A Programming Approach

The Little SAS® Book

a primer

SIXTH EDITION

Lora D. Delwiche and Susan J. Slaughter

The correct bibliographic citation for this manual is as follows: Delwiche, Lora D., and Susan J. Slaughter. 2019. *The Little SAS® Book: A Primer, Sixth Edition*. Cary, NC: SAS Institute Inc.

The Little SAS® Book: A Primer, Sixth Edition

ISBN 978-1-64295-616-0 (Hardcover)
ISBN 978-1-64295-283-4 (Paperback)
ISBN 978-1-64295-342-8 (Web PDF)
ISBN 978-1-64295-343-5 (EPUB)
ISBN 978-1-64295-344-2 (Kindle)

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513-2414

Originally published October 2019. Revised March 2022.

Contents

Introducing SAS Software

SAS software is used by millions of people all over the world—in over 147 countries, at over 83,000 sites. SAS (pronounced sass) is both a company and software. When people say SAS, they sometimes mean the software running on their computers and sometimes mean the company, SAS Institute.

People often ask what SAS stands for. Originally the letters S-A-S stood for Statistical Analysis System (not to be confused with Scandinavian Airlines System, San Antonio Shoemakers, or the Society for Applied Spectroscopy). But SAS products quickly became so diverse that SAS officially dropped the name Statistical Analysis System and became simply SAS.

SAS products The roots of SAS software reach back to the 1970s when it started out as a software package for statistical analysis, but SAS didn't stop there. By the mid-1980s SAS had already branched out into graphics, online data entry, and compilers for the C programming language. In the 1990s, the SAS family tree grew to include tools for visualizing data, administering data warehouses, and building interfaces to the World Wide Web. In the new century, SAS has continued to grow with products designed for cleansing messy data, discovering and developing drugs, detecting money laundering, and building systems for artificial intelligence and machine learning.

While SAS has a diverse family of products, most of these products are integrated. That is, they can be put together like building blocks to construct a seamless system. For example, you might use SAS/ACCESS software to read data stored in an external database such as Oracle, analyze it using SAS/ETS software (econometrics and time series software for modeling and forecasting), use ODS Graphics to produce sophisticated plots, and then forward the results in an email message to your colleagues, all in a single computer program. To find out more about the products that are available from SAS, visit the website:

www.sas.com

Learning SAS In addition to this and other books, there are online resources for learning SAS. SAS Institute has many how-to tutorials and complete courses covering a broad range of topics. Some of these are free, while others are available for a fee. If you don't have access to SAS software at your workplace or school, then there is another way you can practice what you learn. You can set up an account to use SAS OnDemand for Academics which runs on servers hosted by SAS Institute. SAS OnDemand for Academics is available for academic, noncommercial use only.

Operating environments SAS software runs in a wide range of operating environments. You can take a program written on a personal computer and run it on a UNIX server after changing only the file-handling statements that are specific to each operating environment. And because SAS programs are as portable as possible, SAS programmers are as portable as possible, too. If you know SAS in one operating environment, you can switch to another operating environment without having to relearn SAS.

SASware Ballot SAS puts a high percentage of its revenue into research and development, and each year SAS users help determine how that money will be spent by contributing ideas for the SASware Ballot. The ballot is a list of suggestions for new features and enhancements. Anyone can submit an idea and thereby influence the future development of SAS software. To contribute your own ideas or to vote for ones that you like, search the internet for "SASware Ballot."

About This Book

Who needs this book This book is for all new SAS users in business, government, and academia, and for anyone who will be conducting data analysis using SAS software. You need no prior experience with SAS, but if you have some experience you may still find this book useful for learning techniques you missed or for reference.

What this book covers This book introduces you to the SAS language with lots of practical examples, clear and concise explanations, and as little technical jargon as possible. Most of the features covered here come from Base SAS, which contains the core of features used by all SAS programmers. One exception is Chapter 9, which includes procedures from SAS/STAT. Other exceptions appear in Chapters 2 and 10, which cover importing and exporting data from other types of software; some methods require SAS/ACCESS Interface to PC Files.

We have tried to include every feature of Base SAS that a beginner is likely to need. Some readers may be surprised that certain topics, such as macros, are included because they are normally considered advanced. But they appear here because sometimes new users need them. However, that doesn't mean that you need to know everything in this book. On the contrary, this book is designed so that you can read just those sections you need to solve your problems. Even if you read this book from cover to cover, you may still find yourself returning to refresh your memory as new programming challenges arise.

What this book does not cover To use this book you need no prior knowledge of SAS, but you must know something about your local computer and operating environment. The SAS language is virtually the same from one operating environment to another, but some differences are unavoidable. For example, every operating environment has a different way of storing and accessing files, and file names and paths are case sensitive in UNIX but not in Microsoft Windows. Your employer may have rules such as limits for the size of files that you can print. This book addresses operating environments when relevant, but no book can answer every question about your local system. You must have either a working knowledge of your computer system or someone you can turn to with questions.

As a SAS programmer, you have a choice about which interface you use to write your programs, and how you run them. This edition of *The Little SAS Book* is designed to work with all of the interfaces that are included with Base SAS: SAS Studio, SAS Enterprise Guide, and the SAS windowing environment (also known as Display Manager), in addition to batch submission. (SAS OnDemand for Academics uses the SAS Studio interface.) Each of these methods offers its own unique set of features. This book mentions a few of the differences, but is not a comprehensive introduction. See Section 1.5 for a brief description of each method and recommendations about how to learn more.

This book is not a replacement for the SAS Documentation, or the many SAS publications. We encourage you to turn to them for details that are not covered in this book. You can find the complete SAS Documentation at SAS Institute's support website:

support.sas.com

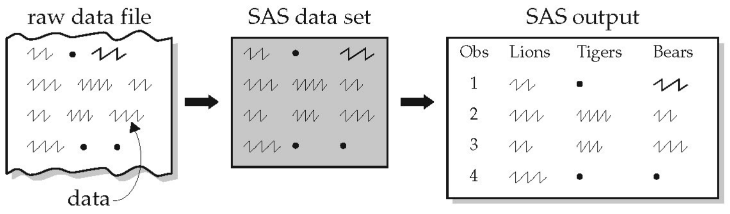
We cover only a few of the many SAS statistical procedures. Fortunately, the statistical procedures share many of the same statements, options, and output, so these few can serve as an introduction to the others. Once you have read Chapter 9, we think that other statistical procedures will feel familiar.

Unfortunately, a book of this type cannot provide a thorough introduction to statistical concepts such as degrees of freedom, or crossed and nested effects. There are underlying assumptions about your data that must be met for the tests to be valid. Experimental design and careful selection of models are critical. Interpretation of the results can often be difficult and subjective. We assume that readers who are interested in statistical computing already know something about statistics. People who want to use statistical procedures but are unfamiliar with these concepts should consult a statistician, seek out an introductory statistics text, or, better yet, take a course in statistics.

Modular sections Our goal in writing this book is to make learning SAS as easy and enjoyable as possible. Let's face it—SAS is a big topic. You may have already spent some time staring at a screen full of documentation until your eyes become blurry. We can't condense all of SAS into this little book, but we can condense topics into short, readable sections.

This entire book consists of two-page sections, each section a complete topic. This way, you can easily skip over topics that do not apply to you. Of course, we think *every* section is important, or we would not have included it. You probably don't need to know everything in this book, however, to complete your job. By presenting topics in short digestible sections, we believe that learning SAS will be easier and more fun—like eating three meals a day instead of one giant meal a week.

Graphics Wherever possible, graphic illustrations either identify the contents of the section or help explain the topic. A box with rough edges indicates a raw data file, and a box with nice smooth edges indicates a SAS data set. The squiggles inside the box indicate data—any old data—and a period indicates a missing value. The arrow between boxes of these types means that the section explains how to get from data that look like one box to data that look like the other. Some sections have graphics that depict printed output. These graphics look like a stack of papers with headers printed at the top of the page.



Typographical conventions For the most part, SAS doesn't care whether your programs are written in uppercase or lowercase, but in this book we have used uppercase and lowercase to tell you something. The statements on the left below show the syntax, or general form, while the statements on the right show an example of actual statements as they might appear in a SAS program.

Syntax	**Example**

`PROC PRINT DATA = `*`data-set-name`*`;`
`   VAR `*`variable-list`*`;`
	`PROC PRINT DATA = bigcats;`
	`   VAR Lions Tigers;`

Notice that the keywords PROC PRINT, DATA, and VAR are the same on both sides and that the descriptive terms *data-set-name* and *variable-list* on the syntax side have been replaced with an actual data set name and variable names in the example.

In this book, all SAS keywords appear in uppercase letters. A keyword is an instruction to SAS and must be spelled correctly. Anything written in lowercase italics is a description of what goes in that spot in the statement, not what you actually type. Things that the programmer has made up such as a variable name, a name for a SAS data set, a comment, or a title, appear in lowercase or mixed case. See Sections 1.2, 2.2, and 2.7 for further discussion of the significance of case in SAS programs.

Indention This book contains many SAS programs, each complete and executable. Programs are formatted in a way which makes them easy for you to read and understand. You do not have to format your programs this way, as SAS is very flexible, but attention to some of these details will make your programs easier to read. Easy-to-read programs are time-savers for you, or the consultant you hire at $200 per hour, when you need to go back and decipher the program months or years later.

The structure of programs is shown by indenting all statements after the first in a step. This is a simple way to make your programs more readable, and it's a good habit to form. SAS doesn't really care where statements start or even if they are all on one line. In the following program, the INFILE and INPUT statements are indented, indicating that they belong with the DATA statement:

```
* Read animals' weights from file. Print the results.;
DATA animals;
   INFILE 'c:\MyRawData\Zoo.dat';
   INPUT Lions Tigers;
RUN;

PROC PRINT DATA = animals;
RUN;
```

Data and programs used in this book You can access the data and programs that are used in the examples by linking to either of the author pages for this book at:

support.sas.com/delwiche

or

support.sas.com/slaughter

From there, you can select **Example Code and Data** to download a file containing the data and programs from this book.

Last, we have tried to make this book as readable as possible and, we hope, even enjoyable. Once you master the contents of this small book you will no longer be a beginning SAS programmer.

About These Authors

With over 25 years of experience, Lora D. Delwiche (right) enjoys teaching people about SAS software and likes solving challenging problems using SAS. She has spent most of her career at the University of California, Davis, using SAS in support of teaching and research.

Susan J. Slaughter (left) discovered SAS software in graduate school over 25 years ago. Since then, she has used SAS in a variety of business and academic settings. She now works as a consultant through her company, Avocet Solutions.

With coauthor Rebecca Ottesen, Lora and Susan have also written *Exercises and Projects for the Little SAS Book, Sixth Edition*, a companion to this book.

Learn more about these authors by visiting their author pages, where you can download free book excerpts, access example code and data, read the latest reviews, get updates, and more:
support.sas.com/delwiche
support.sas.com/slaughter

3

"Contrariwise," continued Tweedledee," if it was so, it might be; and if it were so, it would be; but as it isn't, it ain't. That's logic."

LEWIS CARROLL

From *Alice Through the Looking Glass* by Lewis Carroll. Public domain.

CHAPTER 3

Working with Your Data

3.1 Using the DATA Step to Modify Data

The DATA step is a very powerful tool for manipulating data. Using the DATA step, you can read raw data files, modify existing SAS data sets, and combine SAS data sets. You can also create new variables making use of many available operators and SAS functions, use conditional logic, subset data, and a whole host of other things. There is so much you can do with the DATA step that we could easily write an entire book on the topic. This chapter discusses many of the common operations that you might want to perform on your data. Chapter 6 covers various methods for combining data sets.

DATA steps with INPUT statements We have already seen many examples in Chapter 2 of using the INPUT statement to read raw data files. If you want to do any other data manipulation, such as create new variables, you can do that in the same DATA step as the INPUT statement. Just remember that DATA steps execute line by line, so most other statements need to come after the INPUT statement. The following shows the general form of the DATA step when reading raw data files:

```
DATA new-data-set;
   INFILE raw-data-file;
   INPUT variables;
   Other DATA step statements go here;
```

DATA steps with SET statements When your data are already in a SAS data set, but you want to manipulate the data more, you use a SET statement. The SET statement brings the data into the DATA step one observation at a time, and processes all the observations automatically. To read a SAS data set, start with the DATA statement specifying the name of the new data set. Then follow with the SET statement specifying the name of the old data set you want to read. If you don't want to create a new data set, you can specify the same name in the DATA and SET statements. Then the results of the DATA step will overwrite the old data set named in the SET statement as long as the DATA step does not have any errors. The following shows the general form of the DATA and SET statements:

```
DATA new-data-set;
   SET old-data-set;
   Other DATA step statements go here;
```

Any assignment, subsetting IF, or other DATA step statements usually follow the SET statement. For example, the following creates a new data set, FRIDAY, which is a replica of the SALES data set, except it has an additional variable, Total:

```
DATA friday;
   SET sales;
   Total = Popcorn + Peanuts;
RUN;
```

Example The following raw data give information about hotels in Kyoto, Japan. The hotel name is followed by the nightly rate for two people in yen and the distance from Kyoto Station in kilometers.

```
The Grand West Arashiyama  32200  9.5
Kyoto Sharagam             48000  3.3
The Palace Side Hotel      10200  3.8
Rinn Fushimiinari          41000  2.9
Rinn Nijo Castle           18000  3.3
Suiran Kyoto              102000 11.0
```

This program reads the raw data file KyotoHotels.dat and creates a permanent SAS data set. After the INPUT statement is a simple assignment statement (covered in more detail in the next section) that creates a new variable, USD, which multiplies the value of the variable Yen by the exchange rate of 0.0089.

```
LIBNAME hotels 'c:\MySASLib';
DATA hotels.kyotohotels;
  INFILE 'c:\MyRawData\KyotoHotels.dat';
  INPUT Hotel $ 1-25 Yen Kilometers;
  USD = Yen * 0.0089;
RUN;
```

Here is the SAS data set KYOTOHOTELS created in the program:

	Hotel	Yen	Kilometers	USD
1	The Grand West Arashiyama	32200	9.5	286.58
2	Kyoto Sharagam	48000	3.3	427.20
3	The Palace Side Hotel	10200	3.8	90.78
4	Rinn Fushimiinari	41000	2.9	364.90
5	Rinn Nijo Castle	18000	3.3	160.20
6	Suiran Kyoto	102000	11.0	907.80

Example This program uses the SET statement to read the SAS data set KYOTOHOTELS created in the previous example, and to create a new temporary SAS data set HOTELS. Then, after the SET statement, a simple assignment statement creates a new variable, Miles, which multiplies the value of the variable Kilometers by 0.62.

```
LIBNAME hotels 'C:\MySASLib';
DATA hotels;
  SET hotels.kyotohotels;
  Miles = Kilometers * 0.62;
RUN;
```

Here is the SAS data set HOTELS:

	Hotel	Yen	Kilometers	USD	Miles
1	The Grand West Arashiyama	32200	9.5	286.58	5.890
2	Kyoto Sharagam	48000	3.3	427.20	2.046
3	The Palace Side Hotel	10200	3.8	90.78	2.356
4	Rinn Fushimiinari	41000	2.9	364.90	1.798
5	Rinn Nijo Castle	18000	3.3	160.20	2.046
6	Suiran Kyoto	102000	11.0	907.80	6.820

3.2 Creating and Modifying Variables

If someone were to compile a list of the most popular things to do with SAS software, creating and modifying variables would surely be near the top. Fortunately, SAS is flexible and uses a common-sense approach to these tasks. You create and redefine variables with assignment statements using this basic form:

```
variable = expression;
```

On the left side of the equal sign is a variable name, either new or old. On the right side of the equal sign may appear a constant, another variable, or a mathematical expression. Here are examples of these basic types of assignment statements:

Type of expression	Assignment statement
numeric constant	`Qwerty = 10;`
character constant	`Qwerty = 'ten';`
a variable	`Qwerty = OldVar;`
addition	`Qwerty = OldVar + 10;`
subtraction	`Qwerty = OldVar - 10;`
multiplication	`Qwerty = OldVar * 10;`
division	`Qwerty = OldVar / 10;`
exponentiation	`Qwerty = OldVar ** 10;`

Whether the variable Qwerty is numeric or character depends on the expression that defines it. When the expression is numeric, Qwerty will be numeric. When it is character, Qwerty will be character.

When deciding how to interpret your expression, SAS follows the standard mathematical rules of precedence: SAS performs exponentiation first, then multiplication and division, followed by addition and subtraction. You can use parentheses to override that order. Here are two similar SAS statements showing that a couple of parentheses can make a big difference:

Assignment statement	Result
`x = 10 * 4 + 3 ** 2;`	`x = 49`
`x = 10 * (4 + 3 ** 2);`	`x = 130`

While SAS can read expressions with or without parentheses, people often can't. If you use parentheses, your programs will be a lot easier to read.

Example The following comma-separated values (CSV) data are from a survey of home gardeners. Gardeners were asked to estimate the number of pounds they harvested for four crops: tomatoes, zucchini, peas, and grapes.

```
Name,Tomato,Zucchini,Peas,Grapes
Gregor,10,2,40,0
Molly,15,5,10,1000
Luther,50,10,15,50
Susan,20,0,.,20
```

This program reads the data from a file called Garden.csv using PROC IMPORT, then it modifies the data in a DATA step:

```
* Read csv file with PROC IMPORT;
PROC IMPORT DATAFILE = 'c:\MyRawData\Garden.csv' OUT = homegarden REPLACE;
RUN;
* Modify homegarden data set with assignment statements;
DATA homegarden;
   SET homegarden;
   Zone = 14;
   Type = 'home';
   Zucchini = Zucchini * 10;
   Total = Tomato + Zucchini + Peas + Grapes;
   PerTom = (Tomato / Total) * 100;
RUN;
```

This program contains five assignment statements. The first creates a new variable, Zone, equal to a numeric constant, 14. The variable Type is set equal to a character constant, home. The variable Zucchini is multiplied by 10 because that just seems natural for the prolific zucchini. Total is the sum for all the types of plants. PerTom is not a genetically engineered tomato but the percentage of harvest which were tomatoes. The SAS data set HOMEGARDEN contains all the variables, old and new:

	Name	Tomato	Zucchini	Peas	Grapes	Zone	Type	Total	PerTom
1	Gregor	10	20	40	0	14	home	70	14.2857
2	Molly	15	50	10	1000	14	home	1075	1.3953
3	Luther	50	100	15	50	14	home	215	23.2558
4	Susan	20	0	.	20	14	home	.	.

Notice that the variable Zucchini appears only once because the new value replaced the old value. The other four assignment statements each created a new variable. When a variable is new, SAS adds it to the data set you are creating. When a variable already exists, SAS replaces the original value with the new one. Using an existing name has the advantage of not cluttering your data set with a lot of similar variables. However, you don't want to overwrite a variable unless you are really sure you won't need the original value later.

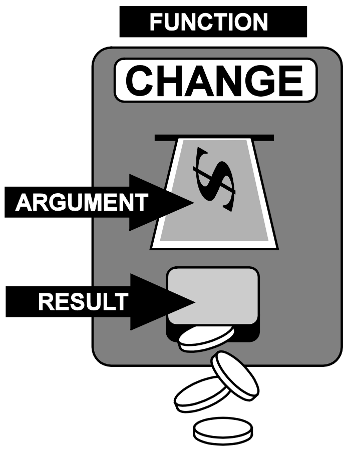
The variable Peas had a missing value for the last observation. Because of this, the variables Total and PerTom, which are calculated from Peas, were also set to missing and this message appeared in the log:

```
NOTE: Missing values were generated as a result of performing an operation
      on missing values.
```

This message is a flag that often indicates an error. However, in this case it is not an error but simply the result of incomplete data collection. If you want to add only nonmissing values, you can use the SUM function discussed in Section 11.7.

3.3 Using SAS Functions

Sometimes a simple expression, using only arithmetic operators, does not give you the new value you are looking for. This is where functions are handy, simplifying your task because SAS has already done the programming for you. All you need to do is plug the right values into the function and out comes the result—like putting a dollar in a change machine and getting back four quarters.



SAS has hundreds of functions in general areas including:

Character	Macro
Character String Matching	Mathematical
Date and Time	Probability
Descriptive Statistics	Random Number
Distance	State and ZIP Code
Financial	Variable Information

The next two sections list the most common SAS functions along with examples.

Functions perform a calculation on, or a transformation of, the arguments given in parentheses following the function name. SAS functions have the following general form:

```
function-name(argument, argument, ...)
```

All functions must have parentheses even if they don't require any arguments. Arguments are separated by commas and can be variable names, constant values such as numbers or characters enclosed in quotation marks, or expressions. The following statement computes DateOfBirth as a SAS date value using the function MDY and the variables MonthBorn, DayBorn, and YearBorn. The MDY function takes three arguments, one each for the month, day, and year:

```
DateOfBirth = MDY(MonthBorn, DayBorn, YearBorn);
```

Functions can be nested, where one function is the argument of another function. For example, the following statement calculates NewValue using two nested functions, INT and LOG:

```
NewValue = INT(LOG(10));
```

The result for this example is 2, the integer portion of the natural log of the numeric constant 10 (2.3026). Just be careful when nesting functions that each parenthesis has a mate.

Example Data from a pumpkin carving contest illustrate the use of several functions. The contestants' names are followed by their age, type of pumpkin (carved or decorated), date of entry, and scores from three judges. Here is the SAS data set CONTEST that was created in Section 2.10:

	Name	Age	Type	Date	Score1	Score2	Score3
1	Alicia Grossman	13	c	22216	7.8	6.5	7.2
2	Matthew Lee	9	D	22218	6.5	5.9	6.8
3	Elizabeth Garcia	10	C	22217	8.9	7.9	8.5
4	Lori Newcombe	6	D	22218	6.7	.	4.9
5	Jose Martinez	7	d	22219	8.9	9.5	10.0
6	Brian Williams	11	C	22217	7.8	8.4	8.5

The following program reads the SAS data set CONTEST, creates two new variables (AvgScore and DayEntered), and transforms another (Type):

```
LIBNAME pump 'c:\MySASLib';
*Use SAS functions to create and modify variables;
DATA pumpkin;
   SET pump.contest;
   AvgScore = MEAN(Score1, Score2, Score3);
   DayEntered = DAY(Date);
   Type = UPCASE(Type);
RUN;
```

The variable AvgScore is created using the MEAN function, which returns the mean of the nonmissing arguments. This differs from simply adding the arguments together and dividing by their number, which would return a missing value if any of the arguments were missing.

The variable DayEntered is created using the DAY function, which returns the day of the month. SAS has all sorts of functions for manipulating dates, and what's great about them is that you don't have to worry about things like leap year—SAS takes care of that for you.

The variable Type is transformed using the UPCASE function. SAS is case sensitive when it comes to variable values; a 'd' is not the same as 'D'. The data file has both lowercase and uppercase letters for the variable Type, so the function UPCASE is used to make all the values uppercase.

Here is the SAS data set PUMPKIN created in the above program:

	Name	Age	Type	Date	Score1	Score2	Score3	AvgScore	DayEntered
1	Alicia Grossman	13	C	22216	7.8	6.5	7.2	7.16667	28
2	Matthew Lee	9	D	22218	6.5	5.9	6.8	6.40000	30
3	Elizabeth Garcia	10	C	22217	8.9	7.9	8.5	8.43333	29
4	Lori Newcombe	6	D	22218	6.7	.	4.9	5.80000	30
5	Jose Martinez	7	D	22219	8.9	9.5	10.0	9.46667	31
6	Brian Williams	11	C	22217	7.8	8.4	8.5	8.23333	29

Notice that the values for the Date variable are shown as the number of days since January 1, 1960. Section 4.6 discusses how to format these values into readable dates.

3.4 Selected SAS Character Functions

Function name	**Syntax**[1]	**Definition**
Character		
ANYALPHA	ANYALPHA(*arg,start*)	Returns position of first occurrence of any alphabetic character at or after optional start position
ANYDIGIT	ANYDIGIT(*arg,start*)	Returns position of first occurrence of any numeral at or after optional start position
ANYSPACE	ANYSPACE(*arg,start*)	Returns position of first occurrence of a whitespace character at or after optional start position
CAT	CAT(*arg-1,arg-2,…arg-n*)	Concatenates two or more character strings together leaving leading and trailing blanks
CATS	CATS(*arg-1,arg-2,…arg-n*)	Concatenates two or more character strings together stripping leading and trailing blanks
CATX	CATX('*separator-string*', *arg-1,arg-2,…arg-n*)	Concatenates two or more character strings together stripping leading and trailing blanks and inserting a separator string between arguments
COMPRESS	COMPRESS(*arg*, '*char*')	Removes spaces or optional characters from character string
INDEX	INDEX(*arg*, '*string*')	Returns starting position for string of characters
LEFT	LEFT(*arg*)	Left aligns a SAS character expression
LENGTH	LENGTH(*arg*)	Returns the length of an argument not counting trailing blanks (missing values have a length of 1)
PROPCASE	PROPCASE(*arg*)	Converts first character in word to uppercase and remaining characters to lowercase
SCAN	SCAN(*arg, n*)	Returns the *n*th word from a character string
SUBSTR	SUBSTR(*arg,position,n*)	Extracts a substring from an argument starting at *position* for *n* characters or until end if no *n*[2]
TRANSLATE	TRANSLATE(*source,to-1, from-1,...to-n,from-n*)	Replaces *from* characters in *source* with *to* characters (one-to-one replacement only—you can't replace one character with two, for example)
TRANWRD	TRANWRD(*source,from,to*)	Replaces *from* character string in *source* with *to* character string
TRIM	TRIM(*arg*)	Removes trailing blanks from character expression
UPCASE	UPCASE(*arg*)	Converts all letters in argument to uppercase

[1] arg is short for argument, which means a literal value, variable name, or expression.

[2] SUBSTR has a different function when on the left side of an equal sign.

Function name	Example	Result	Example	Result
Character				
ANYALPHA	`a='123 E St, #2';` `x=ANYALPHA(a);`	`x=5`	`a='123 E St, #2 ';` `y=ANYALPHA(a,10);`	`y=0`
ANYDIGIT	`a='123 E St, #2';` `x=ANYDIGIT(a);`	`x=1`	`a='123 E St, #2 ';` `y=ANYDIGIT(a,10);`	`y=12`
ANYSPACE	`a='123 E St, #2 ';` `x=ANYSPACE(a);`	`x=4`	`a='123 E St, #2 ';` `y=ANYSPACE(a,10);`	`y=10`
CAT	`a=' cat'; b='dog ';` `x=CAT(a,b);`	`x=' catdog '`	`a='cat '; b=' dog';` `y=CAT(a,b);`	`y='cat  dog'`
CATS	`a=' cat';b='dog ';` `x=CATS(a,b);`	`x='catdog'`	`a='cat ';b=' dog';` `y=CATS(a,b);`	`y='catdog'`
CATX	`a=' cat';b='dog ';` `x=CATX(' ',a,b);`	`x='cat dog'`	`a=' cat ';b='dog ';` `y=CATX('&',a,b);`	`y='cat&dog'`
COMPRESS	`a=' cat & dog';` `x=COMPRESS(a);`	`x='cat&dog'`	`a=' cat & dog';` `y=COMPRESS(a,'&');`	`y=' cat dog'`
INDEX	`a='123 E St, #2';` `x=INDEX(a,'#');`	`x=11`	`a='123 E St, #2';` `y=INDEX(a,'St');`	`y=7`
LEFT	`a=' cat';` `x=LEFT(a);`	`x='cat  '`	`a='  my cat';` `y=LEFT(a);`	`y='my cat  '`
LENGTH	`a='my cat';` `x=LENGTH(a);`	`x=6`	`a=' my cat ';` `y=LENGTH(a);`	`y=7`
PROPCASE	`a='MyCat';` `x=PROPCASE(a);`	`x='Mycat'`	`a='TIGER';` `y=PROPCASE(a);`	`y='Tiger'`
SCAN	`a='my cat';` `x=SCAN(a,1);`	`x='my'`	`a='my cat can';` `y=SCAN(a,-1);`	`y='can'`
SUBSTR[2]	`a='(916)734-6281';` `x=SUBSTR(a,2,3);`	`x='916'`	`y=SUBSTR('1cat',2);`	`y='cat'`
TRANSLATE	`a='6/16/99';` `x=TRANSLATE` `(a,'-','/');`	`x='6-16-99'`	`a='my cat can';` `y=TRANSLATE` `(a, 'r','c');`	`y='my rat ran'`
TRANWRD	`a='Main Street';` `x=TRANWRD` `(a,'Street','St');`	`x='Main St'`	`a='my cat can';` `y=TRANWRD` `(a,'cat','rat');`	`y='my rat can'`
TRIM	`a='my  '; b='cat';` `x=TRIM(a)||b;`[3]	`x='mycat  '`	`a='my cat '; b='s';` `y=TRIM(a)||b;`	`y='my cats '`
UPCASE	`a='MyCat';` `x=UPCASE(a);`	`x='MYCAT'`	`y=UPCASE('Tiger');`	`y='TIGER'`

[3] The concatenation operator || concatenates character strings.

3.5 Selected SAS Numeric Functions

Function name	**Syntax**[4]	**Definition**
Numeric		
INT	INT(*arg*)	Returns the integer portion of argument
LOG	LOG(*arg*)	Natural logarithm
LOG10	LOG10(*arg*)	Logarithm to the base 10
MAX	MAX(*arg-1,arg-2,…arg-n*)	Largest nonmissing value
MEAN	MEAN(*arg-1,arg-2,…arg-n*)	Arithmetic mean of nonmissing values
MIN	MIN(*arg-1,arg-2,…arg-n*)	Smallest nonmissing value
N	N(*arg-1,arg-2,…arg-n*)	Number of nonmissing values
NMISS	NMISS(*arg-1,arg-2,…arg-n*)	Number of missing values
RAND	RAND('UNIFORM')	Generates a random number between 0 and 1 using the uniform distribution[5]
ROUND	ROUND(*arg, round-off-unit*)	Rounds to nearest round-off unit
SUM	SUM(*arg-1,arg-2,…arg-n*)	Sum of nonmissing values
Date[6]		
DATEJUL	DATEJUL(*julian-date*)	Converts a Julian date to a SAS date value
DAY	DAY(*date*)	Returns the day of the month from a SAS date value
MDY	MDY(*month,day,year*)	Returns a SAS date value from month, day, and year values
MONTH	MONTH(*date*)	Returns the month (1–12) from a SAS date value
QTR	QTR(*date*)	Returns the yearly quarter (1–4) from a SAS date value
TODAY	TODAY()	Returns the current date as a SAS date value
WEEKDAY	WEEKDAY(*date*)	Returns day of week (1=Sunday) from SAS date value
YEAR	YEAR(*date*)	Returns year from a SAS date value
YRDIF	YRDIF(*start-date,end-date*,'AGE')	Computes difference in years between two SAS date values taking leap years into account

[4] *arg* is short for argument, which means a literal value, variable name, or expression.

[5] The RAND function can generate random numbers using distributions besides UNIFORM, and each distribution may have optional parameters. To generate the same random numbers each time you run your program, use the CALL STREAMINIT statement.

[6] A SAS date value is the number of days since January 1, 1960.

Function name	Example	Result	Example	Result
Numeric				
INT	`x=INT(4.32);`	`x=4`	`y=INT(5.789);`	`y=5`
LOG	`x=LOG(1);`	`x=0.0`	`y=LOG(10);`	`y=2.30259`
LOG10	`x=LOG10(1);`	`x=0.0`	`y=LOG10(10);`	`y=1.0`
MAX	`x=MAX(9.3,8,7.5);`	`x=9.3`	`y=MAX(-3,.,5);`	`y=5`
MEAN	`x=MEAN(1,4,7,2);`	`x=3.5`	`y=MEAN(2,.,3);`	`y=2.5`
MIN	`x=MIN(9.3,8,7.5);`	`x=7.5`	`y=MIN(-3,.,5);`	`y=-3`
N	`x=N(1,.,7,2);`	`x=3`	`y=N(.,4,.,.);`	`y=1`
NMISS	`x=NMISS(1,.,7,2);`	`x=1`	`y=NMISS(.,4,.,.);`	`y=3`
RAND	`x=RAND('UNIFORM');`	`x=0.48592`	`y=RAND('UNIFORM');`	`y=0.93748`
ROUND	`x=ROUND(12.65);`	`x=13`	`y=ROUND(12.65,.1);`	`y=12.7`
SUM	`x=SUM(3,5,1);`	`x=9.0`	`y=SUM(4,7,.);`	`y=11`
Date				

DATEJUL	`a=60001;`
	`x=DATEJUL(a);`
		`x=0`	`a=60365;`
			`y=DATEJUL(a);`
				`y=364`

DAY	`a=MDY(4,18,2012);`
	`x=DAY(a);`
		`x=18`	`a=MDY(9,3,60);`
			`y=DAY(a);`
				`y=3`

MDY	`x=MDY(1,1,1960);`	`x=0`	`m=2;  d=1;  y=60;`
			`Date=MDY(m,d,y);`
				`Date=31`

MONTH	`a=MDY(4,18,2012);`
	`x=MONTH(a);`
		`x=4`	`a=MDY(9,3,60);`
			`y=MONTH(a);`
				`y=9`

QTR	`a=MDY(4,18,2012);`
	`x=QTR(a);`
		`x=2`	`a=MDY(9,3,60);`
			`y=QTR(a);`
				`y=3`

TODAY	`x=TODAY();`	`x=`*today's date*	`y=TODAY()-1;`	`y=`*yesterday's date*

WEEKDAY	`a=MDY(4,13,2012);`
	`x=WEEKDAY(a);`
		`x=6`	`a=MDY(4,18,2012);`
			`y=WEEKDAY(a);`
				`y=4`

YEAR	`a=MDY(4,13,2012);`
	`x=YEAR(a);`
		`x=2012`	`a=MDY(1,1,1960);`
			`y=YEAR(a);`
				`y=1960`

YRDIF	`a=MDY(4,13,2000);`
	`b=MDY(4,13,2012);`
	`x=YRDIF(a,b,'AGE');`
		`x=12.0`	`a=MDY(4,13,2000);`
			`b=MDY(8,13,2012);`
			`y=YRDIF(a,b,'AGE');`
				`y=12.3342`

3.6 Using IF-THEN and DO Statements

Frequently, you want an assignment statement to apply to some observations, but not all—under some conditions, but not others. This is called conditional logic, and you do it with IF-THEN statements:

```
IF condition THEN action;
```

The *condition* is an expression comparing one thing to another, and the *action* is what SAS should do when the expression is true, often an assignment statement. For example:

```
IF Model = 'Berlinetta' THEN Make = 'Ferrari';
```

This statement tells SAS to set the variable Make equal to Ferrari whenever the variable Model equals Berlinetta. The terms on either side of the comparison may be constants, variables, or expressions. Those terms are separated by a comparison operator, which may be either symbolic or mnemonic. The decision of whether to use symbolic or mnemonic operators depends on your personal preference. Here are the basic comparison operators:

Symbolic	**Mnemonic**	**Meaning**
=	EQ	equals
¬ =, ^ =, or ~ =	NE	not equal
>	GT	greater than
<	LT	less than
> =	GE	greater than or equal
< =	LE	less than or equal
=:		starts with

The IN operator also makes comparisons, but it works a bit differently. IN compares the value of a variable to a list of values. Here is an example:

```
IF Model IN ('Model T', 'Model A') THEN Make = 'Ford';
```

This statement tells SAS to set the variable Make equal to Ford whenever the value of Model is Model T or Model A.

A single IF-THEN statement can only have one action. If you add the keywords DO and END, then you can execute more than one action. For example:

```
IF condition THEN DO;         IF Model = 'DMC-12' THEN DO;
   action;                       Make = 'DeLorean';
   action;                       BodyStyle = 'coupe';
END;                          END;
```

The DO statement causes all SAS statements coming after it to be treated as a unit until a matching END statement appears. Together, the DO statement, the END statement, and all the statements in between are called a DO group.

You can also specify multiple conditions with the keywords AND and OR:

```
IF condition AND condition THEN action;
```

For example:

```
IF Make = 'Alfa Romeo' AND Model = 'Tipo B' THEN Seats = 1;
```

Like the comparison operators, AND and OR may be symbolic or mnemonic:

Symbolic	**Mnemonic**	**Meaning**
&	`AND`	all comparisons must be true
|, ¦, or !	`OR`	at least one comparison must be true

Be careful with long strings of comparisons; they can be a logical maze.

Example The following tab-delimited data show information about rare antique cars sold at auction. The data values are the make, model, the year the car was made, the number of seats , and the selling price in millions of dollars:

```
Make            Model             YearMade  Seats     MillionsPaid
DeDion          LaMarquise        1884      4         4.6
Rolls-Royce     Silver Ghost      1912      4         1.7
Mercedes-Benz   SSK               1929      2         7.4
                F-88              1954      .         3.2
Ferrari         250 Testa Rossa   1957      2         16.3
```

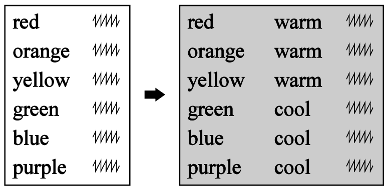
This program uses PROC IMPORT to read the data from a file called Auction.txt, then a DATA step makes some modifications to the data.

```
PROC IMPORT DATAFILE = 'c:\MyRawData\Auction.txt' OUT = oldcars REPLACE;
RUN;
*Use IF-THEN statements to create and modify variables;
DATA oldcars;
   SET oldcars;
   IF YearMade < 1890 THEN Veteran = 'Yes';
   IF Model = 'F-88' THEN DO;
      Make = 'Oldsmobile';
      Seats = 2;
   END;
RUN;
```

This program contains two IF-THEN statements. The first IF-THEN creates a new variable named Veteran and gives it a value of Yes for any car made before 1890. The second IF-THEN uses DO and END to fill in missing data for the model F-88. The resulting SAS data set OLDCARS looks like this:

	Make	Model	YearMade	Seats	MillionsPaid	Veteran
1	DeDion	LaMarquise	1884	4	4.6	Yes
2	Rolls-Royce	Silver Ghost	1912	4	1.7	
3	Mercedes-Benz	SSK	1929	2	7.4	
4	Oldsmobile	F-88	1954	2	3.2	
5	Ferrari	250 Testa Rossa	1957	2	16.3	

3.7 Grouping Observations with IF-THEN/ELSE Statements



One common use of IF-THEN statements is for grouping observations. For example, you might have data for each day but need a report by season, or perhaps you have data for each census tract but want to analyze it by state. There are many possible reasons for grouping data, so sooner or later you'll probably need to do it.

There are several ways to create a grouping variable (including using a PUT function with a user-defined format, covered in Section 4.14), but the simplest and most common method is with a series of IF-THEN statements. By adding the keyword ELSE to your IF statements, you can tell SAS that these statements are related.

IF-THEN/ELSE logic takes this basic form:

```
IF condition THEN action;
   ELSE IF condition THEN action;
   ELSE IF condition THEN action;
```

Notice that the ELSE statement is simply an IF-THEN statement with an ELSE tacked onto the front. You can have any number of these statements.

IF-THEN/ELSE logic has two advantages when compared to a simple series of IF-THEN statements without any ELSE statements. First, it is more efficient, using less computer time; once an observation satisfies a condition, SAS skips the rest of the series. Second, ELSE logic ensures that your groups are mutually exclusive so you don't accidentally have an observation fitting into more than one group.

Sometimes the last ELSE statement in a series is a little different, containing just an action, with no IF or THEN. Note the final ELSE statement in this series:

```
IF condition THEN action;
   ELSE IF condition THEN action;
   ELSE action;
```

An ELSE of this kind becomes a default, which is automatically executed for all observations failing to satisfy any of the previous IF statements. You can have only one of these statements, and it must be the last in the IF-THEN/ELSE series.

When creating character variables using IF-THEN/ELSE statements, you may need to include a LENGTH statement in your program to define the length of the variable you are creating. Without a LENGTH (or ATTRIB) statement, a character variable's length is determined by the first occurrence of the new variable name. For example, if you had the following statements:

```
IF Temperature > 100 THEN Status = 'Hot';
   ELSE Status = 'Cold';
```

The Status variable would have a length of three, because the word Hot is only three characters. This would lead to the value Cold being truncated to Col. Adding the following LENGTH statement before the first occurrence of the Status varable fixes this problem.

```
LENGTH Status $4;
IF Temperature > 100 THEN Status = 'Hot';
   ELSE Status = 'Cold';
```

Example Here are data from a survey of home improvements. Each record contains three data values: owner's name, description of the work done, and cost of the improvements in dollars:

```
Owner   Description                 Cost
Bob     kitchen cabinet face-lift   1253.00
Shirley bathroom addition           11350.70
Silvia  paint exterior              .
Al      backyard gazebo             3098.63
Norm    paint interior              647.77
Kathy   second floor addition       75362.93
```

This program reads the tab-delimited file called Home.txt using PROC IMPORT then assigns a grouping variable called CostGroup in a DATA step. This variable has a value of high, medium, low, or TBD, depending on the value of Cost:

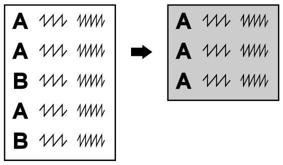
```
PROC IMPORT DATAFILE = 'c:\MyRawData\Home.txt' OUT = homeimp REPLACE;
RUN;
DATA homeimprovements;
   SET homeimp;
   *Group observations by cost;
   LENGTH CostGroup $6;
   IF Cost = . THEN CostGroup = 'TBD';
      ELSE IF Cost < 2000 THEN CostGroup = 'low';
      ELSE IF Cost < 10000 THEN CostGroup = 'medium';
      ELSE CostGroup = 'high';
RUN;
```

Notice that there are four statements in this IF-THEN/ELSE series, one for each possible value of the variable CostGroup. The first statement deals with observations that have missing data for the variable Cost. Without this first statement, observations with a missing value for Cost would be incorrectly assigned a CostGroup of low. SAS considers missing values to be smaller than nonmissing values, smaller than any printable character for character variables, and smaller than negative numbers for numeric variables. Unless you are sure that your data contain no missing values, you should allow for missing values when you write IF-THEN/ELSE statements. The LENGTH statement before the IF-THEN/ELSE statements sets the length of the new variable CostGroup to six, ensuring that no values will be truncated.

The data set HOMEIMPROVEMENTS looks like this:

	Owner	Description	Cost	CostGroup
1	Bob	kitchen cabinet face-lift	1253	low
2	Shirley	bathroom addition	11350.7	high
3	Silvia	paint exterior	.	TBD
4	Al	backyard gazebo	3098.63	medium
5	Norm	paint interior	647.77	low
6	Kathy	second floor addition	75362.93	high

3.8 Subsetting Your Data in a DATA Step



Often programmers find that they want to use some of the observations in a data set and exclude the rest. A common way to do this is with a subsetting IF statement in a DATA step. Here is the basic form of a subsetting IF:

```
IF expression;
```

Consider this example:

```
IF Sex = 'f';
```

At first subsetting IF statements may seem odd. People naturally ask, "IF Sex = 'f', then what?" The subsetting IF looks incomplete, as if a careless typist pressed the delete key too long. But it is really a special case of the standard IF-THEN statement. In this case, the action is merely implied. If the expression is true, then SAS continues with the DATA step. If the expression is false, then no further statements are processed for that observation; that observation is not added to the data set being created; and SAS moves on to the next observation. You can think of the subsetting IF as a kind of on-off switch. If the condition is true, then the switch is on and the observation is processed. If the condition is false, then that observation is turned off.

DELETE statements do the opposite of subsetting IFs. While the subsetting IF statement tells SAS which observations to include, the DELETE statement tells SAS which observations to exclude:

```
IF expression THEN DELETE;
```

The following two statements are equivalent (assuming there are only two values for the variable Sex, and no missing data):

```
IF Sex = 'f';        IF Sex = 'm' THEN DELETE;
```

You can also subset data using the WHERE statement in the DATA step. The WHERE statement is similar to the IF statement and can be more efficient. But you can only use the WHERE statement when selecting observations from existing SAS data sets and only for variables that already exist in the data set. (There is also a WHERE= data set option which is covered in Section 6.12.) The WHERE statement has the following general form:

```
WHERE expression;
```

Example A local zoo maintains a database about the feeding of the animals. A portion of the data appears below. For each group of animals the data include the scientific class, the enclosure those animals live in, and whether they get fed in the morning, afternoon, or both:

```
Animal,Class,Enclosure,FeedTime
bears,Mammalia,E2,both
elephants,Mammalia,W3,am
flamingos,Aves,W1,pm
frogs,Amphibia,E8,pm
kangaroos,Mammalia,W4,am
lions,Mammalia,W6,pm
snakes,Reptilia,E9,pm
tigers,Mammalia,W9,both
zebras,Mammalia,W2,am
```

This program reads the data from a comma-delimited data file called Zoo.csv, creating a permanent SAS data set, ZOO. Then it uses a subsetting IF statement in a separate DATA step to select only observations where the animal class is Mammalia:

```
LIBNAME feed'c:\MySASLib';
PROC IMPORT DATAFILE = 'c:\MyRawData\Zoo.csv' OUT = feed.zoo REPLACE;
RUN;
*Choose only mammals;
DATA mammals;
   SET feed.zoo;
   IF Class = 'Mammalia';
   IF Enclosure =: 'E' THEN Area = 'East';
     ELSE IF Enclosure =: 'W' THEN Area = 'West';
RUN;
```

After the subsetting IF statement is a series of IF-THEN/ELSE statements that create a new variable Area based on the starting character in the variable Enclosure. Note that the Area variable appears in the resulting data set even though the statements creating it come after the subsetting IF statement. Observations are written to the data set at the end of the DATA step unless there is an OUTPUT statement (discussed in Sections 3.10 and 3.11) in the DATA step. The MAMMALS data set looks like this:

	Animal	Class	Enclosure	FeedTime	Area
1	bears	Mammalia	E2	both	East
2	elephants	Mammalia	W3	am	West
3	kangaroos	Mammalia	W4	am	West
4	lions	Mammalia	W6	pm	West
5	tigers	Mammalia	W9	both	West
6	zebras	Mammalia	W2	am	West

These notes appear in the log stating that although nine observations were read from the original data set, the resulting data set MAMMALS contains only six observations:

```
NOTE: There were 9 observations read from the data set FEED.ZOO.
NOTE: The data set WORK.MAMMALS has 6 observations and 5 variables.
```

It is always a good idea to check the SAS log when you subset observations to make sure that you ended up with what you expected.

In the program above, you could substitute this statement:

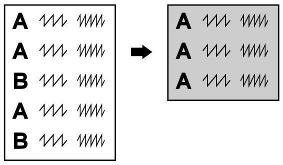
```
IF Class = 'Aves' OR Class = 'Amphibia' OR Class = 'Reptilia' THEN DELETE;
```

for the statement:

```
IF Class = 'Mammalia';
```

But you would have to do a lot more typing. Generally, you use the subsetting IF when it is easier to specify a condition for including observations, and use the DELETE statement when it is easier to specify a condition for excluding observations.

3.9 Subsetting Your Data Using PROC SQL



If you are familiar with Structured Query Language (SQL), you will be pleased to know that there is an SQL procedure in SAS. A common task in SQL is querying data and producing subsets of data. In SQL documentation, data sets are called tables, observations are called rows, and variables are called columns, but they are the same thing. The basic form of PROC SQL for subsetting data is:

```
PROC SQL;
   CREATE TABLE new-data-set AS
     SELECT variable-list
     FROM old-data-set
     WHERE expression;
QUIT;
```

The procedure starts with the keywords PROC SQL followed by a semicolon. Next comes the SQL statement containing four clauses. The CREATE TABLE … AS clause denotes the name of the new SAS data set that you will create. The SELECT clause lists the variables (or columns) you want to keep. The FROM clause indicates the name of the old SAS data set you are reading. Then the WHERE clause states which observations (or rows) you want to keep. Note that between the PROC SQL and QUIT statements, there is only one statement ending in a semicolon. But this one statement contains many clauses (CREATE TABLE, SELECT, FROM, and WHERE). If you leave out the CREATE TABLE clause, then instead of creating a new SAS data set, you will just get a display of the results. PROC SQL ends with a QUIT statement instead of a RUN statement. Unlike most procedures, with PROC SQL, SAS immediately executes whatever you submit without waiting for a RUN statement. You can continue to submit more SQL statements because PROC SQL keeps running until it encounters a QUIT statement or a new DATA or PROC step.

The SELECT clause In the SELECT clause, separate the names of the variables you want to keep with commas or, if you want to keep all the variables, simply use an asterisk (*):

```
SELECT Lion, Weight, Sex          or          SELECT *
```

You can also calculate new variables in the SELECT clause using an expression and the keyword AS to give the variable a name:

```
SELECT Lion, Weight * 0.454 AS Kilos, Sex
```

The WHERE clause Use the WHERE clause in PROC SQL to select which rows, or observations, you want to appear in the new data set. The syntax for the WHERE clause is similar to the subsetting IF statement discussed in the previous section. You can use any of the comparison operators discussed in Section 3.6. For example, the following WHERE clauses would both select observations for lions, tigers and bears:

```
WHERE animal = 'lions' OR animal = 'tigers' OR animal = 'bears'
WHERE animal IN ('lions', 'tigers', 'bears')
```

If you want to include a calculated variable in the WHERE clause, then precede the variable name with the keyword CALCULATED:

```
WHERE CALCULATED Kilos > 200
```

Example The SAS data set ZOO, which was created in the previous section, contains data on the feeding of animals at a local zoo. For each group of animals the data include the scientific class, the enclosure those animals live in, and whether they get fed in the morning, afternoon, or both:

	Animal	Class	Enclosure	FeedTime
1	bears	Mammalia	E2	both
2	elephants	Mammalia	W3	am
3	flamingos	Aves	W1	pm
4	frogs	Amphibia	E8	pm
5	kangaroos	Mammalia	W4	am
6	lions	Mammalia	W6	pm
7	snakes	Reptilia	E9	pm
8	tigers	Mammalia	W9	both
9	zebras	Mammalia	W2	am

The following program uses the SQL procedure to create a new temporary SAS data set, MAMMALS, which contains all the variables (as indicated by the asterisk after the SELECT keyword) in the ZOO data set, but only observations where the value of the variable Class is Mammalia.

```
LIBNAME feed'c:\MySASLib';
*Choose only mammals;
PROC SQL;
   CREATE TABLE mammals AS
     SELECT *
     FROM feed.zoo
     WHERE Class = 'Mammalia';
QUIT;
```

Here is the new data set MAMMALS:

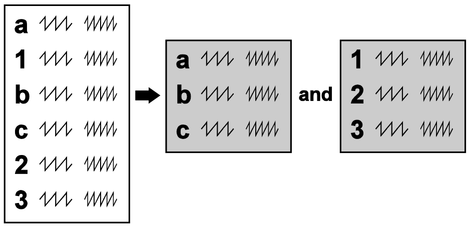
	Animal	Class	Enclosure	FeedTime
1	bears	Mammalia	E2	both
2	elephants	Mammalia	W3	am
3	kangaroos	Mammalia	W4	am
4	lions	Mammalia	W6	pm
5	tigers	Mammalia	W9	both
6	zebras	Mammalia	W2	am

The following note appears in the log.

```
NOTE: Table WORK.MAMMALS created, with 6 rows and 4 columns.
```

When you use the SQL procedure to subset data, the SAS log tells you how many rows and columns the new data set contains. But, unlike using a subsetting IF statement in a DATA step, it does not tell you how many variables and observations were in the old data set.

3.10 Writing Multiple Data Sets Using OUTPUT Statements



Up to this point, all the DATA steps in this book have created a single data set. Most of the time this is what you want. However, there may be times when it is more efficient or more convenient to create multiple data sets in a single DATA step. You can do this by simply putting more than one data set name in your DATA statement. The statement below tells SAS to create three data sets named LIONS, TIGERS, and BEARS:

```
DATA lions tigers bears;
```

If that is all you do, then SAS will write all the observations to all the data sets, and you will have three identical data sets. Normally, of course, you want to create different data sets. You can do that with an OUTPUT statement.

Every DATA step has an implied OUTPUT statement at the end, which tells SAS to write the current observation to the output data set before returning to the beginning of the DATA step to process the next observation. You can override this implicit OUTPUT statement with your own OUTPUT statement. However, once you put an OUTPUT statement in your DATA step, it is no longer implied, and SAS writes an observation only when it encounters an OUTPUT statement. Here is the basic form of the OUTPUT statement:

```
OUTPUT data-set-name;
```

If you leave out the data set name, then the observation will be written to all data sets named in the DATA statement. OUTPUT statements can be used alone or in IF-THEN statements.

```
IF gender = 'F' THEN OUTPUT females;
```

Example The SAS data set ZOO, which was created in Section 3.8, contains data on the feeding of animals at a local zoo. For each group of animals the data include the scientific class, the enclosure those animals live in, and whether they get fed in the morning, afternoon, or both:

	Animal	Class	Enclosure	FeedTime
1	bears	Mammalia	E2	both
2	elephants	Mammalia	W3	am
3	flamingos	Aves	W1	pm
4	frogs	Amphibia	E8	pm
5	kangaroos	Mammalia	W4	am
6	lions	Mammalia	W6	pm
7	snakes	Reptilia	E9	pm
8	tigers	Mammalia	W9	both
9	zebras	Mammalia	W2	am

To help with feeding the animals, the following program creates two data sets, one for morning feedings and one for afternoon feedings:

```
LIBNAME feed'c:\MySASLib';
*Create data sets for morning and afternoon feedings;
DATA morning afternoon;
   SET feed.zoo;
   IF FeedTime = 'am' THEN OUTPUT morning;
      ELSE IF FeedTime = 'pm' THEN OUTPUT afternoon;
      ELSE IF FeedTime = 'both' THEN OUTPUT;
RUN;
```

This DATA statement creates a data set named MORNING and a data set named AFTERNOON. Then the IF-THEN/ELSE statements tell SAS which observations to put in each data set. Because the final OUTPUT statement does not specify a data set, SAS adds those observations to both data sets. The log contains these notes saying that SAS read one input file and wrote two data sets:

```
NOTE: There were 9 observations read from the data set FEED.ZOO.
NOTE: The data set WORK.MORNING has 5 observations and 4 variables.
NOTE: The data set WORK.AFTERNOON has 6 observations and 4 variables.
```

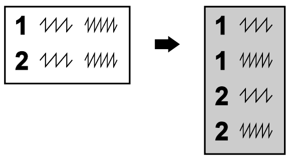
Here are the MORNING and AFTERNOON data sets that are created:

	Animal	Class	Enclosure	FeedTime
1	bears	Mammalia	E2	both
2	elephants	Mammalia	W3	am
3	kangaroos	Mammalia	W4	am
4	tigers	Mammalia	W9	both
5	zebras	Mammalia	W2	am

	Animal	Class	Enclosure	FeedTime
1	bears	Mammalia	E2	both
2	flamingos	Aves	W1	pm
3	frogs	Amphibia	E8	pm
4	lions	Mammalia	W6	pm
5	snakes	Reptilia	E9	pm
6	tigers	Mammalia	W9	both

OUTPUT statements have other uses besides writing multiple data sets in a single DATA step and can be used any time you want to explicitly control when SAS writes observations to a data set (see the next two sections).

3.11 Making Several Observations from One Using OUTPUT Statements



Usually, SAS writes an observation to a data set at the end of the DATA step, but you can override this default using the OUTPUT statement. If you want to write several observations for each pass through the DATA step, you can put an OUTPUT statement in a DO loop (see the next section) or just use several OUTPUT statements. The OUTPUT statement gives you control over when an observation is written to a SAS data set. If your DATA step doesn't have an OUTPUT statement, then it is implied at the end of the step. Once you have an OUTPUT statement, it is no longer implied, and SAS writes an observation only when it encounters an OUTPUT statement.

Example A minor league baseball team has a comma-delimited file containing information about upcoming games. The file contains the opposing team name, game date, whether it is a home (H) or away (A) game, and if it is a single (S) or doubleheader(D).

```
Team,GameDate,Location,Type
Columbia Peaches,04/15/2020,A,S
Walla Walla Sweets,04/17/2020,H,D
Gilroy Garlics,4/18/2020,H,S
Sacramento Tomatoes,4/21/2020,A,S
```

The team manager needs a file that contains one observation for each game, so the observations for doubleheaders need to be repeated. This program first uses PROC IMPORT to read the data file. In the DATA step, if the entry is for a doubleheader, the observation is output two times. If not, the observation is output only once.

```
PROC IMPORT DATAFILE = 'c:\MyRawData\Schedule.csv' OUT = GameDates
            REPLACE;
RUN;
DATA Games;
  SET GameDates;
  *If a doubleheader, output twice;
  IF Type = 'D' THEN DO;
    OUTPUT;
    OUTPUT;
  END;
  *Else if not a doubleheader output only once;
  ELSE OUTPUT;
RUN;
```

Here is the GAMES data set created from the above program:

	Team	GameDate	Location	Type
1	Columbia Peaches	04/15/2020	A	S
2	Walla Walla Sweets	04/17/2020	H	D
3	Walla Walla Sweets	04/17/2020	H	D
4	Gilroy Garlics	04/18/2020	H	S
5	Sacramento Tomatoes	04/21/2020	A	S

Example Here's how you can use OUTPUT statements to create several observations from a single pass through the DATA step. The following data are for ticket sales at three movie theaters. After the month are the theaters' names and sales for all three theaters:

```
Jan Varsity 56723 Downtown 69831 Super-6 70025
Feb Varsity 62137 Downtown 43901 Super-6 81534
Mar Varsity 49982 Downtown 55783 Super-6 69800
```

For the analysis you want to perform, you need to have the theater name as one variable and the ticket sales as another variable. The month should be repeated once for each theater.

The following program has three INPUT statements all reading from the same raw data file. The first INPUT statement reads values for Month, Location, and Tickets, and then holds the data line using the trailing at sign (@). The OUTPUT statement that follows writes an observation. The next INPUT statement reads the second set of data for Location and Tickets, and again holds the data line. Another OUTPUT statement writes another observation. Month still has the same value because it isn't in the second INPUT statement. The last INPUT statement reads the last values for Location and Tickets, this time releasing the data line for the next iteration through the DATA step. The final OUTPUT statement writes the third observation for that iteration of the DATA step. The program has three OUTPUT statements for the three observations created in each iteration of the DATA step:

```
* Create three observations for each data line read
*   using three OUTPUT statements;
DATA theaters;
   INFILE 'c:\MyRawData\Movies.dat';
   INPUT Month $ Location $ Tickets @;
   OUTPUT;
   INPUT Location $ Tickets @;
   OUTPUT;
   INPUT Location $ Tickets;
   OUTPUT;
RUN;
```

Here is the THEATERS data set created in the above program. Notice that there are three observations in the data set for each line in the raw data file, and that the value for Month is repeated:

	Month	Location	Tickets
1	Jan	Varsity	56723
2	Jan	Downtown	69831
3	Jan	Super-6	70025
4	Feb	Varsity	62137
5	Feb	Downtown	43901
6	Feb	Super-6	81534
7	Mar	Varsity	49982
8	Mar	Downtown	55783
9	Mar	Super-6	69800

3.12 Using Iterative DO, DO WHILE, and DO UNTIL Statements

We introduced the DO statement in Section 3.6 where you can conditionally execute a group of statements called a DO group. In this section we will talk about using DO statements that can execute the same DO group more than once. The iterative DO statement executes the DO group a set number of times. The number of times the DO WHILE and DO UNTIL statements execute the DO group depend on the value of a specified expression.

Iterative DO All DO groups start with the DO statement and end with an END statement. Between the DO and END, you can have any number of other SAS statements including even other DO groups. The general form of a DO group with an iterative DO statement is:

```
DO index-variable = specification for iteration;
   statement(s);
END;
```

The iterative DO statement can take on many different forms. All forms start with the keyword DO, followed by an index variable, then a specification for how the index variable should be incremented. The index variable is often a new variable that you create for the purpose of looping through the DO group. This variable is added to the data set unless you drop it. In one form, you can simply list the values you want the index variable to take as it executes the DO group.

```
DO index-variable = item-1, item-2, … ;
```

The items in the list can be all numeric constants, all character constants (enclosed in quotes), or even variable names. For example, the following DO statement would iterate four times, for the four values of Year in the list:

```
DO Year = 2003, 2006, 2012, 2017;
```

You can also increment numeric values automatically by specifying start and stop values. Use the BY option to specify the amount of the increment. If you do not use the BY option, then the index variable will increment by one. The DO group is executed until the value of the index variable passes the stop value. Here is the general form:

```
DO index-variable = start-value TO stop-value BY increment;
```

In the following statement, the index variable, X, will loop through the DO group with values 10, 11, 12, 13, 14, 15, and 16. When X reaches 17, it has passed the stop value and the DO group stops executing.

```
DO X = 10 TO 16;
```

In the next statement, X will loop through the DO group with the values 10, 12, 14, and 16. When X reaches 18, it has passed the stop value and the DO group stops executing.

```
DO X = 10 TO 16 BY 2;
```

DO WHILE and DO UNTIL The DO WHILE and DO UNTIL statements work a little differently than the iterative DO. The DO WHILE statement will continue looping as long as the expression is still true. If the expression is never true, then the DO group will never execute.

```
DO WHILE (expression);
```

The DO UNTIL statement will continue looping until the expression becomes true, so it will always execute at least once:

```
DO UNTIL (expression);
```

The following statements will cause a DO group to execute when Age is less than 65:

```
DO UNTIL (Age GE 65);              DO WHILE (Age < 65);
```

If the value of Age starts out greater than or equal to 65, then the DO UNTIL statement will execute the DO group one time, whereas the DO WHILE statement will never execute the DO group. If the value of Age never gets to be 65 or higher, then SAS will just keep going, and going, and going.

Example Suppose you have $100 to invest and you want to know how many years it would take for your savings to pass $1,000. You choose five different interest rates to test and assume that all the interest earned is reinvested. The following program has a DO UNTIL loop nested inside an iterative DO group. The iterative DO group is executed five times with the values of InterestRate set to 0.02, 0.03, 0.04, 0.05, and 0.06. Each iteration of the DO group starts with statements that set the value of Savings to the initial amount of 100, and the value of Years to zero since we want to start counting over for each interest rate.

```
DATA numyears;
   DO InterestRate = 0.02 TO 0.06 BY 0.01;
     *Initialize Savings and Year for each interest rate;
     Savings = 100;
     Years = 0;

     *Find number of years until savings greater than $1000;
     DO UNTIL (Savings > 1000);
        Years = Years + 1;
        Savings = Savings + (InterestRate * Savings);
     END;

     *Write results to years data set;
     OUTPUT;
   END;
RUN;
```

The DO UNTIL loop executes until the value of Savings is greater than $1,000. Each time through the loop, one is added to the variable Years, and the interest earned (InterestRate times Savings) is added to the current value of Savings. After each iteration of the DO UNTIL loop, the value of Savings is evaluated and when it is greater than 1,000 SAS exits the loop and goes on to the next DATA step statement. The OUTPUT statement after the DO UNTIL loop, but inside the iterative DO group, outputs current values of the variables to the NUMYEARS data set. Here is the data set:

	InterestRate	Savings	Years
1	0.02	1014.43	117
2	0.03	1003.01	78
3	0.04	1011.50	59
4	0.05	1040.13	48
5	0.06	1028.57	40

3.13 Working with SAS Dates

Dates can be tricky to work with. Some months have 30 days, some 31, some 28, and don't forget leap year. SAS dates simplify all this. A SAS date is a numeric value equal to the number of days since January 1, 1960. The table below lists four dates and their values as SAS dates:

Date	**SAS date value**	**Date**	**SAS date value**
January 1, 1959	-365	January 1, 1961	366
January 1, 1960	0	January 1, 2020	21915

SAS has special tools for working with dates (as well as time and datetime values): informats for reading dates, functions for manipulating dates, and formats for printing dates. A table of selected date informats, formats, and functions appears in the next section.

Informats To read variables that are dates in raw data files, you use formatted style input. SAS has a variety of date informats for reading dates in many different forms. All of these informats convert your data to a number equal to the number of days since January 1, 1960. The INPUT statement below tells SAS to read a variable named BirthDate using the ANYDTDTE9. informat:

```
INPUT BirthDate ANYDTDTE9.;
```

ANYDTDTE*w*. is a special informat that can read dates in almost any form. If a date is ambiguous such as 01-02-03, then SAS uses the value of the DATESTYLE= system option to determine the order of month, day and year. The default value of DATESTYLE= is MDY (month, day, then year).

Setting the default century for input When SAS sees a date with a two-digit year like 07/04/76, SAS has to decide in which century the year belongs. Is the year 1976, 2076, or perhaps 1776? The system option YEARCUTOFF= specifies the first year of a hundred-year span for SAS to use. At the time this book was written, the default value for this option was 1926. You can change this value with an OPTIONS statement. To avoid problems, you may want to specify the YEARCUTOFF= option whenever you input data containing two-digit years. This statement tells SAS to interpret two-digit dates as occurring between 1950 and 2049:

```
OPTIONS YEARCUTOFF = 1950;
```

Dates in SAS expressions Once a variable has been read with a SAS date informat, it can be used in arithmetic expressions like other numeric variables. For example, if a library book is due in three weeks, you could find the due date by adding 21 days to the date it was checked out:

```
DueDate = CheckDate + 21;
```

You can use a date as a constant in a SAS expression. Put the date in DATE*w*. format (such as 01JAN1960). Then add quotation marks followed by the letter D. The assignment statement below creates a variable named EarthDay21, which is equal to the SAS date value for April 22, 2021:

```
EarthDay21 = '22APR2021'D;
```

Functions SAS date functions perform a number of handy operations. The statement below uses three functions to compute age from a variable named BirthDate.

```
CurrentAge = INT(YRDIF(BirthDate, TODAY(), 'AGE'));
```

The YRDIF function, with the 'AGE' argument, computes the number of years between the variable BirthDate and the current date (from the TODAY function). Then the INT function returns the integer portion of the value.

Formats If you print a SAS date value, SAS will by default print the actual value—the number of days since January 1, 1960. Since this is not very meaningful to most people, SAS has a variety of formats for printing dates in different forms. The FORMAT statement below tells SAS to print the variable BirthDate using the WORDDATE18. format:

```
FORMAT BirthDate WORDDATE18.;
```

FORMAT statements can go in either DATA steps or PROC steps. If the FORMAT statement is in a DATA step, then the format association is permanent and is stored with the SAS data set. If the FORMAT statement is in a PROC step, then it is temporary—affecting only the results from that procedure. Formats are covered in more detail in Section 4.6.

Example A local library has a data file containing details about library cards. Each record contains the card holder’s name, birthdate, the date that the card was issued, and the due date for the last book borrowed.

```
A. Jones    1-1-60     9-15-96    18JUN20
R. Grandage 03/18/1988 31 10 2007 5jul2020
K. Kaminaka 052903     20200124   12-MAR-20
```

The program below reads the raw data, and then computes the variable DaysOverDue by subtracting DueDate from the current date. The card holder's current age is computed. Then an IF statement uses a date constant to identify cards issued after January 1, 2020.

```
DATA librarycards;
   INFILE 'c:\MyRawData\Library.dat' TRUNCOVER;
   INPUT Name $11. + 1 BirthDate MMDDYY10. +1 IssueDate ANYDTDTE10.
      DueDate DATE11.;
   DaysOverDue = TODAY() - DueDate;
   CurrentAge = INT(YRDIF(BirthDate, TODAY(), 'AGE'));
   IF IssueDate > '01JAN2020'D THEN NewCard = 'yes';
RUN;
PROC PRINT DATA = librarycards;
   FORMAT Issuedate MMDDYY8. DueDate WEEKDATE17.;
   TITLE 'SAS Dates without and with Formats';
RUN;
```

Here is the output from PROC PRINT. Notice that the variable BirthDate is printed without a date format, while IssueDate and DueDate use formats. Because DaysOverDue and CurrentAge are computed using the TODAY() function, their values will change depending on the day the program is run. The value of DaysOverDue is negative for books due in the future.

SAS Dates without and with Formats

Obs	Name	BirthDate	IssueDate	DueDate	DaysOverDue	CurrentAge	NewCard
1	A. Jones	0	09/15/96	Mon, Jun 18, 2020	0	60	
2	R. Grandage	10304	10/31/07	Thu, Jul 5, 2020	-17	32	
3	K. Kaminaka	15854	01/24/20	Mon, Mar 12, 2020	98	17	yes

3.14 Selected Date Informats, Functions, and Formats

Informats	Definition	Width range	Default width
ANYDTDTE*w*.	Reads dates in various date forms	5–32	9
DATE*w*.	Reads dates in form: *ddmmmyy* or *ddmmmyyyy*	7–32	7
DDMMYY*w*.	Reads dates in form: *ddmmyy* or *ddmmyyyy*	6–32	6
JULIAN*w*.	Reads Julian dates in form: *yyddd* or *yyyyddd*	5–32	5
MMDDYY*w*.	Reads dates in form: *mmddyy* or *mmddyyyy*	6–32	6

Functions	Syntax	Definition
DATEJUL	DATEJUL(*julian-date*)	Converts a Julian date to a SAS date value[7]
DAY	DAY(*date*)	Returns the day of the month from a SAS date value
MDY	MDY(*month,day,year*)	Returns a SAS date value from month, day, and year values
MONTH	MONTH(*date*)	Returns the month (1–12) from a SAS date value
QTR	QTR(*date*)	Returns the yearly quarter (1–4) from a SAS date value
TODAY	TODAY()	Returns the current date as a SAS date value
WEEKDAY	WEEKDAY(*date*)	Returns day of week (1=Sunday) from SAS date value
YEAR	YEAR(*date*)	Returns year from a SAS date value
YRDIF	YRDIF(*start-date,end-date,* 'AGE')	Computes difference in years between two SAS date values taking leap years into account

Formats	Definition	Width range	Default width
DATE*w*.	Writes SAS date values in form: *ddmmmyy*	5–11	7
EURDFDD*w*.	Writes SAS date values in form: *dd.mm.yy*	2–10	8
JULIAN*w*.	Writes a Julian date from a SAS date value	5–7	5
MMDDYY*w*.	Writes SAS date values in form: *mmddyy* or *mmddyyyy*	2–10	8
WEEKDATE*w*.	Writes SAS date values in form: *day-of-week, month-name dd, yy* or *yyyy*	3–37	29
WORDDATE*w*.	Writes SAS date values in form: *month-name dd, yyyy*	3–32	18

[7] A SAS date value is the number of days since January 1, 1960.

Informats	Input data	INPUT statement	Results

ANYDTDTE*w*.	`1jan1961`
	`01/01/61`
		`INPUT Day ANYDTDTE10.;`	`366`
			`366`

DATE*w*.	`1jan1961`	`INPUT Day DATE10.;`	`366`

DDMMYY*w*.	`01.01.61`
	`02/01/61`
		`INPUT Day DDMMYY8.;`	`366`
			`367`

JULIAN*w*.	`61001`	`INPUT Day JULIAN7.;`	`366`
MMDDYY*w*.	`01-01-61`	`INPUT Day MMDDYY8.;`	`366`

Functions	Example	Result	Example	Results

DATEJUL	`a=60001;`
	`x=DATEJUL(a);`
		`x=0`	`a=60365;`
			`y=DATEJUL(a);`
				`y=364`

DAY	`a=MDY(4,18,2020);`
	`x=DAY(a);`
		`x=18`	`a=MDY(9,3,60);`
			`y=DAY(a);`
				`y=3`

MDY	`x=MDY(1,1,1960);`	`x=0`	`m=2; d=1; y=60;`
			`Date=MDY(m,d,y);`
				`Date=31`

MONTH	`a=MDY(4,18,2020);`
	`x=MONTH(a);`
		`x=4`	`a=MDY(9,3,60);`
			`y=MONTH(a);`
				`y=9`

QTR	`a=MDY(4,18,2020);`
	`x=QTR(a);`
		`x=2`	`a=MDY(9,3,60);`
			`y=QTR(a);`
				`y=3`

TODAY	`x=TODAY();`	`x=today's date`	`y=TODAY()-1;`	`y=yesterday's date`

WEEKDAY	`a=MDY(4,13,2020);`
	`x=WEEKDAY(a);`
		`x=2`	`a=MDY(4,18,2020);`
			`y=WEEKDAY(a);`
				`y=7`

YEAR	`a=MDY(4,13,2000);`
	`x=YEAR(a);`
		`x=2000`	`a=MDY(1,1,1960);`
			`y=YEAR(a);`
				`y=1960`

YRDIF	`a=MDY(4,13,2000);`
	`b=MDY(4,13,2020);`
	`x=YRDIF(a,b,'AGE');`
		`x=20`	`a=MDY(4,13,2000);`
			`b=MDY(8,13,2020);`
			`y=YRDIF(a,b,'AGE');`
				`y=20.3342`

Formats	Input data	FORMAT statement[8]	Results

DATE*w*.	`366`	`FORMAT Birth DATE7.;`
		`FORMAT Birth DATE9.;`
			`01JAN61`
			`01JAN1961`

EURDFDD*w*.	`366`	`FORMAT Birth EURDFDD8.`
		`FORMAT Birth EURDFDD10.;`
			`01.01.61`
			`01.01.1961`

JULIAN*w*.	`366`	`FORMAT Birth JULIAN5.;`
		`FORMAT Birth JULIAN7.;`
			`61001`
			`1961001`

MMDDYY*w*.	`366`	`FORMAT Birth MMDDYY6.;`
		`FORMAT Birth MMDDYY10.;`
			`010161`
			`01/01/1961`

WEEKDATE*w*.	`366`	`FORMAT Birth WEEKDATE9.;`
		`FORMAT Birth WEEKDATE29.;`
			`Sunday`
			`Sunday, January 1, 1961`

WORDDATE*w*.	`366`	`FORMAT Birth WORDDATE12.;`
		`FORMAT Birth WORDDATE18.;`
			`Jan 1, 1961`
			`January 1, 1961`

[8] Formats can be used in PUT statements and PUT functions in DATA steps, and in FORMAT statements in either DATA or PROC steps.

3.15 Using RETAIN and Sum Statements

When variables are assigned values through either an INPUT or assignment statement, those variables are set to missing at the beginning of each iteration of the DATA step. The RETAIN and sum statements change this behavior. If a variable appears in a RETAIN statement, then its value will be retained from one iteration of the DATA step to the next. A sum statement also retains a value, but then it adds the value to an expression.

RETAIN statement Use the RETAIN statement when you want SAS to preserve a variable's value from the previous iteration of the DATA step. The RETAIN statement can appear anywhere in the DATA step and has the following form, where all variables to be retained are listed after the RETAIN keyword:

```
RETAIN variable-list;
```

You can also specify an initial value, instead of missing, for the variables. All variables listed before an initial value will start the first iteration of the DATA step with that value:

```
RETAIN variable-list initial-value;
```

Sum statement A sum statement also retains values from the previous iteration of the DATA step, but you use it for the special cases where you simply want to cumulatively add the value of an expression to a variable. A sum statement, like an assignment statement, contains no keywords. It has the following form:

```
variable + expression;
```

No, there is no typo here and no equal sign either. This statement adds the value of the expression to the variable while retaining the variable's value from one iteration of the DATA step to the next. The variable must be numeric and has the initial value of zero. This statement can be rewritten using the RETAIN statement and SUM function as follows:

```
RETAIN variable 0;
variable = SUM(variable, expression);
```

As you can see, a sum statement is really a special case of using RETAIN.

Example This example illustrates the use of both the RETAIN and sum statements. The minor league baseball team, the Walla Walla Sweets, has the following data about their games. The month and day the game was played, and the team played are followed by the number of hits and runs for the game:

```
Month Day  Team                 Hits Runs
6     19   Columbia Peaches     8    3
6     20   Columbia Peaches     10   5
6     23   Plains Peanuts       3    4
6     24   Plains Peanuts       7    2
6     25   Plains Peanuts       12   8
6     30   Gilroy Garlics       4    4
7     1    Gilroy Garlics       9    4
7     4    Sacramento Tomatoes  15   9
7     4    Sacramento Tomatoes  10   10
7     5    Sacramento Tomatoes  2    3
```

The team wants two additional variables in their data set. One shows the cumulative number of runs for the season, and the other shows the maximum number of runs in a game to date. The following program reads the tab-delimited file using PROC IMPORT, then in a DATA step, uses a sum statement to compute the cumulative number of runs, and the RETAIN statement and MAX function to determine the maximum number of runs in a game to date:

```
PROC IMPORT DATAFILE = 'c:\MyRawData\Games.txt' OUT = gamestats REPLACE;
RUN;
* Using RETAIN and sum statements to find most runs and total runs;
DATA gamestats;
   SET gamestats;
   RETAIN MaxRuns;
   MaxRuns = MAX(MaxRuns, Runs);
   RunsToDate + Runs;
RUN;
```

The variable MaxRuns is set equal to the maximum of its value from the previous iteration of the DATA step (since it appears in the RETAIN statement) or the value of the variable Runs. The variable RunsToDate adds the number of runs per game, Runs, to itself while retaining its value from one iteration of the DATA step to the next. This produces a cumulative record of the number of runs.

Here is the resulting SAS data set:

	Month	Day	Team	Hits	Runs	MaxRuns	RunsToDate
1	6	19	Columbia Peaches	8	3	3	3
2	6	20	Columbia Peaches	10	5	5	8
3	6	23	Plains Peanuts	3	4	5	12
4	6	24	Plains Peanuts	7	2	5	14
5	6	25	Plains Peanuts	12	8	8	22
6	6	30	Gilroy Garlics	4	4	8	26
7	7	1	Gilroy Garlics	9	4	8	30
8	7	4	Sacramento Tomatoes	15	9	9	39
9	7	4	Sacramento Tomatoes	10	10	10	49
10	7	5	Sacramento Tomatoes	2	3	10	52

3.16 Simplifying Programs with Arrays

Sometimes you want to do the same thing to many variables. You may want to take the log of every numeric variable or change every occurrence of zero to a missing value. You could write a series of assignment statements or IF statements, but if you have a lot of variables to transform, using arrays will simplify and shorten your program.

An array is an ordered group of similar items. You might think your local grocery store has a nice array of fruits to choose from. In SAS, an array is a group of variables. You can define an array to be any group of variables you like, as long as they are either all numeric or all character. The variables can be existing ones, or they can be new variables that you want to create.

Arrays are defined using the ARRAY statement in the DATA step. The ARRAY statement has the following general form:

```
ARRAY name (n) $ variable-list;
```

In this statement, *name* is a name you give to the array, and *n* is the number of variables in the array. Following the (*n*) is a list of variable names. The number of variables in the list must equal the number given in parentheses. (You may use { } or [ ] instead of parentheses if you like.) This is called an explicit array, where you explicitly state the number of variables in the array. Use a $ if the variables are character, and have not previously been defined.

The array itself is not stored with the data set; it is defined only for the duration of the DATA step. You can give the array any name, as long as it does not match any of the variable names in your data set or any SAS keywords. Also, array names must be 32 characters or fewer, start with a letter or underscore, and contain only letters, numerals, or underscores.

To reference a variable using the array name, give the array name and the subscript for that variable. The first variable in the variable list has subscript 1, the second has subscript 2, and so on. So if you have an array defined as:

```
ARRAY veg (4) Carrots Tomatoes Onions Celery;
```

VEG(1) is the variable Carrots, VEG(2) is the variable Tomatoes, VEG(3) is the variable Onions, and VEG(4) is the variable Celery. This is all just fine, but simply defining an array doesn't do anything for you. You want to be able to use the array to make things easier for you.

Example The radio station KBRK is conducting a survey asking people to rate five different songs. Songs are rated on a scale of 1 to 5, where 1 equals change the station when it comes on, and 5 equals turn up the volume when it comes on. If listeners had not heard the song or didn't care to comment on it, a 9 was entered for that song. Here are the data:

```
City,Age,wj,kt,tr,filp,ttr
Albany,54,3,9,4,4,9
Richmond,33,2,9,3,3,3
Oakland,27,3,9,4,2,3
Richmond,41,3,5,4,5,5
Berkeley,18,4,4,9,3,2
```

The listener's city of residence, age, and their responses to all five songs are listed. The following program first reads the comma-delimited file using PROC IMPORT and creates a permanent SAS data set, SONGS. Then, in a DATA step, the program changes all the 9s to missing values. (The variables are named using the first letters of the words in the song's title.)

```
* Create a permanent SAS data set;
LIBNAME radio 'c:\MySASLib';
PROC IMPORT DATAFILE = 'c:\MyRawData\KBRK.csv' OUT = radio.songs REPLACE;
RUN;

* Change all 9s to missing values;
DATA fixsongs;
   SET radio.songs;
   ARRAY song (5) wj kt tr filp ttr;
   DO i = 1 TO 5;
      IF song(i) = 9 THEN song(i) = .;
   END;
RUN;
```

An array, SONG, is defined as having five variables, the variables representing the five songs. Next comes an iterative DO statement. All statements between the DO statement and the END statement are executed, in this case, five times, once for each variable in the array.

The variable I is used as an index variable and is incremented by 1 each time through the DO loop. The first time through the DO loop, the variable I has a value of 1 and the IF statement would read `IF song(1)=9 THEN song(1)=.;`, which is the same as `IF wj=9 THEN wj=.;`. The second time through, I has a value of 2 and the IF statement would read `IF song(2)=9 THEN song(2)=.;`, which is the same as `IF kt=9 THEN kt=.;`. This continues through all five variables in the array.

Here is the SAS data set FIXSONGS created in the program:

	City	Age	wj	kt	tr	filp	ttr	i
1	Albany	54	3	.	4	4	.	6
2	Richmond	33	2	.	3	3	3	6
3	Oakland	27	3	.	4	2	3	6
4	Richmond	41	3	5	4	5	5	6
5	Berkeley	18	4	4	.	3	2	6

Notice that the array members SONG(1) to SONG(5) did not become part of the data set, but the variable I did. You could have written five IF statements instead of using arrays and accomplished the same result. In this program it would not have made a big difference, but if you had 100 songs in your survey instead of five, then using arrays would clearly be a better solution.

3.17 Using Shortcuts for Lists of Variable Names

While writing SAS programs, you will often need to write a list of variable names. If you only have a handful of variables, you might not feel a need for a shortcut. But if, for example, you need to define an array with 100 elements, you might be a little grumpy after typing in the 49th variable name knowing you still have 51 more to go. You might even think, "There must be an easier way." Well, there is.

You can use an abbreviated list of variable names almost anywhere you can use a regular variable list. In functions, abbreviated lists must be preceded by the keyword OF (for example, `SUM(OF Cat8 - Cat12)`). Otherwise, you simply replace the regular list with the abbreviated one.

Numbered range lists Variables that start with the same characters and end with consecutive numbers can be part of a numbered range list. The numbers can start and end anywhere as long as the number sequence is complete. For example, the following INPUT statement shows a variable list and its abbreviated form:

Variable list	Abbreviated list
`INPUT Cat8 Cat9 Cat10 Cat11 Cat12;`	`INPUT Cat8 - Cat12;`

Name range lists Name range lists depend on the internal order, or position, of the variables in the SAS data set. This is determined by the order of appearance of the variables in the DATA step. For example, in this DATA step, the internal variable order would be Y A C M R B:

```
DATA example;
   INFILE 'c:\MyRawData\TestData.dat';
   INPUT y a c m r;
   b = c + r;
RUN;
```

To specify a name range list, put the first variable, then two hyphens, then the last variable. The following PUT statements show the variable list and its abbreviated form using a named range:

Variable list	Abbreviated list
`PUT y a c m r b;`	`PUT y -- b;`

If you are not sure of the internal order, you can find out using PROC CONTENTS with the POSITION option. The following program will list the variables in the permanent SAS data set DISTANCE sorted by position:

```
LIBNAME mydir 'c:\MySASLib';
PROC CONTENTS DATA = mydir.distance POSITION;
RUN;
```

Use caution when including name range lists in your programs. Although they can save on typing, they may also make your programs more difficult to understand and to debug.

Name prefix lists Variables that start with the same characters can be part of a name prefix list, and can be used in some SAS statements and functions. For example:

Variable list	**Abbreviated list**
`DogBills = SUM(DogVet,DogFood,Dog_Care);`	`DogBills = SUM(OF Dog:);`

Special SAS name lists The special name lists, _ALL_, _CHARACTER_, and _NUMERIC_ can also be used any place you want either all the variables, all the character variables, or all the numeric variables in a SAS data set. These name lists are useful when you want to do something like compute the mean of all the numeric variables for an observation `(MEAN(OF _NUMERIC_))`, or list the values of all variables in an observation `(PUT _ALL_;)`.

Example The radio station KBRK wants to modify the DATA step from the previous section, which changes all 9s to missing values. Now, instead of changing the original variables, they create new variables (Song1 through Song5), which will have the new missing values. This program also computes the average score using the MEAN function. Here is the permanent SAS data SONGS:

	City	Age	wj	kt	tr	filp	ttr
1	Albany	54	3	9	4	4	9
2	Richmond	33	2	9	3	3	3
3	Oakland	27	3	9	4	2	3
4	Richmond	41	3	5	4	5	5
5	Berkeley	18	4	4	9	3	2

Here is the new program:

```
LIBNAME radio 'c:\MySASLib';
DATA fixsongs;
   SET radio.songs;
   ARRAY new (5) Song1 - Song5;
   ARRAY old (5) wj -- ttr;
   DO i = 1 TO 5;
      IF old(i) = 9 THEN new(i) = .;
         ELSE new(i) = old(i);
   END;
   AvgScore = MEAN(OF Song1 - Song5);
RUN;
```

Note that both ARRAY statements use abbreviated variable lists; array NEW uses a numbered range list and array OLD uses a name range list. Inside the iterative DO loop, the Song variables (array NEW) are set equal to missing if the original variable (array OLD) had a value of 9. Otherwise, they are set equal to the original values. After the DO loop, a new variable, AvgScore, is created using an abbreviated variable list in the function MEAN. The resulting DATA set includes variables from both the OLD array (wj -- ttr) and NEW array (Song1 - Song5):

	City	Age	wj	kt	tr	filp	ttr	Song1	Song2	Song3	Song4	Song5	i	AvgScore
1	Albany	54	3	9	4	4	9	3	.	4	4	.	6	3.66667
2	Richmond	33	2	9	3	3	3	2	.	3	3	3	6	2.75000
3	Oakland	27	3	9	4	2	3	3	.	4	2	3	6	3.00000
4	Richmond	41	3	5	4	5	5	3	5	4	5	5	6	4.40000
5	Berkeley	18	4	4	9	3	2	4	4	.	3	2	6	3.25000

3.18 Using Variable Names with Special Characters

Traditionally, SAS variable names must start with a letter or underscore, must be 32 characters or less, and cannot contain any special characters including spaces. If you find this too restrictive, then you will be glad know that it is possible to change the rules that SAS uses for variable names.

VALIDVARNAME= system option The system option VALIDVARNAME= controls which set of rules are used for variable names. If VALIDVARNAME= is set to V7, then variable names must conform to the traditional rules described above. If VALIDVARNAME= is set to ANY, then variable names may contain special characters including spaces, and may start with any character. Either way, variable names must still be 32 or fewer characters long. There are several ways that the value of the VALIDVARNAME= system option can be set (see Section 1.7). To find the default value for your SAS session, submit the following and read the SAS log:

```
PROC OPTIONS OPTION = VALIDVARNAME;
RUN;
```

To set the rules for naming variables for your current SAS session, use the OPTIONS statement

```
OPTIONS VALIDVARNAME = value;
```

where *value* is V7 to use traditional SAS naming rules, or ANY to use the more liberal rules.

Name Literals If you are using ANY rules, and you have variable names that contain spaces or special characters, then you must use the name literal form for variable names in your programs. Simply enclose the name in quotes followed by the letter N:

```
'variable-name'N
```

Example The following tab-delimited file contains information about camping equipment: the item name, country of origin, the online price, and the store price. Notice that some of the column headings contain spaces or special characters.

```
Item                Country of Origin Online$  Store$
3 Person Dome Tent  China             308      359
8 Person Cabin Tent USA               399      399
Camp Bag            USA               119      129
Down Mummy Bag      China             449      469
Deluxe Sleep Pad    Canada            169      179
Ultra-light Pad     USA               69       74
```

The following program sets VALIDVARNAME= equal to ANY and reads the file using PROC IMPORT. Then in a DATA step, it uses the name literal form of the variable names to subset the data using an IF statement, and it creates a new variable that is the difference between the store and online prices.

```
*Read data using ANY rules for variable names;
OPTIONS VALIDVARNAME = ANY;
PROC IMPORT DATAFILE = 'c:\MyRawData\CampEquip.txt'
     OUT = campequipment_any REPLACE;
RUN;
```

```
DATA campequipment_any;
   SET campequipment_any;
   IF 'Country of Origin'N = 'USA';
   PriceDiff = 'Store$'N - 'Online$'N;
RUN;
```

Here is the data set CAMPEQUIPMENT_ANY. Notice the special characters and spaces in the variable names.

	Item	Country of Origin	Online$	Store$	PriceDiff
1	8 Person Cabin Tent	USA	399	399	0
2	Camp Bag	USA	119	129	10
3	Ultra-light Pad	USA	69	74	5

If you decide that you don't want to use name literals, then you could choose to rename the variables so that the names conform to V7 rules. You can do this using a RENAME data set option. (See Section 6.10.)

Another option is to use V7 naming rules when creating the data set. If V7 rules are in place, then PROC IMPORT will convert spaces and special characters in headings to underscores when creating variable names. The following program is like the first one only with VALIDVARNAME= set to V7. Notice how now, instead of spaces and special characters, the variable names contain underscores and the name literal form of the variable name is not needed.

```
*Read data using V7 rules for variable names;
OPTIONS VALIDVARNAME = V7;
PROC IMPORT DATAFILE = 'c:\LSB6\Data\CampEquip.txt'
     OUT = CampEquipment_V7 REPLACE;
RUN;
DATA CampEquipment_V7;
   SET CampEquipment_V7;
   IF Country_of_Origin = 'USA';
   PriceDiff = Store_ - Online_;
RUN;
```

Here is the data set CAMPEQUIPMENT_V7.

	Item	Country_of_Origin	Online_	Store_	PriceDiff
1	8 Person Cabin Tent	USA	399	399	0
2	Camp Bag	USA	119	129	10
3	Ultra-light Pad	USA	69	74	5

If you are reading data files (either through PROC IMPORT or the XLSX LIBNAME engine) that contain headings which include spaces or special characters, we recommend that you always specify the VALIDVARNAME= rules that you want to use in an OPTIONS statement. That way your programs will always run no matter what the default value is for VALIDVARNAME= on your system.