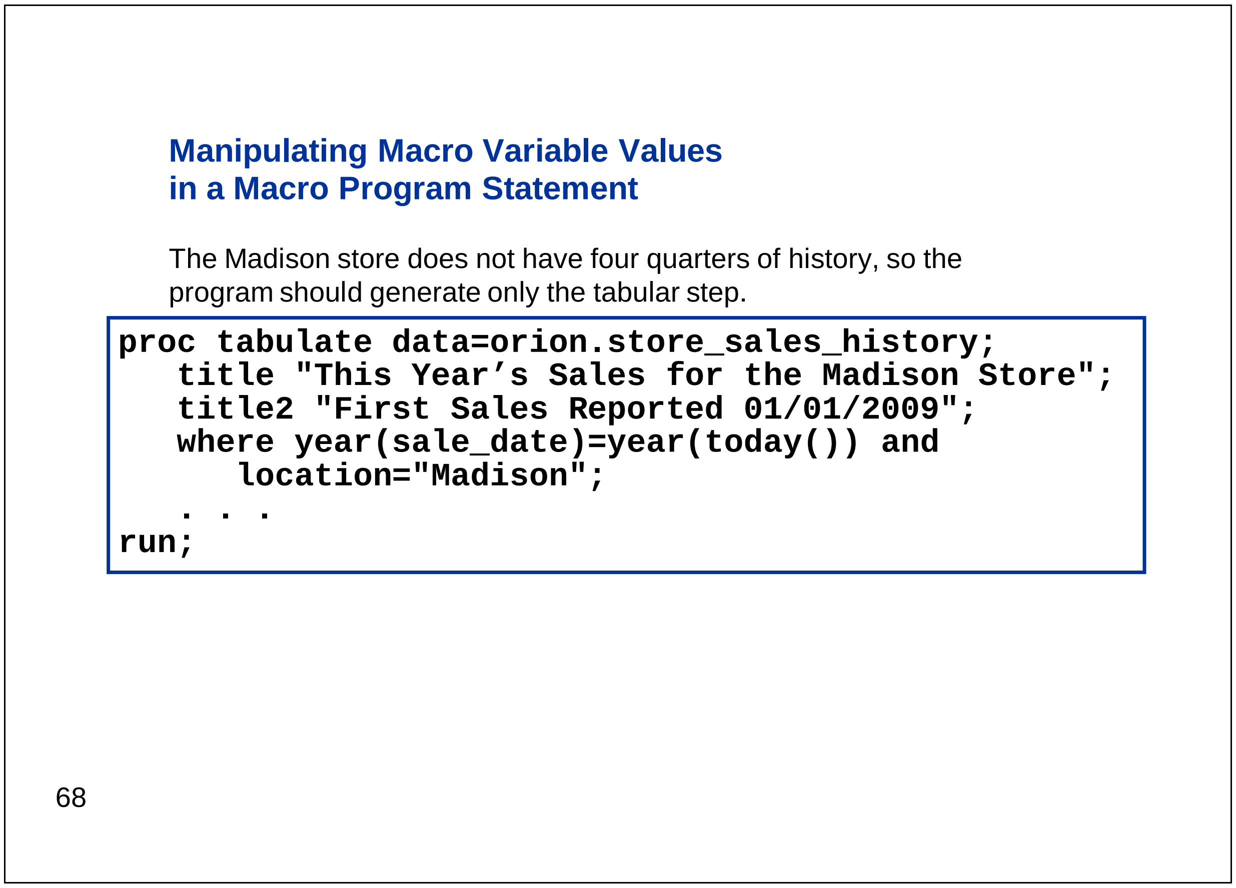
Manipulating Macro Variable Values in a Macro Program Statement

The Ann Arbor store has several years of history, so the program should generate both report steps.

```
proc tabulate data=orion.store_sales_history;
   title "This Year’s Sales for the Ann Arbor Store";
   title2 "First Sales Reported 04/04/2007";
   where year(sale_date)=year(today()) and
      location="Ann Arbor";
   . . .
run;

proc gchart data=orion.store_sales_history;
   title "Historical Sales for the Ann Arbor Store";
   where location="Ann Arbor";
   . . .
run;
```

67

Now this program will actually generate quite a bit of code, so we’ll only look at a couple of sections. We’re using PROC TABULATE to generate the tabular summary and PROC GCHART for the graphical portion. We have a store located in Ann Arbor that’s been open for a few years. We have more than enough data to meet the four-quarter requirement to create the graph. So, the code for Ann Arbor includes both the PROC TABULATE and the PROC GCHART steps.



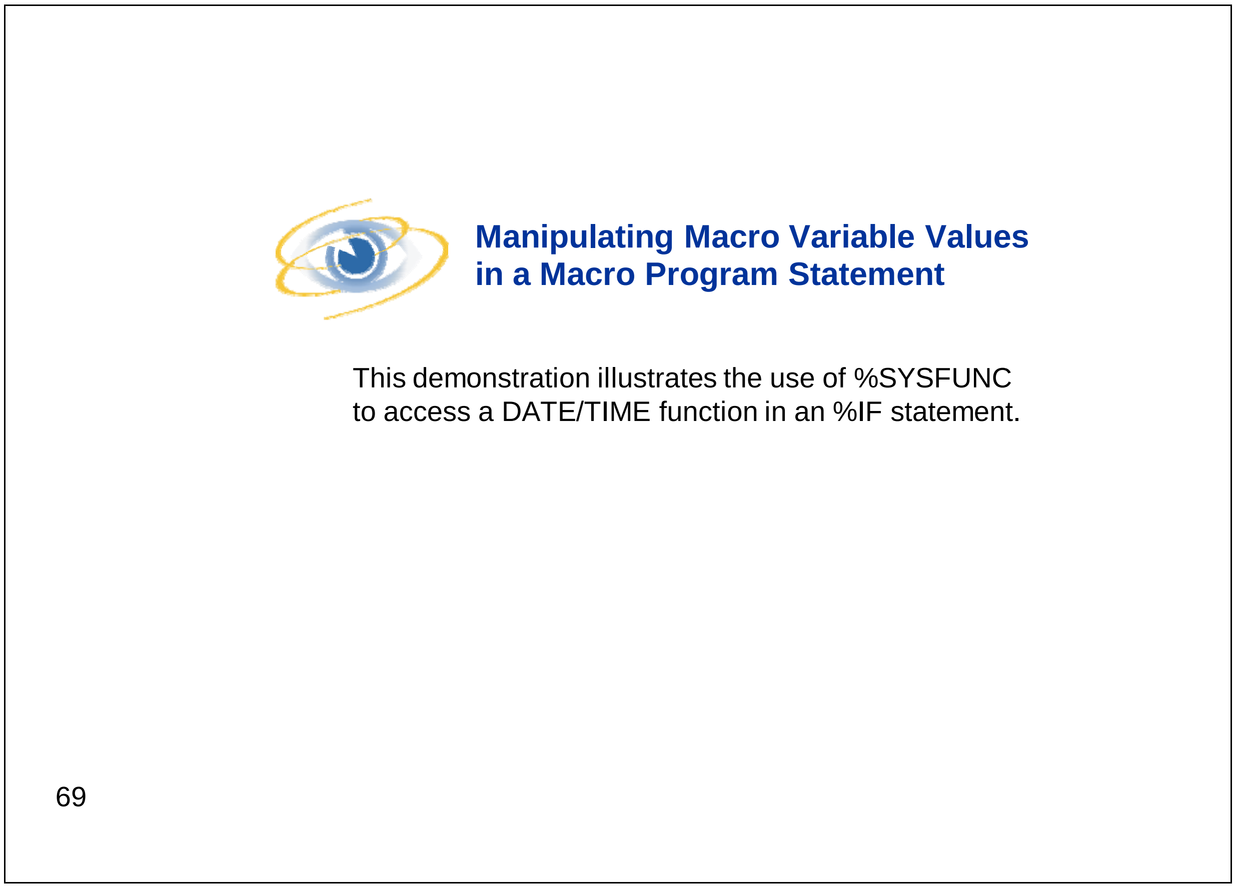
Manipulating Macro Variable Values in a Macro Program Statement

The Madison store does not have four quarters of history, so the program should generate only the tabular step.

```
proc tabulate data=orion.store_sales_history;
   title "This Year’s Sales for the Madison Store";
   title2 "First Sales Reported 01/01/2009";
   where year(sale_date)=year(today()) and
      location="Madison";
   . . .
run;
```

68

We have another store, located in Madison, that opened more recently and doesn’t have four quarters of historical data. The code for Madison, therefore, includes only the PROC TABULATE step. According to our criteria, there isn’t enough historical data to create the graph. The PROC GCHART code wasn’t generated for the Madison store because the condition wasn’t met.



Manipulating Macro Variable Values in a Macro Program Statement

This demonstration illustrates the use of %SYSFUNC to access a DATE/TIME function in an %IF statement.

69

Let’s switch to the SAS session to look at the program that generates this code.

Here’s the program to create the report. We’re using a macro definition because we’re going to be using both %DO and %IF statements in this case. First, we’re using PROC SUMMARY to determine the earliest sale date for each location. We need that date so that we can determine whether there is enough historical data for the graphical part of the report. We’re using the CLASS statement to request statistics for each location, and specifying the MIN statistic to get the earliest sale date. I’m going to run this one step so that we can take a quick look at the data set that’s generated because we’re going to be using it to create several macro variables. It’s stored in the Work library, so I’ll navigate there and we can take a look. Here’s the data. Now I’ll switch the view so that we’re seeing the variable names rather than the labels. We’ve got one observation for each store location and a variable named **`FIRST_SALE`**, which is the earliest sale date for that store. We’re going to use this data to create two series of macro variables. The first series will contain the location name stored here in the variable **`LOCATION`**. The second series will contain the first sale date for each location, which is stored here in the variable **`FIRST_SALE`**. Here we’re seeing values with a date format applied, but the underlying values are numeric SAS dates.

Let’s close the data set and go back to the program. To create those macro variables, we’re using the output data set from PROC SUMMARY in a DATA _NULL_ step with calls to the SYMPUTX routine. Remember that the first argument to CALL SYMPUTX determines the name of the macro variable and the second argument determines its value.

So the first series of variables will be created in this first call to SYMPUTX. They will be named &LOCATION1, &LOCATION2, and so on, up to the number of locations in the data. We're using the automatic variable _N_ to append an appropriate number to the prefix LOCATION. The value for this series of macro variables, specified here in the second argument, will be the value of the variable **`LOCATION`** for the current observation. We're also using the optional third argument to SYMPUTX to specify that this series of variables be stored in the local symbol table.

Down here, the second series of macro variables will be named &FIRSTSALE1, &FIRSTSALE2, and so on, and will have values corresponding to the earliest sale date for the given location, which is the value of the variable **`FIRST_SALE`**. Remember that the variable **`FIRST_SALE`** is a SAS date value, so the macro variables will store those numeric values as text.

Finally, we want to capture the total number of store locations. We need to loop through each store location to create the report, so we'll use that value as the ending point for the %DO loop. Here, we're using the variable **`DONE`**, which was named in the END= option in the SET statement. It will have a value of 1 when we've read the last observation from the data set. At that point, _N_ will be equal to the total number of locations in the data, so we have a final call to SYMPUTX to create a macro variable named &NUMLOCS with that value.

Next we've got a %DO loop to iterate through each of the locations in the data. We want to create the tabular report for every location, so the PROC TABULATE step is next, and it's not conditional. We won't look at the details of this step, except to say that we're using indirect macro variable references using the index variable from the %DO loop to subset the data for the appropriate location here in the WHERE statement. This WHERE statement is also selecting the current year's sales only.

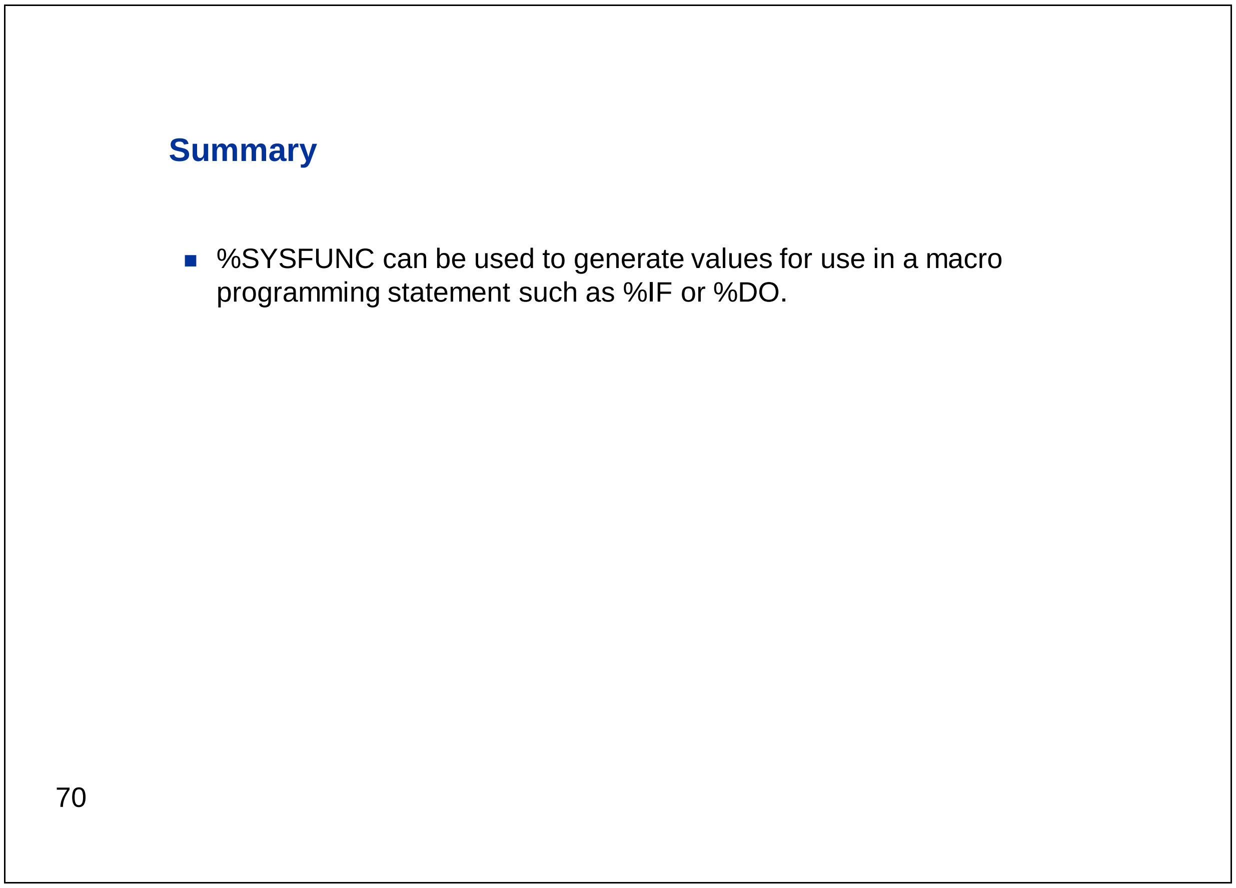
Let's talk a little about this TITLE statement. Here, we're using %SYSFUNC with the PUTN function to write out the date of the first sale for this location as a calendar date. We're using the MMDDYY10. format to write that date in the form of a numeric month, day, and year. Remember that the FIRSTSALE macro variables are storing unformatted SAS date values as text. We will need them in that form shortly, but here we want to write a calendar date. If you've used the PUT function before, you might wonder why we're using PUTN instead of PUT. The reason is that PUT is one of the functions that can't be used with %SYSFUNC. However, there are two alternative functions, PUTN and PUTC, that can be used with %SYSFUNC. You use PUTN to return a value using a numeric format and PUTC to apply a character format. Similarly, the INPUT function can't be used with %SYSFUNC, but INPUTN and INPUTC can be.

Let's move on. Following the PROC TABULATE report, we want to generate a bar chart only if the time between the first sale at the store and today's date spans at least four quarters. Here we're using %SYSFUNC again. Previously we used the DATE/TIME function INTNX to increment a SAS date value by a specified amount. Here, we're using a different DATE/TIME function, INTCK. It will return the number of intervals between two dates. So INTNX and INTCK are opposites, in a sense. In this case, the type of interval that we want to increment is quarters, so that's the first argument to the function. The second argument is the date we want to start with. In this case we have an indirect reference to the FIRSTSALE macro variable for this location. The third argument is the date that we want to end with. Here we have another instance of %SYSFUNC to return the current date using the TODAY function. These stores are all currently open and generating sales, so we're only going to use the current date although we could do something more complicated if necessary. Now, the INTCK function will return the number of times that a new quarter started between the first sale date for the current store location and today's date. As long as that value is greater than or equal to 3, we will have data spanning four quarters. Although the first and last quarter might be incomplete, we can have only part of the first quarter and part of the fourth, but the data will span four quarters. So, if the value returned by INTCK is greater than or equal to 3, we'll generate the PROC GCHART code. Notice that this step includes the same WHERE condition to subset the data for the current store location that we had in the PROC TABULATE step.

Following the GCHART step, we have a %END statement to end the %IF statement. Now we're finished with processing for the current store location, so we have another %END statement to end the %DO loop. That completes the macro definition, so we've got the %MEND statement. Down here at the bottom, we're directing the output to HTML, closing the listing destination, and then calling the macro.

Let's run the program and see what happens. Notice that we've got the MPRINT and MLOGIC options turned on. It will take a few seconds for the program to run.

There, it's finished, so let's take a look at the log. I'll scroll down to where the macro begins executing. Here we see the MLOGIC message that tells us that the macro is beginning execution. After that we see the PROC SUMMARY and DATA _NULL_ steps that we used to capture the location names and dates. Next we see the %DO loop beginning with the first location. We can see from the MPRINT messages that the first location is Ann Arbor. Here's the generated PROC TABULATE step. Following that, we see in the MLOGIC message that the %IF condition based on the date was true for Ann Arbor. We have four or more quarters of data, so the macro facility adds the PROC GCHART step for this location. Next we go to the next iteration, which is for the Bloomington store. This location also has enough data so we see that both steps are generated. On the third iteration, however, which is Columbus, the %IF condition is false. So the macro facility does not generate the bar chart code for Columbus, and we go on to the next location. It's Madison, which also has insufficient data, so we only see the PROC TABULATE step. Let's look at the HTML output. Here we see that we did get both elements of the report for Ann Arbor and Bloomington. If we scroll down further, we see that only the tabular report was created for Columbus and Madison.

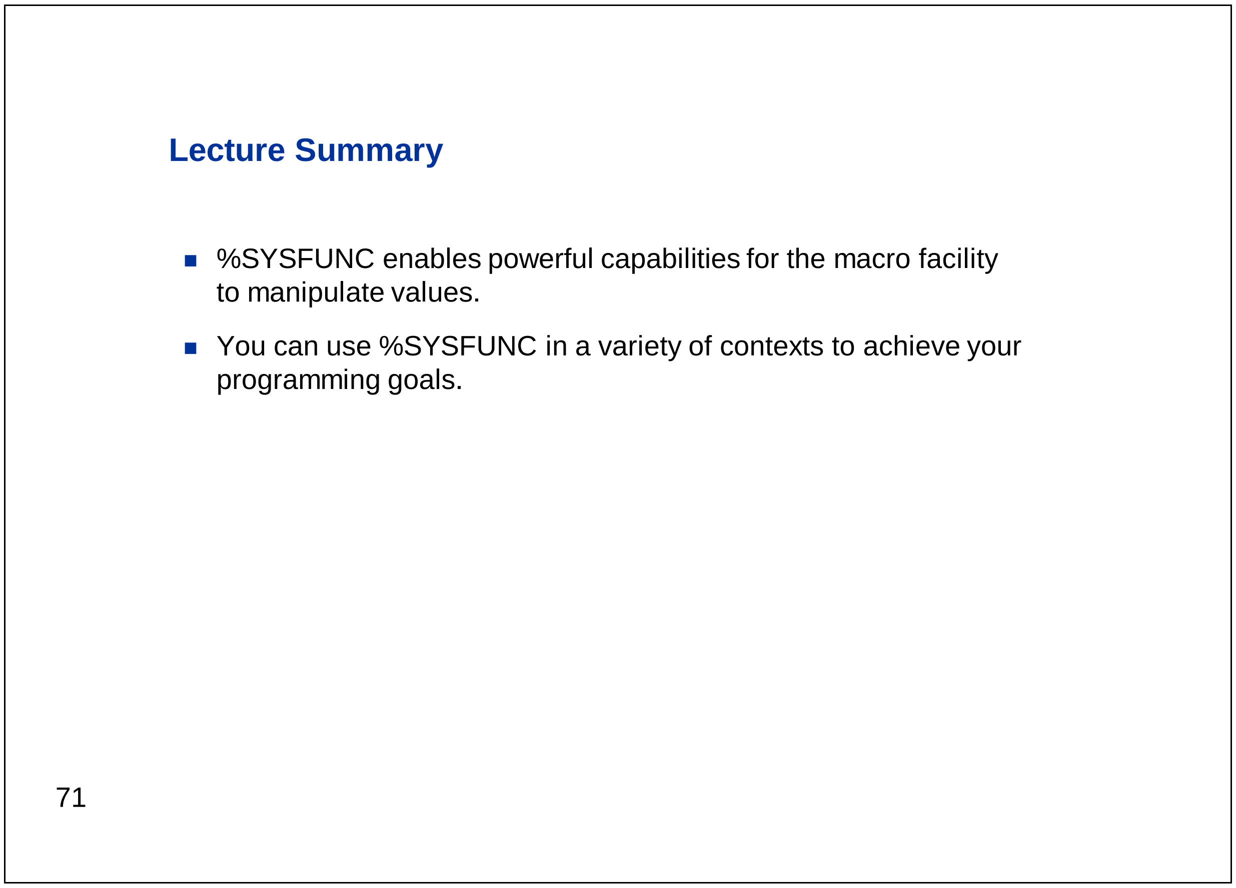


Summary

- %SYSFUNC can be used to generate values for use in a macro programming statement such as %IF or %DO.

70

Here in the final section, we’ve seen a couple examples using %SYSFUNC to generate values for use in macro programming statements.

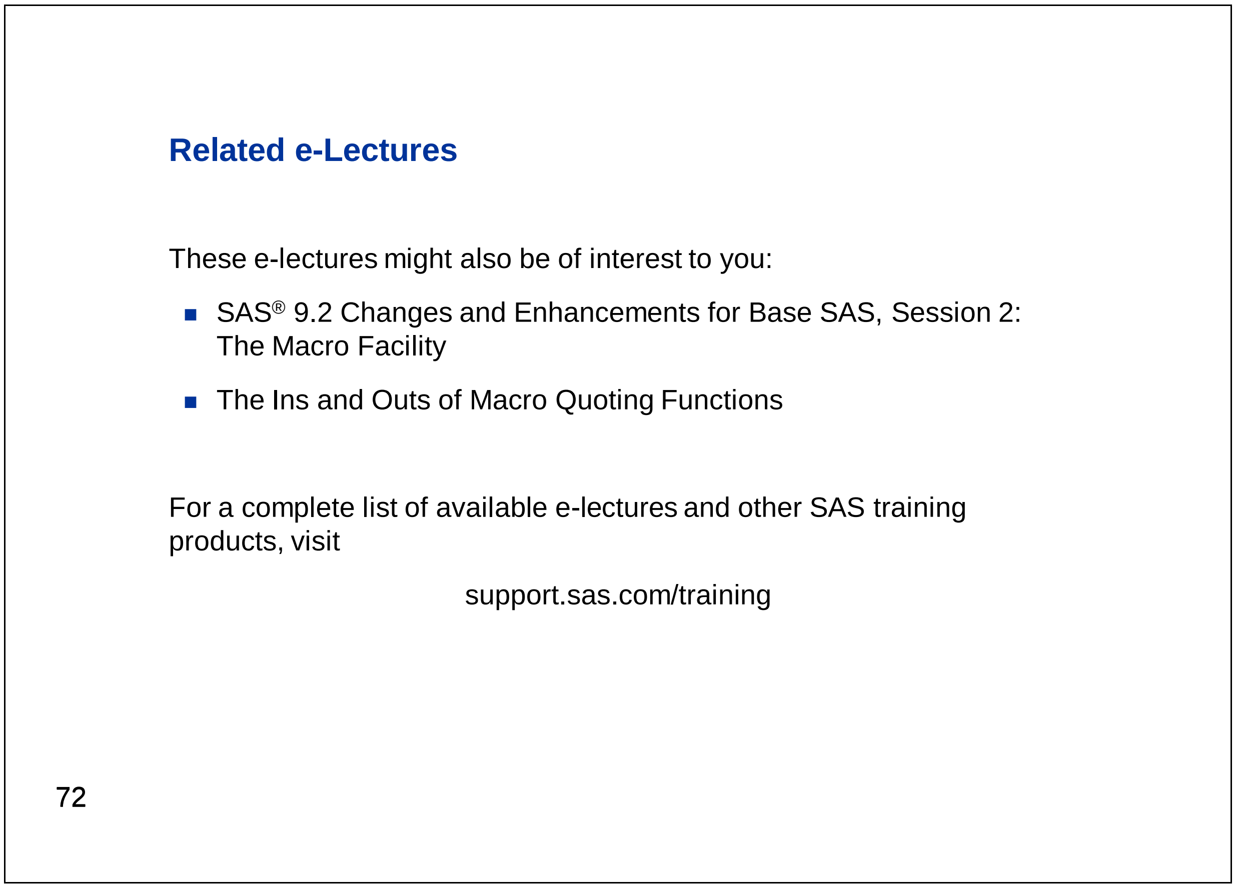


Lecture Summary

- %SYSFUNC enables powerful capabilities for the macro facility to manipulate values.
- You can use %SYSFUNC in a variety of contexts to achieve your programming goals.

71

Even though we’ve seen only a few examples of applications for %SYSFUNC, I hope you’re excited about the possibilities it opens up for you in your SAS programs. It enables you to take advantage of a broad range of SAS language function capabilities in the macro facility. As we’ve seen, you can use %SYSFUNC in a variety of programming situations: to create macro variables, to generate text in a SAS language statement, or to generate results for use in a macro language statement.



Related e-Lectures

These e-lectures might also be of interest to you:

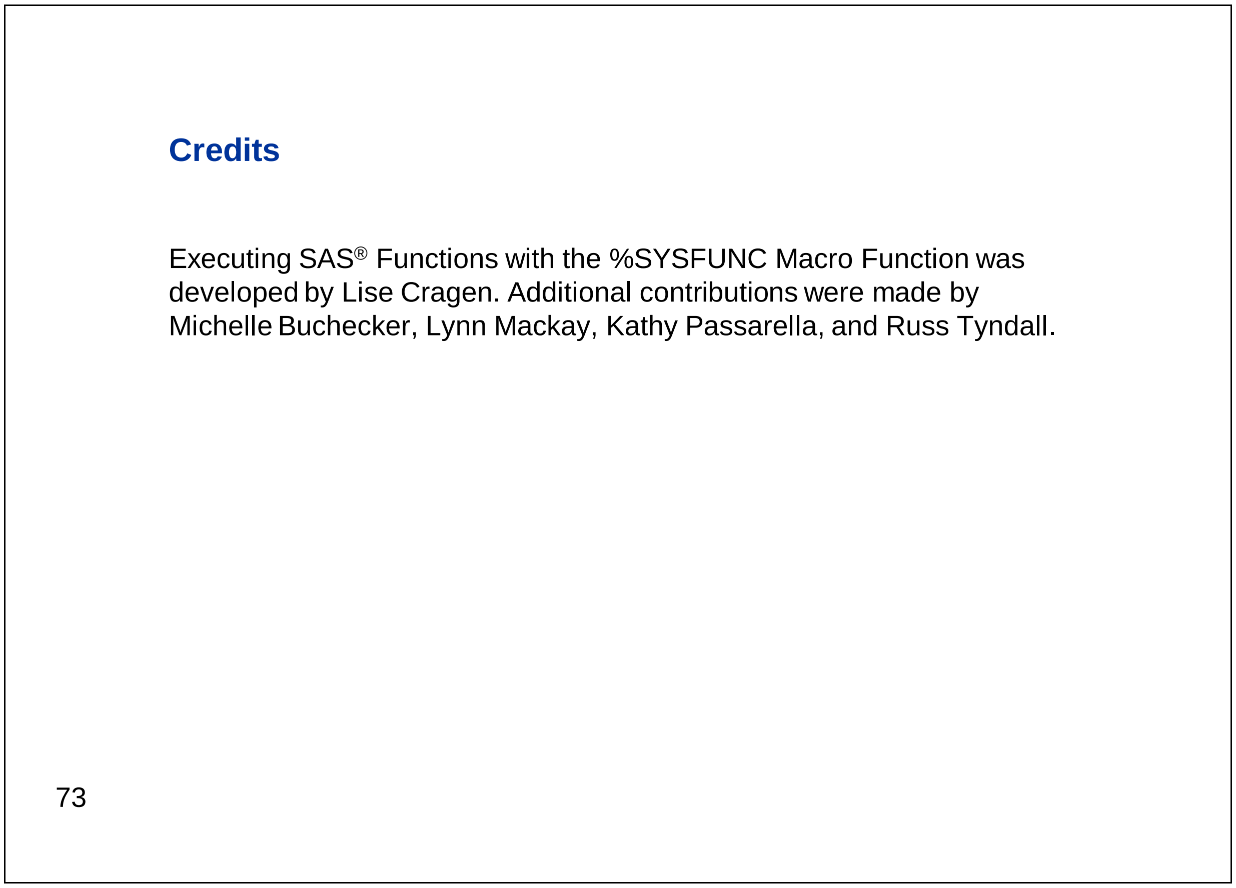
- SAS® 9.2 Changes and Enhancements for Base SAS, Session 2: The Macro Facility
- The Ins and Outs of Macro Quoting Functions

For a complete list of available e-lectures and other SAS training products, visit

support.sas.com/training

72

Listed here are other lectures that might interest you. For a complete list of available e-lectures and other SAS training products, please visit the SAS Web site at support.sas.com/training.



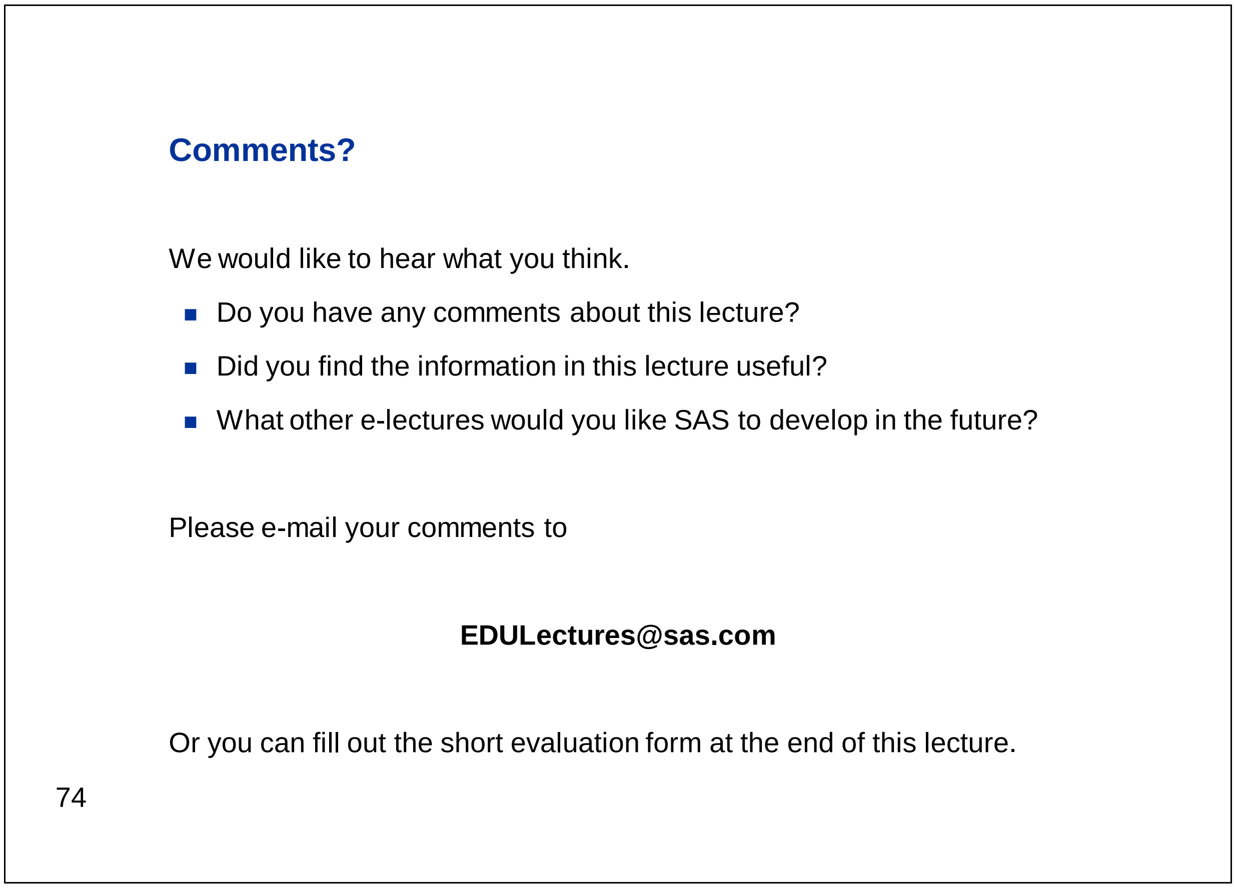
Credits

Executing SAS® Functions with the %SYSFUNC Macro Function was developed by Lise Cragen. Additional contributions were made by Michelle Buchecker, Lynn Mackay, Kathy Passarella, and Russ Tyndall.

73

This concludes the SAS e-Lecture *Executing SAS® Functions with the %SYSFUNC Macro Function.* I hope you will find the material in this lecture to be helpful in your SAS programming efforts.

I'd like to close by thanking everyone who contributed to the creation of this e-lecture.



Comments?

We would like to hear what you think.

- Do you have any comments about this lecture?
- Did you find the information in this lecture useful?
- What other e-lectures would you like SAS to develop in the future?

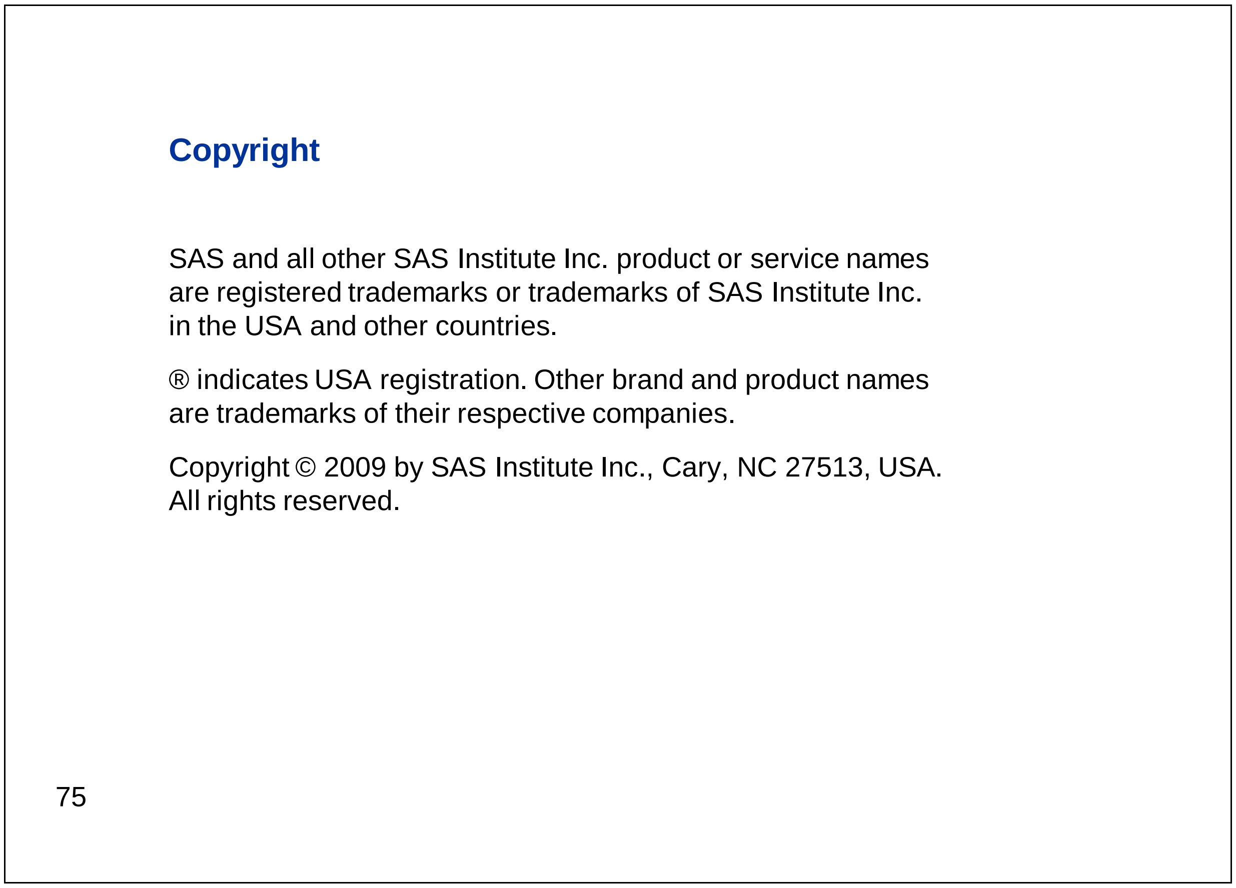
Please e-mail your comments to

EDULectures@sas.com

Or you can fill out the short evaluation form at the end of this lecture.

74

If you have any comments about this lecture or e-lectures in general, we would appreciate receiving your input. You can use the e-mail address listed here to provide that feedback, or you can complete the short evaluation form available at the end of this lecture.



Copyright

75

Thank you for your time.

Appendix A Lecture and Demonstration Programs

1. Creating Macro Variables in a %LET Statement

Section 2, Slide 44

```
%let monthname=%sysfunc(today(),monname.);
%let monthnum=%sysfunc(today(),month.);
options symbolgen;

proc sort data=orion.employee_payroll out=&monthname._Anniv;
   by employee_hire_date;
   where month(employee_hire_date)=&monthnum;
run;

title "Employees with Anniversaries in &monthname";

proc print data=&monthname._Anniv label;
   id employee_id;
   var employee_hire_date;
   format employee_hire_date worddate.;
   label employee_id='Employee ID' employee_hire_date='Date of Hire';
run;

options nosymbolgen;
```

2. Manipulating Macro Variable Values in a SAS Language Statement

Section 3, Slide 52

```
%macro USCityData;

proc sql noprint;
   select count(distinct city) into :numcities
      from orion.employee_addresses
      where country='US';
   select distinct city into :cityname1-:cityname%sysfunc(left(&numcities))
      from orion.employee_addresses
      where country='US';
quit;

data
   %do citycount=1 %to &numcities;
      %sysfunc(compress(&&cityname&citycount,,ps))
   %end;
   ;
   set orion.employee_addresses;
   where country='US';
   select (city);
   %do citycount=1 %to &numcities;
      when ("&&cityname&citycount") output
             %sysfunc(compress(&&cityname&citycount,,ps));
   %end;
   otherwise;
   end;
run;

%mend;

options mprint;
%uscitydata
```

3. Manipulating Macro Variable Values in a SAS Language Statement

Section 3, Slide 56

```
%macro builddata(months);

data Last&months.Months;
set
   %do diff=&months %to 1 %by -1;
      ORION.%sysfunc(intnx(months,%sysfunc(today()),-&diff),monyy.)
   %end;
   ;
run;

%mend;

options mprint;
%builddata(6)
```

4. Manipulating Macro Variable Values in a Macro Program Statement

Section 4, Slide 64

```
%macro builddataw(months);
   %local dslist;

   %do diff=&months %to 1 %by -1;
      %let dsname=ORION.%sysfunc(intnx(months,%sysfunc(today()),
                                  -&diff),monyy.);
      %if %sysfunc(exist(&dsname))=0 %then %do;
         %put WARNING:  The dataset &dsname does not exist and will
                        not be included in the results.;
      %end;
      %else %do;
         %let dslist=&dslist &dsname;
      %end;
   %end;

data Last&months.Months;
   set &dslist;
run;

%mend;

%builddataw(36)
```

5. Manipulating Macro Variable Values in a Macro Program Statement

Section 4, Slide 69

```
%macro SalesReport;

proc summary data=orion.store_order_history nway;
   var sale_date;
   class location;
   output out=First_Order Min=First_Order;
run;

data _null_;
set first_order end=done;
   call symputx('Location'!!left(_n_),Location);
   call symputx('FirstOrder'!!left(_n_),First_order);
   if done then call symputx('NumLocs',_n_);
run;

%do locno=1 %to &numlocs;

proc tabulate data=orion.store_order_history;
   title "This Year's Sales for the &&Location&locno Store";
   title2 "First Sales Reported %sysfunc(putn(&&firstorder&locno,mmddyy10.))";
   where year(sale_date)=year(today()) and
      location="&&location&locno";
   class product_category;
   var quantity total_retail_price sale_date;
   tables product_category='' all='All Products',
         quantity='Total Quantity'*sum=''*f=8.
         total_retail_price='Total Retail Price'*sum=''*f=dollar8.
         /box='Product Category';
run;

%if %sysfunc(intck(quarters,&&firstorder&locno,%sysfunc(today())))>4
   %then %do;

proc gchart data=orion.store_order_history;
   title "Historical Sales for the &&location&locno Store";
   where location="&&location&locno";
   vbar sale_date / sumvar=total_retail_price type=sum discrete;
   format sale_date yyq.;
run;

%end;

%end;

%mend salesreport;

options symbolgen mprint mlogic;
ods listing close;
ods html;

%salesreport

ods html close;
ods listing;
```

Appendix B SAS Functions Not Available with %SYSFUNC

SAS Functions Not Available with %SYSFUNC and %QSYSFUNC		
All Variable Information Functions	ALLCOMB	ALLPERM
DIF	DIM	HBOUND
IORCMSG	INPUT	LAG
LBOUND	LEXCOMB	LEXCOMBI
LEXPERK	LEXPERM	MISSING
PUT	RESOLVE	SYMGET

Note: Instead of INPUT and PUT, which are not available with %SYSFUNC and %QSYSFUNC, use INPUTN, INPUTC, PUTN, and PUTC. ■

Note: The Variable Information functions include functions such as VNAME and VLABEL. For a complete list, see **Definitions of Functions and CALL Routines** in **SAS Language Reference: Dictionary**. ■