



A Programming
Approach

The Little SAS® Book

a p r i m e r
s i x t h e d i t i o n

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

Copyright © 2019, SAS Institute Inc., Cary, NC, USA

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)

All rights reserved. Produced in the United States of America.

For a hard-copy book: No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.

For a Web download or e-book: Your use of this publication shall be governed by the terms established by the vendor at the time you acquire this publication.

The scanning, uploading, and distribution of this book via the Internet or any other means without the permission of the publisher is illegal and punishable by law. Please purchase only authorized electronic editions and do not participate in or encourage electronic piracy of copyrighted materials. Your support of others' rights is appreciated.

U.S. Government Restricted Rights Notice: Use, duplication, or disclosure of this software and related documentation by the U.S. government is subject to the Agreement with SAS Institute and the restrictions set forth in FAR 52.227-19, Commercial Computer Software-Restricted Rights (June 1987).

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

October 2019

SAS Institute Inc. provides a complete selection of books and electronic products to help customers use SAS software to its fullest potential. For more information about our e-books, e-learning products, CDs, and hard-copy books, visit the SAS Books Web site at **support.sas.com/bookstore or call 1-800-727-3228.**

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

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

Contents

Acknowledgments

Introducing SAS Software

About This Book

About These Authors

Chapter 1 Getting Started Using SAS Software

1.1 The SAS Language

1.2 SAS Data Sets

1.3 DATA and PROC Steps

1.4 The DATA Step's Built-in Loop

1.5 Choosing a Method for Running SAS

1.6 Reading the SAS Log

1.7 Using SAS System Options

Chapter 2 Accessing Your Data

2.1 Methods for Getting Your Data into SAS

2.2 SAS Data Libraries and Data Sets

2.3 Listing the Contents of a SAS Data Set

2.4 Reading Excel Files with the IMPORT Procedure

2.5 Accessing Excel Files Using the XLSX LIBNAME Engine

2.6 Reading Delimited Files with the IMPORT Procedure

2.7 Telling SAS Where to Find Your Raw Data

2.8 Reading Raw Data Separated by Spaces

2.9 Reading Raw Data Arranged in Columns

2.10 Reading Raw Data Not in Standard Format

2.11 Selected Informats

2.12 Mixing Input Styles

2.13 Reading Messy Raw Data

2.14 Reading Multiple Lines of Raw Data per Observation

2.15 Reading Multiple Observations per Line of Raw Data

2.16 Reading Part of a Raw Data File

2.17 Controlling Input with Options in the INFILE Statement

2.18 Reading Delimited Files with the DATA Step

Chapter 3 Working with Your Data

3.1 Using the DATA Step to Modify Data

3.2 Creating and Modifying Variables

3.3 Using SAS Functions

3.4 Selected SAS Character Functions

3.5 Selected SAS Numeric Functions

3.6 Using IF-THEN and DO Statements

3.7 Grouping Observations with IF-THEN/ELSE Statements

3.8 Subsetting Your Data in a DATA Step

3.9 Subsetting Your Data Using PROC SQL

3.10 Writing Multiple Data Sets Using OUTPUT Statements

3.11 Making Several Observations from One Using OUTPUT Statements

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

3.13 Working with SAS Dates

3.14 Selected Date Informats, Functions, and Formats

3.15 Using RETAIN and Sum Statements

3.16 Simplifying Programs with Arrays

3.17 Using Shortcuts for Lists of Variable Names

3.18 Using Variable Names with Special Characters

Chapter 4 Sorting, Printing, and Summarizing Your Data

4.1 Using SAS Procedures

4.2 Subsetting in Procedures with the WHERE Statement

4.3 Sorting Your Data with PROC SORT

4.4 Changing the Sort Order for Character Data

4.5 Printing Your Data with PROC PRINT

4.6 Changing the Appearance of DataValues with Formats

4.7 Selected Standard Formats

4.8 Creating Your Own Formats with PROC FORMAT

4.9 Writing a Report to a Text File

4.10 Summarizing Your Data Using PROC MEANS

4.11 Writing Summary Statistics to a SAS Data Set

4.12 Producing One-Way Frequencies with PROC FREQ

- 4.13 Producing Crosstabulations with PROC FREQ
- 4.14 Grouping Data with User-Defined Formats
- 4.15 Producing Tabular Reports with PROC TABULATE
- 4.16 Adding Statistics to PROC TABULATE Output
- 4.17 Enhancing the Appearance of PROC TABULATE Output
- 4.18 Changing Headers in PROC TABULATE Output
- 4.19 Producing Simple Output with PROC REPORT
- 4.20 Using DEFINE Statements in PROC REPORT
- 4.21 Creating Summary Reports with PROC REPORT
- 4.22 Adding Summary Breaks to PROC REPORT Output
- 4.23 Adding Statistics to PROC REPORT Output
- 4.24 Adding Computed Variables to PROC REPORT Output

Chapter 5 Enhancing Your Output with ODS

- 5.1 Concepts of the Output Delivery System
- 5.2 Creating HTML Output
- 5.3 Creating RTF Output
- 5.4 Creating PDF Output
- 5.5 Creating Text Output
- 5.6 Customizing Titles and Footnotes
- 5.7 Customizing PROC PRINT with the STYLE= Option
- 5.8 Customizing PROC REPORT with the

STYLE= Option

5.9 Customizing PROC TABULATE with the
STYLE= Option

5.10 Adding Trafficlighting to Your Output

5.11 Selected Style Attributes

5.12 Tracing and Selecting Procedure Output

5.13 Creating SAS Data Sets from Procedure
Output

Chapter 6 Modifying and Combining SAS Data Sets

6.1 Stacking Data Sets Using the SET
Statement

6.2 Interleaving Data Sets Using the SET
Statement

6.3 Combining Data Sets Using a One-to-One
Match Merge

6.4 Combining Data Sets Using a One-to-Many
Match Merge

6.5 Using PROC SQL to Join Data Sets

6.6 Merging Summary Statistics with the
Original Data

6.7 Combining a Grand Total with the Original
Data

6.8 Adding Summary Statistics to Data Using
PROC SQL

6.9 Updating a Master Data Set with
Transactions

6.10 Using SAS Data Set Options

6.11 Tracking and Selecting Observations with
the IN= Option

6.12 Selecting Observations with the WHERE=
Option

6.13 Changing Observations to Variables Using

PROC TRANSPOSE

6.14 Using SAS Automatic Variables

Chapter 7 Writing Flexible Code with the SAS Macro Facility

7.1 Macro Concepts

7.2 Substituting Text with Macro Variables

7.3 Concatenating Macro Variables with Other Text

7.4 Creating Modular Code with Macros

7.5 Adding Parameters to Macros

7.6 Writing Macros with Conditional Logic

7.7 Using %DO Loops in Macros

7.8 Writing Data-Driven Programs with CALL SYMPUTX

7.9 Writing Data-Driven Programs with PROC SQL

7.10 Debugging Macro Errors

Chapter 8 Visualizing Your Data

8.1 Concepts of ODS Graphics

8.2 Creating Bar Charts with PROC SGPlot

8.3 Creating Histograms and Density Curves with PROC SGPlot

8.4 Creating Box Plots with PROC SGPlot

8.5 Creating Scatter Plots with PROC SGPlot

8.6 Creating Series Plots with PROC SGPlot

8.7 Creating Fitted Curves with PROC SGPlot

8.8 Controlling Axes and Reference Lines in PROC SGPlot

8.9 Controlling Legends and Insets in PROC SGPlot

8.10 Customizing Graph Attributes in PROC

SGPLOT

8.11 Creating Paneled Graphs with PROC SGPANEL

8.12 Specifying Image Properties and Saving Graphics Output

Chapter 9 Using Basic Statistical Procedures

9.1 Examining the Distribution of Data with PROC UNIVARIATE

9.2 Creating Statistical Graphics with PROC UNIVARIATE

9.3 Producing Statistics with PROC MEANS

9.4 Testing Means with PROC TTEST

9.5 Creating Statistical Graphics with PROC TTEST 258

9.6 Testing Categorical Data with PROC FREQ

9.7 Creating Statistical Graphics with PROC FREQ

9.8 Examining Correlations with PROC CORR

9.9 Creating Statistical Graphics with PROC CORR

9.10 Using PROC REG for Simple Regression Analysis

9.11 Creating Statistical Graphics with PROC REG

9.12 Using PROC ANOVA for One-Way Analysis of Variance

9.13 Reading the Output of PROC ANOVA

Chapter 10 Exporting Your Data

10.1 Methods for Exporting Your Data

10.2 Writing Delimited Files with the EXPORT Procedure

10.3 Writing Delimited Files Using ODS

10.4 Writing Microsoft Excel Files with the EXPORT Procedure

10.5 Writing Microsoft Excel Files Using ODS

10.6 Writing Raw Data Files with the DATA Step

Chapter 11 Debugging Your SAS Programs

11.1 Writing SAS Programs That Work

11.2 Fixing Programs That Don't Work

11.3 Searching for the Missing Semicolon

11.4 Note: INPUT Statement Reached Past the End of a Line

11.5 Note: Lost Card

11.6 Note: Invalid Data

11.7 Note: Missing Values Were Generated

11.8 Note: Numeric Values Have Been Converted to Character (or Vice Versa)

11.9 DATA Step Produces Wrong Results but No Error Message

11.10 Error: Invalid Option, Error: The Option Is Not Recognized, or Error: Statement IsNot Valid

11.11 Note: Variable Is Uninitialized or Error: Variable Not Found

11.12 SAS Truncates a Character Variable

11.13 Saving Memory or Disk Space

Index

Acknowledgments

Over the years, many people have helped make this book a reality. We are grateful to everyone who has contributed both to this edition, and to editions past. It takes a family to produce a book including: reviewers, copyeditors, designers, publishing specialists, marketing specialists, and of course our editors. Special thanks go to our readers. We love hearing from you and meeting you at conferences. Without you, of course, there would be no reason for us to write.

The Little SAS Book Family Tree

John West

Stephenie Joyner, Dan Heath

Aimee Rodriguez, Mike Boyd

Brent Cohen, Cynthia Zender, Paul Kent

Stacey Hamilton, Bob Rodriguez, Chris

Hemedinger

Catherine Connolly, Randy Poindexter, Robert Harris

Denise Jones, Sanjay Matange, Amy Peters, Julie Palmieri

Jennifer Dilley, Sally Painter, Michelle Bucheker
Ginny Matsey, Rebecca Ottesen, Peter Ruzsa,
David D. Baggett

Sanford T. Gayle, Patrice Cherry, Helen Carey,
Sian Roberts, Todd Folsom

Deanna Warner, Carol Linden, Anthony House,
Robina G. Thornton
Cate Parrish, Susan C. Tideman, Kevin Hobbs,
Ted Meleky
Allison McMahill, Sandy McNeill, Darrell Barton,
Candy Farrell, Sandy Owens
Michael Williams, Heather B. Dees, Kathy Kiraly,
Mike Pezzoni
Carole Beam, Jason Moore, Ginger Carey, Phil
Gibbs, Morris Vaughan
David Schlotzhauer, Missy Hannah, Janice Bloom,
Jennifer M. Ginn, Jan Squillace
Mary Beth Steinbach, Jake Jacobs, Nancy
Mitchell, Linda Walters, Dina Fiorentino
Laurin Smith, Maggie Underberg, Julie McKnight,
Patsy J. Poole, Lorilyn Russell
Karen Perkins, Paul Grant, Kent Reeve, Gina
Repole, Blanche W. Phillips, Elizabeth Maldonado
Matthew R. Clark, Kris Rinne, Caroline Brickley
SAS Technical Support Staff
Our Families
Our Readers

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 are other ways you can practice what you learn. SAS University Edition is available for free download and installation on your personal computer. Or, you can set up an account to use SAS OnDemand for Academics which runs on servers hosted by SAS Institute. Both SAS University Edition and SAS OnDemand for Academics are 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. 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 University Edition and SAS OnDemand for Academics use 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.

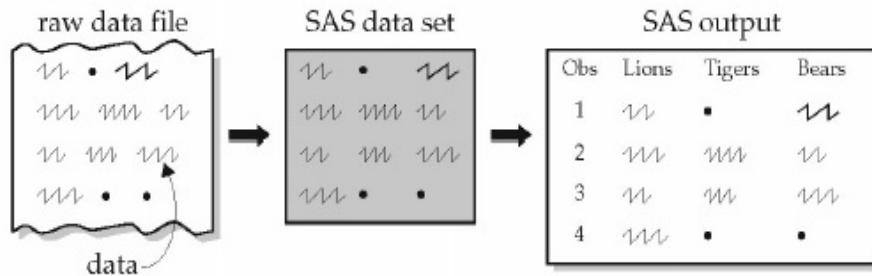
Unfortunately, a book of this type cannot provide a thorough introduction to statistical concepts such as degrees of freedom, or crossed and nested effects. There are underlying assumptions about your data that must be met for the tests to be valid. Experimental design and careful selection of models 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 SAS doesn't care whether your programs are written in uppercase or lowercase, so you can write your programs any way you want. 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

```
PROC PRINT DATA = data-set-name;
      VAR variable-list;
```

Example

```
PROC PRINT DAT
      VAR Lions Tig
```

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. Anything in lowercase or mixed case letters (and not in italics) is something that the programmer has made up such as a variable name, a name for a SAS data set, a comment, or a title. See Section 1.2 for further discussion of the significance of case in SAS names.

Indentation 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:

<http://support.sas.com/delwiche>

<http://support.sas.com/slaughter>

1

“An honest tale speeds best
being plainly told.”

WILLIAM SHAKESPEARE, *KING
RICHARD III*

From *King Richard III* by
William Shakespeare. Public
domain.

CHAPTER 1

Getting Started Using SAS Software

- [1.1 The SAS Language](#)
- [1.2 SAS Data Sets](#)
- [1.3 DATA and PROC Steps](#)
- [1.4 The DATA Step's Built-in Loop](#)
- [1.5 Choosing a Method for Running SAS](#)
- [1.6 Reading the SAS Log](#)
- [1.7 Using SAS System Options](#)

1.1 The SAS Language

Many software applications are either menu driven, or command driven (enter a command—see the result). SAS is neither. With SAS, you use statements to write a series of instructions called a SAS program. The program communicates what you want to do and is written using the SAS language. There are some menu-driven front ends to SAS, for example, SAS Enterprise Guide, which make SAS appear like a point-and-click program. However, these front ends still use the SAS language to write programs for you. You will have much more flexibility using SAS if you learn to write your own programs using the SAS language. Maybe learning a new language is the last thing you want to do, but be assured that although there are parallels between SAS and languages that you know (be they

English or Java), SAS is much easier to learn.

SAS programs A SAS program is a sequence of statements executed in order. A statement gives information or instructions to SAS and must be appropriately placed in the program. An analogy to a SAS program is a trip to the bank. You enter your bank, stand in line, and when you finally reach the teller's window, you say what you want to do. The statements you give can be written down in the form of a program:

I would like to make a withdrawal.
My account number is 0937.
I would like \$200.
Give me five 20s and two 50s.

Note that you first say what you want to do; then give all the information the teller needs to carry out your request. The order of the subsequent statements might not be important, but you must start with the general statement of what you want to do. You would not, for example, go up to a bank teller and say, "Give me five 20s and two 50s." This is not only bad form, but would probably make the teller's heart skip a beat or two. You must also make sure that all the subsequent statements belong with the first. You would not say, "I want the largest box you have" when making a withdrawal from your checking account. That statement belongs with "I would like to open a safe deposit box." A SAS program is an ordered set of SAS statements like the ordered set of instructions you use when you go to the bank.

SAS statements As with any language, there are a few rules to follow when writing SAS programs. Fortunately for us, the rules for writing SAS programs are much fewer and simpler than those for English.

The most important rule is

Every SAS statement ends with a

semicolon.

This sounds simple enough. But while children generally outgrow the habit of forgetting the period at the end of a sentence, SAS programmers never seem to outgrow forgetting the semicolon at the end of a SAS statement. Even the most experienced SAS programmer will at least occasionally forget the semicolon. You will be two steps ahead if you remember this simple rule.

Layout of SAS programs There really aren't any rules about how to format your SAS program. While it is helpful to have a neat looking program with each statement on a line by itself and indentations to show the various parts of the program, it isn't necessary.

- ◆ SAS statements can be in upper- or lowercase.
- ◆ Statements can continue on the next line (as long as you don't split words in two).
- ◆ Statements can be on the same line as other statements.
- ◆ Statements can start in any column.

So you see, SAS is so flexible that it is possible to write programs so disorganized that no one can read them, not even you. (Of course, we don't recommend this.)

Comments To make your programs more understandable, you can insert comments into your programs. It doesn't matter what you put in your comments —SAS doesn't look at it. You could put your favorite cookie recipe in there if you want. However, comments are generally used to annotate the program, making it easier for someone to read your program and understand what you have done and why.

There are two styles of comments that you can use. One style starts with an asterisk (*) and ends with a semicolon (;). The other style starts with a slash-asterisk /*) and ends

with an asterisk-slash (*/). The following program shows both styles of comment:

```
* Convert miles to kilometers;  
DATA distance;  
    Miles = 26.22;  
    Kilometers = 1.61 * Miles;  
RUN;  
  
PROC PRINT DATA = distance; /* Print the results */  
RUN;
```

Some operating environments interpret a slash-asterisk /*) in the first column as the end of a job. For this reason, we chose the asterisk-semicolon style of comment for this book. However, comments using the slash-asterisk style do have some advantages. Because they do not use a semicolon, they can contain embedded semicolons, and can be placed inside SAS statements

Programming tips People who are just starting to learn a programming language often get frustrated because their programs do not work correctly the first time they write them. Writing programs should be done in small steps. Don't try to tackle a long complicated program all at once. If you start small, build on what works, and always check your results along the way, you will increase your programming efficiency. Sometimes programs that do not produce errors are still incorrect. This is why it is vital to check your results as you go even when there are no errors. If you do get errors, don't worry. Most programs simply don't work the first time, if for no other reason than that you are human. You forget a semicolon, misspell a word, have your fingers in the wrong place on the keyboard. It happens. Often one small mistake can generate a whole list of errors. If you build your programs piece by piece, programs are much easier to correct when something goes wrong. Also, as you write programs, it is a good habit to save them frequently. That way, you won't lose your work

if unexpected problems occur.

1.2 SAS Data Sets

Before you run an analysis, before you write a report, before you do anything with your data, SAS must be able to read your data. Generally speaking, SAS wants your data to be in a special form called a SAS data set. (See Section 2.1 for exceptions.) Getting your data into a SAS data set is usually quite simple as SAS is very flexible and can read almost any data. Once your data have been read into a SAS data set, SAS keeps track of what is where and in what form. All you have to do is specify the name and location of the data set you want, and SAS figures out what is in it.

Variables and observations Data, of course, are the primary constituent of any data set. In traditional SAS terminology the data consist of variables and observations. Adopting the terminology of relational databases, SAS data sets are also called tables, observations are also called rows, and variables are also called columns. Below you see a rectangular table containing a small data set. Each line represents one observation, while Id, Name, Height, and Weight are variables. The data point Charlie is one of the values of the variable Name and is also part of the second observation.

Variables (Also Called Columns)				
	Id	Name	Height	Weight
Observations (Also Called Rows)	1	53	Susie	42
	2	54	Charlie	46
	3	55	Calvin	40
	4	56	Lucy	46
	5	57	Dennis	44
	6	58		50

Data types Raw data come in many different forms, but

SAS simplifies this. In Base SAS there are just two data types: numeric and character. Numeric fields are, well, numbers. They can be added and subtracted, can have any number of decimal places, and can be positive or negative. In addition to numerals, numeric fields can contain plus signs (+), minus signs (-), decimal points (.), or E for scientific notation. Character data are everything else. They may contain numerals, letters, or special characters (such as \$ or !) and can be up to 32,767 characters long.

If a variable contains letters or special characters, it must be a character variable. However, if it contains only numerals, then it may be numeric or character. You should base your decision on how you will use the variable. Sometimes data that consist solely of numerals make more sense as character data than as numeric. US five-digit postal codes, for example, are made up of numerals, but it just doesn't make sense to add or subtract postal codes. Such values make more sense as character data. In the preceding data set, Name is obviously a character variable, and Height and Weight are numeric. Id, however, could be either numeric or character. It's your choice.

Missing data Sometimes despite your best efforts, your data may be incomplete. The value of a particular variable may be missing for some observations. In those cases, missing character data are represented by blanks, and missing numeric data are represented by a single period (.). In the preceding data set, the value of Weight for observation 5 is missing, and its place is marked by a period. The value of Name for observation 6 is missing and is just left blank.

Size of SAS data sets Prior to SAS 9.1, SAS data sets could contain up to 32,767 variables. Beginning with SAS 9.1, the maximum number of variables in a SAS data set is limited by the resources available on your computer—but SAS data sets with more than 32,767 variables cannot be used with earlier versions of SAS. The number of

observations, no matter which version of SAS you are using, is limited only by your computer's capacity to handle and store them.

SAS data libraries and data set members SAS data sets are stored and accessed via SAS data libraries. A SAS data library is a collection of one or more SAS data sets that are stored in the same location. Some SAS data libraries are defined by default, but you can also define your own. (See Section 2.2.) The individual data sets within a SAS data library are called its members.

Rules for names of variables, SAS data libraries, and data set members You make up names for the variables in your data, for SAS data libraries, and for the data set members themselves. It is helpful to make up names that identify what the data represent, especially for variables. While the variable names A, B, and C might seem like perfectly fine, easy-to-type names when you write your program, the names Sex, Height, and Weight will probably be more helpful when you go back to look at the program six months later.

The specific rules for variable names depend on the value of the SAS system option VALIDVARNAME= on your system. (See Section 1.7 for more information about system options.) However, if you stick with the following rules, then your variable names will always be valid:

- ◆ Make **variable names 32 characters or fewer in length.**
- ◆ Start names with a letter or an underscore (_).
- ◆ Include only letters, numerals, or underscores (_).
No %\$!*&#@#, please.</div><div data-bbox="283 808 850 894" data-label="Text"><p>These rules apply when VALIDVARNAME=V7. If you have VALIDVARNAME=ANY, then the rules are the same except that variable names can begin with and contain any character including blanks. (See Section 3.18 for how to</p></div>

use variable names containing special characters.) Names for SAS data libraries and data set members follow the rules listed above except that libraries are limited to 8 characters in length. The VALIDVARNAME= system option has no effect on names of data libraries and members. It is possible that in the future some of these rules could change.

Capitalization of SAS names Another important point is that SAS is insensitive to case so you can use uppercase, lowercase, or mixed case—whichever looks best to you. SAS doesn't care. The data set name heightweight is the same as HEIGHTWEIGHT or HeightWeight. Likewise, the variable name BirthDate is the same as BIRTHDATE and birThDaTe. However, there is one difference for variable names. SAS remembers the case of the first occurrence of each variable name and uses that case when printing results. That is why, in this book, we use mixed case for variable names but lowercase for other SAS names.

Documentation stored in SAS data sets In addition to your actual data, SAS data sets contain information about the data set such as its name, the date that you created it, and the version of SAS that you used to create it. SAS also stores information about each variable, including its name, label (if any), type (numeric or character), length (or storage size), and position within the data set. This information is called the descriptor portion of the data set, and it makes SAS data sets self-documenting. You can use PROC CONTENTS to produce a report about the descriptor portion of a SAS data set. (See Section 2.3.)

1.3 DATA and PROC Steps



SAS programs are constructed from two basic building blocks: DATA steps and PROC steps. A typical program starts with a DATA step to create or modify a SAS data set and then passes the data to a PROC step for processing. Here is a simple program that converts miles to kilometers in a DATA step and prints the results with a PROC step:

```
DATA step   [ DATA distance;
              Miles = 26.22;
              Kilometers = 1.61 * Miles;
            ]
PROC step   [ PROC PRINT DATA = distance;
              RUN;
```

DATA and PROC steps are made up of statements. A step may have as few as one or as many as hundreds of statements. Most statements work in only one type of step—in DATA steps but not PROC steps, or vice versa. A common mistake made by beginners is to try to use a statement in the wrong kind of step. You’re less likely to make this mistake if you remember that, generally speaking, DATA steps read and modify data while PROC steps analyze data, perform utility functions, or print reports.

DATA steps start with the DATA statement, which starts, not surprisingly, with the keyword DATA. This keyword is followed by a name that you make up for a SAS data set. The DATA step above produces a SAS data set named DISTANCE. In addition to reading data from external, raw data files, DATA steps can modify existing SAS data sets

and include DO loops, IF-THEN/ELSE logic, and a large assortment of numeric and character functions. DATA steps can also combine data sets in just about any way you want, including concatenation and match-merge.

Procedures, on the other hand, start with a PROC statement in which the keyword PROC is followed by the name of the procedure (PRINT, SORT, or MEANS, for example). Most SAS procedures have only a handful of possible statements. Like following a recipe, you use basically the same statements or ingredients each time. SAS procedures do everything from simple sorting and printing to analysis of variance and 3-D graphics.

A step ends when SAS encounters a new step (marked by a DATA or PROC statement); a RUN, QUIT, STOP, or ABORT statement; or, if you are running in batch mode, the end of the program. RUN statements tell SAS to run all the preceding lines of the step and are among those rare, global statements that are not part of a DATA or PROC step. It is not necessary to insert a RUN statement between steps, but including them is a good habit. Having RUN statements between steps make your programs easier to read and debug. In the program above, SAS knows that the DATA step has ended when it reaches the PROC statement. The PROC step ends with a RUN statement, which coincides with the end of the program.

While a typical program starts with a DATA step to input or modify data and then passes the data to a PROC step, that is certainly not the only pattern for mixing DATA and PROC steps. Just as you can stack building blocks in any order, you can arrange DATA and PROC steps in any order. A program could even contain only DATA steps or only PROC steps.

To review, the table below outlines the basic differences between DATA and PROC steps:

DATA steps	PROC steps
▶ begin with DATA statements	▶ begin with PROC statements
▶ read and modify data	▶ perform specific analysis or task
▶ create a SAS data set	▶ produce results or report

As you read this table, keep in mind that it is a simplification. Because SAS is so flexible, the differences between DATA and PROC steps are, in reality, more blurry. The table above is not meant to imply that PROC steps never create SAS data sets (most do), or that DATA steps never produce reports (they can). Nonetheless, you will find it much easier to write SAS programs if you understand the basic functions of DATA and PROC steps.

1.4 The DATA Step's Built-in Loop

DATA steps read and modify data, and they do it in a way that is flexible, giving you lots of control over what happens to your data. However, DATA steps also have an underlying structure, an implicit, built-in loop. You don't tell SAS to execute this loop: SAS does it automatically. Memorize this:

**DATA steps execute line by line and
observation by observation.**

This basic concept is rarely stated explicitly. Consequently, new users often grow into old users before they figure this out on their own.

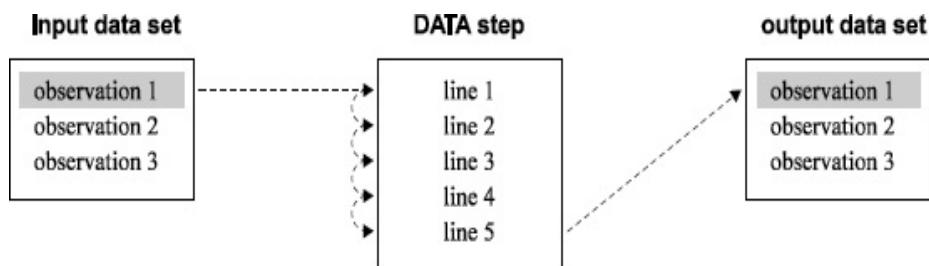
The idea that DATA steps execute line by line is fairly straightforward and easy to understand. It means that, by default, SAS executes line one of your DATA step before it executes line two, and line two before line three, and so on. That seems common sense, and yet new users frequently run into problems because they try to use a variable before they create it. If a variable named Z is the product of X and Y, then you better make sure that the statements creating X

and Y come before the statement creating Z.

What is not so obvious is that while DATA steps execute line by line, they also execute observation by observation. That means SAS takes the first observation and runs it all the way through the DATA step (line by line, of course) before looping back to pick up the second observation. In this way, SAS sees only one observation at a time.

Imagine a SAS program running in slow motion: SAS reads observation number one from your input data set. Then SAS executes your DATA step using that observation. If SAS reaches the end of the DATA step without encountering any serious errors, then SAS writes the current observation to a new, output data set and returns to the beginning of the DATA step to process the next observation. After the last observation has been written to the output data set, SAS terminates the DATA step and moves on to the next step, if there is one. End of slow motion; please return to normal gigahertz.

This diagram illustrates how an observation flows through a DATA step:



SAS reads observation number one and processes it using line one of the DATA step, then line two, and so on, until SAS reaches the end of the DATA step. Then SAS writes the observation in the output data set. This diagram shows the first execution of the line-by-line loop. Once SAS finishes with the first observation, it loops back to the top of the DATA step and picks up observation two. After SAS outputs the last observation, it automatically stops.

Here is an analogy. DATA step processing is a bit like

voting. When you arrive at your polling place, you stand in line behind other people who have come to vote. When you reach the front of the line you are asked standard questions: “What is your name? Where do you live?” Then you sign your name, and you cast your vote. In this analogy, the people are observations, and the voting process is the DATA step. People vote one at a time (or observation by observation). Each voter’s choices are secret, and peeking at your neighbor’s ballot is definitely frowned upon. In addition, each person completes each step of the process in the same order (line by line). You cannot cast your vote before you give your name and address. Everything must be done in the proper order.

If this seems a bit too structured, don’t worry. DATA steps are very flexible. SAS offers a number of ways to override the line-by-line and observation-by-observation structure. These include the RETAIN statement (see Section 3.15) and the OUTPUT statement (see Sections 3.10 and 3.11). It is even possible to have a DATA step that does not read or modify existing data. Here is the DATA step from the previous section. If you look at it, you may notice something interesting.

```
DATA distance;  
  Miles = 26.22;  
  Kilometers = 1.61 * Miles;  
RUN;
```

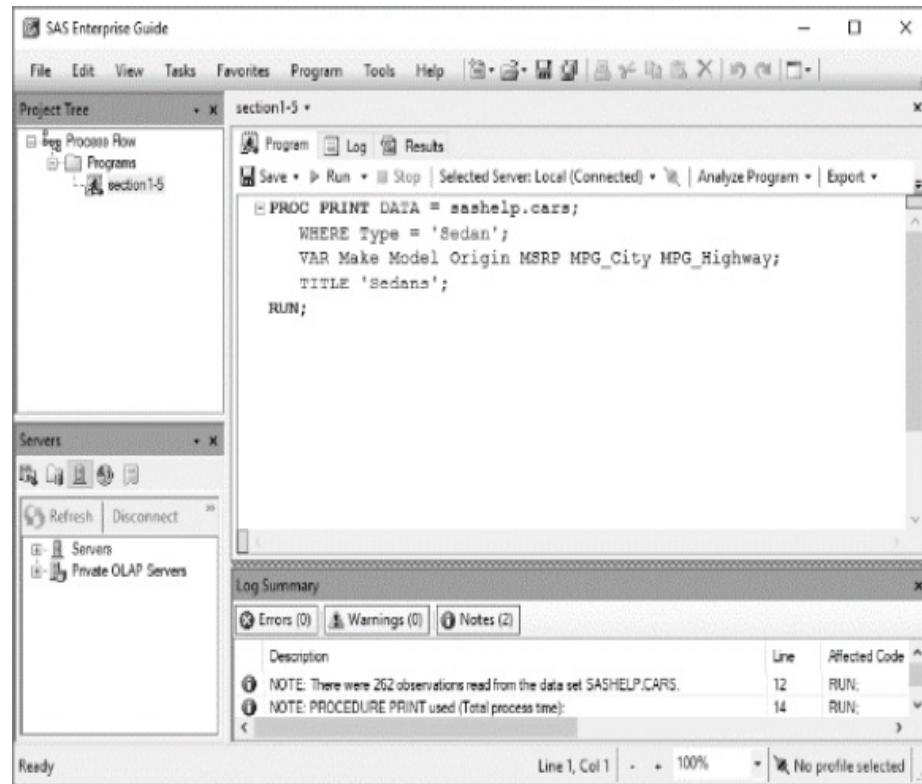
This simple DATA step has no input data set. Consequently, SAS will execute the DATA step once, output a single observation, and then terminate the DATA step because there are no more observations to process. The DATA step is covered in more detail in Chapters 2, 3, and 6.

1.5 Choosing a Method for Running SAS

So far we have talked about writing SAS programs, but simply writing a program does not give you any results.

Just like writing a letter to your representative in Congress does no good unless you mail it, a SAS program does nothing until you run it.

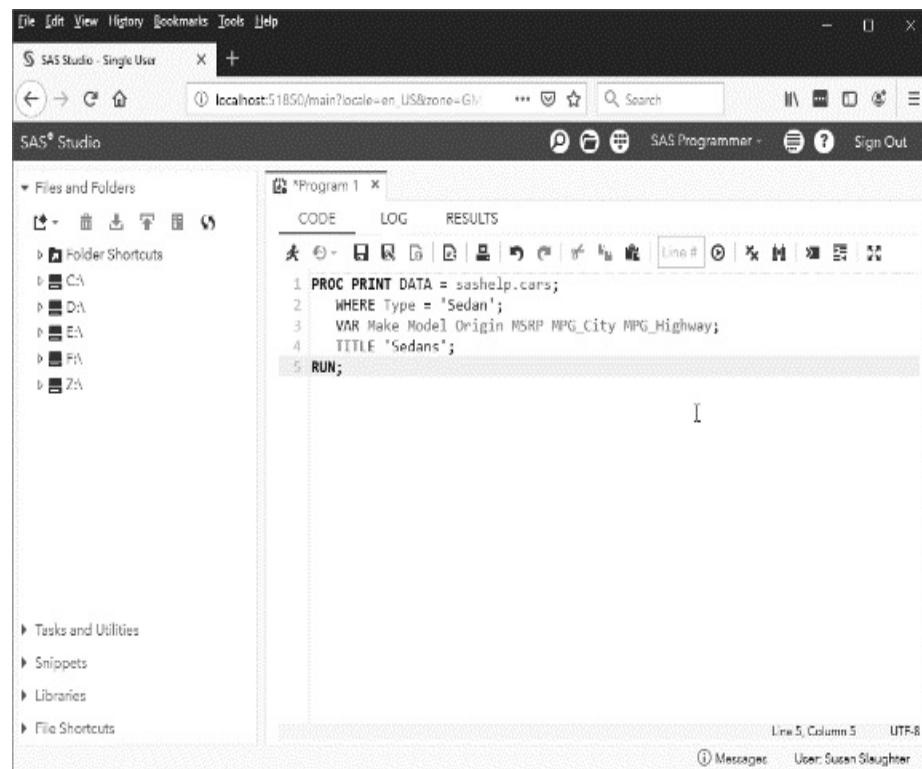
In recent years, there has been a proliferation of ways to run SAS programs. The following methods are included with Base SAS: SAS Enterprise Guide, SAS Studio, the SAS windowing environment, and batch or background mode. The best method for you will depend on your preferences and your operating environment. If you are using SAS at a large site with many users, then ask around and find out which method is the most accepted. If you are using SAS on your own personal computer, then choose the method that suits you.



SAS Enterprise Guide Using tasks and queries in SAS Enterprise Guide, you can write and run SAS programs without ever writing a single line of code. Tasks and queries are point-and-click windows for generating SAS code. However, you can also open a Program window and write your own programs. The program editor in SAS

Enterprise Guide color codes your program and displays automatic syntax help as you type. There is a program analyzer that will generate a diagram of your program to help you visualize the various steps and how they fit together. If you have SAS installed on more than one computer (maybe on your local machine and on a server), then you can choose where SAS runs your code. In SAS Enterprise Guide, you can store related programs, SAS logs, results, and references to data in a single file called a project. SAS Enterprise Guide displays projects in elegant process flow diagrams, so it is easy to see how the various data sets, programs, and tasks work together. You can view and edit SAS data sets in a Data Grid. If your program creates a new SAS data set, then by default SAS Enterprise Guide will automatically display it. SAS Enterprise Guide runs only under Windows.

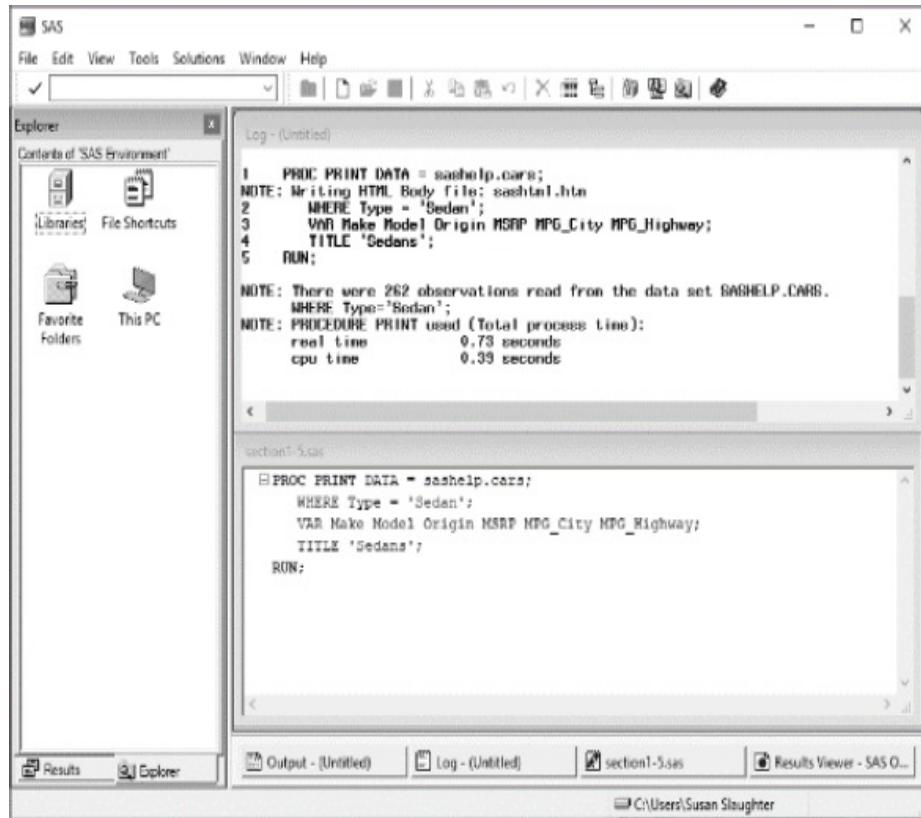
For a good introduction to SAS Enterprise Guide, search the internet for the video “Getting Started with SAS Enterprise Guide.”



SAS Studio SAS Studio runs in web browsers. With SAS Studio, you can access SAS on your local machine or on a server. Like SAS Enterprise Guide, SAS Studio has tasks and queries that can generate code for you, or you can open a Program window and type your own program. When you write programs, SAS Studio color codes your program and displays automatic syntax help. You can view SAS data sets, and if your program creates a new SAS data set, then by default SAS Studio will automatically display it.

SAS Studio is the interface for SAS University Edition and SAS OnDemand for Academics which are available free for academic, noncommercial use. When you download SAS University Edition, it installs a virtual Linux server on your local computer along with SAS software. SAS OnDemand for Academics runs on Linux servers hosted by SAS Institute. Regardless of which operating environment your local computer runs (Windows, Linux, or OS X for Macs), SAS University Edition and SAS OnDemand for Academics still run on Linux servers. You access those Linux servers via SAS Studio from your local computer.

For a good introduction to SAS Studio, search the internet for the video “Getting Started with SAS Studio.”



SAS windowing environment Also known as Display Manager, the SAS windowing environment is the oldest interactive interface for running SAS. The SAS windowing environment has fewer features than SAS Enterprise Guide or SAS Studio, but you can still write and edit SAS programs, run programs, and view and print your results. The SAS windowing environment has a syntax-sensitive editor that color codes your programs. There are windows for performing tasks such as managing SAS files, accessing SAS Help and Documentation, and importing or exporting data. In addition, you can view and edit SAS data sets using the Viewtable window. The SAS windowing environment does not automatically display new SAS data sets that your programs create. However, you can open data sets yourself by navigating to them in the SAS Explorer window (located on the left) and double-clicking their icons.

For a good introduction to the SAS windowing environment, search the internet for the video “Getting

Started with the SAS Windowing Environment.”

Batch or background mode With batch or background mode, you save your SAS program in a file. You can create that file using one of the preceding SAS interfaces, or a text editor such as Microsoft Notepad. Then you submit the file for processing by SAS. Your SAS program may start executing immediately, or it may start at a time you specify (midnight, for example), or it could be put in a queue behind other jobs. After you submit your job, you can work on other things, or better yet, go to a baseball game and let SAS work while you play. Batch processing is a convenient way to set up production jobs to run automatically on a schedule, maybe once a week or once a month. When your job is complete, the results will be placed in a file or files, which you can display or print at any time.

To find out how to submit SAS programs for batch processing, check the SAS Documentation for your operating environment, or check with other SAS users at your site. Even sites with the same operating environment may have different ways of submitting jobs in batch mode.

1.6 Reading the SAS Log

Every time you submit a SAS program, SAS writes messages in your log. Many SAS programmers ignore the SAS log and go straight to the output. That’s understandable, but dangerous. It is possible—and sooner or later it happens to all of us—to get bogus results that look just fine. The only way to know they are bad is to check the SAS log. Just because it runs doesn’t mean it’s right.

Where to find the SAS log The location of the SAS log varies depending on the operating environment you use, the method you use to submit your program (SAS windowing environment, SAS Enterprise Guide, SAS Studio, or batch),

and local settings. If you submit a program in the SAS windowing environment, you will, by default, see the SAS log in your Log window. In SAS Enterprise Guide and SAS Studio, by default, you will see a Log tab.

If you submit your program in batch mode, the log will be written to a file that you can view or print using your operating environment's commands for viewing and printing. The name given to the log file is generally some permutation of the name you gave to the original program. For example, if you named your SAS program Marathon.sas, then it is a good bet that your log file will be Marathon.log.

What the log contains People tend to think of the SAS log as either a rehash of their program or as just a lot of gibberish. OK, we admit, there is some technical trivia in the SAS log, but there is also plenty of important information. Here is a simple program that converts miles to kilometers and prints the result:

```
* Create a SAS data set named distance;  
* Convert miles to kilometers;  
DATA distance;  
    Miles = 26.22;  
    Kilometers = 1.61 * Miles;  
RUN;  
* Print the results;  
PROC PRINT DATA = distance;  
RUN;
```

If you run this program, SAS will produce a log similar to this:

① NOTE: Copyright (c) 2016 by SAS Institute Inc., Cary, NC, USA.

NOTE: This session is executing on the W64_8HOME platform.

NOTE: SAS initialization used:

real time	1.40 seconds
cpu time	0.96 seconds

① 1 * Create a SAS data set named distance;

2 * Convert miles to kilometers;

3 DATA distance;

4 Miles = 26.22;

5 Kilometers = 1.61 * Miles;

6 RUN;

③ NOTE: The data set WORK.DISTANCE has 1 observations and 2 variables.

④ NOTE: DATA statement used (Total process time):

real time	0.03 seconds
cpu time	0.03 seconds

② 7 * Print the results;

8 PROC PRINT DATA = distance;

9 RUN;

NOTE: There were 1 observations read from the data set WORK.DISTANCE

④ NOTE: PROCEDURE PRINT used (Total process time):

real time	0.01 seconds
cpu time	0.00 seconds

The SAS log above is a blow-by-blow account of how SAS executes the program.

- ① It starts with notes about the version of SAS and your SAS site number.
- ② It contains the original program statements with line numbers added on the left.
- ③ The DATA step is followed by a note containing the name of the SAS data set created (WORK.DISTANCE), and the number of observations (1) and variables (2). A quick glance is enough to assure you that you did not lose any observations or accidentally create a lot of unwanted variables.

- ④ Both DATA and PROC steps produce a note about the computer resources used. At first you probably won't care in the least. But if you run on a multiuser system or have long jobs with large data sets, these statistics may start to pique your interest. If you ever find yourself wondering why your job takes so long to run, a glance at the SAS log will tell you which steps are the culprits.

If there were error messages, they would appear in the log, indicating where SAS got confused and what action it took. You may also find warnings and other types of notes which sometimes indicate errors and other times just provide useful information. Chapter 11 discusses several of the more common errors that SAS users encounter.

1.7 Using SAS System Options

System options are parameters you can change that affect SAS—how it works, what the output looks like, how much memory is used, error handling, and a host of other things. SAS makes many assumptions about how you want it to work. This is good. You do not want to specify every little detail each time you use SAS. However, you may not always like the assumptions SAS makes. System options give you a way to change some of these assumptions.

OPTIONS procedure Not all options are available for all operating environments. You can see a list of system options and their current values by using the OPTIONS procedure. To use the OPTIONS procedure, submit the following SAS program and view the results in the SAS log:

```
PROC OPTIONS;  
RUN;
```

The list of options produced by PROC OPTIONS is long and can be overwhelming. If you know the specific options you are interested in, then you can request just those values. This statement will display just the MISSING, VALIDVARNAME, and YEARCUTOFF system options:

```
PROC OPTIONS OPTION = (MISSING  
VALIDVARNAME YEARCUTOFF);  
RUN;
```

Changing option settings There are several possible ways to specify system options:

1. Create a SAS configuration file that contains settings for the system options. This file is accessed by SAS every time SAS is started. Configuration files are created by systems administrators. (This could be you if you are using a personal computer.)
2. Specify system options at the time you start up SAS from your system's prompt (called the invocation).
3. Specify system options in an autoexec file, or in the Edit Autoexec File window in SAS Studio, or in an autoexec process flow in SAS Enterprise Guide.
4. Use the OPTIONS statement as a part of your SAS program. If you are using the SAS windowing environment, you can specify options in the SAS System Options window.

The methods are listed here in order of increasing precedence; method 2 will override method 1, method 3 will override method 2, and so on. In the SAS windowing environment, the SAS System Options window and the OPTIONS statement will override each other—so whichever was used last will be in effect. The OPTIONS statement is covered in this section. To find out more about other methods check the SAS Documentation for your operating environment.

OPTIONS statement The OPTIONS statement starts

with the keyword OPTIONS and follows with a list of options and their values. For example:

```
OPTIONS LEFTMARGIN = 1IN NODATE;
```

The OPTIONS statement is one of the special SAS statements that do not belong to either a PROC or a DATA step. This global statement can appear anywhere in your SAS program, but it usually makes the most sense to let it be the first line in your program. This way you can easily see which options are in effect. If the OPTIONS statement is in a DATA or PROC step, then it affects that step and any following steps. Subsequent OPTIONS statements in a program override any previous ones. System options that are set in this way apply only to your current program or session. Once you exit SAS, the system options will be set back to their default values.

Selected general options Here are some system options you might want to use:

CENTER NOCENTER	controls whether output is center justified. Default is CENTER.
----------------------	--

DATASTMTCHK = <i>value</i>	Specifies which SAS statement is prohibited from being specified member name to protect against input data set. See Section 11.3 example. Possible values are COREKEYWORDS (the default), ALLKEYWORDS, or NONE.
-------------------------------	---

DATESTYLE = <i>value</i>	Determines order of month, day, the ANYDTDTE. informat is a Possible values include MDY (month, day, year), DMY, and YMD. See Section 3
--------------------------	---

MISSING = '*character*' Specifies the character (which can be blank) to print for missing numbers. Default is a period (.).

YEARCUTOFF = *n* Specifies the first year of a 100-year range. This option is used by date informats and functions to read a two-digit year. See Section 1.2 for details. Default varies.

VALIDVARNAME= '*value*' Specifies the rules for valid SAS variable names. Possible values include V7. See Section 1.2 for rules for V7. Variable names can use special characters, including spaces, but names containing special characters must use a name form '*variable-name'N*'. See Section 1.2 for examples. Default varies.

Selected options for printing The following options affect the appearance of results in formats meant for printing, such as PDF and RTF (in other words not HTML):

DATE | NODATE controls whether or not today's date appear at the top of each page. Default is DATE.

NUMBER | NONUMBER controls whether or not page numbers appear on each page of SAS output. Default is NUMBER.

ORIENTATION = '*orientation*' specifies the orientation for printed output, either LANDSCAPE or PORTRAIT.

Default is PORTRAIT.

PAGENO = *n* starts numbering output page
Default is 1.

RIGHTMARGIN = *n* specifies the size of the margin
LEFTMARGIN = *n* (0.75in or 2cm) to be used for
TOPMARGIN = *n* designed for printing. Default
BOTTOMMARGIN = *n*

2

“Practice is the best of all instructors.”

PUBLIUS SYRUS, CIRCA 42 B.C

“We all learned by doing, by experimenting (and often failing), and by asking questions.”

JAY JACOB WIND

From *Bartlett's Familiar Quotations* 13th edition, by John Bartlett, copyright 1955 by Little Brown & Company. Public domain.

From the SAS L Listserv, March 15, 1994. Reprinted by permission of the author.

CHAPTER 2

Accessing Your Data

- 2.1 Methods for Getting Your Data into SAS
- 2.2 SAS Data Libraries and Data Sets
- 2.3 Listing the Contents of a SAS Data Set
- 2.4 Reading Excel Files with the IMPORT Procedure
- 2.5 Accessing Excel Files Using the XLSX LIBNAME Engine
- 2.6 Reading Delimited Files with the IMPORT Procedure
- 2.7 Telling SAS Where to Find Your Raw Data
- 2.8 Reading Raw Data Separated by Spaces
- 2.9 Reading Raw Data Arranged in Columns
- 2.10 Reading Raw Data Not in Standard Format
- 2.11 Selected Informats
- 2.12 Mixing Input Styles
- 2.13 Reading Messy Raw Data
- 2.14 Reading Multiple Lines of Raw Data per Observation
- 2.15 Reading Multiple Observations per Line of Raw Data
- 2.16 Reading Part of a Raw Data File
- 2.17 Controlling Input with Options in the INFILE Statement
- 2.18 Reading Delimited Files with the DATA Step

2.1 Methods for Getting Your Data into SAS



Data come in many different forms. Your data may be handwritten on a piece of paper, or typed into a raw data file on your computer. Perhaps your data are in a database file on your personal computer, or in a data warehouse on the server at your office. Wherever your data reside, there is a way for SAS to use them. You may need to convert your data from one form to another, or SAS may be able to use your data in their current form. This section outlines several methods for getting your data into SAS. Most of these methods are covered in this book, but a few of the more advanced methods are merely mentioned so that you know they exist. We do not attempt to cover all methods available for getting your data into SAS, as new methods are continually being developed, and creative SAS users can always come up with clever methods that work for their own situations. However your data are stored, there should be at least one method in this book that will work for you.

Methods for accessing your data can be put into five general categories:

- ◆ using SAS data sets
- ◆ entering data directly into SAS data sets
- ◆ creating SAS data sets from raw data files
- ◆ converting other software's data files into SAS data sets
- ◆ reading other software's data files directly

Using SAS data sets If your data are already stored as SAS data sets, all you have to do is tell SAS which data set to use. SAS data sets are stored and accessed via SAS data libraries. A SAS data library is a collection of one or more

SAS data sets that are stored in the same location. Some SAS data libraries are defined by default, but you can also define your own. Individual data sets within a SAS data library are called members. See the next section for more information about SAS data libraries and data set members.

Entering data directly into SAS data

sets Sometimes the best method for getting your data into SAS is to enter the data directly into SAS data sets through your keyboard.

- ◆ The Viewtable window is part of the SAS windowing environment and is included with Base SAS software. Viewtable allows you to enter your data in a tabular format. You can define variables (also called columns) and give them attributes such as name, length, and type (character or numeric).
- ◆ The Data Grid is part of SAS Enterprise Guide, which is included with Base SAS, but only for Windows. As with Viewtable, you can define variables, give them attributes, and then enter your data.

Creating SAS data sets from raw data files Much of this chapter is devoted to reading raw data files (also referred to as text, ASCII, sequential, or flat files). You can always read a raw data file using Base SAS software. If your data are not already in a raw data file, chances are you can convert your data into a raw data file. There are two general methods for reading raw data files:

- ◆ The DATA step is so versatile that it can read almost any type of raw data file. This method is covered in this chapter starting with Section 2.7.
- ◆ The IMPORT procedure, covered in Section 2.6, is available with Base SAS running on the UNIX and Windows operating environments. This is an easy way to read particular types of raw data files

including tab-delimited and comma-separated values (CSV) files.

Converting other software's data files into SAS

data sets Each software application has its own form for data files. While this is useful for software developers, it is troublesome for software users—especially when your data are in one application, but you need to analyze them with another. There are several options for converting data:

- ◆ The IMPORT procedure, available for UNIX and Windows operating environments, can be used to convert Microsoft Excel (covered in Section 2.4), Lotus, dBase, Stata, SPSS, JMP, Paradox, and Microsoft Access files into SAS data sets. All of these except JMP require that you have SAS/ACCESS Interface to PC Files installed on your SAS server.
- ◆ If you don't have SAS/ACCESS software, then you can always create a raw data file from your application and read the raw data file with either the DATA step or the IMPORT procedure. Many applications can create CSV files, which are easily read using the IMPORT procedure (covered in Section 2.6) or the DATA step (covered in Section 2.18).

Reading other software's data files directly Under certain circumstances, you may be able to read data without converting to a SAS data set. This method is particularly useful when you have many people updating data files, and you want to make sure that you are using the most current data.

- ◆ The SAS/ACCESS products allow you to read data without converting your data into SAS data sets. There are SAS/ACCESS products for most of the popular database management systems including Oracle, DB2, MySQL, and Teradata. This method of

data access is not covered in this book.

- ◆ SAS/ACCESS Interface to PC Files also allows you to read some PC file types directly without converting them to SAS data sets including Excel (covered in Section 2.5), Access, and JMP. One method that SAS uses to access PC files in their native format is called a LIBNAME engine. To tell SAS the type of file you want to read, you specify a LIBNAME statement that includes the option for that particular engine. See the SAS Documentation for more information on LIBNAME engines.
- ◆ There are also LIBNAME engines that allow you to read data directly and are part of Base SAS software (so you don't need SAS/ACCESS Interface to PC Files). There are engines for SPSS, OSIRIS, old versions of SAS data sets, and SAS data sets in transport format. Check the SAS Documentation for your operating environment for a complete list of available engines.

Given all these methods for getting your data into SAS, you are sure to find at least one method that will work for you —probably more.

2.2 SAS Data Libraries and Data Sets

It's not always obvious, but all SAS data sets have a two-level name such as WORK.BANANA. The first level is called its libref (short for SAS data library reference), and the second level is the member name that uniquely identifies the data set within the library. A SAS data library is a collection of SAS data sets residing in the same location. So a libref is like a nickname for that location. Sometimes a libref refers to a physical location, such as a hard drive or flash drive. Other times it refers to a logical

location such as a directory or folder.

Both the libref and member name follow the standard rules for SAS names. They must start with a letter or underscore and contain only letters, numerals, or underscores.

However, librefs cannot be longer than 8 characters while member names can be up to 32 characters. (There are system options that allow you to bend these rules, and the defaults may change, so check the SAS Documentation if you have a question.)

Built-in SAS data libraries SAS comes with some built-in libraries. The exact libraries available to you will depend on the SAS products you have installed and whether you have any locally defined libraries, but will always include these:

WORK is a special library for temporary SAS data sets. The library is erased when you exit SAS. WORK is the default library if you don't specify a libref in the data step.

SASHELP contains sample data sets that you can use, and is read-only.

SASUSER is a library provided for you to save your data sets. The library is permanent on most systems. However, on some UNIX systems, it may be temporary or permanent. If you have more than one SAS server, then you must have a SASUSER library on each server.

Temporary SAS data sets If you create a data set that you won't need later, it's a good idea to make it temporary so your disks don't get cluttered. Here is the DATA step shown in Chapter 1:

`DATA distance;`

```
Miles = 26.22;  
Kilometers = 1.61 * Miles;  
RUN;
```

This DATA step creates a temporary data set. Notice that the data set name does not include the libref WORK. Because the DATA statement uses a one-level name, SAS assigns the default library, WORK, and uses DISTANCE as the member name within that library. The log contains this note showing the complete, two-level name:

NOTE: The data set WORK.DISTANCE has 1 observations
and 2 variables.

Permanent SAS data sets using direct referencing In general, if you use a data set more than once, it is more efficient to save it as a permanent SAS data set than to create a new temporary SAS data set every time you want to use the data. To use direct referencing, just take your operating environment's file path and name, enclose it in quotation marks, and put it in your program. The quotation marks tell SAS that this is a permanent SAS data set. Here is the general form of a DATA statement for direct referencing in different operating environments. (Keep in mind that SAS University Edition and SAS OnDemand for Academics use Linux.)

Windows: DATA '*drive:\directory\filename*';

UNIX or DATA '*/home/path/filename*';
Linux:

z/OS: DATA '*data-set-name*';

This program is the same as the preceding one except that it includes the file path and name enclosed in quotes (using

the syntax for the Windows operating environment):

```
DATA 'c:\MySASLib\distance';
  Miles = 26.22;
  Kilometers = 1.61 * Miles;
RUN;
```

This time the log contains this note showing the file path and name:

```
NOTE: The data set c:\MySASLib\distance has 1
observations and 2 variables.
```

The SAS log does not show the libref, but if you check your SAS libraries you will see that SAS has created a library with a name like WC000001 and it contains a data set named DISTANCE. Librefs are temporary, but this data set is permanent because it is not in the WORK library.

Permanent SAS data sets with a LIBNAME

statement If you read or write many data sets in the same SAS data library, it is easier (and less error prone) to use a LIBNAME statement than to keep typing your file path and name over and over. Here is the general form of LIBNAME statements for different operating environments:

Windows: LIBNAME *libref*'*drive:\directory*';

UNIX or LIBNAME *libref*'*/home/path*';
Linux:

z/OS: LIBNAME *libref*'*data-set-name*';

This program is the same as the preceding two except that it uses a LIBNAME statement to define a libref. Notice that the DATA statement uses the two-level name, MARATHON.DISTANCE.

```
LIBNAME marathon 'c:\MySASLib';
DATA marathon.distance;
    Miles = 26.22;
    Kilometers = 1.61 * Miles;
RUN;
```

This time the log contains this note with the two-level data set name:

NOTE: The data set MARATHON.DISTANCE has 1 observations and 2 variables.

This is a permanent SAS data set because the libref is not WORK. If you view the directory of files on your computer, you will not see a file named MARATHON.DISTANCE. That is because operating environments have their own systems for naming files. When run under Windows or UNIX, this data set will be called distance.sas7bdat. Under z/OS, the filename would be the *data-set-name* specified in the LIBNAME statement. The examples in this section all create SAS data sets, the next section shows an example, which accesses an existing permanent SAS data set.

2.3 Listing the Contents of a SAS Data Set

To use a SAS data set, all you need to do is tell SAS the name and location of the data set you want, and SAS will figure out what is in it. SAS can do this because SAS data sets are self-documenting, which is another way of saying that SAS automatically stores information about the data set (called the descriptor portion) along with the data. There is an easy way to see the information about your data set, use PROC CONTENTS.

PROC CONTENTS is a simple procedure. You just type the keywords PROC CONTENTS and specify the data set you want with the DATA= option:

PROC CONTENTS DATA = *data-set*;

Example The SASHELP data library contains sample data sets and is provided with Base SAS. The following PROC CONTENTS produces a report about the CARS data set in the SASHELP data library. Because the SASHELP library is defined by default, no LIBNAME statement is needed.

* Use PROC CONTENTS to describe the CARS data set in the SASHELP library;

```
PROC CONTENTS DATA = SASHELP.CARS;
```

```
RUN;
```

The output from PROC CONTENTS is like a table of contents for your data set:

The CONTENTS Procedure

① Data Set Name	SASHELP.CARS	② Observatio
Member Type	DATA	③ Variables
Engine	V9	Indexes
④ Created	06/24/2015 22:15:37	Observation
Last Modified	06/24/2015 22:15:37	Deleted Observation
Protection		Compressed

Data Set Type		Sorted
⑤ Label	2004 Car Data	
Data Representation	WINDOWS_64	
Encoding	us-ascii ASCII (ANSI)	

Engine/Host Dependent Information	
Data Set Page Size	65536
Number of Data Set Pages	2
First Data Page	1
Max Obs per Page	430
Obs in First Data Page	413
Number of Data Set Repairs	0
ExtendObsCounter	YES
Filename	C:\Program

	Files\SASHome\SASFoundation\9.4\core\s7bdat
Release Created	9.0401M3
Host Created	X64_SRV12

Alphabetic List of Variables and Attributes					
#	⑥ Variable	⑦ Type	⑧ Len	⑨ Format	⑩
9	Cylinders	Num	8		
5	DriveTrain	Char	5		
8	EngineSize	Num	8		Enc
10	Horsepower	Num	8		
7	Invoice	Num	8	DOLLAR8.	
15	Length	Num	8		Len
11	MPG_City	Num	8		MPG
12	MPG_Highway	Num	8		MPG
6	MSRP	Num	8	DOLLAR8.	

1	Make	Char	13		
2	Model	Char	40		
4	Origin	Char	6		
3	Type	Char	8		
13	Weight	Num	8	Wei	
14	Wheelbase	Num	8	Wh	

Sort Information	
Sortedby	Make Type
Validated	YES
Character Set	ANSI

The output starts with information about your data set and then describes each variable.

For the data set

- ① Data set name

For each variable

- ⑥ Variable name

- | | |
|---|---|
| <p>② Number of observations</p> <p>③ Number of variables</p> <p>④ Date created</p> <p>⑤ Data set label (if any)</p> | <p>⑦ Type (numeric or character)</p> <p>⑧ Length (storage size in bytes)</p> <p>⑨ Format for printing (if any)</p> <p>⑩ Variable label (if any)</p> |
|---|---|

2.4 Reading Excel Files with the IMPORT Procedure

SAS offers many ways to access data in Microsoft Excel files. In fact, entire books have been written about exchanging data between SAS and Excel. SAS Studio, SAS Enterprise Guide and the SAS windowing environment each have point-and-click windows that will import Excel files for you, but writing a program gives you more control and is easier to repeat. If you have SAS/ACCESS Interface to PC Files and are running in the Windows or UNIX operating environment, then you can use the IMPORT procedure to read Excel files. PROC IMPORT will scan your Excel worksheet and automatically determine the variable types (character or numeric), will assign lengths to the character variables, and can recognize most date formats. Also, if you want, you can use the first row in your worksheet for the variable names.

Here is the general form of the IMPORT procedure for reading Excel files:

```
PROC IMPORT DATAFILE = 'filename' OUT = data-set
```

```
DBMS = identifier REPLACE;
```

where *filename* is the file you want to read and *data-set* is the name of the SAS data set you want to create. The REPLACE option tells SAS to replace the SAS data set named in the OUT= option if it already exists. The DBMS= option tells SAS the type of Excel file to read and may not be necessary.

DBMS identifiers There are several DBMS identifiers you can use to read Excel files. Three commonly used identifiers are: EXCEL, XLS, and XLSX. In the UNIX operating environment, use the XLS identifier for older style files (.xls extension), and the XLSX identifier for newer style files (.xlsx extension). In the Windows operating environment, in addition to the XLS and XLSX identifiers, you can use the EXCEL identifier to read all types of Excel files. The EXCEL identifier uses different technology to read files than do the XLS and XLSX identifiers, so the results may be different. By default, the XLS and XLSX identifiers look at more data rows to determine the column type than does the EXCEL identifier. Not all of these identifiers may work for you if your Windows computer has a mixture of 64-bit and 32-bit applications. In addition, some computer configurations may require that a PC Files Server be installed. The PC Files Server uses the EXCELCS identifier. See the SAS Documentation for more information.

Optional statements If you have more than one sheet in your file, then you can specify which sheet to read using the following statement:

```
SHEET = 'sheet-name';
```

If you want to read only specific cells in the sheet, you can specify a range. The range can be a named range (if defined), or you can specify the upper-left (UL) and lower-

right (LR) cells for the range as follows:

```
RANGE = 'sheet-name$UL:LR';
```

By default, the IMPORT procedure will take the variable names from the first row of the spreadsheet. If you do not want this, then you can add the following statement to the procedure and SAS will give your variables names like A, B, C, and so on.

```
GETNAMES = NO;
```

When using the EXCEL identifier, if you have a column that contains both numeric and character values, then by default, the numbers will be converted to missing values. To read the numbers as character values instead of converting them to missing values, use the following statement:

```
MIXED = YES;
```

Example Suppose you have the following Microsoft Excel spreadsheet which contains data about magnolia trees. For each type of tree the file includes the scientific and common names, maximum height, age at first blooming when planted from seed, whether evergreen or deciduous, and color of flowers.

	A	B	C	D	E	F
1	ScientificName	CommonName	MaxHeight	AgeBloom	Type	Color
2	M. grandiflora	Southern Magnolia	80	15	E	white
3	M. campbellii		80	20	D	rose
4	M. liliiflora	Lily Magnolia	12	4	D	purple
5	M. soulangiana	Saucer Magnolia	25	3	D	pink
6	M. stellata	Star Magnolia	10	3	D	white

The following program reads the Microsoft Excel file using the IMPORT procedure with the XLSX DBMS identifier and the REPLACE option, and creates a permanent data set named MAGNOLIA.

```
LIBNAME sasfiles 'c:\MySASLib';
```

```

* Read an Excel spreadsheet using PROC IMPORT;
PROC IMPORT DATAFILE = 'c:\MyExcel\Trees.xlsx'
OUT = sasfiles.magnolia
DBMS = XLSX REPLACE;
RUN;

```

Here is the MAGNOLIA data set. You can view the data in the Output Data tab in SAS Enterprise Guide or SAS Studio, or in the Viewtable window in the SAS windowing environment.

	ScientificName	CommonName	MaxHeight	AgeBlock
1	M. grandiflora	Southern Magnolia	80	1
2	M. campbellii		80	2
3	M. liliiflora	Lily Magnolia	12	
4	M. soulangiana	Saucer Magnolia	25	
5	M. stellata	Star Magnolia	10	

Notice that the variable names were taken from the first row in the spreadsheet. If you have spaces or special characters in the first row, see Section 3.18.

You should always check the SAS log when you create data sets to make sure nothing went wrong. These notes tell you that the MAGNOLIA data set has five observations and six variables.

NOTE: The import data set has 5 observations and 6 variables.

NOTE: SASFILES.MAGNOLIA was successfully created.

2.5 Accessing Excel Files Using the XLSX LIBNAME Engine

The preceding section covered one way to convert an Excel file into a SAS data set, the IMPORT procedure. The XLSX LIBNAME engine goes a step further. With the XLSX LIBNAME engine, you can read from and write to an Excel file without converting it to a SAS data set. This engine works for files created by any version of Microsoft Excel 2007 or later on the Windows or UNIX operating environments. You must have SAS 9.4M2 or higher and SAS/ACCESS Interface to PC Files software. With this engine, you can read an existing worksheet, replace a worksheet, or add a new worksheet, but you cannot update individual values inside a worksheet.

The XLSX LIBNAME engine uses the first line in your data file for the variable names, scans each full column to determine the variable type (character or numeric), assigns lengths to character variables, and recognizes dates, and numeric values containing commas or dollar signs. While the XLSX LIBNAME engine does not offer many options, because you are using an Excel worksheet like a SAS data set, you can use many standard data set options. For example, you can use the RENAME= data set option to change the names of variables. (See Section 6.10.)

Here is the general form of a LIBNAME statement with the XLSX engine option:

`LIBNAME libref XLSX 'filename';`

Notice that the only difference between this LIBNAME statement and one used to access a SAS data library (Section 2.2) is that the XLSX option has been added. The libref is a name you make up and must follow the rules for

names of SAS data libraries (eight characters or fewer, start with a letter or underscore; and contain only letters, numerals or underscores). The filename is the name used by your operating environment.

Keep in mind that, unlike some other LIBNAME engines, the XLSX LIBNAME engine only locks your Excel file while it actively accesses it. The file quickly becomes available to other people and processes that can read and edit it. You do not need to release the file, but if you know you are done with it, it is a good practice to clear the libref with this statement:

```
LIBNAME libref CLEAR;
```

Example This example uses the same Excel spreadsheet as the previous section. The spreadsheet contains data about magnolia trees: the scientific and common names, maximum height, age at first blooming when planted from seed, whether evergreen or deciduous, and color of flowers.

	A	B	C	D	E	F
1	ScientificName	CommonName	MaxHeight	AgeBloom	Type	Color
2	M. grandiflora	Southern Magnolia	80	15	E	white
3	M. campbellii		80	20	D	rose
4	M. liliiflora	Lily Magnolia	12	4	D	purple
5	M. soulangiana	Saucer Magnolia	25	3	D	pink
6	M. stellata	Star Magnolia	10	3	D	white
7						

With the XLSX LIBNAME engine, SAS can read the spreadsheet directly, without first converting it to a SAS data set. Here is a PROC PRINT that prints the data from the Excel spreadsheet. A simple PROC PRINT like this one prints the values for all variables and all observations.

```
* Read an Excel spreadsheet using the XLSX  
LIBNAME engine;
```

```
LIBNAME exfiles XLSX 'c:\MyExcel\Trees.xlsx';
```

```
PROC PRINT DATA = exfiles.sheet1;
```

```
TITLE 'PROC PRINT of Excel File';
```

```
RUN;
```

Here are the results of the PROC PRINT. Notice that the variable names were taken from the first row in the spreadsheet. If you have spaces or special characters in the first row, see Section 3.18.

PROC PRINT of Excel File

Obs	ScientificName	CommonName	MaxHeight	AgeBlock
1	M. grandiflora	Southern Magnolia	80	
2	M. campbellii		80	
3	M. liliiflora	Lily Magnolia	12	
4	M. soulangiana	Saucer Magnolia	25	
5	M. stellata	Star Magnolia	10	

If you want to convert an Excel spreadsheet to a SAS data set, you can do that too. Here is a DATA step that reads the Excel spreadsheet (using a SET statement in a DATA step, which is covered in more detail in Section 3.1) and creates a permanent SAS data set named MAGNOLIA.

```
* Read Excel into a permanent SAS data set;
```

```
LIBNAME exfiles XLSX 'c:\MyExcel\Trees.xlsx';
```

```
LIBNAME sasfiles 'c:\MySASLib';
```

```
DATA sasfiles.magnolia;
```

```
SET exfiles.sheet1;
```

```
RUN;
```

Here is the MAGNOLIA data set:

	ScientificName	CommonName	MaxHeight	AgeBlock
1	M. grandiflora	Southern Magnolia	80	1
2	M. campbellii		80	2
3	M. liliiflora	Lily Magnolia	12	
4	M. soulangiana	Saucer Magnolia	25	
5	M. stellata	Star Magnolia	10	

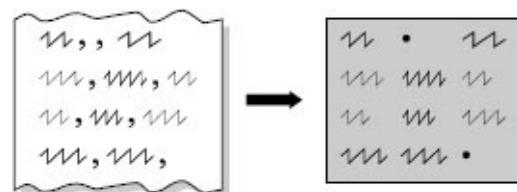
The SAS log shows that no observations or variables were lost:

NOTE: The import data set has 5 observations and 6 variables.

NOTE: There were 5 observations read from the data set EXFILES.sheet1.

NOTE: The data set SASFILES.MAGNOLIA has 5 observations and 6 variables.

2.6 Reading Delimited Files with the IMPORT Procedure



The IMPORT procedure reads Excel files (Section 2.4), but also reads many other types of files. Delimited files are raw data files that have a special character separating data values. Many types of software can save data as delimited files, often with commas or tab characters for delimiters. The IMPORT procedure for reading delimited files is part of Base SAS in the UNIX and Windows operating environments. You can also read delimited files with a DATA step (Section 2.18).

PROC IMPORT will scan your data file (the first 20 rows by default) and automatically determine the variable types (character or numeric), will assign lengths to the character variables, and can recognize some date formats. PROC IMPORT will treat two consecutive delimiters in your data file as a missing value, will read values enclosed in quotation marks, and assign missing values to variables when it runs out of data on a line. Also, if you want, you can use the first line in your data file for the variable names.

Here is the general form of the IMPORT procedure for delimited files:

```
PROC IMPORT DATAFILE = 'filename' OUT = data-set;
```

where *filename* is the file you want to read and *data-set* is the name of the SAS data set you want to create. SAS will determine the file type based on the extension of the file.

Type of File	Extension	Default Identifier
Comma-delimited	.csv	CS
Tab-delimited	.txt	TA

Delimiters other than commas or tabs

DI

If your file does not have the proper extension, or your file is of type DLM, then you must use the DBMS= option in the PROC IMPORT statement. Use the REPLACE option if you already have a SAS data set with the name you specified in the OUT= option, and you want to overwrite it. Here is the general form of PROC IMPORT with both the REPLACE and the DBMS options:

```
PROC IMPORT DATAFILE = 'filename' OUT = data-set
               DBMS = identifier REPLACE;
```

Optional statements Some types of files need a few more instructions to be read correctly. If the data do not start in the first line of the file, use the DATAROWS statement. If the delimiter is not a comma, tab, or space, use the DELIMITER statement. If your file contains only data and no headings, use the GETNAMES=NO statement to assign default variable names. Lastly, if your data file has all missing values or non-representative data in the first 20 data rows, you may need the GUESSINGROWS statement to make sure variables are assigned the correct data type and length.

DATAROWS = *n*; starts reading data in row *n*. Default is 1.

DELIMITER = 'character'; specifies delimiter for DLM files. Default is space.

GETNAMES = NO; tells SAS whether to use the first line of the input file for variable names. Default is YES. If NO, then variables are named VAR1, VAR2, VAR3, and so on.

GUESSINGROWS = n ; uses n rows to determine variable types. Default is 20.

Here is the general form of PROC IMPORT with the GETNAMES=NO statement:

```
PROC IMPORT DATAFILE = 'filename' OUT = data-set
    REPLACE;
    GETNAMES = NO;
```

Example The following example uses data about Jerry's Coffee Shop where Jerry employs local bands to attract customers. Jerry keeps records of the number of customers present for each band. The data are the band name, and the number of customers at 8 p.m., 9 p.m., 10 p.m., and 11 p.m. Notice that one of the bands, "Stop, Drop, and Rock-N-Roll," has commas in the name of the band. When a data value contains the delimiter, then the value must be enclosed in quotation marks.

```
BandName,EightPM,NinePM,TenPM,ElevenPM
Lupine Lights,45,63,70,
Awesome Octaves,17,28,44,12
"Stop, Drop, and Rock-N-Roll",34,62,77,91
The Silveyville Jazz Quartet,38,30,42,43
Catalina Converts,56,,65,34
```

Here is the program that will read this data file:

```
PROC IMPORT DATAFILE
    = 'c:\MyRawData\Bands2.csv' OUT = music REPLACE;
    RUN;
```

Here is the MUSIC data set created by the program. You can view the data in the Output Data tab in SAS Enterprise Guide or SAS Studio, or in the Viewtable window in the SAS windowing environment. Notice that PROC IMPORT used the first row of data for variable names. If you have spaces or special characters in the first row, see Section 3.18.

	BandName	EightPM	NinePM	TenPM
1	Lupine Lights	45	63	
2	Awesome Octaves	17	28	
3	Stop, Drop, and Rock-N-Roll	34	62	
4	The Silveyville Jazz Quartet	38	30	
5	Catalina Converts	56	.	

The SAS log contains these notes describing the new data set:

NOTE: 5 records were read from the infile

'c:\MyRawData\Bands2.csv'.

The minimum record length was 23.

The maximum record length was 41.

NOTE: The data set WORK.MUSIC has 5 observations and
5 variables.

2.7 Telling SAS Where to Find Your Raw Data

The rest of this chapter covers reading raw data using the DATA step. If your data are already stored as SAS data sets, as Excel spreadsheets, as CSV or tab-delimited files, or in a database such as Oracle, then you can probably skip this topic. However, if your data are in raw data files (also referred to as text, ASCII, sequential, or flat files), then you can use the power and flexibility of the DATA step to read your data.

A raw data file can be viewed using simple text editors or system commands. On personal computers, raw data files will either have no program associated with them, or they will be associated with simple editors like Microsoft Notepad. In some operating environments, you can use commands to list the file, such as the cat or more commands in UNIX. Spreadsheet files are examples of data files that are not raw data. If you try using a text editor to look at a spreadsheet file, you will probably see lots of funny special characters you can't find on your keyboard. It may cause your computer to beep and chirp, making you wish you had that private office down the hall. It looks nothing like the nice neat rows and columns you see when you use your spreadsheet software to view the same file.

The first step toward reading raw data files is telling SAS where to find the raw data. Your raw data may be either internal to your SAS program (also called instream), or in a separate file. Either way, you must tell SAS where to find your data.

Internal raw data If you type raw data directly into your SAS program, then the data are internal to your program. You may want to do this when you have small amounts of data, or when you are testing a program with a small test data set. Use the DATALINES statement to indicate internal data. The DATALINES statement must be the last statement in the DATA step. All lines in the SAS program following the DATALINES statement are considered data until SAS encounters a semicolon. The

semicolon can be on a line by itself or at the end of a SAS statement that follows the data lines. Any statements following the data are part of a new step. The CARDS statement is synonymous with the DATALINES statement.

The following SAS program illustrates the use of the DATALINES statement. (The DATA statement simply tells SAS to create a SAS data set named USPRESIDENTS, and the INPUT statement tells SAS how to read the data. The INPUT statement is discussed in Sections 2.8 through 2.18.)

```
* Read internal data into SAS data set uspresidents;  
DATA uspresidents;  
    INPUT President $ Party $ Number;  
    DATALINES;  
    Adams F 2  
    Lincoln R 16  
    Grant R 18  
    Kennedy D 35  
    ;  
    RUN;
```

If you run this program, the following note appears in the SAS log telling you that the data set USPRESIDENTS in the WORK library contains four observations and three variables.

NOTE: The data set WORK.USPRESIDENTS has 4 observations and 3 variables.

External raw data files Usually you will want to keep data in external files, separating the data from the program. This eliminates the chance that you might accidentally alter the data when you are editing your SAS program. Use the INFILE statement to tell SAS the filename (and path, if appropriate) of the file containing the data. The INFILE statement follows the DATA statement and must precede the INPUT statement. After the INFILE keyword, enclose

the file path and name in quotation marks. Here is the general form for different operating environments. (If you are using SAS University Edition or SAS OnDemand for Academics, then use the Linux style of statement.)

Windows: INFILE 'c:\MyDir\President.dat';

UNIX or Linux: INFILE '/home/mydir/president.dat';

z/OS: INFILE 'MYID.PRESIDEN.DAT';

Suppose the following data are in a file called President.dat in the directory MyRawData on the C drive (Windows):

Adams F 2
Lincoln R 16
Grant R 18
Kennedy D 35

You could use this program with an INFILE statement to read the external data file:

```
* Read data from external file into SAS data set;  
DATA uspresidents;  
  INFILE 'c:\MyRawData\President.dat';  
  INPUT President $ Party $ Number;  
  RUN;
```

Whenever you read data from an external file, SAS lists some valuable information about the file in the SAS log. Always check this information as it could indicate problems. The following is an excerpt from the SAS log after reading the external file. A simple comparison of the number of records read by the INFILE statement (four in this case) with the number of observations (also four) in the SAS data set can tell you a lot about whether SAS read your data correctly.

NOTE: The infile 'c:\MyRawData\President.dat' is:

Filename=c:\MyRawData\President.dat,
RECFM=V,LRECL=32767, File Size (bytes)=49

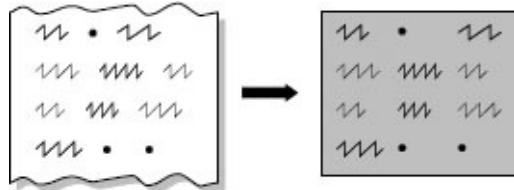
NOTE: 4 records were read from the infile 'c:\MyRawData\President.dat'.

The minimum record length was 9.

The maximum record length was 12.

NOTE: The data set WORK.USPRESIDENTS has 4 observations
and 3 variables.

2.8 Reading Raw Data Separated by Spaces



If the values in your raw data file are all separated by at least one space, then using list input (also called free formatted or space-delimited input) to read the data may be appropriate. List input is an easy way to read raw data into SAS, but with ease come a few limitations. By default, you must read all the data in a record—no skipping over unwanted values. Any missing data must be indicated with a period. Character data, if present, must be simple—no embedded spaces, and no values greater than 8 characters in length. If the data file contains dates or other values that need special treatment, then list input would not be appropriate. This may sound like a lot of restrictions, but a surprising number of data files can be read using list input.

The INPUT statement, which is part of the DATA step, tells SAS how to read your raw data. To write an INPUT statement using list input, simply list the variable names after the keyword INPUT in the order they appear in the data file. Generally, variable names must be 32 characters or fewer, start with a letter or an underscore, and contain only letters, underscores, or numerals. If the values are

character (not numeric), then place a dollar sign (\$) after the variable name. Leave at least one space between names, and remember to place a semicolon at the end of the statement. The following is an example of a simple list-style INPUT statement:

```
INPUT Name $ Age Height;
```

This statement tells SAS to read three data values. The \$ after Name indicates that it is a character variable, whereas the variables Age and Height are both numeric.

If your data file does not quite satisfy the rules for list input, you will be glad to know that there are various ways to work around the limitations. Many of those are discussed later in this chapter. One of the most common problems is character variables with lengths longer than 8. You can tell SAS to use a longer length with a LENGTH statement. This statement, for example, tells SAS to create a variable called Name that is character and has a length of 12:

```
LENGTH Name $ 12;
```

In SAS programs, the attributes of a variable are set when SAS first encounters that variable. So if a LENGTH statement comes before an INPUT statement, then SAS will use that length.

Example Your hometown has been overrun with toads this year. A local resident, having heard of frog jumping in California, had the idea of organizing a toad jump to cap off the annual town fair. For each contestant you have the toad's name, weight, and the jump distance from three separate attempts. If the toad is disqualified for any jump, then a period is used to indicate missing data. Here is what the data file ToadJump.dat looks like:

```
Lucky 2.3 1.9 . 3.0  
Spot 4.6 2.5 3.1 .5  
Toadzilla 7.1 .. 3.8  
Hop 4.5 3.2 1.9 2.6
```

Noisy 3.8 1.3 1.8 1.5

Winner 5.7 ...

This data file does not look very neat, but it does meet most of the requirements for list input: no embedded spaces, all values are separated by at least one space, and missing data are indicated by a period. However, there is one problem. The name of one toad, Toadzilla, is longer than 8 characters. Adding a LENGTH statement to the program will allow SAS to read the toads' names with list input.

Here is the SAS program that will read the data:

```
* Create a SAS data set named toads;  
* Read the data file ToadJump.dat using list input;  
DATA toads;  
    LENGTH ToadName $ 9;  
    INFILE 'c:\MyRawData\ToadJump.dat';  
    INPUT ToadName Weight Jump1 Jump2 Jump3;  
RUN;
```

The LENGTH statement creates a variable named ToadName that is character (indicated by the dollar sign) and has a length of 9. Then the INPUT statement reads the variables ToadName, Weight, Jump1, Jump2, and Jump3. Because SAS already knows that ToadName is character, you don't need to include a dollar sign in the INPUT statement. All the other variables are numeric.

It is always important to check data sets you create to make sure they are correct. Here is the TOADS data set. You can view the data in the Output Data tab in SAS Enterprise Guide or SAS Studio, or in the Viewtable window in the SAS windowing environment.

	ToadName	Weight	Jump1	Jump2
1	Lucky	2.3	1.9	

2	Spot	4.6	2.5	E
3	Toadzilla	7.1	.	.
4	Hop	4.5	3.2	I
5	Noisy	3.8	1.3	I
6	Winner	5.7	.	.

These notes appear in the log:

NOTE: 6 records were read from the infile
'c:\MyRawData\ToadJump.dat'.

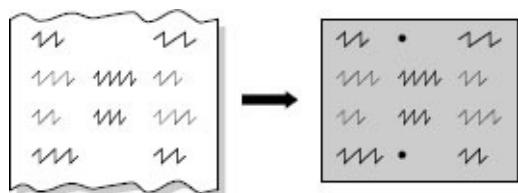
The minimum record length was 16.

The maximum record length was 21.

NOTE: The data set WORK.TOAD\$ has 6 observations and 5 variables.

You can use list input to read files with other delimiters too, such as commas or tabs. See Section 2.18.

2.9 Reading Raw Data Arranged in Columns



Some raw data files do not have spaces (or other delimiters) between all the values or periods for missing data—so the files can't be read using list input. But if each of the variable's values is always found in the same place in the

line of data, then you can use column input as long as all the values are character or standard numeric. Standard numeric data contain only numerals, decimal points, plus and minus signs, and E for scientific notation. Numbers with embedded commas or dates, for example, are not standard.

Column input has the following advantages over list input:

- ◆ spaces are not required between values
- ◆ missing values can be left blank
- ◆ character data can have embedded spaces
- ◆ you can skip unwanted variables

Data files with street addresses, which often have embedded blanks, are good candidates for column input. The street Martin Luther King Jr. Boulevard should be read as one variable not five, as it would be with list input. Data which can be read with column input can often also be read with formatted input (discussed in Sections 2.10 and 2.11) or a combination of input styles (Section 2.12).

To write an INPUT statement using column input, after the keyword INPUT, list the first variable's name. If the variable is character, leave a space; then place a \$. After the \$, or variable name if it is numeric, leave a space; then list the column or range of columns for that variable. The columns are positions of the characters or numbers in the data line and are not to be confused with columns like those you see in a spreadsheet. Repeat this for all the variables you want to read. Here is a simple column-style INPUT statement:

```
INPUT Name $ 1-10 Age 11-13 Height 14-18;
```

The first variable, Name, is character, and the data values are in columns 1 through 10. The second variable, Age, is numeric since it is not followed by a \$ and is in columns 11 through 13. The third variable, Height, is also numeric and

is in columns 14 through 18.

Example The local minor league baseball team, the Walla Walla Sweets, is keeping records about concession sales. A ballpark favorite are the sweet onion rings which are sold at the concession stands and also by vendors in the bleachers. The ballpark owners have a feeling that in games with lots of hits and runs more onion rings are sold in the bleachers than at the concession stands. They think they should send more vendors out into the bleachers when the game heats up, but need more evidence to back up their feelings.

For each home game they have the following information: name of opposing team, number of onion ring sales at the concession stands and in the bleachers, the number of hits for each team, and the final score for each team. The following is a sample of the data file named OnionRing.dat. For reference, a ruler showing the column numbers has been placed above the data:

-----	1	-----	2	-----	3	-----	4
Columbia Peaches	35	67	1	10	2	1	
Plains Peanuts	210	2	5	0	2		
Gilroy Garlics	151035	12	11	7	6		
Sacramento Tomatoes	124	85	15	4	9	1	

Notice that the data file has the following characteristics, all making it a prime candidate for column input. All the values line up in columns, the team names have embedded blanks, missing values are blank, and in one case there is no space between data values. (Those Gilroy Garlics fans must really love onion rings.)

The following program shows how to read these data using column input:

```
* Create a SAS data set named sales;  
* Read the data file OnionRing.dat using column input;  
DATA sales;  
  INFILE 'c:\MyRawData\OnionRing.dat';
```

```

INPUT VisitingTeam $ 1-20 CSales 21-24
BSales 25-28
OurHits 29-31 TheirHits 32-34
OurRuns 35-37 TheirRuns 38-40;
RUN;

```

The variable VisitingTeam is character (indicated by a \$) and reads the visiting team's name in columns 1 through 20. The variables CSales and BSales read the concession and bleacher sales in columns 21 through 24 and 25 through 28, respectively. The number of hits for the home team, OurHits, and the visiting team, TheirHits, are in columns 29 through 31 and 32 through 34, respectively. The number of runs for the home team, OurRuns, is in columns 35 through 37, while the number of runs for the visiting team, TheirRuns, is in columns 38 through 40.

Here is the SALES data set. You can view the data in the Output Data tab in SAS Enterprise Guide or SAS Studio, or in the Viewtable window in the SAS windowing environment.

	VisitingTeam	CSale s	BSale s	OurHi ts	TheirHi ts	Ou
1	Columbia Peaches	35	67	1	10	
2	Plains Peanuts	210	.	2	5	
3	Gilroy Garlics	15	1035	12	11	
4	Sacramento Tomatoes	124	85	15	4	

These notes appear in the log:

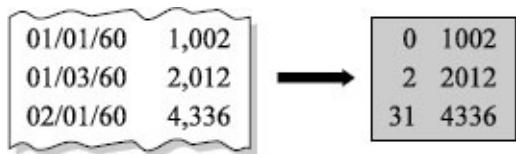
NOTE: 4 records were read from the infile
'c:\MyRawData\OnionRings.dat'.

The minimum record length was 40.

The maximum record length was 40.

NOTE: The data set WORK.SALES has 4 observations and 7 variables.

2.10 Reading Raw Data Not in Standard Format



Sometimes raw data are not straightforward numeric or character. For example, we humans easily read the number 1,000,001 as one million and one, but your trusty computer sees it as a character string. While the embedded commas make the number easier for us to interpret, they make the number impossible for the computer to recognize without some instructions. In SAS, informats are used to tell the computer how to interpret these types of data.

Informats are useful anytime you have nonstandard data. (Standard numeric data contain only numerals, decimal points, plus and minus signs, and E for scientific notation.) Numbers with embedded commas or dollar signs are nonstandard data. Other examples include data in hexadecimal or packed decimal formats. SAS has informats for reading these types of data as well.

Dates are perhaps the most common nonstandard data. Using date informats, SAS will convert conventional forms of dates like 10-31-2023 or 31OCT23 into a number, the number of days since January 1, 1960. This number is referred to as a SAS date value. (Why January 1, 1960? Who knows? Maybe 1960 was a good year for the SAS founders.) This turns out to be extremely useful when you

want to do calculations with dates. For example, you can easily find the number of days between two dates by subtracting one from the other. See Section 3.13 for more about dates.

There are three general types of informats: character, numeric, and date. A table of selected SAS informats appears in the next section. The three types of informats have these general forms:

Character	Numeric	D
$\$informatw.$	$informatw.d$	$informat$

The \$ indicates character informats, *informat* is the name of the informat, *w* is the total width, and *d* is the number of decimal places (numeric informats only). The period is an important part of the informat name. Without a period, SAS may try to interpret the informat as a variable name, which by default, cannot contain any special characters except the underscore. Two informats do not have names: $\$w.$, which reads standard character data, and $w.d$, which reads standard numeric data.

Use informats by placing the informat after the variable name in the INPUT statement; this is called formatted input. Here is an INPUT statement using formatted input:

```
INPUT Name $10. Age 3. Height 5.1 BirthDate  
      MMDDYY10.;
```

The columns read for each variable are determined by the starting point and the width of the informat. SAS always starts with the first column; so the data values for the first variable, Name, which has an informat of \$10., are in columns 1 through 10. Now the starting point for the second variable is column 11, and SAS reads values for

Age in columns 11 through 13. The values for the third variable, Height, are in columns 14 through 18. Those five columns include the decimal place and the decimal point itself (150.3 for example). The values for the last variable, BirthDate, are in columns 19 through 28.

Example This example illustrates the use of informats for reading data. The data contain the results from a pumpkin-carving contest. Each line includes the contestant's name, age, type (carved or decorated), the date the pumpkin was entered, and the scores from each of three judges.

Alicia Grossman	13	c	10-28-2020	7.8	6.5	7.2
Matthew Lee	9	D	10-30-2020	6.5	5.9	6.8
Elizabeth Garcia	10	C	10-29-2020	8.9	7.9	8.5
Lori Newcombe	6	D	10-30-2020	6.7		4.9
Jose Martinez	7	d	10-31-2020	8.9	9.5	10.0
Brian Williams	11	C	10-29-2020	7.8	8.4	8.5

The following program reads the data from a file named Pumpkin.dat, and uses a LIBNAME statement to create a permanent SAS data set named CONTEST. Please note there are many ways to input these data, so if you imagined something else, that's OK.

```
LIBNAME pump 'c:\MySASLib';
* Create a permanent SAS data set named contest;
* Read the file Pumpkin.dat using formatted input;
DATA pump.contest;
  INFILE 'c:\MyRawData\Pumpkin.dat';
    INPUT Name $16. Age 3. +1 Type $1. +1 Date
          MMDDYY10.
          (Score1 Score2 Score3) (4.1);
RUN;
```

The variable Name has an informat of \$16., meaning that it is a character variable 16 columns wide. Variable Age has an informat of 3., is numeric, three columns wide, and has no decimal places. The +1 skips over one column. Variable Type is character, and it is one column wide. Variable Date

has an informat MMDDYY10. and reads dates in the form 10-31-2020 or 10/31/2020, each 10 columns wide. The remaining variables, Score1 through Score3, all require the same informat, 4.1. Putting the variables and the informat in separate sets of parentheses saves typing.

Here is the CONTEST data set. You can view the data in the Output Data tab in SAS Enterprise Guide or SAS Studio, or in the Viewtable window in the SAS windowing environment.

	Name	Age	Type	Date	Score1	Sc
1	Alicia Grossman	13	c	22216	7.8	
2	Matthew Lee	9	D	22218	6.5	
3	Elizabeth Garcia	10	C	22217	8.9	
4	Lori Newcombe	6	D	22218	6.7	
5	Jose Martinez	7	d	22219	8.9	
6	Brian Williams	11	C	22217	7.8	

Notice that these dates are printed as the number of days since January 1, 1960. Section 3.13 discusses how to format these into readable dates.

These notes appear in the log:

NOTE: 6 records were read from the infile
'c:\MyRawData\Pumpkin.dat'.

The minimum record length was 44.

The maximum record length was 44.

NOTE: The data set PUMP.CONTEST has 6 observations and 7 variables.

2.11 Selected Informats

Informat	Definition	Width range
Character		
\$CHARw.	Reads character data—does not trim leading or trailing blanks	1–32,767
\$UPCASEw. .	Converts character data to uppercase	1–32,767
\$w.	Reads character data—trims leading blanks	1–32,767
Date, Time, and Datetime¹		
ANYDTD TEw.	Reads dates in various date forms	5–32
DATEw.	Reads dates in form: <i>ddmmmyy</i> or <i>ddmmmyyyy</i>	7–32

DATETIME w.	Reads datetime values in the form: <i>ddmmmyy hh:mm:ss.ss</i>	13–40
DDMMYY w. .	Reads dates in form: <i>ddmmyy</i> or <i>ddmmyyyy</i>	6–32
JULIAN w.	Reads Julian dates in form: <i>yyddd</i> or <i>yyyyddd</i>	5–32
MMDDYY w. .	Reads dates in form: <i>mmdyy</i> or <i>mmdyyyy</i>	6–32
STIMER w. .	Reads time in form: <i>hh:mm:ss.ss</i> (or <i>mm:ss.ss</i> , or <i>ss.ss</i>)	1–32
TIME w. <i>d</i>	Reads time in form: <i>hh:mm:ss.ss</i> (or <i>hh:mm</i>) using a 24-hour clock	5–32

Numeric

COMMA w. .d	Removes embedded commas and \$, converts left parentheses to minus sign	1–32
COMMAX w.d	Like COMMAw.d but reverses role of comma and period	1–32
PERCENT w.	Converts percentages to proportions	1–32
w.d	Reads standard numeric data	1–32

¹ SAS date values are the number of days since January 1, 1960.
 Time values are the number of seconds past midnight, and
 datetime values are the number of seconds past midnight January
 1, 1960.

Informat	Input data	INPUT statement	Result
Character			
\$CHARw.	my cat my cat	INPUT Animal \$CHAR10.;	my cat
\$UPCASEw .	my cat	INPUT Name \$UPCASE10.;	MY
\$w.	my cat my cat	INPUT Animal \$10.;	my cat
Date, Time, and Datetime			
ANYDTD TEw.	1jan1961 01/01/61	INPUT Day ANYDTDTE10.;	366
DATEw.	1jan1961 1 jan	INPUT Day DATE10.;	366

	61		
DATETIME w.	1jan1960 10:30:15 1jan1961,10:30:15	INPUT Dt DATETIME18.;	378 316
DDMMYY w.	01.01.61 02/01/61	INPUT Day DDMMYY8.;	366
JULIANw.	61001 1961001	INPUT Day JULIAN7.;	366
MMDDYY w.	01-01-61 01/01/61	INPUT Day MMDDYY8.;	366
STIMERw.	10:30 10:30:15	INPUT Time STIMER8.;	630
TIMEw. d	10:30 10:30:15	INPUT Time TIME8.;	378

Numeric

COMMAw .d	\$1,000,001 (1,234)	INPUT Income COMMA10.;	10 -12
COMMAXw .d	\$1.000.001 (1.234,25)	INPUT Value COMMAX10.;	10 -12
PERCENTw .d	5% (20%)	INPUT Value PERCENT5.;	0.0

w.d	1234 -12.3	INPUT Value 5.1;	123
-----	------------	------------------	-----

2.12 Mixing Input Styles

Each of the three major input styles has its own advantages. List style is the easiest; column style is a bit more work; and formatted style is the hardest of the three. However, column and formatted styles do not require spaces (or other delimiters) between variables and can read embedded blanks. Formatted style can read special data such as dates. Sometimes you use one style, sometimes another, and sometimes the easiest way is to use a combination of styles. SAS is so flexible that you can mix and match any of the input styles for your own convenience.

Example The following raw data contain information about U.S. national parks: name, state (or states as the case may be), year established, and size in acres:

Yellowstone	ID/MT/WY	1872	4,065,493
Everglades	FL	1934	1,398,800
Yosemite	CA	1864	760,917
Great Smoky Mountains	NC/TN	1926	520,269
Wolf Trap Farm	VA	1966	130

You could write the INPUT statement for these data in many ways, which is the point of this section. This program shows one way to do it:

```
* Create a SAS data set named nationalparks;
* Read a data file NatPark.dat mixing input styles;
DATA nationalparks;
  INFILE 'c:\MyRawData\NatPark.dat';
  INPUT ParkName $ 1-22 State $ Year @40 Acreage
    COMMA9.;
```

RUN;

Notice that the variable ParkName is read with column style input, State and Year are read with list input, and Acreage is read with formatted input.

Here is the NATIONALPARKS data set:

	ParkName	State	Yea
1	Yellowstone	ID/MT/WY	1872
2	Everglades	FL	1908
3	Yosemite	CA	1890
4	Great Smoky Mountains	NC/TN	1916
5	Wolf Trap Farm	VA	1962

Sometimes programmers run into problems when they mix input styles. When SAS reads a line of raw data, it uses a pointer to mark its place, but each style of input uses the pointer a little differently. With list input, SAS automatically scans to the next non-blank field and starts reading. With column style input, SAS starts reading in the exact column you specify. But with formatted input, SAS just starts reading—wherever the pointer is, that is where SAS reads.

Sometimes you need to move the pointer explicitly, and you can do that by using the column pointer, $@n$, where n is the number of the column SAS should move to.

In the preceding program, the column pointer @40 tells SAS to move to column 40 before reading the value for Acreage. If you removed the column pointer from the INPUT statement, as shown in the following statement, then SAS would start reading Acreage right after Year:

```
INPUT ParkName $ 1-22 State $ Year Acreage
      COMMA9.;
```

Here is the data set with truncated values of the variable Acreage:

	ParkName	State	Yea
1	Yellowstone	ID/MT/WY	18
2	Everglades	FL	19
3	Yosemite	CA	18
4	Great Smoky Mountains	NC/TN	19
5	Wolf Trap Farm	VA	19

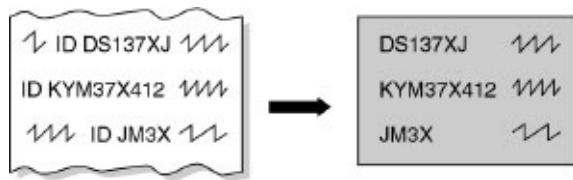
Because Acreage was read with formatted input, SAS started reading right where the pointer was. Here is the data file with a column ruler for counting columns at the top and asterisks marking the place where SAS started reading the values of Acreage:

```
-----1-----2-----3-----4-----5
Yellowstone      ID/MT/WY 1872 * 4,065,493
Everglades       FL 1934 *    1,398,800
Yosemite        CA 1864 *    760,917
Great Smoky Mountains NC/TN 1926 *    520,269
```

The COMMA9. informat told SAS to read nine columns, and SAS did that even when those columns were completely blank.

The column pointer, $@n$, has other uses, too, and can be used anytime you want SAS to skip backward or forward within a line of data. You could use it, for example, to skip over unneeded data, or to read a variable twice using different informats.

2.13 Reading Messy Raw Data



Sometimes you need to read data that just don't line up in nice columns or have predictable lengths. When you have these types of messy files, ordinary list, column, or formatted input simply aren't enough. You need more tools in your bag: tools like the $@'character'$ column pointer, and the colon and ampersand modifiers.

The $@'character'$ column pointer In the preceding section we showed how you can use the $@$ column pointer to move to a particular column before reading data. However, sometimes you don't know the starting column of the data, but you do know that it always comes after a particular character or word. For these types of situations, you can use the $@'character'$ column pointer. For example, suppose you have a data file that has information about dog ownership. Nothing in the file lines up, but you know that the breed of the dog always follows the word Breed. You could read the dog's breed using the following INPUT statement:

```
INPUT @'Breed' DogBreed $;
```

The colon modifier The above INPUT statement will work just fine as long as the dog's breed is 8 characters or less (the default length for character variables) and contains no spaces. So if the dog is a Poodle you're fine, but if the dog is a Rottweiler or Shih Tzu, it won't work. If you assign the variable an informat such as \$20. in the INPUT statement, then SAS will read for 20 columns whether or not there is a space in those columns. So the DogBreed variable may include unwanted characters, which appear after the dog's breed on the data line. If you want SAS to read only until it encounters a space or the end of the data line, you can use a colon modifier on the informat. (This works with other delimiters besides spaces. See Section 2.18.) To use a colon modifier, simply put a colon (:) before the informat (such as :\$20. instead of \$20.). For example, given this line of raw data:

My dog Sam Breed Rottweiler Vet Bills \$478

the following table shows the results you would get using different INPUT statements:

Statement	Value of variable DogBreed
INPUT @'Breed ' DogBreed \$;	Rottweil
INPUT @'Breed ' DogBreed \$20.;	Rottweiler Vet Bill
INPUT @'Breed ' DogBreed :\$20.;	Rottweiler

The ampersand modifier The colon modifier tells SAS

to read a data value until it reaches a space. If you have embedded spaces in your data values, you may still be able to read your data. The ampersand modifier (&) tells SAS to read a data value until it reaches two or more spaces in a row. Insert the ampersand after the name of the variable with the embedded spaces.

```
INPUT @'Breed' DogBreed & $;
```

Example Each year engineering students from around the USA and Canada build concrete canoes and hold regional and national competitions. Part of the competition involves racing the canoes. The following data contain the final results of a men's sprint competition. The data lines start with the name of the canoe, followed by the school, and the time.

```
Bellatorum School CSULA Time 1:40.5
The Kraken School ASU Time 1:45.35
Black Widow School UoA Time 1:33.7
Koicrete School CSUF Time 1:40.25
Khaos School UNLV Time 2:03.45
Max School UCSD Time 1:26.47
Hakuna Matata School UCLA Time 1:20.64
Prospector School CPSLO Time 1:12.08
Andromeda School CPP Time 1:25.1
Kekoapohaku School UHM Time 1:24.49
```

You can see that some canoe names contain embedded spaces, and there are two spaces after each canoe name. Because the canoe names have different lengths, the school names do not line up in the same column. Also, the time values are sometimes six characters and sometimes seven. This SAS program reads the canoe name, school, and time:

```
DATA canoerelations;
INFILE 'c:\MyRawData\Canoes.dat';
INPUT CanoeName & $13. @'School' School $
```

```
@'Time' RaceTime :STIMER8.;  
RUN;
```

This INPUT statement uses both an ampersand and an informat (\$13.) to read the values of CanoeName. The result is that SAS will read CanoeName until it encounters two or more spaces and then stop even if the length of that CanoeName is less than the specified informat. Then the INPUT statement uses @'School' and @'Time' to position the column pointer to read the school name and time. The school names are all less than the default length of 8 characters so no informat is needed to read them, but times are non-standard data and require an informat. Because the time is not always the same number of characters, a colon modifier is added to the informat (:STIMER8.). Without the colon modifier, SAS would go to a new data line to try to read the time values when it ran out of characters on a line of data.

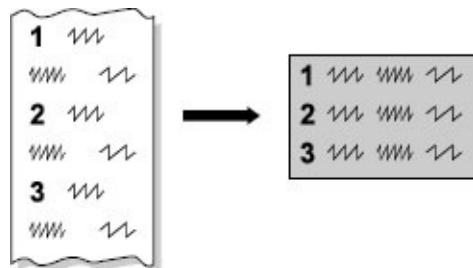
Here is the CANOERESULTS data set:

	CanoeName	School
1	Bellatorum	CSULA
2	The Kraken	ASU
3	Black Widow	UoA
4	Koicrete	CSUF
5	Khaos	UNLV
6	Max	UCSD

7	Hakuna Matata	UCLA
8	Prospector	CPSLO
9	Andromeda	CPP
10	Kekoapohaku	UHM

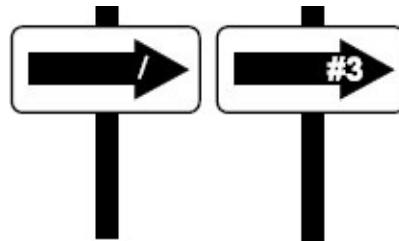
Note that the times in this output are printed in seconds.
See Section 3.14 for how to format these values into minutes and seconds.

2.14 Reading Multiple Lines of Raw Data per Observation



In a typical raw data file each line of data represents one observation, but sometimes the data for each observation are spread over more than one line. Since SAS will automatically go to the next line if it runs out of data before it has read all the variables in an INPUT statement, you could just let SAS take care of figuring out when to go to a new line. But if you know that your data file has multiple lines of raw data per observation, it is better for you to explicitly tell SAS when to go to the next line than to make SAS figure it out. That way you won't get a suspicious SAS-went-to-a-new-line note in your log (Section 11.4). To tell SAS when to skip to a new line, you simply add line pointers to your INPUT statement.

The line pointers, slash (/) and pound-*n* (#*n*), are like road signs telling SAS, “Go this way.” To read more than one line of raw data for a single observation, you simply insert a slash into your INPUT statement when you want to skip to the next line of raw data. The #*n* line pointer performs the same action except that you specify the line number. The *n* in #*n* stands for the number of the line of raw data for that observation; so #2 means to go to the second line for that observation, and #4 means go to the fourth line. You can even go backward using the #*n* line pointer, reading from line 4 and then from line 3, for example. The slash is simpler, but #*n* is more flexible.



Example A colleague is planning his next summer vacation, and he wants to go someplace where the weather is just right. He obtains data from a meteorological database. Unfortunately, he has not quite figured out how to export from this database and makes a rather odd file.

The file contains information about temperatures for the month of July for Alaska, Florida, and North Carolina. (If your colleague chooses the last state, maybe he can visit SAS headquarters.) The first line contains the city and state, the second line lists the normal high temperature and normal low (in degrees Fahrenheit), and the third line contains the record high and low:

```
Nome AK
55 44
88 29
Miami FL
```

90 75
97 65
Raleigh NC
88 68
105 50

The following program reads the weather data from a file named Temperature.dat:

```
* Create a SAS data set named highlow;  
* Read the data file using line pointers;  
DATA highlow;  
  INFILE 'c:\MyRawData\Temperature.dat';  
  INPUT City $ State $  
    / NormalHigh NormalLow  
    #3 RecordHigh RecordLow;  
RUN;
```

The INPUT statement reads the values for City and State from the first line of data. Then the slash tells SAS to move to column 1 of the next line of data before reading NormalHigh and NormalLow. Likewise, the #3 tells SAS to move to column 1 of the third line of data for that observation before reading RecordHigh and RecordLow. As usual, there is more than one way to write this INPUT statement. You could replace the slash with #2 or replace #3 with a slash.

Here is the HIGHLOW data set:

	City	State	NormalHigh	NormalLow	RecordHigh	
1	Nome	AK	55	44		8
2	Miami	FL	90	75		9
3	Raleigh	NC	88	68		10

These notes appear in the log:

NOTE: 9 records were read from the infile

'c:\MyRawData\Temperature.dat'.

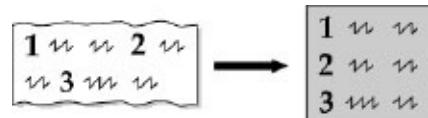
The minimum record length was 5.

The maximum record length was 10.

NOTE: The data set WORK.HIGHLOW has 3 observations and 6 variables.

Notice that while nine records were read from the INFILE statement, the SAS data set contains just three observations. Usually, this would set off alarms in your mind, but here it confirms that indeed three data lines were read for every observation just as planned. You should always check your log, particularly when using line pointers.

2.15 Reading Multiple Observations per Line of Raw Data



There ought to be a Murphy's law of data: whatever form data can take, it will. Normally SAS assumes that each line of raw data represents no more than one observation. When you have multiple observations per line of raw data, you can use **double trailing at signs (@@)** at the end of your **INPUT statement**. This line-hold specifier is like a stop sign telling SAS, "Stop, hold that line of raw data." SAS will hold that line of data, continuing to read observations until it either runs out of data or reaches an INPUT statement that does not end with a double trailing @.



Example Suppose you have a colleague who is planning a vacation and has obtained a file containing data about rainfall for the three cities he is considering. The file contains the name of each city, the state, average rainfall (in inches) for the month of July, and average number of days with measurable precipitation in July. The raw data look like this:

```
Nome AK 2.5 15 Miami FL 6.75  
18 Raleigh NC . 12
```

Notice that in this data file the first line stops in the middle of the second observation. The following program reads these data from a file named Precipitation.dat, and uses an @@ so SAS does not automatically go to a new line of raw data for each observation:

```
* Input more than one observation from each record;  
DATA rainfall;  
  INFILE 'c:\MyRawData\Precipitation.dat';  
    INPUT City $ State $ NormalRain MeanDaysRain  
    @@;  
  RUN;
```

Here is the RAINFALL data set:

	City	State	NormalRain	MeanDaysRain
1	Nome	AK		2.50

2	Miami	FL	6.75
3	Raleigh	NC	.

These notes will appear in the log:

NOTE: 2 records were read from the infile
'c:\MyRawData\Precipitation.dat'

The minimum record length was 18.

The maximum record length was 28.

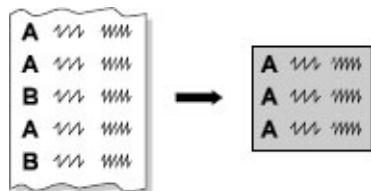
NOTE: SAS went to a new line when INPUT statement reached
past the end of a line.

NOTE: The data set WORK.RAINFALL has 3 observations and 4
variables.

While only two records were read from the raw data file,
the RAINFALL data set contains three observations. The
log also includes a note saying SAS went to a new line
when the INPUT statement reached past the end of a line.
This means that SAS came to the end of a line in the
middle of an observation and continued reading with the
next line of raw data. Normally these messages would
indicate a problem, but in this case they are exactly what
you want.

Section 3.11 shows a different way to produce multiple
observations from a single line of raw data, a technique that
works when some data values must be repeated.

2.16 Reading Part of a Raw Data File



At some time you may find that you need to read a small fraction of the records in a large data file. For example, you might be reading U.S. census data and want only female heads-of-household who have incomes above \$225,000 and live in Walla Walla, Washington. You could read all the records in the data file and then throw out the unneeded ones, but that would waste time.

Luckily, you don't have to read all the data before you tell SAS whether to keep an observation. Instead, you can read just enough variables to decide whether to keep the current observation, and then end the INPUT statement with an at sign (@), called a trailing at. This tells SAS to hold that line of raw data. While the trailing @ holds that line, you can test the observation with an IF statement to see if it's one you want to keep. If it is, then you can read data for the remaining variables with a second INPUT statement. Without the trailing @, SAS would automatically start reading the next line of raw data with each INPUT statement.

The trailing @ is similar to the column pointer, @n, introduced in Section 2.12. By specifying a number after the @ sign, you tell SAS to move to a particular column. By using an @ without specifying a column, it is as if you are telling SAS, "Stay tuned for more information. Don't touch that dial!" SAS will hold that line of data until it reaches either the end of the DATA step, or an INPUT statement that does not end with a trailing @.

Example You want to read part of a raw data file containing local traffic data for freeways and surface streets. The data include information about the type of street, name of street, the average number of vehicles per hour traveling that street during the morning, and the average number of vehicles per hour for the evening. Here are the raw data:

freeway 408

3684 3459

surface Martin Luther King Jr. Blvd.	1590	1234
surface Broadway	1259	1290
surface Rodeo Dr.	1890	2067
freeway 608	4583	3860
freeway 808	2386	2518
surface Lake Shore Dr.	1590	1234
surface Pennsylvania Ave.	1259	1290

Suppose you want to see only the freeway data at this point so you read the raw data file, Traffic.dat, with this program:

```
* Use a trailing @, then delete surface streets;
DATA freeways;
  INFILE 'c:\MyRawData\Traffic.dat';
  INPUT Type $ @;
  IF Type = 'surface' THEN DELETE;
  INPUT Name $ 9-38 AMTraffic PMTraffic;
RUN;
```

Notice that there are two INPUT statements. The first reads the character variable Type and then ends with an @. The trailing @ holds each line of data while the IF statement tests it. The second INPUT statement reads Name (in columns 9 through 38), AMTraffic, and PMTraffic. If an observation has a value of surface for the variable Type, then the second INPUT statement never executes. Instead, SAS returns to the beginning of the DATA step to process the next observation and does not add the unwanted observation to the FREEWAYS data set. (Do not pass go, do not collect \$200.)

Here is the FREEWAYS data set:

	Type	Name	AMTraffic
1	freeway	408	3684

2	freeway	608	4583
3	freeway	808	2386

When you run this program, the log will contain the following two notes, one saying that eight records were read from the input file and another saying that the new data set contains only three observations:

NOTE: 8 records were read from the infile
'c:\MyRawData\Traffic.dat'.

The minimum record length was 47.
The maximum record length was 47.

NOTE: The data set WORK.FREeways has 3 observations and
4 variables.

The other five observations had a value of surface for the variable Type and were deleted by the IF statement.

This example used a DELETE statement to delete observations for surface streets. Instead of specifying which observations to delete, you could specify which observations to keep. To keep only observations for freeways, you would use a subsetting IF statement. See Section 3.8 for more about subsetting IF statements.

Trailing @ versus double trailing @ The double trailing @, discussed in the previous section, is similar to the trailing @. Both are line-hold specifiers; the difference is how long they hold a line of data for input. The trailing @ holds a line of data for subsequent INPUT statements, but releases that line of data when SAS returns to the top of the DATA step to begin building the next observation. The double trailing @ holds a line of data for subsequent INPUT statements even when SAS starts building a new observation. In both cases, the line of data is released if SAS reaches a subsequent INPUT statement that does not

contain a line-hold specifier.

2.17 Controlling Input with Options in the INFILE Statement

So far in this chapter, we have seen ways to use the INPUT statement to read many different types of raw data. When reading raw data files, SAS makes certain assumptions. For example, SAS starts reading with the first data line and, if SAS runs out of data on a line, it automatically goes to the next line to read values for the rest of the variables. Most of the time this is OK, but some data files can't be read using the default assumptions. The options in the INFILE statement change the way SAS reads raw data files. The following options are useful for reading particular types of data files. Place these options after the filename in the INFILE statement.

FIRSTOBS= The FIRSTOBS= option tells SAS at what line to begin reading data. This is useful if you have a data file that contains descriptive text or header information at the beginning, and you want to skip over these lines before reading the data. The following data file, for example, has a description of the data in the first two lines:

```
Ice-cream sales data for the summer
Flavor   Location  Boxes sold
Chocolate 213      123
Vanilla   213      512
Chocolate 415      242
```

The following program uses the FIRSTOBS= option to tell SAS to start reading data on the third line of the file:

```
DATA icecream;
  INFILE 'c:\MyRawData\IceCreamSales.dat'
    FIRSTOBS = 3;
  INPUT Flavor $ 1-9 Location BoxesSold;
```

RUN;

OBS= The OBS= option can be used anytime you want to read only a part of your data file. It tells SAS to stop reading when it gets to that line in the raw data file. Note that it does not necessarily correspond to the number of observations. If, for example, you are reading two raw data lines for each observation, then an OBS=100 would read 100 data lines, and the resulting SAS data set would have 50 observations. The OBS= option can be used with the FIRSTOBS= option to read lines from the middle of the file. For example, suppose the ice-cream sales data had a remark at the end of the file that was not part of the data.

```
Ice-cream sales data for the summer
Flavor   Location  Boxes sold
Chocolate 213      123
Vanilla   213      512
Chocolate 415      242
Data verified by Blake White
```

With FIRSTOBS=3 and OBS=5, SAS will start reading this file on the third data line and stop reading after the fifth data line.

```
DATA icecream;
  INFILE 'c:\MyRawData\IceCreamSales2.dat'
  FIRSTOBS = 3 OBS=5;
  INPUT Flavor $ 1-9 Location BoxesSold;
RUN;
```

MISSOVER By default, SAS will go to the next data line to read more data if SAS has reached the end of the data line and there are still more variables in the INPUT statement that have not been assigned values. The **MISSOVER** option tells SAS not to go to the next line of data when it runs out of data. Instead, assign missing values to any remaining variables. The following data file illustrates where this option may be useful. This file

contains test scores for a self-paced course. Since not all students complete all the tests, some have more scores than others.

```
Nguyen 89 76 91 82  
Ramos 67 72 80 76 86  
Robbins 76 65 79
```

The following program reads the data for the five test scores, assigning missing values to tests not completed:

```
DATA class102;  
  INFILE 'c:\MyRawData\AllScores.dat' MISSOVER;  
  INPUT Name $ Test1 Test2 Test3 Test4 Test5;  
  RUN;
```

TRUNCOVER You need the TRUNCOVER option when you are reading data using column or formatted input and some data lines are shorter than others. If a variable's field extends past the end of the data line, then, by default, SAS will go to the next line to start reading the variable's value. This option tells SAS to read data for the variable until it reaches the end of the data line, or the last column specified in the format or column range, whichever comes first. The next file contains addresses and must be read using column or formatted input because the street names have embedded blanks. Note that the data lines have different lengths:

```
John Garcia 114 Maple Ave.  
Sylvia Chung 1302 Washington Drive  
Martha Newton 45 S.E. 14th St.
```

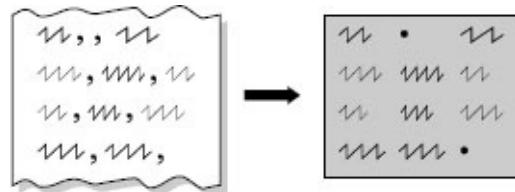
This program uses column input to read the address file. Because some of the addresses stop before the end of the variable Street's field (columns 22 through 37), you need the TRUNCOVER option. Without the TRUNCOVER option, SAS would try to go to the next line to read the data for Street on the first and third records.

```
DATA homeaddress;
```

```
INFILE 'c:\MyRawData\Address.dat' TRUNCOVER;  
INPUT Name $ 1-15 Number 16-19 Street $ 22-37;  
RUN;
```

TRUNCOVER is similar to MISSOVER. Both will assign missing values to variables if the data line ends before the variable's field starts. But when the data line ends in the middle of a variable field, TRUNCOVER will take as much as is there, whereas MISSOVER will assign the variable a missing value.

2.18 Reading Delimited Files with the DATA Step



We suspect that by now you have realized that with SAS there is usually more than one way to accomplish the same result. Section 2.6 showed how to read delimited data files using the IMPORT procedure; now we are going to show how to read delimited files using the DATA step. The INFILE statement has two options that make it easy to read delimited data files: the DLM= option and the DSD option.

The DLM= option If you read your data using list input, SAS expects your file to have spaces between data values. The **DELIMITER=**, or **DLM=**, option in the INFILE statement allows you to use other delimiters. To read data with commas as delimiters, add the DLM= option to your INFILE statement like this:

```
INFILE 'filename' DLM = ',';
```

You can use the DLM= option to specify tabs as delimiters too. In ASCII, 09 is the hexadecimal equivalent of a tab character, and the notation '09'X means a hexadecimal 09.

If your computer uses EBCDIC (IBM mainframes) instead of ASCII, then use DLM='05'X. Here is a DLM= option with the ASCII tab specified as the delimiter.

```
INFILE 'filename' DLM = '09'X;
```

The comma and tab characters are common delimiters for data files, but you can read data files with any delimiter character by simply enclosing the delimiter character in quotation marks after the DLM= option (for example, DLM='&'). If your delimiter is a string of characters, then use the DLMSTR= option.

By default, SAS interprets two or more delimiters in a row as a single delimiter. If your file has missing values, and two delimiters in a row indicate a missing value, then you will also need the DSD option in the INFILE statement.

The DSD option The DSD (Delimiter-Sensitive Data) option for the INFILE statement does three things for you. First, it ignores delimiters in data values enclosed in quotation marks. Second, it does not read quotation marks as part of the data value. Third, it treats two delimiters in a row as a missing value. The DSD option assumes that the delimiter is a comma. If your delimiter is not a comma, then you can use the DLM= option with the DSD option to specify the delimiter. For example, to read a tab-delimited ASCII file with missing values indicated by two consecutive tab characters use this:

```
INFILE 'filename' DLM = '09'X DSD;
```

Comma-separated values files, or CSV files, are a common type of file that can be read with the DSD option. Many programs, such as Microsoft Excel, can save data in CSV format. These files have commas for delimiters and consecutive commas for missing values; if there are commas in any of the data values, then those values are enclosed in quotation marks.

Example The following example illustrates how to read a CSV file using the DSD option. Jerry's Coffee Shop employs local bands to attract customers. Jerry keeps records of the number of customers for each band, for each night they play in his shop. The band's name is followed by the number of customers present at 8 p.m., 9 p.m., 10 p.m., and 11 p.m.

```
Lupine Lights,45,63,70,  
Awesome Octaves,17,28,44,12  
"Stop, Drop, and Rock-N-Roll",34,62,77,91  
The Silveyville Jazz Quartet,38,30,42,43  
Catalina Converts,56,,65,34
```

Notice that one group's name has embedded commas, and is enclosed in quotation marks. Also, the last group has a missing data point for the 9 p.m. hour as indicated by two consecutive commas. Here is the program that will read this data file:

```
DATA music;  
  INFILE 'c:\MyRawData\Bands.csv' DLM = ',' DSD  
  MISSOVER;  
  INPUT BandName :$30. EightPM NinePM TenPM  
        ElevenPM;  
  RUN;
```

Notice that for BandName the INPUT statement uses an informat with a colon modifier. The colon modifier tells SAS to read for the length of the informat (30 for BandName), or until it encounters a delimiter, whichever comes first.

This data file is missing one data value at the end of the first line. By default, SAS would go to the next line of data to find the last data value. The MISSOVER option on the INFILE statement tells SAS not to go to the next line of data when it runs out of values. It is prudent when using the DSD option to add the MISSOVER option if there is any chance that you have missing data values at the end of your

data lines.

Here is the MUSIC data set:

	BandName	EightPM	NinePM	TenP
1	Lupine Lights	45	63	
2	Awesome Octaves	17	28	
3	Stop, Drop, and Rock-N-Roll	34	62	
4	The Silveyville Jazz Quartet	38	30	
5	Catalina Converts	56	.	

The SAS log contains these notes describing the new data set:

NOTE: 5 records were read from the infile

'c:\MyRawData\Bands.csv'.

The minimum record length was 23.

The maximum record length was 41.

NOTE: The data set WORK.MUSIC has 5 observations and
5 variables.

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](#)
- [3.2 Creating and Modifying Variables](#)
- [3.3 Using SAS Functions](#)
- [3.4 Selected SAS Character Functions](#)
- [3.5 Selected SAS Numeric Functions](#)
- [3.6 Using IF-THEN and DO Statements](#)
- [3.7 Grouping Observations with IF-THEN/ELSE Statements](#)
- [3.8 Subsetting Your Data in a DATA Step](#)
- [3.9 Subsetting Your Data Using PROC SQL](#)
- [3.10 Writing Multiple Data Sets Using OUTPUT Statements](#)
- [3.11 Making Several Observations from One Using OUTPUT Statements](#)
- [3.12 Using Iterative DO, DO WHILE, and DO UNTIL Statements](#)
- [3.13 Working with SAS Dates](#)
- [3.14 Selected Date Informats, Functions, and Formats](#)
- [3.15 Using RETAIN and Sum Statements](#)
- [3.16 Simplifying Programs with Arrays](#)
- [3.17 Using Shortcuts for Lists of Variable Names](#)
- [3.18 Using Variable Names with Special Characters](#)

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	Kilometres
1	The Grand West Arashiyama	32200	
2	Kyoto Sharagam	48000	
3	The Palace Side Hotel	10200	
4	Rinn Fushimiinari	41000	
5	Rinn Nijo Castle	18000	
6	Suiran Kyoto	102000	

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

multiples 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
1	The Grand West Arashiyama	32200	9.5
2	Kyoto Sharagam	48000	3.3
3	The Palace Side Hotel	10200	3.8
4	Rinn Fushimiinari	41000	2.9
5	Rinn Nijo Castle	18000	3.3
6	Suiran Kyoto	102000	11.0

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
---------------------------------	---------------

<code>x = 10 * 4 + 3 ** 2;</code>	<code>x = 49</code>
-----------------------------------	---------------------

<code>x = 10 * (4 + 3 ** 2);</code>	<code>x = 130</code>
-------------------------------------	----------------------

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
1	Gregor	10	20	40	0	14	home
2	Molly	15	50	10	1000	14	home
3	Luther	50	100	15	50	14	home
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.

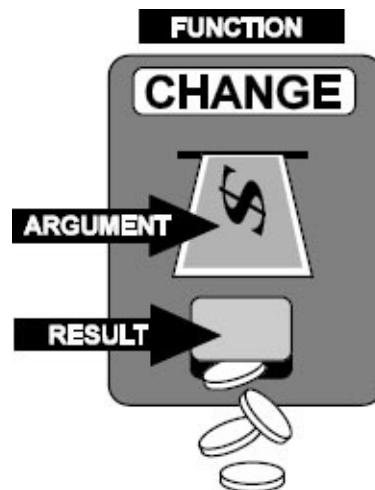
The variable Peas had a missing value for the last observation. Because of this, the variables Total and PerTom, which are calculated from Peas, were also set to missing and this message appeared in the log:

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	8.5	9.0

2	Matthew Lee	9	D	22218	6.5
3	Elizabeth Garcia	10	C	22217	8.9
4	Lori Newcombe	6	D	22218	6.7
5	Jose Martinez	7	d	22219	8.9
6	Brian Williams	11	C	22217	7.8

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	Score 1	Score 2	Score 3	Avg score
1	Alicia Grossman	13	C	22216	7.8	6.5	7.2	7.16
2	Matthew Lee	9	D	22218	6.5	5.9	6.8	6.40
3	Elizabeth Garcia	10	C	22217	8.9	7.9	8.5	8.43
4	Lori Newcombe	6	D	22218	6.7	.	4.9	5.80
5	Jose Martinez	7	D	22219	8.9	9.5	10.0	9.46
6	Brian Williams	11	C	22217	7.8	8.4	8.5	8.23

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		
ANYALNUM	ANYALNUM(<i>arg,start</i>)	Returns position of first occurrence of any alphabetic character or numeral at or after optional start position
ANYALPHA	ANYALPH(<i>arg,start</i>)	Returns position of first occurrence of any alphabetic character at or after optional position
ANYDIGIT	ANYDIGIT(<i>arg,start</i>)	Returns position of first occurrence of any numeral at or after optional start position
ANYSPACE	ANYSPAC(<i>arg,start</i>)	Returns position of first occurrence of a whitespace character at or after optional position
CAT	CAT(<i>arg-1,arg-2,...arg-n</i>)	Concatenates two or more character strings together leaving leading and trailing blanks
CATS	CATS(<i>arg-1,arg-</i>	Concatenates two or more

		$2, \dots, arg-n)$	character strings together stripping leading and trailing blanks
	CATX	CATX('separator-string', $arg-1, arg-2, \dots, arg-n)$	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 sequence of characters
	LEFT	LEFT(arg)	Left aligns a SAS character expression
H	LENGTH	LENGTH(arg)	Returns the length of an argument not counting trailing blanks (missing values have a length of 0)
A	PROPCASE	PROPCASE(arg)	Converts first character in word to uppercase and remaining characters to lowercase
R	SUBST	SUBSTR(arg, position, n)	Extracts a substring from an argument starting at position n characters or until end if no

TRANS LATE	TRANSLA TE(<i>source,to-1, from -1,. from- 1,...to-n,from-n</i>)	Replaces <i>from</i> characters in <i>s</i> with <i>to</i> characters (one-to-one replacement only—you can't replace one character with two, for example)
TRANWRD	TRANWRD(<i>source,from,to</i>)	Replaces <i>from</i> character string <i>source</i> with <i>to</i> character string
TRIM	TRIM(<i>arg</i>)	Removes trailing blanks from character expression
UPCASE	UPCASE(<i>a rg</i>)	Converts all letters in argument <i>a</i> 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	Res
Character				

NUM	ANYAL	a='123 E St, #2 '; x=ANYALNU M(a);	x=1	a='123 E St, #2 '; y=ANYALNU M(a,10);	y=12
PHA	ANYAL	a='123 E St, #2 '; x=ANYALPH A(a);	x=5	a='123 E St, #2 '; y=ANYALPH A(a,10);	y=0
GIT	ANYDI	a='123 E St, #2 '; x=ANYDIGIT (a);	x=1	a='123 E St, #2 '; y=ANYDIGIT(a,10);	y=12
ACE	ANYSP	a='123 E St, #2 '; x=ANYSPAC E(a);	x=4	a='123 E St, #2 '; y=ANYSPACE (a,10);	y=10
CAT	CAT	a=' cat';b='dog '; x=CAT(a,b);	x=' catdog '	a='cat ';b=' dog'; y=CAT(a,b);	y='cat
CATS	CATS	a=' cat';b='dog '; x=CATS(a,b);	x='catdog'	a='cat ';b=' dog'; y=CATS(a,b);	y='cat

	CATX	a=' cat';b='dog'; x=CATX(' ,a,b);	x='cat dog'	a=' cat';b='dog'; y=CATX('&,a,b);	y='cat '
RESS	COMP	a=' cat & dog'; x=COMPRES S(a);	x='cat&do g'	a=' cat & dog'; y=COMPRESS (a,'&');	y=' cat dc
	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
	LENGTH	a='my cat'; x=LENGTH(a);	x=6	a=' my cat '; y=LENGTH(a);	y=7
ASE	PROPC	a='MyCat'; x=PROPCAS E(a);	x='Mycat'	a='TIGER'; y=PROPCASE (a);	y='Ti
2	SUBSTR	a='(916)734- 6281';	x='916'	y=SUBSTR('1c at',2);	y='cat

		x=SUBSTR(a, 2,3);		
LATE	TRANS	a='6/16/99'; x=TRANSLA TE (a,'-','/');	x='6-16- 99'	a='my cat can'; y=TRANSLAT E (a, 'r','c');
WRD	TRAN	a='Main Street'; x=TRANWR D (a,'Street','St');	x='Main St'	a='my cat can'; y=TRANWRD (a,'cat','rat');
	TRIM	a='my '; b='cat'; x=TRIM(a) b; 3	x='mycat '	a='my cat '; b='s'; y=TRIM(a) b;
E	UPCAS	a='MyCat'; x=UPCASE(a) ;	x='MYCA T'	y=UPCASE('T iger');

³ The concatenation operator || concatenates character strings.

3.5 Selected SAS Numeric Functions

Function name	Syntax ⁴	Definition
Numeric		
T IN	INT(<i>arg</i>)	Returns the integer portion of <i>arg</i>
G LO	LOG(<i>arg</i>)	Natural logarithm
G10)	LOG10(<i>arg</i>)	Logarithm to the base 10
AX M	MAX(<i>arg-1,arg-2,...arg-n</i>)	Largest nonmissing value
EAN M	MEAN(<i>arg-1,arg-2,...arg-n</i>)	Arithmetic mean of nonmissing values
N MI	MIN(<i>arg-1,arg-2,...arg-n</i>)	Smallest nonmissing value
N N	N(<i>arg-1,arg-2,...arg-n</i>)	Number of nonmissing values
MISS N	NMISS(<i>arg-1,arg-2,...arg-n</i>)	Number of missing values

R	AND	RAND('UNIFORM')	Generates a random number between 0 and 1 using the uniform distribution
R	ROUND	ROUND(argument, round-off-unit)	Rounds to nearest round-off unit
SUM	M	SUM(argument-1,arg-2,...arg-n)	Sum of nonmissing values

Date⁶

D	ATEJUL	DATEJUL(<i>julian-date</i>)	Converts a Julian date to a SAS date value
D	AY	DAY(<i>date</i>)	Returns the day of the month from a SAS date value
M	DY	MDY(<i>month,day,year</i>)	Returns a SAS date value from month, day, and year values
M	MONTH	MONTH(<i>date</i>)	Returns the month (1–12) from a SAS date value
Q	TR	QTR(<i>date</i>)	Returns the yearly quarter (1–4) from a SAS date value
T	O	TODAY()	Returns the current date as a SAS date value
W	EEDAY	WEEKDAY(<i>date</i>)	Returns day of week (1=Sunday) from a SAS date value

AR	YE)	YEAR(<i>date</i>)	Returns year from a SAS date value
RDIF	Y YRDIFF(<i>start-date,end-date,'AGE'</i>)		Computes difference in years between two SAS date values taking leap year 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=2.3
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=0.9
ROUND	x=ROUND(12.65);	x=13	y=ROUND(12.65,.1);	y=12.
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=
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=yest y's date
WEEKDAY	a=MDY(4,13,2012); x=WEEKDAY(a); ;	x=6	a=MDY(4,18,2012); ; y=WEEKDAY(a);	y=4
YEAR	YE AR	a=MDY(4,13,2012); x=YEAR(a);	x=2012 a=MDY(1,1,1960); y=YEAR(a);	y=196
YRDIF	Y RDIF	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=12.0

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
$\neg =$, $\wedge =$, or $\sim =$	NE	not equal
>	GT	greater than
<	LT	less than
\geq	GE	greater than or eq

\leq 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 Paid	Model	Year Made	Seats	Millions
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	Seat s	Millio n
1	DeDion	LaMarquise	1884	4	
2	Rolls-Royce	Silver Ghost	1912	4	
3	Mercedes-Benz	SSK	1929	2	
4	Oldsmobile	F-88	1954	2	
5	Ferrari	250 Testa Rossa	1957	2	

3.7 Grouping Observations with IF-THEN/ELSE Statements

red	###	
orange	###	warm
yellow	###	warm
green	###	cool
blue	###	cool
purple	###	cool

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 variable 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;

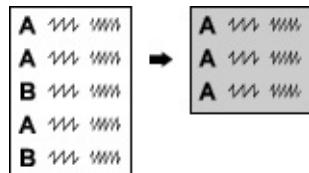
```

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

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	FeedT
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

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;

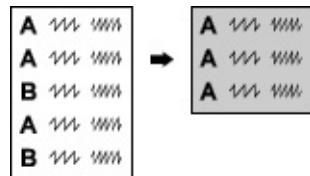
for the statement:

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	
1	bears	Mammalia	E2	b
2	elephants	Mammalia	W3	a
3	flamingos	Aves	W1	p
4	frogs	Amphibia	E8	p
5	kangaroos	Mammalia	W4	a
6	lions	Mammalia	W6	p
7	snakes	Reptilia	E9	p
8	tigers	Mammalia	W9	b
9	zebras	Mammalia	W2	a

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	
1	bears	Mammalia	E2	bc
2	elephants	Mammalia	W3	an
3	kangaroos	Mammalia	W4	an
4	lions	Mammalia	W6	pr
5	tigers	Mammalia	W9	bc
6	zebras	Mammalia	W2	an

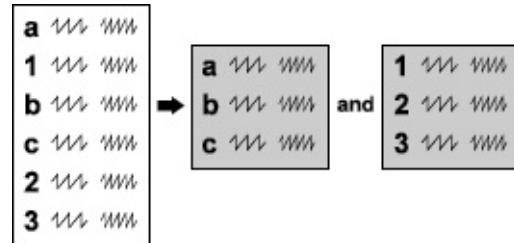
The following note appears in the log.

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	
1	bears	Mammalia	E2	b
2	elephants	Mammalia	W3	a
3	flamingos	Aves	W1	p
4	frogs	Amphibia	E8	p
5	kangaroos	Mammalia	W4	a
6	lions	Mammalia	W6	p
7	snakes	Reptilia	E9	p
8	tigers	Mammalia	W9	b
9	zebras	Mammalia	W2	a

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	
1	bears	Mammalia	E2	b

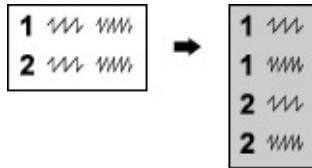
2	elephants	Mammalia	W3	a
3	kangaroos	Mammalia	W4	a
4	tigers	Mammalia	W9	b
5	zebras	Mammalia	W2	a

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

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	Loca
1	Columbia Peaches	04/15/2020	A
2	Walla Walla Sweets	04/17/2020	H
3	Walla Walla Sweets	04/17/2020	H
4	Gilroy Garlics	04/18/2020	H
5	Sacramento Tomatoes	04/21/2020	A

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
1	Jan	Varsity
2	Jan	Downtown
3	Jan	Super-6
4	Feb	Varsity
5	Feb	Downtown
6	Feb	Super-6
7	Mar	Varsity
8	Mar	Downtown
9	Mar	Super-6

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
1	0.02	1014.43
2	0.03	1003.01
3	0.04	1011.50
4	0.05	1040.13
5	0.06	1028.57

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
January 1, 1959	-365	January 1, 1961	366
January 0	0	January 2191!	

1, 1960

1, 2020

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.;
```

ANYDTDTEw. 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 DATEw. 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	Currge
1	A. Jones	0	09/15/96	Mon, Jun 18, 2020	0	
2	R. Grandage	10304	10/31/07	Thu, Jul 5, 2020	-17	
3	K. Kaminaka	15854	01/24/20	Mon, Mar 12, 2020	98	

3.14 Selected Date Informats, Functions, and Formats

Informats	Definition	Width range	Default width
ANYDTDTEw.	Reads dates in various date forms	32	5–9
DATEw.	Reads dates in form: <i>ddmmmyy</i> or <i>ddmmmyyyy</i>	32	7–7
DDMMYY	Reads dates in form: <i>ddmmyy</i>	6–32	6

w.		or <i>ddmmyyyy</i>			
JU LIANw.		Reads Julian dates in form: <i>yyddd</i> or <i>yyyyddd</i>	32	5–	
M MDDYYw .		Reads dates in form: <i>mmdyy</i> or <i>mmdyyyy</i>	32	6–	

Functions	Syntax	Definition
DA TEJUL	DATEJUL(<i>j</i> <i>ulian-date</i>)	Converts a Julian date to a SAS date value ⁷
DAY	DAY(<i>date</i>)	Returns the day of the month from a SAS date value
MDY	MDY(<i>month,day,year</i>)	Returns a SAS date value from month, day, and year values
MONTH	MONTH(<i>date</i>)	Returns the month (1–12) as a SAS date value
QT	QTR(<i>date</i>)	Returns the yearly quarter (1–4) for a SAS date

			value
DAY	TO	TODAY()	Returns the current date as SAS date value
WEEKDAY	YEAR	WEEKDAY(<i>date</i>)	Returns day of week (1=Sunday) SAS date value
DIF	YR	YEAR(<i>date</i>)	Returns year from a SAS date value
		YRDIF(<i>start-date,end-date,'AGE'</i>)	Computes difference in years between two SAS date values taking leap years into account

Formats	Definition	Width range	Default width
w. DATE	Writes SAS date values in form: <i>ddmmmyy</i>	11	5–
DFDDw. EUR	Writes SAS date values in form: <i>dd.mm.yy</i>	10	2–
ANw. JULI	Writes a Julian date from a SAS date value	7	5–

MM DDYYw.	Writes SAS date values in form: <i>mmdyy</i> or <i>mmddyyyy</i>	10	2–
WE EKDATEw.	Writes SAS date values in form: <i>day-of-week, month-name dd,</i> <i>yy or yyyy</i>	37	3–
WO RDDATEw.	Writes SAS date values in form: <i>month-name dd, yyyy</i>	32	3–

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

Information	Input data	INPUT statement	Result
ANYDTD TEw.	1jan1961 01/01/61	INPUT Day ANYDTDTE10.;	366 366
DATEw.	1jan1961	INPUT Day DATE10.;	366

DDMMYY w.	01.01.61	INPUT Day DDMMYY8.;	366
.	02/01/61		367
JULIAN w.	61001	INPUT Day JULIAN7.;	366
MMDDYY w.	01-01-61	INPUT Day MMDDYY8.;	366

Functions	Example	Result	Example	R
DATEJUL	a=60001; x=DATEJUL(a);	x=0	a=60365; y=DATEJUL(a);	y=
DAY	a=MDY(4,18,2020); x=DAY(a);	x=18	a=MDY(9,3,60); y=DAY(a);	y=
MDY	x=MDY(1,1,1960);	x=0	m=2; d=1; y=60; Date=MDY(m,d,y);	D

MONT H	a=MDY(4,18,20 20); x=MONTH(a);	x=4	a=MDY(9,3,60); y=MONTH(a);	y=
QTR	a=MDY(4,18,20 20); x=QTR(a);	x=2	a=MDY(9,3,60); y=QTR(a);	y=
TODA Y	x=TODAY();	x=today 's date	y=TODAY()-1; y='s d	y='s d
WEEK DAY	a=MDY(4,13,202 0); x=WEEKDAY(a);	x=2	a=MDY(4,18,202 0); y=WEEKDAY(a);	y=7
YEAR	a=MDY(4,13,200 0); x=YEAR(a);	x=2000	a=MDY(1,1,1960); y=YEAR(a);	y=1
YRDIF	a=MDY(4,13,200 0); b=MDY(4,13,202 0); x=YRDIF(a,b,'A GE');	x=20	a=MDY(4,13,200 0); b=MDY(8,13,202 0); y=YRDIF(a,b,'A GE');	y=2

Formats	Input data	FORMAT statement ⁸	Results
DATEw.	366	FORMAT Birth DATE7.; FORMAT Birth DATE9.;	01JAN61 01JAN1961
EURDFDD w.	366	FORMAT Birth EURDFDD8. FORMAT Birth EURDFDD10.;	01.01.61 01.01.1961
JULIANw.	366	FORMAT Birth JULIAN5.; FORMAT Birth JULIAN7.;	61001 1961001
MMDDYY w.	366	FORMAT Birth MMDDYY6.; FORMAT Birth MMDDYY10.;	010161 01/01/1961
WEEKDA TEw.	366	FORMAT Birth WEEKDATE9.; FORMAT Birth	Sunday Sunday, Janu 1961

		WEEKDATE29.;	
WORDDA TEw.	366	FORMAT Birth WORDDATE12.;	Jan 1, 1961
		FORMAT Birth WORDDATE18.;	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:

	Mont	h	Day	Team	Hits	Runs	MaxRur	s
1	6	19	Columbia Peaches		8	3		

2	6	20	Columbia Peaches	10	5	
3	6	23	Plains Peanuts	3	4	
4	6	24	Plains Peanuts	7	2	
5	6	25	Plains Peanuts	12	8	
6	6	30	Gilroy Garlics	4	4	
7	7	1	Gilroy Garlics	9	4	
8	7	4	Sacramento Tomatoes	15	9	
9	7	4	Sacramento Tomatoes	10	10	1
10	7	5	Sacramento Tomatoes	2	3	1

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
1	Albany	54	3	.	4	
2	Richmond	33	2	.	3	
3	Oakland	27	3	.	4	
4	Richmond	41	3	5	4	
5	Berkeley	18	4	4	.	

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

```
INPUT Cat8 Cat9  Cat10 Cat11  
      Cat12;
```

Abbreviated list

```
INPUT Cat8 - C
```

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

```
PUT y a c m r b;
```

Abbreviated list

```
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

```
DogBills =  
SUM(DogVet,DogFood,Dog_Care);
```

Abbreviated list

```
DogBills = SU  
Dog:);
```

Special SAS name lists

The special name lists, `_ALL_`, `_CHARACTER_`, and `_NUMERIC_` can also be used anywhere 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
1	Albany	54	3	9	4
2	Richmond	33	2	9	3
3	Oakland	27	3	9	4

4	Richmond	41	3	5	4
5	Berkeley	18	4	4	9

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 -
```

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):

1	Albany	54	3			4	9	3	.	4	4
				9	4						
2	Richmo nd	33	2			3	3	2	.	3	3
				9	3						
3	Oaklan d	27	3			2	3	3	.	4	2
				9	4						
4	Richmo nd	41	3			5	5	3	5	4	5
				5	4						
5	Berkele y	18	4			3	2	4	4	.	3
				4	9						

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\$	Stor
1	8 Person Cabin Tent	USA	399	:
2	Camp Bag	USA	119	:
3	Ultra-light Pad	USA	69	:

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_	Stc
1	8 Person Cabin Tent	USA		399
2	Camp Bag	USA		119
3	Ultra-light Pad	USA		69

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.

4

“Once in a while the simple
things work right off.”

PHIL GALLAGHER

From the SAS L Listserve,
1994. Reprinted by
permission of the author.

CHAPTER 4

Sorting, Printing, and Summarizing Your Data

- 4.1 Using SAS Procedures
- 4.2 Subsetting in Procedures with the WHERE Statement
- 4.3 Sorting Your Data with PROC SORT
- 4.4 Changing the Sort Order for Character Data
- 4.5 Printing Your Data with PROC PRINT
- 4.6 Changing the Appearance of DataValues with Formats
- 4.7 Selected Standard Formats
- 4.8 Creating Your Own Formats with PROC FORMAT
- 4.9 Writing a Report to a Text File
- 4.10 Summarizing Your Data Using PROC MEANS
- 4.11 Writing Summary Statistics to a SAS Data Set
- 4.12 Producing One-Way Frequencies with PROC FREQ
- 4.13 Producing Crosstabulations with PROC FREQ
- 4.14 Grouping Data with User-Defined Formats
- 4.15 Producing Tabular Reports with PROC TABULATE
- 4.16 Adding Statistics to PROC TABULATE Output
- 4.17 Enhancing the Appearance of PROC

TABULATE Output

4.18 Changing Headers in PROC TABULATE Output

4.19 Producing Simple Output with PROC REPORT

4.20 Using DEFINE Statements in PROC REPORT

4.21 Creating Summary Reports with PROC REPORT

4.22 Adding Summary Breaks to PROC REPORT Output

4.23 Adding Statistics to PROC REPORT Output

4.24 Adding Computed Variables to PROC REPORT Output

4.1 Using SAS Procedures



Using a procedure, also called a PROC, is like filling out a form. Someone else designed the form, and all you have to do is fill in the blanks and choose from a list of options. Each PROC has its own unique form with its own list of options. But while each procedure is unique, there are similarities too. This section discusses some of those similarities.

All procedures have required statements, and most have optional statements. For example, the only statement required in PROC PRINT is the PROC statement:

PROC PRINT;

However, by adding optional statements you could make this procedure a dozen lines long or even more.

PROC statement All procedures start with the keyword

PROC followed by the name of the procedure, such as PRINT or CONTENTS. Options, if there are any, follow the procedure name. The DATA= option tells SAS which data set to use as input for that procedure. The DATA= option is, of course, optional. If you skip it, then SAS will use the most recently created data set, which is not necessarily the same as the most recently used. Sometimes it is easier to specify the data set you want than to figure out which data set SAS will use by default. In this case, SAS will use a temporary SAS data set named BANANA:

```
PROC CONTENTS DATA = banana;
```

To use a permanent SAS data set, add the libref to the data set name. In this case, SAS will use a data set named BANANA that is stored in a SAS data library named TROPICAL:

```
PROC CONTENTS DATA = tropical.banana;
```

You can also refer to a permanent SAS data set directly by placing your operating environment's name for it between quotation marks.

```
PROC CONTENTS DATA = 'c:\MySASLib\banana';
```

See Section 2.2 for a more complete discussion of how to reference SAS data libraries and data set members.

BY statement The BY statement is required for only one procedure, PROC SORT. In PROC SORT the BY statement tells SAS how to arrange the observations. In all other procedures, the BY statement is optional, and tells SAS to perform a separate analysis for each combination of values of the BY variables rather than treating all observations as one group. For example, this statement tells SAS to run a separate analysis for each state:

```
BY State;
```

All procedures, except PROC SORT, assume that your data are already sorted by the variables in your BY statement. If

your observations are not already sorted, then use PROC SORT to do the job.

TITLE and FOOTNOTE statements The TITLE statement specifies titles to print at the top of your procedure output. FOOTNOTE works the same way, but prints at the bottom of the page. These global statements are not technically part of any step. You can put them anywhere in your program, but since they apply to the procedure output it generally makes sense to put them with the procedure. The most basic TITLE statement consists of the keyword TITLE followed by your title enclosed in quotation marks. SAS doesn't care if the two quotation marks are single or double as long as they are the same:

```
TITLE 'This is a title';
```

If you find that your title contains an apostrophe, use double quotation marks around the title, or replace the single apostrophe with two:

```
TITLE "Here's another title";  
TITLE 'Here"s another title';
```

You can specify up to ten titles or footnotes by adding numbers to the keywords TITLE and FOOTNOTE:

```
FOOTNOTE3 'This is the third footnote';
```

When you specify a new title or footnote, it replaces the old title or footnote with the same number and cancels those with a higher number. For example, a new TITLE2 cancels an existing TITLE3, if there is one. Titles and footnotes stay in effect until you replace them with new ones or cancel them with a null statement. The following null statement would cancel all current titles:

```
TITLE;
```

LABEL statement Sometimes variable names are not as informative as you would like. In those cases, you can use the LABEL statement to create descriptive labels, up to 256

characters long, for each variable. This statement creates labels for the variables ReceiveDate and ShipDate:

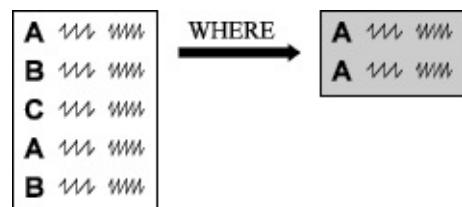
```
LABEL ReceiveDate = 'Date order was received'  
      ShipDate = 'Date merchandise was shipped';
```

If you put a LABEL statement in a DATA step, the labels will be saved with the data; but if you put a LABEL statement in a PROC, then the labels will be used only by that particular step.

Customizing output You have a lot of control over the output produced by procedures. Using system options, you can set many features such as centering, dates, and paper orientation. (See Section 1.7.) With the Output Delivery System, you can also change the overall style of your output, produce output in different formats such as PDF or RTF, or change specific attributes of your output such as font or color. (See Chapter 5.)

Output data sets Most procedures produce some kind of report, but sometimes you would like the results of the procedure saved as a SAS data set so you can perform further analysis. You can create SAS data sets from any procedure output using the **ODS OUTPUT** statement (Section 5.13). Some procedures can also write a SAS data set using an **OUTPUT statement** or **OUT= option**.

4.2 Subsetting in Procedures with the WHERE Statement



One optional statement for a PROC that reads a SAS data set is the WHERE statement. The WHERE statement tells a

procedure to use a subset of the data. There are other ways to subset data, as you probably remember, so you could get by without ever using the WHERE statement. However, the WHERE statement is a shortcut. While subsetting IFs work only in DATA steps, the WHERE statement works in PROC steps too.

Unlike subsetting in a DATA step, using a WHERE statement in a procedure does not create a new data set. That is one of the reasons why WHERE statements are sometimes more efficient than other ways of subsetting. (The WHERE= data set option is similar to a WHERE statement. See Section 6.12 for more information.)

The basic form of a WHERE statement is:

WHERE *condition*;

Only observations satisfying the condition will be used by the PROC. This may look familiar since it is similar to a subsetting IF. The left side of the condition is a variable name, and the right side is a variable name, a constant, or a mathematical expression. Mathematical expressions can contain the standard arithmetic symbols for addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (**). Between the two sides of the expression, you can use comparison and logical operators; those operators may be symbolic or mnemonic:

Symbolic	Mnemonic	Example
=	EQ	WHERE Region = 'Spain';
\neq , \sim =, \wedge =	NE	WHERE Region \sim = 'Spain';
>	GT	WHERE Rainfall > 20;

<	LT	WHERE Rainfall < AvgRain;
>=	GE	WHERE Rainfall >= AvgRain +
<=	LE	WHERE Rainfall <= AvgRain /
&	AND	WHERE Rainfall > 20 AND Temp > 10;
, , !	OR	WHERE Rainfall > 20 OR Temp < 10;
	IS MISSING	WHERE Region IS MISSING;
	IS NOT MISSING	WHERE Region IS NOT MISSING;
	BETWEEN AND	WHERE Region BETWEEN 'Plain' AND 'Spain';
?	CONTAINS	WHERE Region CONTAINS 'Spain';
=*		WHERE Region =* 'Spa%ne';
	LIKE	WHERE Region LIKE 'Sp%on';
	IN (<i>list</i>)	WHERE Region IN ('Rain', 'Snow', 'Plain');

The sounds-like operator (=<*) uses the Soundex algorithm

to select observations with data values that are phonetically similar in English (such as Spain and Spane). The LIKE operator selects observations that match a specific pattern where a percent sign (%) serves as a wildcard corresponding to any number of characters.

Example The following comma-separated values data contain information about well-known painters. For each artist, the data include the painter's name, primary style, and nationality:

Name,Genre,Nationality
Mary Cassatt,Impressionism,U
Paul Cezanne,Post-impressionism,F
Salvador Dali,Surrealism,S
Henri Matisse,Post-impressionism,F
Claude Monet,Impressionism,F
Berthe Morisot,Impressionism,F
Pablo Picasso,Surrealism,S
Jackson Pollock,Abstract expressionism,U
Pierre Auguste Renoir,Impressionism,F
Vincent van Gogh,Post-impressionism,N

Suppose you wanted a list of just the Impressionist painters. The quick-and-easy way to do this is with PROC PRINT and a WHERE statement. In the following program, a PROC IMPORT reads the data from a file named Artists.csv, and uses a LIBNAME statement to create a permanent SAS data set named STYLE in a directory named MySASLib. Then in the PROC PRINT, a WHERE statement selects just the Impressionists, and a FOOTNOTE statement explains the codes for Nationality.

```
* Import CSV file of artists;  
LIBNAME art 'c:\MySASLib';  
PROC IMPORT DATAFILE =  
'c:\MyRawData\Artists.csv' OUT = art.style REPLACE;  
RUN;  
* Print list of Impressionist painters;
```

```

PROC PRINT DATA = art.style;
  WHERE Genre = 'Impressionism';
  TITLE 'Impressionist Painters';
  FOOTNOTE 'F = France U = USA';
RUN;

```

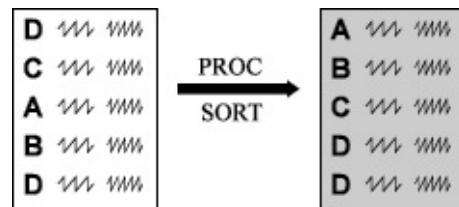
Here are the results. Notice that because the data were subsetted in the PROC PRINT, the numbers in the Obs column reflect the order of observations in the original data set. If you don't like that, you can tell SAS not to print the Obs column. (See Section 4.5.)

Impressionist Painters

Obs	Name	Genre	
1	Mary Cassatt	Impressionism	U
5	Claude Monet	Impressionism	F
6	Berthe Morisot	Impressionism	F
9	Pierre Auguste Renoir	Impressionism	F

F = France U = USA

4.3 Sorting Your Data with PROC SORT



There are many reasons for sorting your data: to organize data for a report, before combining data sets, or before

using a BY statement in another PROC or DATA step. Fortunately, PROC SORT is quite simple. The basic form of this procedure is:

```
PROC SORT;  
    BY variable-list;
```

The variables named in the BY statement are called BY variables. You can specify as many BY variables as you like. With one BY variable, SAS sorts the data based on the values of that variable. With more than one variable, SAS sorts observations by the first variable, then by the second variable within categories of the first, and so on. A BY group is all the observations that have the same values of BY variables. If, for example, your BY variable is Country, then all the observations for Spain form one BY group.

Controlling the output data set The DATA= and OUT= options specify the input and output data sets. If you don't specify the DATA= option, then SAS will use the most recently created data set. If you don't specify the OUT= option, then SAS will replace the original data set with the newly sorted version. This sample statement tells SAS to sort the data set named MESSY, and then put the sorted data into a data set named NEAT:

```
PROC SORT DATA = messy OUT = neat;
```

The **NODUPKEY** option tells SAS to eliminate duplicate observations that have the same values for the BY variables. NODUPKEY applies only to variables listed in the BY statement. So if you want to eliminate duplicate observations that have identical values for all the variables, then you need to include all the variables in your BY statement. If you specify the DUPOUT= option, then SAS will put the deleted observations in that data set. This statement would remove any duplicate observations and put them into a data set named EXTRAOBS:

```
PROC SORT DATA = messy OUT = neat
```

NODUPKEY DUPOUT = extraobs;

Ascending versus descending sorts By default, SAS sorts data in ascending order, from lowest to highest. To have your data sorted in the opposite order, add the keyword DESCENDING to the BY statement before each variable that should be sorted in reverse order. This statement tells SAS to sort first by Country (from A to Z) and then by City (from Z to A) within Country:

BY Country **DESCENDING** City;

Example The following data show the length in feet of whales and sharks. Notice that each line includes data for more than one animal.

```
beluga whale 15 dwarf shark .5 sperm whale 60
basking shark 30 humpback . 50 whale shark 40
gray whale 50 blue whale 100 killer whale 30
mako shark 12 whale shark 40 blue whale 90
```

In the following program, the DATA step reads a raw data file named Lengths.dat and creates a SAS data set named MARINE. Then PROC SORT orders the observations by Family and Name.

```
DATA marine;
  INFILE 'c:\MyRawData\Lengths.dat';
  INPUT Name $ Family $ Length @@;
  RUN;
  * Sort the data and remove duplicates;
  PROC SORT DATA = marine OUT = seasort
    NODUPKEY;
    BY Family Name;
  RUN;
  * Print the sorted data;
  PROC PRINT DATA = seasort;
    TITLE 'Whales and Sharks';
  RUN;
```

The OUT= option writes the sorted data into a new data set

named SEASORT. The NODUPKEY option eliminates observations with duplicate values of the BY variables (Family and Name). The log contains these notes showing that the sorted data set has two fewer observations than the original data set.

NOTE: There were 12 observations read from the data set WORK.MARINE.

NOTE: 2 observations with duplicate key values were deleted.

NOTE: The data set WORK.SEASORT has 10 observations and 3 variables.

The output from PROC PRINT looks like this:

Whales and Sharks

Obs	Name	Family
1	humpback	
2	basking	shark
3	dwarf	shark
4	mako	shark
5	whale	shark
6	beluga	whale
7	blue	whale
8	gray	whale

9	killer	whale
10	sperm	whale

Notice that the humpback, with a missing value for Family, became observation 1. That is because missing values are always lowest for both numeric and character variables. Also, the NODUPKEY option eliminated two duplicate observations, one for the whale shark and one for the blue whale. Because the BY variables were Family and Name, SAS ignored the variable Length, and just kept the first observation for each combination of Family and Name.

4.4 Changing the Sort Order for Character Data

At first glance, sorting character data appears straightforward. After all, everyone knows that "A" comes before "B." However, it is less obvious whether "A" comes before "a." SAS offers dozens of options for controlling the sort order of character data (also called the collating sequence). This section describes a few of them.

ASCII versus EBCDIC The default collating sequence for the z/OS operating environment is EBCDIC. The default collating sequence for most other operating environments is ASCII. From lowest to highest, the basic sort orders for character data are

ASCII	blank	numerals	uppercase letters	low lett
EBCDIC	blank	lowercase letters	uppercase letters	nur

If you work in only one operating environment, this may not matter to you. However, if you need to create a data set on Windows that will be used on z/OS or vice versa, then you might want your data to be sorted in the order expected by that environment. You can use the options SORTSEQ=EBCDIC or SORTSEQ=ASCII to change the sort order:

```
PROC SORT SORTSEQ = EBBCDIC;
```

Other possible values for the SORTSEQ= option include DANISH, FINNISH, ITALIAN, NORWEGIAN, POLISH, SPANISH, and SWEDISH.

Linguistic sorting By default, upper- and lowercase letters will be sorted separately, but this is not the way that people generally sort them. You can use linguistic sorting to produce a more intuitive order. The SORTSEQ=LINGUISTIC option with the STRENGTH=PRIMARY suboption tells SAS to ignore case. To use these options, add them to the PROC SORT statement like this:

```
PROC SORT SORTSEQ = LINGUISTIC (STRENGTH  
= PRIMARY);
```

Here are data that are unsorted, sorted in the default ASCII order, and then sorted ignoring case:

Unsorted order	Default Sort	Linguistic Sort (STRENGTH=PRIMARY)
eva	ANNA	amanda
amanda	Zenobia	ANNA
Zenobia	amanda	eva

ANNA eva Zenobia

When numerals are sorted as character data, the value "10" comes before "2." The NUMERIC_COLLATION=ON suboption tells SAS to treat numerals as their numeric equivalent.

```
PROC SORT SORTSEQ = LINGUISTIC  
(NUMERIC_COLLATION = ON);
```

Here are data that are unsorted, sorted in the default order, and sorted with numeric collation:

Unsorted order	Default Sort	Linguistic Sort (NUMERIC_COLLATION = ON)
1500m freestyle	100m backstroke	50m freestyle
200m breaststroke	1500m freestyle	100m backstroke
100m backstroke	200m breaststroke	200m breaststroke
50m freestyle	50m freestyle	1500m freestyle

Example The following data contain names and addresses:

Name	Street	City	State
Seiki	100 A St.	Anchorage	Alaska

Wong 2 A St. honolulu Hawaii
 Shaw 10 A St. Apt. 10 juneau Alaska
 Smith 10 A St. Apt. 2 Honolulu Hawaii
 Lee 100 A St. Honolulu hawaii
 Jones 10 A St. Apt. 22 Juneau alaska

This program imports the tab-delimited file. Then the data are sorted by the variables State, City, and Street ignoring case and using numeric collation.

```

PROC IMPORT DATAFILE = 'c:\MyRawData\Mail.txt'
OUT = addresses REPLACE;
RUN;
* Sort addresses using linguistic sorting with numeric
collation;
PROC SORT DATA = addresses OUT = sortout
  SORTSEQ = LINGUISTIC (STRENGTH =
PRIMARY NUMERIC_COLLATION = ON);
  BY State City Street;
RUN;
* Print the linguistically sorted data;
PROC PRINT DATA = sortout;
  TITLE 'Addresses Sorted by State, City, and Street';
RUN;

```

Here are the results:

Addresses Sorted by State, City, and Street

Obs	Name	Street	City
1	Seiki	100 A St.	Anchorage
2	Shaw	10 A St. Apt. 10	juneau
3	Jones	10 A St. Apt. 22	Juneau

4	Wong	2 A St.	honolulu
5	Smith	10 A St. Apt. 2	Honolulu
6	Lee	100 A St.	Honolulu

4.5 Printing Your Data with PROC PRINT

The PRINT procedure is perhaps the most widely used SAS procedure. In its simplest form, PROC PRINT prints all the values of all variables for all observations in the SAS data set. SAS decides the best way to format the output, so you don't have to worry about things like how many variables will fit on a page. But there are a few more features of PROC PRINT that you might want to use.

The PRINT procedure starts with the keywords PROC PRINT followed by options:

PROC PRINT *options*;

We recommend always using the DATA= option for clarity in your programs. By default, SAS prints the observation numbers along with the variables' values. If you don't want observation numbers, use the NOOBS option. If you define variable labels with a LABEL statement, and you want to print the labels instead of the variable names, then add the LABEL option as well. The following statement shows all of these options together:

PROC PRINT DATA = *data-set* NOOBS LABEL;

The following are optional statements that sometimes come in handy:

BY *variable-list*;

The BY statement starts a new section of output for each new value of the BY

prints the values of the BY variable each section. The data must be preceded by BY variables.

ID *variable-list*;

When you use the ID statement, the numbers are not printed. Instead, the ID variable list appear on the left of the page.

SUM *variable-list*;

The SUM statement prints sums for variables in the list.

VAR *variable-list*;

The VAR statement specifies which print and their order. Without a VAR statement, all variables in the SAS data set are listed in the order that they occur in the data set.

Example Students from two fourth-grade classes are selling cookies to earn money for a special field trip. The following are the data for the cookie sale. The students' names are followed by their classroom number, the date they turned in their money, the type of cookie (mint patties or chocolate dinosaurs), and the number of boxes sold. Note that each line of data contains information for two students:

Adriana 21 3/21/2020 MP 7 Nathan 14 3/21/2020 CD

19

Matthew 1 3/21/2020 CD 14 Claire 14 3/22/2020 CD

11

Ian 21 3/24/2020 MP 18 Chris 14 3/25/2020 CD 6

Anthony 21 3/25/2020 MP 13 Erika 21 3/25/2020 MP

17

The class earns \$2.50 for each box of cookies sold. The

teachers want a report showing the money earned for each classroom, the money earned by each student, the type of cookie sold, and the date the students returned their money. The following program reads the data, creates a permanent SAS data set named SALES and computes money earned (Profit). Then PROC SORT sorts the data by classroom and creates a temporary data set named SALESSORT. The PROC PRINT uses a BY statement to print the data by Class and a SUM statement to give the totals for Profit. The VAR statement lists the variables to be printed.

```
LIBNAME class 'c:\MySASLib';
DATA class.sales;
    INFILE 'c:\MyRawData\CookieSales.dat';
    INPUT Name $ Class DateReturned MMDDYY10.
        CookieType $ Quantity @@;
        Profit = Quantity * 2.5;
RUN;
PROC SORT DATA = class.sales OUT = salessort;
    BY Class;
RUN;
PROC PRINT DATA = salessort;
    BY Class;
    SUM Profit;
    VAR Name DateReturned CookieType Profit;
    TITLE 'Cookie Sales for Field Trip by Class';
RUN;
```

Here are the results. Notice that the values for the variable DateReturned are printed as their SAS date values. You can use formats, covered in the next section, to print dates in readable forms.

Cookie Sales for Field Trip by Class

Class=14

Obs	Name	DateReturned	CookieType

1	Nathan	21995	CD
2	Matthew	21995	CD
3	Claire	21996	CD
4	Chris	21999	CD
Class			

Class=21

Obs	Name	DateReturned	CookieType
5	Adriana	21995	MP
6	Ian	21998	MP
7	Anthony	21999	MP
8	Erika	21999	MP
Class			

4.6 Changing the Appearance of Data

Values with Formats

0 1002	2 2012	31 4336	→
			Obs Date Sales
			1 01/01/60 1,002
			2 01/03/60 2,012
			3 02/01/60 4,336

When SAS displays your data, it decides which format is best—how many decimal places to show, how much space to allow for each value, and so on. This is very convenient and makes your job much easier, but SAS doesn't always do what you want. Fortunately, you're not stuck with the format SAS thinks is best. You can change the appearance of data values using SAS formats.

SAS has many formats for character, numeric, and date values. For example, you can use the COMMAw.d format to print numbers with embedded commas, the \$w. format to control the number of characters printed, and the MMDDYYw. format to print SAS date values (the number of days since January 1, 1960) in a readable form like 12/03/2023. You can even print your data in more obscure formats like hexadecimal, zoned decimal, and packed decimal, if you like. Using the FORMAT procedure, you can also create your own formats (Section 4.8).

The general forms for SAS formats are:

Character	Numeric	Date
$\$formatw.$	$formatw.d$	$formatw.$

where the \$ indicates character formats, *format* is the name of the format, *w* is the total width including any decimal point, and *d* is the number of decimal places.

FORMAT statement You can associate formats with variables in a FORMAT statement. The FORMAT statement starts with the keyword FORMAT, followed by

the variable name (or names if more than one variable is to be associated with the same format), followed by the format. For example, the following FORMAT statement associates the DOLLAR8.2 format with the variables Profit and Loss and associates the MMDDYY8. format with the variable SaleDate:

```
FORMAT Profit Loss DOLLAR8.2 SaleDate  
      MMDDYY8.;
```

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.

Example In the previous section, results from the fourth-grade cookie sale were printed using the PRINT procedure. The names of the students were printed along with the date they turned in their money, the type of cookie sold, and the profit. You may have noticed that the dates printed were numbers like 21995 and 21999. Using a FORMAT statement in the PRINT procedure, we can print the dates in a readable form. At the same time, we can print the variable Profit using the DOLLAR6.2 format so dollar signs appear before the numbers.

This example uses the permanent SAS data set created in the preceding section. The data are the students' names followed by their classroom, the date they turned in their money, the type of cookie sold (mint patties or chocolate dinosaurs), the number of boxes sold, and profit:

	Name	Class	DateReturned	CookieType	Qu
1	Adriana	21	21995	MP	

2	Nathan	14	21995	CD
3	Matthew	14	21995	CD
4	Claire	14	21996	CD
5	Ian	21	21998	MP
6	Chris	14	21999	CD
7	Anthony	21	21999	MP
8	Erika	21	21999	MP

The following program uses the permanent SAS data set SALES. The FORMAT statement in the PRINT procedure associates the DATE9. format with the variable DateReturned and the DOLLAR6.2 format with the variable Profit:

```

LIBNAME class 'c:\MySASLib';
* Print cookie sales data with formatted values;
PROC PRINT DATA = class.sales;
  VAR Name DateReturned CookieType Profit;
  FORMAT DateReturned DATE9. Profit DOLLAR6.2;
  TITLE 'Cookie Sale Data Using Formats';
  RUN;

```

Here are the results with formatted values for DateReturned and Profit:

Cookie Sale Data Using Formats

Obs	Name	DateReturned	CookieType

1	Adriana	21MAR2020	MP
2	Nathan	21MAR2020	CD
3	Matthew	21MAR2020	CD
4	Claire	22MAR2020	CD
5	Ian	24MAR2020	MP
6	Chris	25MAR2020	CD
7	Anthony	25MAR2020	MP
8	Erika	25MAR2020	MP

4.7 Selected Standard Formats

Format	Definition	Width range	Default width
Character			
\$UPCASE w.	Converts character data to uppercase	1–32767	Length of variable

\$w.	Writes standard character data —does not trim leading blanks (same as \$CHARw.)	1– 32767	Length of variable
Date, Time, and Datetime¹			
DATEw.	Writes SAS date values in form <i>ddmmmyy</i> or <i>ddmmmyyyy</i>	5– 11	7
DATETIME w.d	Writes SAS datetime values in form <i>ddmmmyy:hh:mm:ss.ss</i>	7– 40	16
DTDATE w.	Writes SAS datetime values in form <i>ddmmmyy</i> or <i>ddmmmyyyy</i>	5–9	7
EURDFDD w.	Writes SAS date values in form <i>dd.mm.yy</i> or <i>dd.mm.yyyy</i>	2–10	8
JULIANw.	Writes SAS date values in Julian date form <i>yyddd</i> or	5–7	5

	<i>yyyyddd</i>		
MMDDYY w.	Writes SAS date values in form <i>mm/dd/yy</i> or <i>mm/dd/yyyy</i>	2– 10	8
TIME <i>w.d</i>	Writes SAS time values in form <i>hh:mm:ss.ss</i>	2–20	8
WEEKDA TE <i>w.</i>	Writes SAS date values in form <i>day-of-week, month-name dd,</i> <i>yy, or yyyy</i>	3– 37	29
WORDDA TE <i>w.</i>	Writes SAS date values in form <i>month-name dd, yyyy</i>	3–32	18
Numeric			
BEST <i>w.</i>	SAS chooses the best format —default format for numeric data	1– 32	12
COMMA <i>w.d</i>	Writes numbers with commas	1–32	6

DOLLAR <i>w.d</i>	Writes numbers with a leading \$ and commas separating every three digits	2–32	6	
Ew.	Writes numbers in scientific notation	7–32	12	
EUROXw. <i>d</i>	Writes numbers with a leading € and periods separating every three digits	1–32	6	
PERCENT <i>w.d</i>	Writes numeric data as percentages	4–32	6	
<i>w.d</i>	Writes standard numeric data	1–32	none	

¹ SAS date values are the number of days since January 1, 1960. SAS time values are the number of seconds past midnight, and datetime values are the number of seconds since midnight January 1, 1960.

Format			
	Input data	FORMAT statement	Results
Character			
\$UPCASEw.	my cat	FORMAT Animal \$UPCASE6.;	MY CAT
\$w.	my cat my snake	FORMAT Animal \$8.;	my cat my snak
Date, Time, and Datetime			
DATEw.	8966	FORMAT Birth DATE7.; FORMAT Birth DATE9.;	19JUL84 19JUL1984
DATETIMEw.	12182	FORMAT Start DATETIME13.; FORMAT Start DATETIME18.1;	01JAN60:03:23 01JAN60:03:23:
DTDATEw.	12182	FORMAT Start DTDATE7.; FORMAT Start DTDATE9.;	01JAN60 01JAN1960

EURDFDD w.	8966	FORMAT Birth EURDFDD8.; FORMAT Birth EURDFDD10.;	19.07.84 19.07.1984	
JULIANw.	8966	FORMAT Birth JULIAN5.; FORMAT Birth JULIAN7.;	84201 1984201	
MMDDYY w.	8966	FORMAT Birth MMDDYY8.; FORMAT Birth MMDDYY6.;	7/19/84 071984	
TIMEw.d	12182	FORMAT Start TIME8.; FORMAT Start TIME11.2;	3:23:02 3:23:02.00	
WEEKDA TEw.	8966	FORMAT Birth WEEKDATE15.; FORMAT Birth WEEKDATE29.;	Thu, Jul 19, 84 Thursday, July 1984	
WORDDA TEw.	8966	FORMAT Birth WORDDATE12.; FORMAT Birth WORDDATE18.;	Jul 19, 1984 July 19, 1984	

Numeric			
BESTw.	1200001	FORMAT Value BEST6.;	1.20E6 1200001
		FORMAT Value BEST8.;	
COMMAw. .d	1200001	FORMAT Value COMMA9.; FORMAT Value COMMA12.2;	1,200,001 1,200,001.00
DOLLAR w.d	1200001	FORMAT Value DOLLAR10.; FORMAT Value DOLLAR13.2;	\$1,200,001 \$1,200,001.00
Ew.	1200001	FORMAT Value E7.;	1.2E+06
EUROXw.d	1200001	FORMAT Value EUROX13.2;	€1.200.001,00
PERCENT w.d	0.05	FORMAT Value PERCENT9.2;	5.00%

w.d	23.635	FORMAT Value 6.3; FORMAT Value 5.2;	23.635 23.64
-----	--------	---	-----------------

4.8 Creating Your Own Formats with PROC FORMAT

m 2 f 1 m 3	Obs Sex AgeGroup 1 Male Adult 2 Female Teen 3 Male Senior
--	--

At some time you will probably want to create your own custom formats—especially if you use a lot of coded data. Suppose that you have just completed a survey for your company and to save disk space and time, all the responses to the survey questions are coded. For example, the age categories teen, adult, and senior are coded as numbers 1, 2, and 3. This is convenient for data entry and analysis, but bothersome when it comes time to interpret the results. You could present your results along with a code book, and your company directors could look up the codes as they read the results. But this will probably not get you that promotion you've been looking for. A better solution is to create user-defined formats using PROC FORMAT and print the formatted values instead of the coded values.

The FORMAT procedure creates formats that will later be associated with variables in a FORMAT statement. (Formats can also be used with a PUT function to modify data values in a DATA step. See Section 4.14 for an example.) Here is the general form of PROC FORMAT:

```
PROC FORMAT;
  VALUE name range-1 = 'formatted-text-1'
    range-2 = 'formatted-text-2'
    .
    .
    .
  range-n = 'formatted-text-n';
```

The *name* in the VALUE statement is the name of the format that you are creating. If the format is for character data, the *name* must start with a \$. The *name* can't be longer than 32 characters (including the \$ for character data), it must not start or end with a number, and cannot contain any special characters except an underscore. In addition, the *name* can't be the name of an existing format. Each *range* is the value of a variable that is to be assigned to the text given in quotation marks on the right side of the equal sign. The text can be up to 32,767 characters long. The following are examples of valid range specifications:

```
'A' = 'Asia'
1, 3, 5, 7, 9 = 'Odd'
500000 - HIGH = 'Not Affordable'
13 -< 20 = 'Teenager'
0 <- HIGH = 'Positive Non Zero'
OTHER = 'Bad Data'
```

Character data values should be enclosed in quotation marks ('A' for example). If there is more than one value in the range, then separate the values with a comma or use a hyphen (-) for a continuous range. The keywords LOW and HIGH can be used in ranges to indicate the lowest and the highest nonmissing value for the variable. You can also use the less than symbol (<) in ranges to exclude either end point of the range. The OTHER keyword can be used to assign a format to any values not listed in the VALUE statement.

Example Universe Cars is surveying its customers as to their preferences for car colors. They have information about the customer's age, sex (coded as 1 for male and 2 for female), annual income, and preferred car color (yellow, gray, blue, or white). Here are the data in CSV format:

```
Age,Sex,Income,Color  
19,1,28000,Y  
45,1,130000,G  
72,2,70000,B  
31,1,88000,Y  
58,2,166000,W
```

The following program reads the data; creates three user-defined formats for age, sex, and car color using the FORMAT procedure; and then prints the data using the new formats:

```
PROC IMPORT DATAFILE = 'c:\MyRawData\Cars.csv'  
OUT = carsurvey REPLACE;  
RUN;  
  
PROC FORMAT;  
    VALUE gender 1 = 'Male'  
        2 = 'Female';  
    VALUE agegroup 13 -< 20 = "Teen"  
        20 -< 65 = 'Adult'  
        65 - HIGH = 'Senior';  
    VALUE $col 'W' = 'Moon White'  
        'B' = 'Sky Blue'  
        'Y' = 'Sunburst Yellow'  
        'G' = 'Rain Cloud Gray';  
  
RUN;  
* Print data using user-defined and standard formats;  
PROC PRINT DATA = carsurvey;  
    FORMAT Sex gender. Age agegroup. Color $col.  
Income DOLLAR8.;  
    TITLE 'Survey Results Printed with User-Defined  
Formats';  
RUN;
```

This program creates two numeric formats: GENDER. for the variable Sex and AGEGROUP. for the variable Age. The program creates a character format, \$COL., for the variable Color. Notice that the format names do not end with periods in the VALUE statement, but do in the FORMAT statement.

Here is the output:

Survey Results Printed with User-Defined Formats

Obs	Age	Sex	Income	Color
1	Teen	Male	\$28,000	Sunburst Yellow
2	Adult	Male	\$130,000	Rain Cloud Grey
3	Senior	Female	\$70,000	Sky Blue
4	Adult	Male	\$88,000	Sunburst Yellow
5	Adult	Female	\$166,000	Moon White

This example creates temporary formats that exist only for the current job or session. It is possible to create permanent formats. (See the SAS Documentation for details.) Formats can also be used to group data. (See Section 4.14.)

4.9 Writing a Report to a Text File

PROC PRINT is flexible and easy to use. Still, there are times when PROC PRINT just won't do: when your report to a state agency has to be spaced just like their fill-in-the-

blank form, or when your client insists that the report contain complete sentences, or when you want one page per observation. At those times you can use the flexibility of the DATA step to write a text file. Text files are simple, but with simplicity comes a level of control that is not always possible with other file formats.

You can write data in a DATA step the same way you read data—but in reverse. Instead of using an INFILE statement, you use a FILE statement; instead of INPUT statements, you use PUT statements. This is similar to writing a raw data file in a DATA step (Section 10.6), but to write a report you use the PRINT option telling SAS to include the carriage returns and page breaks needed for printing. Here is the general form of a FILE statement for creating a report:

FILE '*file-specification*' PRINT;

Like INPUT statements, PUT statements can be in list, column, or formatted style, but since SAS already knows whether a variable is numeric or character, you don't have to put a \$ after character variables. If you use list format, SAS will automatically put a space between each variable. If you use column or formatted styles of PUT statements, SAS will put the variables wherever you specify. You can control spacing with the same pointer controls that INPUT statements use: **@*n* to move to column *n*, +*n* to move *n* columns, / to skip to the next line, #*n* to skip to line *n*, and the trailing @ to hold the current line**. In addition to printing variables, you can insert a text string by simply enclosing it in quotation marks.

Example To show how this differs from PROC PRINT, this example uses the permanent SAS data set created in Section 4.5. Two fourth-grade classes have sold cookies to raise money for a field trip. The data are the students' names followed by their classroom, the date they turned in their money, the type of cookie sold (mint patties or

chocolate dinosaurs), the number of boxes sold, and profit:

	Name	Class	DateReturned	CookieType	Qu
1	Adriana	21	21995	MP	
2	Nathan	14	21995	CD	
3	Matthew	14	21995	CD	
4	Claire	14	21996	CD	
5	Ian	21	21998	MP	
6	Chris	14	21999	CD	
7	Anthony	21	21999	MP	
8	Erika	21	21999	MP	

The teachers want a report for each student showing how much money that student earned. They want each student's report on a separate page so it is easy to hand out. Lastly, they want it to be easy for fourth graders to understand, with complete sentences. Here is the program:

```
* Write a report with FILE and PUT statements;  
LIBNAME class 'c:\MySASLib';  
DATA _NULL_;  
SET class.sales;  
FILE 'c:\MyTextFiles\Student.txt' PRINT;
```

```

TITLE;
PUT @5 'Cookie sales report for ' Name 'from
classroom ' Class
// @5 'Congratulations! You sold ' Quantity
'boxes of cookies'
/ @5 'and earned ' Profit DOLLAR6.2 ' for our
field trip.';

PUT _PAGE_;
RUN;

```

Notice that the keyword `_NULL_` appears in the DATA statement instead of a data set name. `_NULL_ tells SAS not to bother writing a SAS data set` (since the goal is to create a text file not a data set), and makes the program run slightly faster. `The FILE statement creates the output file for the report, and the PRINT option tells SAS to include carriage returns and page breaks.` The null TITLE statement tells SAS to eliminate any automatic titles.

The first PUT statement in this program starts with a pointer, `@5`, telling SAS to go to column 5. Then it tells SAS to print the words Cookie sales report for followed by the current value of the variable Name. The variables Name, Class, and Quantity are printed in list style, whereas Profit is printed using formatted style and the DOLLAR6.2 format. A slash line pointer tells SAS to skip to the next line; two slashes skips two lines. You could use multiple PUT statements instead of slashes to skip lines because SAS goes to a new line every time there is a new PUT statement. The statement `PUT _PAGE_` inserts a page break after each student's report. When the program is run, the log will contain these notes:

NOTE: 24 records were written to the file
 'c:\MyRawData\Student.txt'.

NOTE: There were 8 observations read from CLASS.SALES.

The first three pages of the report look like this:

Cookie sales report for Adriana from classroom 21

Congratulations! You sold 7 boxes of cookies and earned \$17.50 for our field trip.

Cookie sales report for Nathan from classroom 14

Congratulations! You sold 19 boxes of cookies and earned \$47.50 for our field trip.

Cookie sales report for Matthew from classroom 14

Congratulations! You sold 14 boxes of cookies and earned \$35.00 for our field trip.

4.10 Summarizing Your Data Using PROC MEANS

One of the first things people usually want to do with their data, after reading it and making sure it is correct, is to look at some simple statistics. Statistics such as the mean value, standard deviation, and minimum and maximum values give you a feel for your data. These types of information can also alert you to errors in your data (a score of 980 in a basketball game, for example, is suspect). The MEANS procedure provides simple statistics for numeric variables.

The MEANS procedure starts with the keywords PROC MEANS, followed by options:

PROC MEANS *options*;

Some options control how your data are summarized:

MAXDEC rounds the value to n decimal places
 $= n$

MISSING treats missing values as valid summary group

Other options request specific summary statistics :

MAX maximum value

MIN minimum value

MEAN mean

MEDIAN median

MODE mode

N number of nonmissing values

NMISS number of missing values

RANGE range

STDDEV standard deviation

SUM sum

If you do not specify any summary statistics, SAS will print the number of nonmissing values, the mean, the standard deviation, and the minimum and maximum values for each variable. More options for PROC MEANS are listed in Section 9.3.

Here are some of the optional statements:

BY *variable-list*;

The BY statement performs a separate analysis for each level of the variables in the *variable-list*. (The data must first be sorted by these variables. You can use PROC SORT to do this.)

CLASS *variable-list*;

The CLASS statement also performs a separate analysis for each level of the variables in the *variable-list*, but its output is more compact than the BY statement, and the data do not have to be sorted first.

VAR *variable-list*;

The VAR statement specifies which variables to use in the analysis. If it is not specified, then SAS uses all numeric variables.

Example A wholesale nursery is selling garden flowers, and they want to summarize their sales figures by month. The data contain the customer ID, date of sale, and number of petunias, snapdragons, and marigolds sold:

756-01 05/04/2020 120 80 110

```
834-01 05/12/2020 90 160 60
901-02 05/18/2020 50 100 75
834-01 06/01/2020 80 60 100
756-01 06/11/2020 100 160 75
901-02 06/19/2020 60 60 60
756-01 06/25/2020 85 110 100
```

The following program reads the data from a file named Flowers.dat, and creates a permanent SAS data set named PLANTS. Then it computes a new variable, Month, which is the month of the sale; and summarizes the data by Month using PROC MEANS with a CLASS statement. The MAXDEC option is set to zero, so no decimal places will be printed.

```
LIBNAME garden 'c:\MySASLib';
DATA garden.plants;
INFILE 'c:\MyRawData\Flowers.dat';
INPUT CustID $ @9 SaleDate MMDDYY10. Petunia
SnapDragon
Marigold;
Month = MONTH(SaleDate);
FORMAT SaleDate MMDDYY10;
RUN;
* Calculate means by Month for flower sales;
PROC MEANS DATA = garden.plants MAXDEC = 0;
CLASS Month;
VAR Petunia SnapDragon Marigold;
TITLE 'Summary of Flower Sales by Month';
RUN;
```

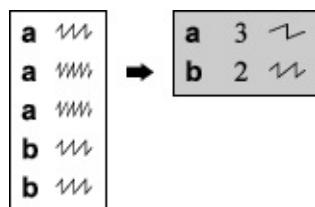
Here are the results of the PROC MEANS:

Summary of Flower Sales by Month

The MEANS Procedure

Month	N Obs	Variable	N	Mean	Std Dev	Minim
5	3	Petunia	3	87	35	
		SnapDragon	3	113	42	
		Marigold	3	82	26	
6	4	Petunia	4	81	17	
		SnapDragon	4	98	48	
		Marigold	4	84	20	

4.11 Writing Summary Statistics to a SAS Data Set



Sometimes you want to save summary statistics in a SAS data set for further analysis, or to merge with other data.

For example, you might want to

plot the hourly temperature in your office to show how it heats up every afternoon causing you to fall asleep, but the instrument you have records data for every minute. The MEANS procedure can condense the data by computing the mean temperature for each hour, and then save the results in a SAS data set so it can be plotted.

There are two methods in PROC MEANS for saving summary statistics in a SAS data set. You can use the OUTPUT destination (covered in Section 5.13), or you can use the OUTPUT statement. The OUTPUT statement has the following form:

```
OUTPUT OUT = data-set output-statistic-list;
```

Here, *data-set* is the name of the SAS data set which will contain the results, and *output-statistic-list* specifies the statistics you want and the associated variable names. You can have more than one OUTPUT statement and multiple output statistic lists. This is one of the possible forms for *output-statistic-list*:

```
statistic(variable-list) = name-list
```

Here, *statistic* can be any of the statistics available in PROC MEANS (SUM, N, MEAN, for example), *variable-list* specifies the variables in the VAR statement you want to output, and

name-list specifies names for the new summary variables.

The new variable names must be

in the same order as their corresponding variables in *variable-list*. For example, the following

PROC MEANS statements produce a new data set called ZOOSUM, which contains one observation with the variables LionWeight, the mean of the lions' weights, and BearWeight,

the mean of the bears' weights:

```
PROC MEANS DATA = zoo NOPRINT;  
  VAR Lions Tigers Bears;  
  OUTPUT OUT = zoosum MEAN(Lions Bears) =  
    LionWeight BearWeight;  
  RUN;
```

The NOPRINT option in the PROC MEANS statement tells SAS there is no need to produce any printed results since we are saving the results in a SAS data set. (Using PROC SUMMARY is the same as using PROC MEANS with the NOPRINT option.)

The SAS data set created in the OUTPUT statement will contain all the variables defined in the *output-statistic-list*; any variables listed in a BY or CLASS statement; plus two

new variables, `_TYPE_` and `_FREQ_`. If there is no BY or CLASS statement, then the data set will have just one observation. If there is a BY statement, then the data set will have one observation for each level of the BY group. CLASS statements produce one observation for each level of interaction of the class variables. The value of the `_TYPE_` variable depends on the level of interaction. The observation where `_TYPE_` has a value of zero is the grand total.

Example This example uses the permanent SAS data set created in the preceding section. The data are sales for a wholesale nursery with the customer ID; date of sale; the number of petunias, snapdragons, and marigolds sold, and the month of sale.

	CustID	SaleDate	Petunia	SnapDragon	Marigold
1	756-01	05/04/2020	120		80
2	834-01	05/12/2020	90		160
3	901-02	05/18/2020	50		100
4	834-01	06/01/2020	80		60
5	756-01	06/11/2020	100		160
6	901-02	06/19/2020	60		60
7	756-01	06/25/2020	85		110

The nursery wants to summarize the data, and then output the results as a SAS data set for further analysis. The following program uses the MEANS procedure with the NOPRINT option to summarize the data by CustID.

Because a CLASS statement is used, the new data set will include one observation for each customer along with one for the grand total. The OUTPUT statement creates a temporary SAS data set named TOTALS. The maximum values are given the variable names MaxP, MaxS, and MaxM, and the sums are given the original variable names Petunia, SnapDragon, and Marigold:

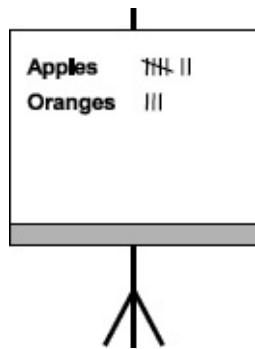
```
LIBNAME garden 'c:\MySASLib';
* Calculate means by CustID, output sum and max to
new data set;
PROC MEANS NOPRINT DATA = garden.plants;
  CLASS CustID;
  VAR Petunia SnapDragon Marigold;
  OUTPUT OUT = totals
    MAX(Petunia SnapDragon Marigold) = MaxP MaxS
    MaxM
    SUM(Petunia SnapDragon Marigold) = Petunia
    SnapDragon Marigold;
RUN;
```

Here is the TOTALS data set. Notice that the _TYPE_ variable has a value of 0 for the grand total, and 1 for the summaries by customer. If you want just the summaries by customer (and no grand total), you could use a BY statement instead of a CLASS statement, but then the data would need to be sorted by CustID first.

	CustID	_TYPE_	_FREQ	MaxP	MaxS	MaxM	Petunia	Snapo
	1	0	7	120	160	110	585	

2	756-01		1		3	120	160	110	305
3	834-01		1		2	90	160	100	170
4	901-02		1		2	60	100	75	110

4.12 Producing One-Way Frequencies with PROC FREQ



A frequency table is a simple list of counts answering the question “How many?” When you have counts for one variable, they are called one-way frequencies. When you combine two or more variables, the counts are called two-way frequencies, three-way frequencies, and so on. This section focuses on one-way frequencies while the next two sections cover more complex examples.

The most obvious reason for using PROC FREQ is to create tables showing the distribution of categorical data values, but PROC FREQ can also reveal irregularities in your data. You could get dizzy proofreading a large data set, but data entry errors are often glaringly obvious in a frequency table. The basic form of PROC FREQ is:

```
PROC FREQ;
  TABLES variable-list;
```

If you skip the TABLES statement, then you will get one-way frequencies for all variables. With a TABLES statement, SAS will produce just the frequencies you specify. This statement would produce two frequency tables: one for the variable YearsEducation and another for Sex:

```
TABLES YearsEducation Sex;
```

You can specify any number of table requests in a single TABLES statement, and you can have as many TABLES statements as you like.

Options, if any, appear after a slash in the TABLES statement and include the following:

MISSING	includes missing values in frequencies and percentages
MISSPRINT	includes missing values in frequencies and percentages
NOCUM	suppresses cumulative frequencies in output
NOPERCENT	suppresses printing of percentages
NOPRINT	suppresses printing of frequency tables
OUT = <i>data-set</i>	writes a data set containing frequencies

The statement below, for example, tells SAS to treat missing values as valid and save the results in a temporary SAS data set named EDFREQS:

```
TABLES YearsEducation / MISSING OUT = edfreqs;
```

PROC FREQ also offers many statistical options. See Section 9.6 for more information.

Example The proprietor of a coffee shop keeps a record of sales. For each drink sold, she records the type of coffee (cappuccino, espresso, kona, or iced coffee), and whether the customer walked in or came to the drive-up window. Here are the data with ten observations per line:

```
esp w cap d cap w kon w ice w kon d esp d kon w ice d  
esp d  
cap w esp d cap d Kon d . d kon w esp d cap w ice w  
kon w  
kon w kon w ice d esp d kon w esp d esp w kon w cap w  
kon w
```

The following program reads the data from a file named Coffee.dat, and creates a permanent SAS data set named ORDERS in a directory named MySASLib (Windows). Then it produces one-way frequencies for two variables: Window and Coffee:

```
LIBNAME drinks 'c:\MySASLib';  
DATA drinks.orders;  
  INFILE 'c:\MyRawData\Coffee.dat';  
  INPUT Coffee $ Window $ @@;  
  RUN;  
  * Print tables for Window and Coffee;  
  PROC FREQ DATA = drinks.orders;  
    TABLES Window Coffee;  
    TITLE 'Coffee Sales by Window and by Type of  
    Drink';  
    RUN;
```

The output contains two tables. The first is a one-way frequency table for the variable Window. You can see that 13 customers came to the drive-up window while 17

walked into the restaurant. In addition to the frequency, the table also shows the percent, cumulative frequency, and cumulative percent for each value of the variable Window.

Coffee Sales by Window and by Type of Drink

The FREQ Procedure

Window	Frequency	Percent	Cumulative Frequency
d	13	43.33	13
w	17	56.67	30

Coffee	Frequency	Percent	Cumulative Frequency
Kon	1	3.45	1
cap	6	20.69	7
esp	8	27.59	15
ice	4	13.79	19
kon	10	34.48	29

Frequency Missing = 1

The second table is a one-way frequency table for Coffee. You can see how many customers ordered each type of drink. Notice that the missing value is noted at the bottom, but not included in the statistics. (Use the MISSING or MISSPRINT options if you want missing values to be included in the counts.) Also, there is one observation with a value of Kon for Coffee. Data values are case sensitive. This is a data entry error; it should be kon.

4.13 Producing Crosstabulations with PROC FREQ

One-way frequencies are a good start, but often you want to know how particular variables interact. You can cross any number of variables to get two-way frequencies, three-way frequencies, and so on, up to n -way frequencies. Tables combining two or more variables are also called crosstabulations or contingency tables.

To produce **crosstabulations**, **separate the variables with asterisks**. This statement produces a two-way table showing frequencies for each combination of values of the variables YearsEducation and Sex:

```
TABLES YearsEducation * Sex;
```

This statement produces a three-way table:

```
TABLES YearsEducation * Sex * Age;
```

Be careful though; reading crosstabulations of three or more levels can feel like playing three-dimensional tic-tac-toe without the benefit of a three-dimensional board.

You can use parentheses to group variables. This statement

produces two, two-way tables: one for YearsEducation by Sex, and another for YearsEducation by Age:

```
TABLES YearsEducation * (Sex Age);
```

You can specify any number of table requests in a single TABLES statement, and you can have as many TABLES statements as you like.

Options, if any, appear after a slash in the TABLES statement. Possible options include some listed in the previous section (MISSPRINT, MISSING, NOPERCENT, NOPRINT, and OUT=) and some that apply specifically to crosstabulations:

CROSSLIST displays crosstabulations in list form

LIST displays crosstabulations in list form
 totals

NOCOL suppresses column percentages in crosstabulations

NOROW suppresses row percentages in crosstabulations

The statement below, for example, tells SAS to leave out all the percentages:

```
TABLES YearsEducation * Sex / NOPERCENT  
         NOROW NOWCOL;
```

Example This example uses the SAS data set created in the previous section. The data are from a coffee shop. For each drink sold, the owner records the type of coffee (cappuccino, espresso, kona, or iced coffee), and whether the customer walked in or came to the drive-up window.

The following program produces two, two-way frequency tables:

```
LIBNAME drinks 'c:\MySASLib';
* Print two tables for Window by Coffee;
PROC FREQ DATA = drinks.orders;
    TABLES Window * Coffee;
    TABLES Window * Coffee / MISSING
NOPERCENT NOCOL NOROW;
    TITLE 'Coffee Sales by Window and Type of Drink';
RUN;
```

Here are the results:

Coffee Sales by Window and Type of Drink

The FREQ Procedure

Table of Window by Coffee					
	Window	Coffee			
		Kon	cap	esp	ice
d	Frequency	1	2	6	2
		3.45	6.90	20.69	6.90
		8.33	16.67	50.00	16.67
		100.00	33.33	75.00	50.00
w	Percent	0	4	2	2
		0.00	13.79	6.90	6.90
					3

	0.00	23.53	11.76	11.76	5
	0.00	66.67	25.00	50.00	9
Total	1 3.45	6 20.69	8 27.59	4 13.79	3
Frequency Missing = 1					

Table of Window by Coffee						
	Window	Coffee				
Frequency		Kon	cap	esp	ice	
d		1	1	2	6	2
w		0	0	4	2	2
Total		1	1	6	8	4

These two tables are the same except that the second one uses options to include missing values in the table instead of in a note at the bottom, and to exclude all the percentages.

4.14 Grouping Data with User-Defined Formats

With user-defined formats you can change how variables

are displayed (covered in Section 4.8), but user-defined formats are also a powerful tool for grouping data. Grouping data this way is a two-step process. First, use the FORMAT procedure to define a format that assigns all the values that you want to group together to a text string. Second, apply the new format to the variable you want to group in a PUT function in a DATA step or a FORMAT statement in a procedure.

PUT function If you want to create a new variable with a user-defined format, you can use a PUT function in a DATA step. The original variable may be numeric or character, but the resulting variable is always character. The general form is:

new-variable = PUT(*old-variable*, *user-defined-format*.);

FORMAT statement You can also group data directly in a procedure without creating a new variable. This is handy for cases where you have a lot of data, and it takes a long time to run a DATA step. Using this method, it is easy to change the groupings by simply creating a new format. This method works for procedures that group data such as PROC FREQ, PROC TABULATE, PROC MEANS with a CLASS statement, and PROC REPORT with GROUP or ACROSS variables. The general form is:

FORMAT *old-variable* *user-defined-format*.;

Example The staff of the local library want to see the type of books people check out by age group. They have the age of the patron in years and the type of book (fiction, mystery, science fiction, biography, non-fiction, or reference). Here are the data with nine observations per line:

```
17 sci 9 bio 28 fic 50 mys 13 fic 32 fic 67 fic 81 non 38  
non  
53 non 16 sci 15 bio 61 fic 52 ref 22 mys 76 bio 37 fic
```

```

86 fic
49 mys 78 non 45 sci 64 bio 8 fic 11 non 41 fic 46 ref
69 fic
34 fic 26 mys 23 sci 74 ref 15 sci 27 fic 23 mys 63 fic
78 non
40 bio 12 fic 29 fic 54 mys 67 fic 60 fic 38 sci 42 fic 80
fic

```

Here is the program that creates three user-defined formats: one for the type of book and two to group the age data in different ways.

```

*Define formats to group the data;
PROC FORMAT;
  VALUE $typ
    'bio','non','ref' = 'Non-Fiction'
    'fic','mys','sci' = 'Fiction';
  VALUE agegpa
    0-18   = '0 to 18'
    19-25  = '19 to 25'
    26-49  = '26 to 49'
    50-HIGH = ' 50+ ';
  VALUE agegpb
    0-25   = '0 to 25'
    26-HIGH = ' 26+ ';
RUN;

```

```

DATA books;
  INFILE 'c:\MyRawData\LibraryBooks.dat';
  INPUT Age Book $ @@;
  BookType = PUT(Book,$typ.);
RUN;
*Create two way table with Age grouped into four
categories;
PROC FREQ DATA = books;
  TITLE 'Patron Age by Book Type: Four Age Groups';
  TABLES BookType * Age / NOPERCENT NOROW
  NOCOL;

```

```

FORMAT Age agegpa.;

RUN;
*Create two way table with Age grouped into two
categories;
PROC FREQ DATA = books;
  TITLE 'Patron Age by Book Type: Two Age Groups';
  TABLES BookType * Age / NOPERCENT NOROW
NOCOL;
FORMAT Age agegpb.;

RUN;

```

In a DATA step, a PUT function uses the format \$TYP. to create a new variable named BookType that groups books into Fiction or Non-Fiction. Then the first PROC FREQ groups age into four categories using the format AGEGPA., while the second PROC FREQ groups age into two categories using the format AGEGPB. Because the NOPERCENT, NOROW, and NOCOL options were added to the TABLES statements, only frequencies appear in the results.

Patron Age by Book Type: Four Age Groups

The FREQ Procedure

Table of BookType by Age				
BookType	Age			
	0 to 18	19 to 25	26 to 49	50 and over
Frequency				
Fiction	6	3	12	
Non-Fiction	3	0	3	

Total	9	3	15
-------	---	---	----

Patron Age by Book Type: Two Age Groups The FREQ Procedure

Table of BookType by Age		
BookType	Age	
Frequency	0 to 25	26+
Fiction	9	22
Non-Fiction	3	11
Total	12	33

4.15 Producing Tabular Reports with PROC TABULATE



Every summary statistic the TABULATE procedure computes can also be produced by other procedures such as PRINT, MEANS, and FREQ, but PROC TABULATE is popular because its reports are pretty. If PROC TABULATE were a box, it would be gift-wrapped.

PROC TABULATE is so powerful that entire books have been written about it, but it is also so concise that you may feel like you're reading hieroglyphics. If you find the syntax of PROC TABULATE a little hard to get used to, that may be because it has roots outside of SAS. PROC TABULATE is based in part on the Table Producing Language, a language developed by the U.S. Department of Labor.

The general form of PROC TABULATE is:

```
PROC TABULATE;  
  CLASS classification-variable-list;  
  TABLE page-dimension, row-dimension, column-  
    dimension;
```

The CLASS statement tells SAS which variables contain categorical data to be used for dividing observations into groups, while the TABLE statement tells SAS how to organize your table and which numbers to compute. Each TABLE statement defines only one table, but you can have multiple TABLE statements. If a variable is listed in a CLASS statement, then, by default, PROC TABULATE produces simple counts of the number of observations in each category of that variable. PROC TABULATE offers many other statistics, and the next section describes how to request those.

Dimensions Each TABLE statement can specify up to three dimensions. Those dimensions, separated by commas, tell SAS which variables to use for the pages, rows, and columns in the report. If you specify only one dimension, then that becomes the column dimension. If you specify two dimensions, then you get rows and columns, but no

page dimension. If you specify three dimensions, then you get pages, rows, and columns.

When you write a TABLE statement, start with the column dimension. Once you have that debugged, add the rows. Once you are happy with your rows and columns, you are ready

to add a page dimension, if you need one. Notice that the order of dimensions in the TABLE statement is page, then row, then column. So, to avoid scrambling your table when you add dimensions, insert the page and row specifications *in front* of the column dimension.

Missing data By default, observations are excluded from tables if they have missing values for variables listed in a CLASS statement (even if that variable is not listed in the TABLE statement). If you want to keep these observations, then simply add the MISSING option to your PROC statement, like this:

```
PROC TABULATE MISSING;
```

Example Here are data about day cruises by boat including the name of each boat, its home port, its type (sailing or power), the kind of vessel (schooner, catamaran, or yacht), the price of an excursion, and the length of the boat in feet.

Name	Port	Type	Vessel	Price	Length
Silent Lady	Maalea	sail	sch	95.00	64
America II	Maalea	sail	yac	72.95	65
Aloha Anai	Lahaina	sail	cat	112.00	60
Ocean Spirit	Maalea	power	cat	62.00	65
Anuenue	Maalea	sail	sch	177.50	52
Hana Lei	Maalea	power	cat	88.99	110
Leilani	Maalea	power	yac	99.99	45
Kalakaua	Maalea	power	cat	69.50	70
Reef Runner	Lahaina	power	yac	59.95	50

Blue Dolphin Maalea sail cat 92.95 65

Suppose that you want a report showing the number of boats by Type and Vessel for each port. The following program reads the tab-delimited file using PROC IMPORT and creates a permanent SAS data set named BOATS. Then PROC TABULATE creates a three-dimensional report with the values of Port for the pages, Type for the rows, and Vessel for the columns.

```
LIBNAME trips 'c:\MySASLib';
PROC IMPORT DATAFILE =
  'c:\MyRawData\Boats.txt' OUT = trips.boats REPLACE;
RUN;
* Tabulations with three dimensions;
PROC TABULATE DATA = trips.boats;
  CLASS Port Type Vessel;
  TABLE Port, Type, Vessel;
  TITLE 'Number of Boats by Port, Type, and Vessel';
RUN;
```

This report has two pages, one for each value of the page dimension. Here is one page:

Number of Boats by Port, Type, and Vessel

Port Maalea		Vessel	
Type		cat	so
	N	N	

power	3
sail	1

The value of the page dimension appears above the top, left corner of the table. You can see that this is the page for the port of Maalea. The heading N tells you that the numbers in this table are simple counts, the number of boats in each group.

4.16 Adding Statistics to PROC TABULATE Output

By default, PROC TABULATE produces simple counts for variables listed in a CLASS statement, but you can request many other statistics in a TABLE statement. You can also concatenate or cross variables within dimensions. In fact, you can write TABLE statements so complicated that even *you* won't know what the report is going to look like until you run it.

While the CLASS statement lists categorical variables, the VAR statement tells SAS which variables contain continuous data. Here is the general form:

```
PROC TABULATE;
  VAR analysis-variable-list;
  CLASS classification-variable-list;
  TABLE page-dimension, row-dimension, column-dimension;
```

While the variables in a CLASS statement can be either numeric or character, variables in a VAR statement must be numeric. You may have both a CLASS statement and a VAR statement, or just one, but all variables listed in a TABLE statement must also appear in either a CLASS or a

VAR statement.

Keywords In addition to variable names, each dimension can contain keywords. Here are a few of the values that PROC TABULATE can compute:

ALL adds a row, column, or page showing the total

MAX highest value

MIN lowest value

MEAN arithmetic mean

MEDIAN median

MODE mode

N number of nonmissing values

NMISS number of missing values

PCTN percentage of observations for that group

PCTSUM percentage of total represented by that group

STDDEV standard deviation

SUM sum

Concatenating, crossing, and grouping Within a dimension, variables and keywords can be concatenated,

crossed, or grouped. To concatenate variables or keywords, simply list them separated by a space; to cross variables or keywords, separate them with an asterisk (*); and to group them, enclose the variables or keywords in parentheses.

The keyword ALL is generally concatenated. To request other statistics, however, cross that keyword with the variable name.

Concatenating: TABLE Type Vessel ALL;

Crossing: TABLE MEAN * Price;

Crossing, grouping, and concatenating: TABLE PCTN *(Type Ves

Example This example uses the permanent SAS data set created in the preceding section. The data contain the name of each boat, its home port, its type (sailing or power), the kind of vessel (schooner, catamaran, or yacht), the price of an excursion, and the boat's length.

	Name	Port	Type	Vessel	Pr
1	Silent Lady	Maalea	sail	sch	
2	America II	Maalea	sail	yac	7
3	Aloha Anai	Lahaina	sail	cat	
4	Ocean Spirit	Maalea	power	cat	

5	Anuenue	Maalea	sail	sch	1
6	Hana Lei	Maalea	power	cat	8
7	Leilani	Maalea	power	yac	9
8	Kalakaua	Maalea	power	cat	
9	Reef Runner	Lahaina	power	yac	5
10	Blue Dolphin	Maalea	sail	cat	9

The following program creates another report about the boats. However, this PROC TABULATE includes a VAR statement. The TABLE statement in this program contains only two dimensions; but it also concatenates, crosses, and groups variables and statistics.

```

LIBNAME trips 'c:\MySASLib';
* Tabulations with two dimensions and statistics;
PROC TABULATE DATA = trips.boats;
  CLASS Type Vessel;
  VAR Price;
  TABLE Type ALL, MEAN*Price*(Vessel ALL);
    TITLE 'Mean Price by Type and Vessel';
  RUN;

```

The row dimension of this table concatenates the classification variable Type with ALL to produce totals. The column dimension, on the other hand, crosses MEAN with the analysis variable Price and with the classification variable Vessel (which happens to be concatenated and grouped with ALL). Here are the results:

Mean Price by Type and Vessel

	Mean		
	Price		
	Vessel		
	cat	sch	yac
Type			
power	73.50	.	79.9
sail	102.48	136.25	72.9
All	85.09	136.25	77.6

4.17 Enhancing the Appearance of PROC

TABULATE Output

When you use PROC TABULATE, SAS wraps your data in tidy little boxes, but there may be times when they just don't look right. Using three simple options, you can enhance the appearance of your output. Think of it as changing the wrapping paper.

FORMAT= option To change the format of all the data cells in your table, use the FORMAT= option in your PROC statement. For example, if you needed the numbers in your table to have commas and no decimal places, you could use this PROC statement:

PROC TABULATE FORMAT = COMMA8.0;
telling SAS to use the COMMA8.0 format for all the data cells in the table.

To apply a format to an individual variable, cross it with the variable name, like this:

*variable-name**FORMAT=*formatw.d*

Then you insert this rather convoluted construction into your TABLE statement:

```
TABLE Region, Sales*FORMAT=COMMA8.0  
Profit*FORMAT=DOLLAR10.2;
```

This TABLE statement applies the COMMA8.0 format to a variable named Sales, and the DOLLAR10.2 format to Profit. If you specify formats in both the PROC and TABLE statements, then the TABLE statement will override the PROC statement.

BOX= and MISSTEXT= options While the FORMAT= option can go in either PROC or TABLE statements, the BOX= and MISSTEXT= options go only in TABLE statements. The BOX= option allows you to insert a brief phrase in the normally empty box that appears in the upper left corner of every PROC TABULATE report. Using this empty space can give your reports a polished look. The MISSTEXT= option, on the other hand, specifies a value for SAS to print in empty data cells. The period that SAS prints, by default, for missing values can seem downright mysterious to someone, perhaps your CEO, who is not familiar with SAS output. You can give them something more meaningful with the MISSTEXT= option. This statement:

```
TABLE Region, MEAN*Sales / BOX='Mean Sales by  
Region' MISSTEXT='No Sales';
```

tells SAS to print the title “Mean Sales by Region” in the upper left corner of the table, and to print the words “No

Sales” in any cells of the table that have no data. The BOX= and MISSTEXT= options must be separated from the dimensions of the TABLE statement by a slash.

Example This example uses the permanent SAS data set created in Section 4.15. The data contain the name of each boat, its home port, its type (sailing or power), the kind of vessel (schooner, catamaran, or yacht), the price of an excursion, and the boat's length.

	Name	Port	Type	Vessel	Pr
1	Silent Lady	Maalea	sail	sch	
2	America II	Maalea	sail	yac	7
3	Aloha Anai	Lahaina	sail	cat	
4	Ocean Spirit	Maalea	power	cat	
5	Anuenue	Maalea	sail	sch	1
6	Hana Lei	Maalea	power	cat	8
7	Leilani	Maalea	power	yac	9
8	Kalakaua	Maalea	power	cat	
9	Reef Runner	Lahaina	power	yac	5
10	Blue Dolphin	Maalea	sail	cat	9

This program is similar to the one in the previous section. However, the variable Length has been added to the VAR and TABLE statements along with the FORMAT=, BOX=, and MISSTEXT= options in the PROC and TABLE statements. Because the BOX= option serves as a title, a null TITLE statement is used to remove the usual title.

```

LIBNAME trips 'c:\MySASLib';
* PROC TABULATE report with options;
PROC TABULATE DATA = trips.boats FORMAT =
DOLLAR7.2;
    CLASS Type Vessel;
    VAR Price Length;
    TABLE Type ALL,
        MEAN * (Price Length*FORMAT=2.0) * (Vessel
ALL)
    /BOX='Full Day Excursions'
    MISSTEXT='none';
    TITLE;
RUN;

```

Here is the enhanced output:

		Mean				
		Price				
		Vessel			Vessel	
		cat	sch	yac	All	cat
Type						
power		\$73.50	none	\$79.97	\$76.09	82

sail	\$102.48	\$136.25	\$72.95	\$110.08	63
All	\$85.09	\$136.25	\$77.63	\$93.08	74

Notice that the data cells for Price now use the DOLLAR7.2 format as specified in the PROC statement, while the cells for Length use the 2.0 format specified in the TABLE statement. The text “Full Day Excursions” appears in the upper left corner, which was empty in the previous section. In addition, the one data cell with no data shows the word “none” instead of a period.

4.18 Changing Headers in PROC TABULATE Output

The TABULATE procedure produces reports with a lot of headers. Sometimes there are so many headers that your reports look cluttered; at other times you may simply feel that a different header would be more meaningful. Before you can change a header, though, you need to understand what type of header it is. PROC TABULATE reports have two basic types of headers: headers that are the values of variables listed in a CLASS statement, and headers that are the names of variables and keywords. You use different methods to change different types of headers.

CLASS variable values To change headers that are the values of variables listed in a CLASS statement, use the FORMAT procedure to create a user-defined format. Then assign the format to the variable in a FORMAT statement (discussed in Section 4.8).

Variable names and keywords You can change headers that are the names of variables using a simple LABEL statement as described in Section 4.1, but that won’t work for keywords. There is another method that

works for both variable names and keywords. In the TABLE statement, you put an equal sign after the variable or keyword followed by the new header enclosed in quotation marks. You can eliminate a header entirely by setting it equal to blank (two quotation marks with nothing in between), and SAS will remove the box for that header. This TABLE statement tells SAS to remove the headers for Region and MEAN, and to change the header for the variable Sales to “Mean Sales by Region.”

```
TABLE Region=", MEAN=/*Sales='Mean Sales by
Region';
```

In some cases SAS leaves the empty box when a row header is set to blank. This happens for statistics and analysis variables (but not class variables). To force SAS to remove the empty box, add the ROW=FLOAT option to the end of your TABLE statement, like this:

```
TABLE MEAN=/*Sales='Mean Sales by Region',
Region=" / ROW=FLOAT;
```

Example This example uses the permanent SAS data set created in Section 4.15. The data contain the name of each boat, its home port, its type (sailing or power), the kind of vessel (schooner, catamaran, or yacht), the price of an excursion, and the boat's length.

	Name	Port	Type	Vessel	Pr
1	Silent Lady	Maalea	sail	sch	
2	America II	Maalea	sail	yac	7
3	Aloha Anai	Lahaina	sail	cat	
4	Ocean Spirit	Maalea	power	cat	

5	Anuenue	Maalea	sail	sch	1
6	Hana Lei	Maalea	power	cat	8
7	Leilani	Maalea	power	yac	9
8	Kalakaua	Maalea	power	cat	
9	Reef Runner	Lahaina	power	yac	5
10	Blue Dolphin	Maalea	sail	cat	9

The following program shows how to change headers. To start with, a FORMAT procedure creates a user-defined format named \$VES. Then the \$VES. format is assigned to the variable Vessel using a FORMAT statement so that the formatted data values will be used for column headers. In the TABLE statement, more headers are changed. The headers for Type, MEAN, and Vessel are all set to blank, while the header for Price is set to “Mean Price by Kind of Vessel”.

```

LIBNAME trips 'c:\MySASLib';
* Changing headers;
PROC FORMAT;
  VALUE $ves 'cat' = 'catamaran'
            'sch' = 'schooner'
            'yac' = 'yacht';
RUN;
PROC TABULATE DATA = trips.boats
FORMAT=DOLLAR7.2;
CLASS Type Vessel;

```

```

VAR Price;
FORMAT Vessel $ves.:
TABLE Type=" ALL,
      MEAN=/*Price='Mean Price by Kind of Vessel'*
(Vessel=" ALL)
/BOX='Full Day Excursions' MISSTEXT='none';
TITLE;
RUN;

```

This program does not require the ROW=FLOAT option because the only variable being set to blank in the row dimension is a class variable. If you put an analysis variable or a statistics keyword in the row dimension and set it equal to blank, then you would need to add the ROW=FLOAT option to remove empty boxes.

Here is the output:

Full Day Excursions	Mean Price by Kind of Vessel		
	catamaran	schooner	yacht
power	\$73.50	none	\$79.
sail	\$102.48	\$136.25	\$72.
All	\$85.09	\$136.25	\$77.

4.19 Producing Simple Output with PROC REPORT



The REPORT procedure shares features with the PRINT, MEANS, TABULATE, and SORT procedures and the DATA step. With all those features rolled into one procedure, it's not surprising that PROC REPORT can be complex—in fact entire books have been written about it—but with all those features comes power.

Here is the general form of a basic REPORT procedure:

```
PROC REPORT;  
    COLUMN variable-list;
```

In its simplest form, the COLUMN statement is similar to a VAR statement in PROC PRINT, telling SAS which variables to include and in what order. If you leave out the COLUMN statement, SAS will, by default, include all the variables in your data set.

Numeric versus character data The type of report you get from PROC REPORT depends, in part, on the type of data you use. If you have at least one character variable in your report, then, by default, you will get a detail report with one row per observation. If, on the other hand, your report includes only numeric variables, then, by default, PROC REPORT will sum those variables. Even dates will be summed, by default, because they are numeric. (You can override this default by assigning one of your numeric variables a usage type of DISPLAY in a DEFINE statement. See the next section.)

Example Here are data about national parks and monuments in the USA. The variables are name, type (NP for national park or NM for national monument), region (East or West), number of museums (including visitor centers), and number of campgrounds.

Dinosaur	NM	West	3	6
Everglades	NP	East	4	2
Grand Canyon	NP	West	3	3
Great Smoky Mountains	NP	East	7	10
Hawaii Volcanoes	NP	West	1	2
Statue of Liberty	NM	East	2	0
Theodore Roosevelt	NP	.	2	2
Yellowstone	NP	West	8	12
Yosemite	NP	West	5	11

The following program creates a permanent SAS data set named NATPARKS, and then runs two reports. The first report has no COLUMN statement so SAS will use all the variables, while the second uses a COLUMN statement to select just the numeric variables.

```

LIBNAME visit 'c:\MySASLib';
DATA visit.natparks;
  INFILE 'c:\MyRawData\Parks.dat';
    INPUT Name $ 1-21 Type $ Region $ Museums
Camping;
RUN;
PROC REPORT DATA = visit.natparks;
  TITLE 'Report with Character and Numeric Variables';
RUN;
PROC REPORT DATA = visit.natparks;
  COLUMN Museums Camping;
  TITLE 'Report with Only Numeric Variables';
RUN;

```

While the two PROC steps are only slightly different, the reports they produce differ dramatically. The first report is almost identical to the output you would get from a PROC PRINT except for the absence of the OBS column. The second report, since it contained only numeric variables, was summed.

Report with Character and Numeric Variables

Name	Type	Region	Museum
Dinosaur	NM	West	
Everglades	NP	East	
Grand Canyon	NP	West	
Great Smoky Mountains	NP	East	
Hawaii Volcanoes	NP	West	
Statue of Liberty	NM	East	
Theodore Roosevelt	NP		
Yellowstone	NP	West	
Yosemite	NP	West	

Report with Only Numeric Variables

Museums	
	35

4.20 Using DEFINE Statements in PROC REPORT

The DEFINE statement is a general purpose statement that specifies options for an individual variable. You can have a DEFINE statement for every variable, but you only need to have a DEFINE statement if you want to specify an option for that particular variable. The general form of a DEFINE statement is:

DEFINE variable / options 'column-header';

In a DEFINE statement, you specify the variable name followed by a slash and any options for that particular variable.

Usage options The most important option is a usage option that tells SAS how that variable is to be used. Possible values of usage options include the following:

ACROSS creates a column for each unique value of the

ANALYS calculates statistics for the variable. This is the usage for numeric variables, and the default st IS SUM.

COMPU TED creates a new variable whose value you calculate in a compute block. See Section 4.24 for a discussion of compute blocks.

DISPLA Y creates one row for each observation in the data set. This is the default usage for character variables.

GROUP creates one row for each unique value of the variable.

ORDER creates one row for each observation with row according to the values of the order variable.

Changing column headers There are several ways to change column headers in PROC REPORT including using a LABEL statement, as described in Section 4.1, or specifying a column header in a DEFINE statement. The following statement tells SAS to arrange a report by the values of the variable Age, and use the words “Age at Admission” as the column header for that variable. Putting a slash in a column header tells SAS to split the header at that point.

```
DEFINE Age / ORDER 'Age at/Admission';
```

Missing data By default, observations are excluded from reports if they have missing values for variables with a usage type of ORDER, GROUP, or ACROSS. If you want to keep these observations, then simply add the MISSING option to your PROC statement, like this:

```
PROC REPORT MISSING;
```

Example This example uses the permanent SAS data set created in the preceding section with data about national parks and monuments in the USA. The variables are name, type (NP for national park or NM for national monument), region (East or West), number of museums (including visitor centers), and number of campgrounds.

	Name	Type	Region	Museu
1	Dinosaur	NM	West	

2	Everglades	NP	East
3	Grand Canyon	NP	West
4	Great Smoky Mountains	NP	East
5	Hawaii Volcanoes	NP	West
6	Statue of Liberty	NM	East
7	Theodore Roosevelt	NP	
8	Yellowstone	NP	West
9	Yosemite	NP	West

The following PROC REPORT contains two DEFINE statements. The first defines Region with a usage type of ORDER. The second specifies a column header for the variable Camping. Camping is a numeric variable and has a default usage of ANALYSIS, so the DEFINE statement does not change its usage. The MISSING option in the PROC statement tells SAS to include observations with missing values of Region.

```

LIBNAME visit 'c:\MySASLib';
* PROC REPORT with ORDER variable, MISSING
option, and column header;
PROC REPORT DATA = visit.natparks MISSING;
COLUMN Region Name Museums Camping;
DEFINE Region / ORDER;
DEFINE Camping / ANALYSIS 'Campgrounds';
TITLE 'National Parks and Monuments Arranged by
Region';
RUN;
```

Here are the results:

National Parks and Monuments Arranged by Region

Region	Name	Museums
	Theodore Roosevelt	2
East	Everglades	4
	Great Smoky Mountains	7
	Statue of Liberty	2
West	Dinosaur	3
	Grand Canyon	3
	Hawaii Volcanoes	1
	Yellowstone	8
	Yosemite	5

4.21 Creating Summary Reports with PROC REPORT

Two different usage types tell the REPORT procedure to “roll up” data into summary groups. While the GROUP usage type produces summary rows, the ACROSS usage

type produces summary columns. However, keep in mind that, if you have any display or order variables in a COLUMN statement, they will override this behavior, and SAS will produce a detail report instead of a summary report.

Group variables Defining a group variable is fairly simple. Just specify the GROUP usage option in a DEFINE statement. By default, analysis variables will be summed. (PROC REPORT can produce many other statistics, see Section 4.23.) The following PROC REPORT tells SAS to produce a report showing the sum of Salary and of Bonus with a row for each value of Department.

Department	Salary	Bonus
A	~~~	~~
B	~~	~

```
PROC REPORT DATA =
employees;
COLUMN Department
Salary Bonus;
DEFINE Department /
GROUP;
RUN;
```

Across variables To define an across variable, you also use a DEFINE statement. However, by default, SAS produces counts rather than sums. To obtain sums for across variables, you must tell SAS which variables to summarize. You do that by putting a comma between the across variable and analysis variable (or variables if you enclose them in parentheses). The following PROC REPORT tells SAS to produce a report showing the sum of Salary and of Bonus with one column for each value of Department.

Department			
A	B	Salary	Bonus
~~~	~~	~~	~

PROC REPORT DATA =  
employees;

```
COLUMN Department ,
(Salary Bonus);
DEFINE Department /
ACROSS;
RUN;
```

**Example** This example uses the permanent SAS data set created in Section 4.19 with data about national parks and monuments in the USA. The variables are name, type (NP for national park or NM for national monument), region, number of museums, and number of campgrounds.

	Name	Type	Region	Museum
1	Dinosaur	NM	West	
2	Everglades	NP	East	
3	Grand Canyon	NP	West	
4	Great Smoky Mountains	NP	East	
5	Hawaii Volcanoes	NP	West	
6	Statue of Liberty	NM	East	
7	Theodore Roosevelt	NP		
8	Yellowstone	NP	West	

9	Yosemite	NP	West	
---	----------	----	------	--

The following program contains two PROC REPORTs. In the first, Region and Type are both defined as group variables. In the second, Region is still a group variable, but Type is an across variable. Notice that the two COLUMN statements are the same except for punctuation added to the second procedure to cross the across variable with the analysis variables.

```

LIBNAME visit 'c:\MySASLib';
* Region and Type as GROUP variables;
PROC REPORT DATA = visit.natparks;
  COLUMN Region Type Museums Camping;
  DEFINE Region / GROUP;
  DEFINE Type / GROUP;
  TITLE 'Summary Report with Two Group Variables';
  RUN;
* Region as GROUP and Type as ACROSS with sums;
PROC REPORT DATA = visit.natparks;
  COLUMN Region Type,(Museums
Camping);
  DEFINE Region / GROUP;
  DEFINE Type / ACROSS;
  TITLE 'Summary Report with a Group and an Across
Variable';
  RUN;

```

Here is the resulting output:

### Summary Report with Two Group Variables

Region	Type	Museums	

East	NM		2
	NP		11
West	NM		3
	NP		17

## Summary Report with a Group and an Across Variable

		Type		
		NM	N	
Region	Museums	Camping	Museums	
East	2	0	11	
West	3	6	17	

## 4.22 Adding Summary Breaks to PROC REPORT Output

Two kinds of statements allow you to insert breaks into a report. The BREAK statement adds a break for each unique value of the variable you specify, while the RBREAK statement does the same for the entire report (or BY-group).

if you are using a BY statement). The general forms of these statements are:

BREAK *location variable / options*;  
RBREAK *location / options*;

where *location* has two possible values—BEFORE or AFTER—depending on whether you want the break to precede or follow that particular section of the report. The options that come after the slash tell SAS what kind of break to insert. Some of the possible options are:

PAGE            starts a new page

SUMMARI    inserts summary statistics for  
ZE                numeric variables

Notice that the BREAK statement requires you to specify a variable, but the RBREAK statement does not. That's because the RBREAK statement produces only one break (at the beginning or end of the report or BY group), while the BREAK statement produces one break for every unique value of the variable you specify. That variable must be either a group or an order variable and, therefore, must also be listed in a DEFINE statement with either the GROUP or ORDER usage option. You can use an RBREAK statement in any report, but you can use BREAK only if you have at least one group or order variable.

**Example** This example uses the permanent SAS data set created in Section 4.19 with data about national parks and monuments in the USA. The variables are name, type (NP for national park or NM for national monument), region (East or West), number of museums (including visitor centers), and number of campgrounds.



	Name	Type	Region	Museum
1	Dinosaur	NM	West	
2	Everglades	NP	East	
3	Grand Canyon	NP	West	
4	Great Smoky Mountains	NP	East	
5	Hawaii Volcanoes	NP	West	
6	Statue of Liberty	NM	East	
7	Theodore Roosevelt	NP		
8	Yellowstone	NP	West	
9	Yosemite	NP	West	

The following program defines Region as an order variable, and then uses both BREAK and RBREAK statements with the AFTER location. The SUMMARIZE option tells SAS to print totals for the numeric variables.

```

LIBNAME visit 'c:\MySASLib';
* PROC REPORT with breaks;
PROC REPORT DATA = visit.natparks;
  COLUMN Name Region Museums Camping;
  DEFINE Region / ORDER;

```

```
BREAK AFTER Region / SUMMARIZE;  
RBREAK AFTER / SUMMARIZE;  
TITLE 'Detail Report with Summary Breaks';  
RUN;
```

Here is the resulting output:

## Detail Report with Summary Breaks

Name	Region	Museums
Everglades	East	4
Great Smoky Mountains		7
Statue of Liberty		2
	East	13
Dinosaur	West	3
Grand Canyon		3
Hawaii Volcanoes		1
Yellowstone		8
Yosemite		5
	West	20

## 4.23 Adding Statistics to PROC REPORT Output

There are several ways to request statistics in the REPORT procedure. An easy method is to insert statistics keywords directly into the COLUMN statement along with the variable names. This is a little like requesting statistics in a TABLE statement in PROC TABULATE, except that instead of using an asterisk to cross a statistics keyword with a variable, you use a comma. In fact, PROC REPORT can produce all the same statistics as PROC TABULATE and PROC MEANS because it uses the same internal engine to compute those statistics. These are a few of the statistics that PROC REPORT can compute:

MAX	highest value
MIN	lowest value
MEAN	arithmetic mean
MEDIAN	median
MODE	mode
N	number of nonmissing values

NMISS	number of missing values
PCTN	percentage of observations for that group
PCTSUM	percentage of a total sum represented by t
STD	standard deviation
SUM	sum

**Applying statistics to variables** To request a statistic for a particular variable, insert a comma between the statistic and the variable in the COLUMN statement. One statistic, N, does not require a comma because it does not apply to a particular variable. If you insert N in a COLUMN statement, then SAS will print the number of observations that contributed to that row of the report. This statement tells SAS to print two columns of data: the median of a variable named Age, and the number of observations in that row.

```
COLUMN Age, MEDIAN N;
```

To request multiple statistics or statistics for multiple variables, put parentheses around the statistics or variables. This statement uses parentheses to request two statistics for the variable Age, and then requests one statistic for two variables, Height and Weight.

```
COLUMN Age,(MIN MAX) (Height Weight),MEAN;
```

**Example** This example uses the permanent SAS data set created in Section 4.19 with data about national parks and monuments in the USA. The variables are name, type, region, number of museums, and number of campgrounds.

The following program contains two PROC REPORTs. Both procedures request the statistics N and MEAN, but the first report defines Type as a group variable, while the second defines Type as an across variable.

```
LIBNAME visit 'c:\MySASLib';
*Statistics in COLUMN statement with two group
variables;
PROC REPORT DATA = visit.natparks;
  COLUMN Region Type N (Museums
Camping),MEAN;
  DEFINE Region / GROUP;
  DEFINE Type / GROUP;
  TITLE 'Statistics with Two Group Variables';
RUN;
*Statistics in COLUMN statement with group and
across variables;
PROC REPORT DATA = visit.natparks;
  COLUMN Region N Type,(Museums
Camping),MEAN;
  DEFINE Region / GROUP;
  DEFINE Type / ACROSS;
  TITLE 'Statistics with a Group and Across Variable';
RUN;
```

Here is the resulting output:

### Statistics with Two Group Variables

			Museums	
Region	Type	N	MEAN	
East	NM	1	2	
	NP	2	5.5	

West	NM	1	3
	NP	4	4.25

## Statistics with a Group and Across Variable

		Type		
		NM		M
		Museums	Camping	Museums
Region	N	MEAN	MEAN	MEAN
East	3	2	0	5.1
West	5	3	6	4.2!

Notice that these reports are similar to the reports in Section 4.21 except that these contain counts and means instead of sums.

## 4.24 Adding Computed Variables to PROC REPORT Output

Unlike most procedures, the REPORT procedure has the ability to compute not only statistics (like sums and means) but also new variables. You do this using a compute block.

Compute blocks start with a COMPUTE statement and end with an ENDCOMP statement. In between, you put the programming statements to calculate your new variable. PROC REPORT uses a limited set of programming statements including assignment statements, IF-THEN/ELSE statements, and DO loops. You do not have to specify a DEFINE statement for the new variable, but if you do, then give it a usage type of COMPUTED. The general form of these statements is:

```
DEFINE new-variable-name / COMPUTED;  
COMPUTE new-variable-name / options;  
      programming statements  
ENDCOMP;
```

**Computing a numeric variable** For a numeric variable, simply name the new variable in the COMPUTE statement. If you use any variables with a type of analysis in the compute block, you must append the variable name with its statistic. The default statistic for an analysis variable is SUM. The following statements compute a variable named Income by adding Salary and Bonus:

```
DEFINE Salary / ANALYSIS SUM;  
DEFINE Bonus / ANALYSIS SUM;  
DEFINE Income / COMPUTED;  
COMPUTE Income;  
      Income = Salary.SUM + Bonus.SUM;  
ENDCOMP;
```

**Computing a character variable** For a character variable, add the CHAR option to the COMPUTE statement. You will probably also want to include the LENGTH option. Lengths for computed character variables range from 1 to 200, with a default of 8. The following statements compute a variable named JobType using IF-THEN/ELSE statements:

```
DEFINE Title / DISPLAY;  
DEFINE JobType / COMPUTED;
```

```
COMPUTE JobType / CHAR LENGTH = 10;  
  IF Title = 'Programmer' THEN JobType = 'Technical';  
  ELSE JobType = 'Other';  
ENDCOMP;
```

**Example** This example uses the permanent SAS data set created in Section 4.19 with data about national parks and monuments in the USA. The variables are name, type, region, number of museums, and number of campgrounds.

The following PROC REPORT computes two variables named Facilities and Note. Facilities is a numeric variable equal to the number of museums plus the number of campgrounds. Note is a character variable, which is equal to "No Camping" for parks that have no campgrounds. Notice that the variables Museums and Camping must be listed in the COLUMN statement because they are used to compute the new variables. In order to exclude them from the report, the program uses the NOPRINT option in DEFINE statements.

```
LIBNAME visit 'c:\MySASLib';  
* COMPUTE new variables that are numeric and  
character;  
PROC REPORT DATA = visit.natparks;  
  COLUMN Name Region Museums Camping  
Facilities Note;  
  DEFINE Museums / ANALYSIS SUM NOPRINT;  
  DEFINE Camping / ANALYSIS SUM NOPRINT;  
  DEFINE Facilities / COMPUTED  
'Campgrounds/and/Museums';  
  DEFINE Note / COMPUTED;  
  COMPUTE Facilities;  
    Facilities = Museums.SUM + Camping.SUM;  
  ENDCOMP;  
  COMPUTE Note / CHAR LENGTH = 10;
```

```

IF Camping.SUM = 0 THEN Note = 'No
Camping';
ENDCOMP;
TITLE 'Report with Two Computed Variables';
RUN;

```

Here is the resulting output:

## Report with Two Computed Variables

Name	Region	Campgrounds and Museums
Dinosaur	West	9
Everglades	East	6
Grand Canyon	West	6
Great Smoky Mountains	East	17
Hawaii Volcanoes	West	3
Statue of Liberty	East	21
Theodore Roosevelt		4
Yellowstone	West	20
Yosemite	West	16

# 5

“Some men see things as they are and say, ‘Why.’ I dream things that never were and say, ‘Why not.’”

ROBERT F. KENNEDY

From *Respectfully Quoted: A Dictionary of Quotations from the Library of Congress*, edited by Suzy Platt, copyright 1992 by Library of Congress. Based on George Bernard Shaw, *Back to Methuselah*, act 1, 1949.

# CHAPTER 5

## Enhancing Your Output with ODS

- [5.1 Concepts of the Output Delivery System](#)
- [5.2 Creating HTML Output](#)
- [5.3 Creating RTF Output](#)
- [5.4 Creating PDF Output](#)
- [5.5 Creating Text Output](#)
- [5.6 Customizing Titles and Footnotes](#)
- [5.7 Customizing PROC PRINT with the STYLE= Option](#)
- [5.8 Customizing PROC REPORT with the STYLE= Option](#)
- [5.9 Customizing PROC TABULATE with the STYLE= Option](#)
- [5.10 Adding Trafficlighting to Your Output](#)
- [5.11 Selected Style Attributes](#)
- [5.12 Tracing and Selecting Procedure Output](#)
- [5.13 Creating SAS Data Sets from Procedure Output](#)

### 5.1 Concepts of the Output Delivery System



You might think that procedures produce output. They don't. Technically, procedures produce only data. Then they send the data to the **Output Delivery System (ODS)** which determines where the output should go and what it should look like when it gets there. That means the question to ask yourself is not whether you want to use ODS—you always use ODS. The question is whether to accept default output or choose something else.

ODS is like a busy airport. Passengers arrive by car and bus. Once at the airport, passengers check baggage, pass security, eventually board a plane, and fly out to their destinations. In ODS, data are like passengers arriving from various procedures. ODS processes each set of data and sends it off to its proper destination. In fact, different types of ODS output are called destinations. What your data look like when they get to their destination is determined by templates. A template is a set of instructions telling ODS how to format your data. These two concepts—destinations and templates—are fundamental to understanding what you can do with ODS.

**Destinations** SAS can send output to many different destinations. If you are satisfied with the default output, then you can skip the rest of this chapter. However, if you want more customized output, the features described in this chapter can help you. Here are the major ODS destinations:

CSV

Comma-separated values

EXCEL

Microsoft Excel

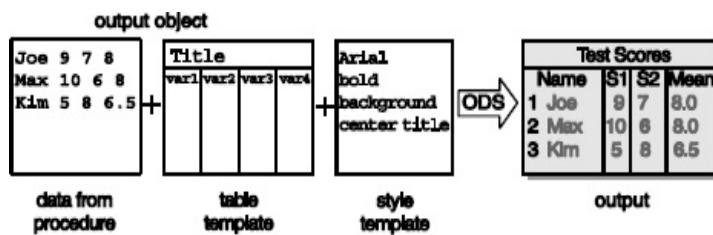
HTML	Hypertext Markup Language
LISTING	plain text output
OUTPUT	SAS data set
PDF	Portable Document Format
POWERPOINT	Microsoft PowerPoint
T	
PS	PostScript
RTF	Rich Text Format
WORD	Microsoft Word document (pre-product 9.4 M6)

HTML is the default destination for SAS Studio. HTML is also the default destination for SAS Enterprise Guide starting with release 8.1 (earlier versions use the proprietary format SASREPORT). Likewise, HTML is the default destination for the SAS windowing environment in Microsoft Windows and UNIX starting with SAS 9.3 (earlier versions use the LISTING destination). If you run SAS programs in batch or in other operating environments, the default destination is LISTING.

Most of these destinations are designed to create output for viewing on a screen or for printing. The CSV (Section 10.3), EXCEL (Section 10.5), and OUTPUT (Section 5.13) destinations, on the other hand, create data files or data

sets.

**Style and table templates** Templates tell ODS how to format and present your data. The two most common types of templates are table templates and style templates (also called table definitions and style definitions). A table template specifies the basic structure of your output (which variable will be in the first column?), while a style template specifies how the output will look (will the headers be blue or red?). ODS combines the data produced by a procedure with a table template and together they are called an output object. The output object is then combined with a style template and sent to a destination to create your final output.



In ODS Graphics, instead of a table template, your data are combined with a graph template, and the final result is graphical output instead of tabular output. See Chapter 8 for more information about ODS Graphics.

You can create your own style templates using the TEMPLATE procedure. However, PROC TEMPLATE's syntax is rather arcane. Fortunately, there are other ways to control and modify output. The quickest and easiest way to change the look of your output is to use one of the many built-in style templates. To view a list of the style templates available on your system, submit the following PROC TEMPLATE statements:

```
PROC TEMPLATE;  
  LIST STYLES;  
  RUN;
```

Here are a few of the built-in style templates:

ANALYSIS	HTMLBL UE	MINIMAL	RTF	ST
BARRETTSB LUE	JOURNA L	PEARL	SASWEB	

Notice that RTF is the name of both a destination and a style. Some styles work better with certain destinations than with others. HTMLBLUE is the default style for HTML output, RTF is the default style for RTF output, and PEARL is the default style for output sent to the PDF and PS destinations.

While all procedures that produce printable output allow you to use style templates to control the overall look of that output, the PRINT, REPORT, and TABULATE procedures allow you to do something special. With these three procedures, you can use the STYLE= option directly in the PROC step to control individual features of your output without having to create a whole new style template. Sections 5.7 to 5.11 discuss how to do this.

## 5.2 Creating HTML Output

When you send output to the HTML destination, you get files in Hypertext Markup Language format. These files are ready to be posted on a website for viewing by your boss or colleagues, but HTML output has other uses too. It can be read into spreadsheets, and even printed or imported into word processors (though some formatting may change). HTML output is what you see in the Results tab in SAS Studio and SAS Enterprise Guide (starting with release 8.1), or in the Results Viewer in the SAS windowing

environment (starting with SAS 9.3), but you can use ODS HTML statements to create HTML files in any SAS environment.

**ODS statement** The general form of the ODS statement to open an HTML file is:

```
ODS HTML PATH = 'path' BODY = 'filename.html'  
options;
```

The PATH= option specifies the location where the output will be written. The BODY= option creates a body file that contains your results, and is synonymous with the FILE= option. A common option is STYLE= which specifies a style template (default is HTMLBLUE). The following statement tells SAS to send output to the HTML destination, save the results in a file named AnnualReport.html in a directory named MyHTMLFiles on the C drive, and use the style named MINIMAL.

```
ODS HTML PATH = 'c:\MyHTMLFiles\' BODY =  
'AnnualReport.html'  
STYLE = MINIMAL;
```

ODS statements are global, and do not belong to either DATA steps or PROC steps. However, if you put them in the wrong place, they won't capture the output that you want. **A good place to put the first ODS statement is just before the step (or steps) whose output you want to capture.**

To close the HTML destination, put this statement after the step (or steps) whose output you want to capture, and following a RUN statement:

```
ODS HTML CLOSE;
```

If the default HTML output gets turned off, you can always turn it back on by submitting this statement:

```
ODS HTML;
```

**Removing procedure titles** Some procedures (such as PROC MEANS and PROC FREQ) include the name of the

procedure in your output. You can remove procedure names by using the ODS NOPROCTITLE statement. This statement works for all destinations, not just HTML.

```
ODS NOPROCTITLE;
```

**Example** This example uses data about selected whales and sharks. The data include the animal's name, type (whale or shark), and average length in feet. Notice that each line contains data for four observations.

```
beluga whale 15 dwarf shark .5 basking shark  
30 humpback whale 50  
whale shark 40 blue whale 100 killer whale  
30 mako shark 12
```

The following program creates a permanent SAS data set named MARINE. The program also contains two procedures: PROC MEANS and PROC PRINT. There are three ODS statements. The first ODS statement opens the Marine.html file in the MyHTMLFiles folder and selects the style BARRETSBLUE. The second ODS statement turns off procedure titles. The third ODS statement closes the file.

```
LIBNAME ocean 'c:\MySASLib';  
DATA ocean.marine;  
    INFILE 'c:\MyRawData\Lengths8.dat';  
    INPUT Name $ Family $ Length @@;  
RUN;  
* Create the HTML file and remove procedure name;  
ODS HTML PATH = 'c:\MyHTMLFiles' BODY =  
'Marine.html'  
    STYLE = BARRETSBLUE;  
ODS NOPROCTITLE;  
PROC MEANS DATA = ocean.marine MEAN MIN  
MAX;  
    CLASS Family;  
    TITLE 'Whales and Sharks';  
RUN;
```

```
PROC PRINT DATA = ocean.marine;  
RUN;  
ODS HTML CLOSE;
```

Here is what the Marine.html file looks like when viewed in a browser:

The screenshot shows a web browser window titled "SAS Output". The address bar indicates the file is located at "C:/myhtmfiles/marine.html". The main content area displays two tables under the heading "Whales and Sharks".

**Analysis Variable : Length**

Family	N Obs	Mean	Minimum	Maximum
shark	4	20.6250000	0.5000000	40.0000000
whale	4	48.7500000	15.0000000	100.0000000

**Whales and Sharks**

Obs	Name	Family	Length
1	beluga	whale	15.0
2	dwarf	shark	0.5
3	basking	shark	30.0
4	humpback	whale	50.0
5	whale	shark	40.0
6	blue	whale	100.0
7	killer	whale	30.0
8	mako	shark	12.0

You may also be able to save HTML output in a file without using ODS statements. The options available depend on the interface you use to run SAS. With a little exploration, you will find various menus, icons, and tasks for creating, exporting, and saving HTML output.

## 5.3 Creating RTF Output

Rich Text Format (RTF) was developed by Microsoft for document interchange. When you create RTF output, you can copy it into a Word document, and edit or resize it like other Word tables. RTF is not a default destination, but you can send output to it using ODS RTF statements.

**ODS statement** The general form of the ODS statement to open an RTF file is:

```
ODS RTF PATH = 'path' FILE = 'filename.rtf' options;
```

The PATH= option specifies the location where the output will be written. The FILE= option creates a file that contains your results and is synonymous with the BODY= option. Here are some of the most commonly used options for this destination:

BODYTITLE	puts titles and footnotes in the main RTF document instead of in Word headers and footers.
COLUMNS = <i>n</i>	requests columnar output where <i>n</i> is the number of columns.
SASDATE	by default, the date and time that appear at the top of RTF output indicate when the document was opened or printed in Word. This option allows you to use the date and time when the current session or job started running.
STARTPAGE = <i>value</i>	controls page breaks. The default value of YES inserts a page break between procedures. A value of NO turns off page breaks. A value of NOW inserts a page break at the present time.

`STYLE = style-name` specifies a style template. The default style template is JOURNAL.

The following statement tells SAS to send output to the RTF destination, save the results in a file named AnnualReport.rtf in a directory named MyRTFFiles on the C drive, and use the style JOURNAL:

```
ODS RTF PATH = 'c:\MyRTFFiles' FILE =  
'AnnualReport.rtf' STYLE = JOURNAL;
```

ODS statements are global, and do not belong to either DATA steps or PROC steps. However, if you put them in the wrong place, they won't capture the output that you want. A good place to put the first ODS statement is just before the step (or steps) whose output you want to capture.

To close the RTF destination, put this statement after the step (or steps) whose output you want to capture, and following a RUN statement:

```
ODS RTF CLOSE;
```

**Example** This example uses the permanent SAS data set created in the preceding section. The data set contains the average lengths, in feet, of selected whales and sharks. The following program contains two procedures: PROC MEANS and PROC PRINT. There are three ODS statements in the program: one to open the RTF file, one to turn off procedure titles, and one to close the RTF file.

```
* Create an RTF file;  
LIBNAME ocean 'c:\MySASLib';  
ODS RTF PATH = 'c:\MyRTFFiles'  
FILE = 'Marine.rtf'  
BODYTITLE STARTPAGE = NO;  
ODS NOPROCTITLE;
```

```
PROC MEANS DATA = ocean.marine MEAN MIN  
MAX;  
    CLASS Family;  
    TITLE 'Whales and Sharks';  
RUN;  
PROC PRINT DATA = ocean.marine;  
RUN;  
ODS RTF CLOSE;
```

Here is what the Marine.rtf file looks like when viewed in Microsoft Word. Because the BODYTITLE option was specified, the titles appear with the tables instead of in a Word header. The STARTPAGE=NO option told SAS not to insert a page break between the two tables. Since no style was specified, SAS used the default style, RTF.

The screenshot shows a Microsoft Word document window titled "Marine.rtf [Compatibility Mode] - Word". The "Layout" tab is selected in the ribbon. The main content of the document is as follows:

*Whales and Sharks*

Analysis Variable : Length				
Family	N Obs	Mean	Minimum	Maximum
shark	4	20.6250000	0.5000000	40.0000000
whale	4	48.7500000	15.0000000	100.0000000

*Whales and Sharks*

Obs	Name	Family	Length
1	beluga	whale	15.0
2	dwarf	shark	0.5
3	basking	shark	30.0
4	humpback	whale	50.0
5	whale	shark	40.0
6	blue	whale	100.0
7	killer	whale	30.0
8	mako	shark	12.0

Page 1 of 1 62 words

You may also be able to create RTF output without using ODS statements. The options available depend on the interface you use to run SAS. With a little exploration, you will find various menus, icons, and tasks for creating, exporting, and saving RTF output.

## 5.4 Creating PDF Output

The PDF destination creates output in Portable Document Format, a format that was developed by Adobe Systems but then became an open standard for document exchange. PDF is not a default destination, but you can send output to

it using ODS PDF statements.

**ODS statement** The general form of the ODS statement to open a PDF file is:

ODS PDF FILE = '*filename.pdf*' *options*;

The FILE= option creates a file that contains your results. If your filename does not include a path, then the file will be saved in a default location. The options available for this destination include:

COLUMNS = *n* requests columnar output where number of columns.

STARTPAGE = *value* controls page breaks. The default inserts a break between procedure NO turns off breaks. A value of N break at that point.

STYLE = *style-name* specifies a style template. The default is PEARL.

The following statement tells SAS to create PDF output, save the results in a file named AnnualReport.pdf in a directory named MyPDFFiles on the C drive, and use the style JOURNAL.

```
ODS PDF FILE = 'c:\MyPDFFiles\AnnualReport.pdf'  
      STYLE = JOURNAL;
```

ODS statements are global, and do not belong to either DATA steps or PROC steps. However, if you put them in the wrong place, they won't capture the output that you want. A good place to put the first ODS statement is just before the step (or steps) whose output you want to capture.

To close the PDF destination, put this statement after the

step (or steps) whose output you want to capture, and following a RUN statement:

```
ODS PDF CLOSE;
```

**Example** This example uses the permanent SAS data set created in Section 5.2. The data set contains the average lengths, in feet, of selected whales and sharks. **The following program contains two procedures: PROC MEANS and PROC PRINT. There are three ODS statements in the program: one to open the PDF file, one to turn off procedure titles, and one to close the PDF file.**

```
* Create the PDF file;  
LIBNAME ocean 'c:\MySASLib';  
ODS PDF FILE = 'c:\MyPDFFiles\Marine.pdf'  
STARTPAGE = NO;  
ODS NOPROCTITLE;
```

```
PROC MEANS DATA = ocean.marine MEAN MIN  
MAX;  
    CLASS Family;  
    TITLE 'Whales and Sharks';  
RUN;  
PROC PRINT DATA = ocean.marine;  
RUN;  
ODS PDF CLOSE;
```

Here is what the Marine.pdf file looks like when viewed in Adobe Acrobat. Because the option STARTPAGE=NO was specified, the output from both procedures appears on the same page. Since no style was specified, SAS used the default style, PEARL.

The screenshot shows a Adobe Acrobat Pro window displaying a PDF file named 'Marine.pdf'. The PDF contains two tables generated by SAS. The first table, titled 'Analysis Variable : Length', provides summary statistics for 'shark' and 'whale' families. The second table lists individual observations ('Obs') with their names ('Name'), families ('Family'), and lengths ('Length').

Family	N Obs	Mean	Minimum	Maximum
shark	4	20.6250000	0.5000000	40.0000000
whale	4	48.7500000	15.0000000	100.0000000

Obs	Name	Family	Length
1	beluga	whale	15.0
2	dwarf	shark	0.5
3	basking	shark	30.0
4	humpback	whale	50.0
5	whale	shark	40.0
6	blue	whale	100.0
7	killer	whale	30.0
8	mako	shark	12.0

You may also be able to create PDF output without using ODS statements. The options available depend on the interface you use to run SAS. With a little exploration, you will find various menus, icons, and tasks for creating, exporting, and saving PDF output.

## 5.5 Creating Text Output

The LISTING destination creates simple text output. Text output consists of basic characters without the special formatting added by applications such as word processors or spreadsheets. Text output has some advantages. It is highly portable, compact when printed, can be easily

edited, and can be read by many software packages including word processors and spreadsheets. LISTING is the default destination when you run SAS in batch or in the z/OS operating environment, but you can always create text output using ODS LISTING statements.

**ODS statement** In the SAS windowing environment and in SAS Enterprise Guide, you can open the LISTING destination by submitting an ODS LISTING statement, like this:

ODS LISTING;

If you submit this statement, then your output will automatically appear in the Output window in the SAS windowing environment or in the Results-Listing tab in SAS Enterprise Guide. You do not need to close the destination in order to see text output in these interfaces. If you want to stop producing text output, submit this statement:

ODS LISTING CLOSE;

In any SAS environment, you can save text output as a separate file by adding a FILE= option to an ODS LISTING statement. The general form of the ODS statement to open a listing file is:

ODS LISTING FILE = '*filename*';

If your filename does not include a path, then the file will be saved in a default location.

The following statement tells SAS to send output to the LISTING destination and save the results in a file named AnnualReport.lst in a directory named MyTextFiles on the C drive. Note that the LISTING destination does not use styles for tabular output.

ODS LISTING FILE =  
'c:\MyTextFiles\AnnualReport.lst';

ODS statements are global, and do not belong to either

DATA steps or PROC steps. However, if you put them in the wrong place, they won't capture the output that you want. A good place to put the first ODS statement is just before the step (or steps) whose output you want to capture.

To close the listing file, put this statement after the step (or steps) whose output you want to capture, and following a RUN statement:

```
ODS LISTING CLOSE;
```

**Example** This example uses the permanent SAS data set created in Section 5.2. The data set contains the average lengths, in feet, of selected whales and sharks. The following program contains two procedures: PROC MEANS and PROC PRINT. There are three ODS statements in the program: one to open the LISTING destination, one to turn off procedure titles, and one to close the destination.

```
* Create the text output and remove procedure name;
LIBNAME ocean 'c:\MySASLib';
ODS LISTING FILE = 'c:\MyTextFiles\Marine.lst';
ODS NOPROCTITLE;
PROC MEANS DATA = ocean.marine MEAN MIN
MAX;
CLASS Family;
TITLE 'Whales and Sharks';
RUN;
PROC PRINT DATA = ocean.marine;
RUN;
ODS LISTING CLOSE;
```

Here is what the text file produced by SAS Enterprise Guide looks like when viewed in Microsoft Notepad. The LISTING destination does not use a style for tabular output. Instead, the results are rendered as plain text.

The screenshot shows a Notepad window titled "Marine.lst - Notepad". The menu bar includes File, Edit, Format, View, and Help. The main content displays two tables related to the "Whales and Sharks" dataset.

**Analysis Variable : Length**

Family	N Obs	Mean	Minimum	Maximum
shark	4	20.6250000	0.5000000	40.0000000
whale	4	48.7500000	15.0000000	100.0000000

**Whales and Sharks**

Obs	Name	Family	Length
1	beluga	whale	15.0
2	dwarf	shark	0.5
3	basking	shark	30.0
4	humpback	whale	50.0
5	whale	shark	40.0
6	blue	whale	100.0
7	killer	whale	30.0
8	mako	shark	12.0

You may also be able to create text output without using ODS statements. The options available depend on the interface you use to run SAS. With a little exploration you, will find various menus and tasks for creating, exporting, and saving text output.

## 5.6 Customizing Titles and Footnotes

In ODS output, your style template tells SAS how titles and footnotes should look. However, you can easily change the appearance of titles and footnotes by inserting a few simple options in your TITLE and FOOTNOTE statements.

The general form for a TITLE or FOOTNOTE statement is:

TITLE *options 'text-string-1' options 'text-string-2' ... options 'text-string-n'*;

FOOTNOTE *options 'text-string-1' options 'text-string-2'  
...options 'text-string-n';*

Text can be broken into pieces with different options for each piece. SAS will concatenate text strings just the way you type them, so be sure to include any necessary blanks. Each option applies to the text string that follows, and stays in effect until another value is specified for that option, or until the end of the statement. Here are the main options that you can choose:

COLOR= specifies a color for the text

BCOLOR= specifies a color for the background of th

HEIGHT= specifies the height of the text

JUSTIFY= requests justification

FONT= specifies a font for the text

BOLD makes text bold

ITALIC makes text italic

**Color** The COLOR= option specifies the color of the text. This statement:

```
TITLE COLOR=BLACK 'Black ' COLOR=GRAY  
'Gray ' COLOR=LTGRAY 'Light Gray';
```

produces this title:



Black Gray Light Gr

SAS supports hundreds of colors ranging from primary colors—red—to more esoteric colors—LIGRPR (light grayish purplish red). These colors can be specified by name—BLUE—or by hexadecimal code—#0000FF. In fact, SAS recognizes more than a half-dozen naming schemes for specifying color. (For more information about colors, search for “SAS color-naming schemes” in the SAS Documentation or on the internet.) Names of colors need quotation marks if the name is longer than 8 characters or contains embedded spaces. RGB hexadecimal codes beginning with a pound sign also require quotation marks. If you want to specify colors by name, here is a list of basic colors that you can start with: AQUA, BLACK, BLUE, FUCHSIA, GREEN, GRAY, LIME, MAROON, NAVY, OLIVE, PURPLE, RED, SILVER, TEAL, WHITE, YELLOW.

**Background color** The BCOLOR= option specifies a background color. This statement uses an RGB hexadecimal code:

```
TITLE BCOLOR = '#C0C0C0' 'This Title Has a Gray  
Background';
```

and produces this title:

**This Title Has a Gray Backgr**

You can choose among the same colors as with the COLOR= option.

**Height** To change the height of the text, use the HEIGHT= option where the value of HEIGHT is a number with units of PT, IN, or CM. This statement:

```
TITLE HEIGHT="12PT" 'Small ' HEIGHT=.25IN
```

```
'Medium ' HEIGHT="1CM" 'Large';  
produces this title:
```



**Small Medium Large**

**Justification** You can control justification of text using the JUSTIFY= option, which can have the values LEFT, CENTER, or RIGHT. You can even mix these options within a single statement. This statement:

```
TITLE JUSTIFY=LEFT 'Left ' JUSTIFY=CENTER 'vs.  
' JUSTIFY=RIGHT 'Right';
```

produces this title:

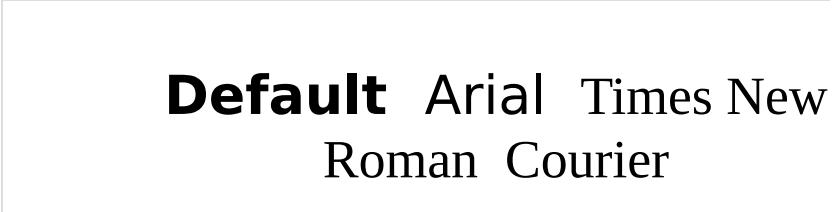


**Left                    vs.**

**Font** Use the FONT= option to specify a font. This statement:

```
TITLE 'Default ' FONT=Arial 'Arial '  
      FONT='Times New Roman' 'Times New Roman '  
      FONT=Courier 'Courier';
```

produces this title:



**Default Arial Times New  
Roman Courier**

The particular fonts available to you depend on your

operating environment and hardware. Courier, Arial, Times, and Helvetica work in most situations.

**Bold and italic** By default, titles are bold. When you change the font, you also turn off the bolding. To turn on bolding, use the BOLD option; to turn on italics use the ITALIC option. There is no option to turn off bolding or italics, so if you want to turn them off, use the FONT= option. Here are the three options together:

```
TITLE FONT = Courier 'Courier ' BOLD 'Courier  
Bold '  
ITALIC 'Courier Bold and Italic';
```

This statement produces this title:

Courier **Bold** *Courier Bold and Italic*

## 5.7 Customizing PROC PRINT with the STYLE= Option

If you want to change the overall look of any output, you can specify a different style template using a STYLE= option in your ODS statement. But what if you want to change the appearance of just the headers, or just one column of your output? With most procedures, you would need to use PROC TEMPLATE to create a custom style template. However, the reporting procedures, PRINT, REPORT, and TABULATE, allow you to change the style of various parts of the table using the STYLE= option in the procedures' own statements.

The general form of the STYLE= option in the PROC PRINT statement is:

```
PROC PRINT STYLE(location-list) = {style-attribute =
```

*value*};

where *location-list* specifies the parts of the table that should take on the style, *style-attribute* is the characteristic that you want to change, and *value* is the way you want the style attribute to look. (See Section 5.11 for a table of attributes and possible values.) For example, the following statement tells SAS to use a gray background for the data in the table.

```
PROC PRINT DATA = mysales STYLE(DATA) =
{BACKGROUND COLOR = GRAY};
```

You can have several *STYLE=* options on one *PROC PRINT* statement, and the same style can apply to several locations. Here are some of the locations that you can specify:

<b>Location</b>	<b>Table region affected</b>
DATA	all the data cells
HEADER	the column headers (variable names)
OBS	the data in the OBS column, or ID column ID statement
OBSHEADER	the header for the OBS or ID column
TOTAL	the data in the totals row produced by a S statement
GRANDTOT AL	the data for the grand total produced by a statement

If you place the STYLE= option in the PROC PRINT statement, the entire table will be affected. For example, if you specify HEADER as the location, then all of the column headers will have the new style. To change the header of just one column, put the STYLE= option in the VAR statement:

```
VAR variable-list / STYLE(location-list) = {style-  
attribute = value};
```

Only the variables listed in the VAR statement will have the specified style. If you want different variables to have different styles, then use multiple VAR statements. Only the DATA and HEADER locations are valid on the VAR statement. You can apply styles to ID statements in the same way.

**Example** The following data are the top five finishers in the men's 5000 meter speed skating event at the 2002 Winter Olympics. The skater's place is followed by his name, country, and time in seconds.

Place	Name	Country	Time
1	Jochem Uytdehaage	Netherlands	374.66
2	Derek Parra	United States	377.98
3	Jens Boden	Germany	381.73
4	Dmitry Shepel	Russia	381.85
5	KC Boutiette	United States	382.97

This program creates a permanent SAS data set named RESULTS. Then a PROC PRINT uses the default destination and style template.

```
LIBNAME skate 'c:\MySASLib';  
PROC IMPORT DATAFILE =  
'c:\MyRawData\Mens5000.csv' OUT = skate.results  
REPLACE;  
RUN;  
PROC PRINT DATA = skate.results;  
ID Place;
```

```

VAR Name Country Time;
TITLE "Men's 5000m Speed Skating";
RUN;

```

Here are the results:

## Men's 5000m Speed Skating

<b>Place</b>	<b>Name</b>	<b>Country</b>
<b>1</b>	Jochem Uytdehaage	Netherlands
<b>2</b>	Derek Parra	United States
<b>3</b>	Jens Boden	Germany
<b>4</b>	Dmitry Shepel	Russia
<b>5</b>	KC Boutiette	United States

The next program also uses the default style template, but the STYLE= option has been added to the PROC PRINT statement giving all the data cells a gray background and white foreground. A STYLE= option was also added to the ID statement to center the values of Place, and to a VAR statement making the values of Name italic.

* Use STYLE= option in PROC, ID, and VAR statements;

PROC PRINT DATA = skate.results

```

STYLE(DATA) = {BACKGROUNDCOLOR =
GRAY COLOR = WHITE};
ID Place / STYLE(DATA) = {TEXTALIGN =
CENTER};
VAR Name / STYLE(DATA) = {FONTSTYLE =

```

ITALIC};

VAR Country Time;

TITLE "Men's 5000m Speed Skating";  
RUN;

Here are the results:

## Men's 5000m Speed Skating

Place	Name	Country
1	<i>Jochem Uytdehaage</i>	Netherlands
2	<i>Derek Parra</i>	United States
3	<i>Jens Boden</i>	Germany
4	<i>Dmitry Shepel</i>	Russia
5	<i>KC Boutiette</i>	United States

## 5.8 Customizing PROC REPORT with the STYLE= Option

With the STYLE= option in an ODS statement you can change the overall style of PROC REPORT output, but you can also use the STYLE= option inside PROC REPORT to change the style of individual features. The STYLE= option in PROC REPORT is similar to the PRINT procedure because you have to specify a location. The general form of the STYLE= option in the PROC REPORT

statement is:

```
PROC REPORT STYLE(location-list) = {style-attribute  
= value};
```

where *location-list* specifies the parts of the table that should take on the style, *style-attribute* is the characteristic that you want to change, and *value* is the way you want the style attribute to look. (See Section 5.11 for a table of attributes and possible values.) For example, to give column headers a green background, you could use this statement:

```
PROC REPORT DATA = mysales STYLE(HEADER) =  
{BACKGROUNDCOLOR = GREEN};
```

You can specify more than one location in a single STYLE= option, and you can have several STYLE= options in one PROC REPORT statement. Here are some of the locations whose appearance you can control in PROC REPORT:

<b>Location</b>	<b>Table region affected</b>
HEADER	column headings
COLUMN	data cells
SUMMAR Y	totals created by SUMMARIZE option in B RBREAK statements

If you put a STYLE= option in a PROC REPORT statement, then it will affect the whole table, for example, all the column headings, all the data cells, or all the summary breaks. You can change part of a report by using

the STYLE= option in other statements. To specify a style for a particular variable, put the STYLE= option in a DEFINE statement. If you use a GROUP or ORDER variable, you may want to add the SPANROWS option to the PROC statement. The SPANROWS option combines cells in the same category into a single cell. These statements tell SAS to use Month as a group variable, and make the background blue for both the header and the data:

```
PROC REPORT DATA = mysales SPANROWS;  
  DEFINE Month / GROUP STYLE(HEADER  
    COLUMN) = {BACKGROUNDCOLOR = BLUE};
```

To specify a style for particular summary breaks, use the STYLE= option in a BREAK or RBREAK statement. This statement tells SAS to use a red background for summary breaks for each value of Month.

```
BREAK AFTER Month / SUMMARIZE  
  STYLE(SUMMARY) = {BACKGROUNDCOLOR =  
    RED};
```

**Example** This example uses the permanent SAS data set created in the preceding section. The data are the top five finishers in the men's 5000 meter speed skating event at the 2002 Winter Olympics. The variables are each skater's place, name, country, and time in seconds.

This program uses the default destination and style template:

```
LIBNAME skate 'c:\MySASLib';  
PROC REPORT DATA = skate.results;  
  COLUMN Country Name Time Place;  
  DEFINE Country / ORDER;  
  TITLE "Men's 5000m Speed Skating";
```

RUN;

Here are the results:

## Men's 5000m Speed Skating

Country	Name	Time
Germany	Jens Boden	38
Netherlands	Jochem Uytdehaage	37
Russia	Dmitry Shepel	38
United States	Derek Parra	37
	KC Boutiette	38

The next program also uses the default style template, but it adds a STYLE= option in the PROC REPORT statement to change the background of all the data cells to gray and the foreground text to white. Then it adds the STYLE= option in the DEFINE statements to italicize the values of Name and to center the values of Place. The SPANROWS option combines the two lines for the United States into one cell.

* Use STYLE= option in PROC and DEFINE statements;

```
PROC REPORT DATA = skate.results SPANROWS
```

```
    STYLE(COLUMN) = {BACKGROUNDCOLOR = GRAY COLOR = WHITE};
```

```
    COLUMN Country Name Time Place;
```

```
    DEFINE Country / ORDER;
```

```
    DEFINE Name / STYLE(COLUMN) = {FONTSTYLE = ITALIC};
```

```
DEFINE Place / STYLE(COLUMN) = {TEXTALIGN  
= CENTER};
```

```
TITLE "Men's 5000m Speed Skating";
```

```
RUN;
```

Here are the results:

## Men's 5000m Speed Skating

Country	Name	Time
Germany	<i>Jens Boden</i>	38
Netherlands	<i>Jochem Uytdehaage</i>	37
Russia	<i>Dmitry Shepel</i>	38
United States	<i>Derek Parra</i>	37
	<i>KC Boutiette</i>	38

## 5.9 Customizing PROC TABULATE with the STYLE= Option

Like the PRINT and REPORT procedures, PROC TABULATE allows you to control the style of individual features with the STYLE= option. However, unlike PROC PRINT and PROC REPORT, you do not have to specify a location. The part of the table affected depends on where you place the STYLE= option. (See Section 5.11 for a table

of style attributes and their possible values.) Here are some of the PROC TABULATE statements that accept the STYLE= option:

Statement	Table region affected
PROC TABULATE	all the data cells
CLASS	class variable name ]
CLASSLEV	class level value hea
TABLE (crossed with elements)	element's data cell
VAR	analysis variable nar

**PROC TABULATE statement** If you place the STYLE= option in the PROC TABULATE statement, all the table's data cells will have the style. For example, if you wanted all the data cells in your table created from the MYSALES SAS data set to have a yellow background, then you would use the following statement:

```
PROC TABULATE DATA = mysales STYLE =
{BACKGROUNDCOLOR = YELLOW};
```

**TABLE statement** If you want some of the data cells to have a different style from the rest, then you need to add the STYLE= option to the TABLE statement and cross the style with the variable or keyword you want to change (similar to having different formats for different parts of the table, as discussed in Section 4.17). Any style assigned in a TABLE statement will override styles assigned in the PROC TABULATE statement. For example, the following

TABLE statement produces a table where the data cells in the ALL column have a red background:

```
TABLE City, Month ALL*{STYLE =  
{BACKGROUNDCOLOR = RED}};
```

**CLASSLEV, VAR, and CLASS statements** The CLASSLEV, VAR, and CLASS statements all require that you place the STYLE= option after a slash (/). Any variable that appears in a CLASSLEV statement must also appear in a CLASS statement. For example, suppose that you had a table with a class variable Month, and you wanted all the values of Month to have a foreground color of green, then you would use this CLASSLEV statement:

```
CLASSLEV Month / STYLE = {COLOR = GREEN};
```

**Example** This example uses the permanent SAS data set created in Section 5.2. The data set contains the average lengths, in feet, of selected whales and sharks.

This program uses the default destination and style template:

```
LIBNAME ocean 'c:\MySASLib';  
  
PROC TABULATE DATA = ocean.marine;  
  
    CLASS Family;  
  
    VAR Length;  
  
    TABLE Family, Length*(Min Max Mean);  
  
    TITLE 'Whales and Sharks';  
  
    RUN;
```

Here are the results:

## Whales and Sharks

--	--

<b>Family</b>	<b>Length</b>	
	<b>Min</b>	<b>Max</b>
<b>shark</b>	0.50	40.00
<b>whale</b>	15.00	100.00

The next program uses the default style template, but adds a STYLE= option to the PROC statement giving the data cells white foreground text and a gray background. Then the STYLE= options in the CLASS and VAR statements apply an italic style to the headings for those variables.

```
* Use STYLE= option in PROC, CLASS, and VAR
statements;
```

```
PROC TABULATE DATA = ocean.marine
```

```
STYLE =
{BACKGROUNDCOLOR =
GRAY COLOR = WHITE};
CLASS Family / STYLE =
{FONTSTYLE = ITALIC};
VAR Length / STYLE =
{FONTSTYLE = ITALIC};
TABLE Family, Length*(Min Max Mean);
```

```
TITLE 'Whales and Sharks';
```

```
RUN;
```

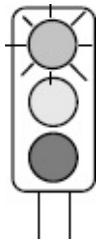
Here are the results:

## Whales and Sharks

--	--

<i><b>Family</b></i>	<i><b>Length</b></i>	
	<b>Min</b>	<b>Max</b>
<b>shark</b>	0.50	40.00
<b>whale</b>	15.00	100.00

## 5.10 Adding Trafficlighting to Your Output



Trafficlighting is a feature that allows you to control the style of cells in the table based on the values of the data in the cells. This way you can draw attention to important values in your report, or highlight relationships between values. Trafficlighting can be used in any of the three reporting procedures: PRINT, REPORT, and TABULATE.

To implement trafficlighting you need to do two things. First, create a user-defined format where the ranges of data values correspond to different values of a style attribute. (See the following section for a table of style attributes and their possible values.) Next, set the style attribute equal to your user-defined format in a STYLE= option. For example, you could create this format:

```
PROC FORMAT;
```

```

VALUE posneg
LOW -< 0 = 'RED'
0-HIGH  = 'BLACK';
RUN;

```

Then with a VAR statement in a PRINT procedure, you could set the value of the COLOR attribute equal to your user-defined format (POSNEG.), like this:

```
VAR Balance / STYLE = {COLOR = posneg.};
```

Then all the data cells for the variable Balance would have red numbers if they are negative and black numbers if they are positive.

**Example** This example uses the permanent SAS data set created in Section 5.7. The data are the top five finishers in the men's 5000 meter speed skating event at the 2002 Winter Olympics. The skater's place is followed by his name, country, and time in seconds.

The following program prints the data using PROC PRINT. The resulting output uses the default destination and the default style.

```

LIBNAME skate 'c:\MySASLib';
PROC PRINT DATA = skate.results;
ID Place;
TITLE "Men's 5000m Speed Skating";
RUN;

```

Here are the results:

### Men's 5000m Speed Skating

Place	Name	Country
1	Jochem Uytdehaage	Netherlands

2	Derek Parra	United States
3	Jens Boden	Germany
4	Dmitry Shepel	Russia
5	KC Boutiette	United States

We can use trafficlighting to give an idea of how these times compare with previous times for this event. Prior to the 2002 Olympics, the world record for the 5000 meter speed skating was 378.72 seconds and the Olympic record was 382.20 seconds. To show which skaters skated faster than these records, the following PROC FORMAT creates a user-defined format, named REC., which assigns the color light gray to times less than the world record, very light gray to times less than the Olympic record, and white to other times. The second VAR statement in the PROC PRINT uses the STYLE= option to set the BACKGROUNDCOLOR attribute of the values for the variable Time equal to the REC. format.

* Create user-defined format for colors;  
**PROC FORMAT;**  
**VALUE rec 0 -< 378.72 = 'LIGHT GRAY'**  
**378.72 -< 382.20 = 'VERY LIGHT GRAY'**  
**382.20 - HIGH = 'WHITE';**

**RUN;**

* Use STYLE= option to apply format in VAR statement;

**PROC PRINT DATA = skate.results;**

**ID Place;**

**VAR Name Country;**

**VAR Time / STYLE = {BACKGROUNDCOLOR =**

rec.};

TITLE "Men's 5000m Speed Skating";  
RUN;

Here is the output showing the different colored backgrounds based on the value of Time. Those skaters who broke the previous world record have a light gray background for Time, those who broke the Olympic record have a very light gray background, and the fifth skater who didn't break any records has a white background.

### Men's 5000m Speed Skating

Place	Name	Country
1	Jochem Uytdehaage	Netherlands
2	Derek Parra	United States
3	Jens Boden	Germany
4	Dmitry Shepel	Russia
5	KC Boutiette	United States

## 5.11 Selected Style Attributes

Attribute	Description	Possible Values
BACKGROUND COLOR	Specifies the background color of the	Any valid color name or hex code.

or BACKGROUND	table or cell.	
BACKGROUNDIMAGE	Specifies a background image to be used for the table or cell. Not valid for RTF.	Any GIF, JPEG, PNG image file ²
COLOR or FOREGROUND	Specifies the color of the text in the cells.	Any valid color name
FLYOVER	Specifies the pop-up text displayed when the cursor is held over the text (HTML)  or if you double-click on the text (PDF).	Any text string enclosed in quotation marks
FONTFAMILY or FONT_FACE	Specifies the font to use for the text	Any valid font name supported by devices supporting Tim

	in the cells.	Courier, Arial, and Helvetica
FONTSIZE or FONT_SIZE	Specifies the relative size of the font for the text in cells. ³	1 to 7
FONTSTYLE or FONT_STYLE	Specifies the style of the font used in the cells.	ITALIC, RC, SLANT (Italic and slant map to the same font)
FONTWEIGHT or FONT_WEIGHT	Specifies the weight of the font used in the cells.	BOLD, MEDIUM, LIGHT
PREIMAGE or POSTIMAGE	Specifies an image that will be inserted either before (PREIMAGE) or	Any GIF, JPEG, or PNG image file (JPEG and PNG are supported only for PREIMAGE)

	after (POSTIMAGE) the text in the cells.	RTF) ²
PRETEXT or POSTTEXT	Specifies text that goes either before (PRETEXT) or after (POSTTEXT)  the text in the cells.	Any text string enclosed in quotation marks
TEXTALIGN or JUST	Specifies the justification of the text  in the cells.	R RIGHT, C CENTER  L LEFT, or (decimal)
URL	Specifies the URL to link to from the  text in the cell. HTML, PDF, and  RTF only.	Any URL

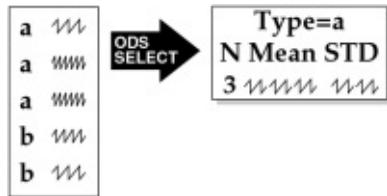
- 
- 1 You can specify colors by name including AQUA, BLACK, BLUE, FUCHSIA, GREEN, GRAY, LIME, MAROON, NAVY, OLIVE, PURPLE, RED, SILVER, TEAL, WHITE, and YELLOW. For an exact color you can use the RGB notation such as #00FF00 for green. To learn more, search for “SAS color-naming schemes” in the SAS Documentation or on the internet.
- 2 For HTML, if you use a simple filename, then the SAS internal browser may not be able to find the file. If you use a complete path and then move the files to a new location, be sure to edit the HTML file to reflect the new location.
- 3 For some destinations, you can specify size in units of measure: cm, in, mm, pt, px (pixels). For example, if you want text that is 24 points, then you would specify FONTSIZE=24pt.

<b>Attribute</b>	<b>STYLE= code</b>	<b>Result</b>
BACKGROUND DCOLOR or BACKGROUND D	STYLE(DATA)=  {BACKGROUNDCO LOR=WHITE};	-----
BACKGROUND DIMAGE	STYLE(DATA)=  {BACKGROUNDIM AGE=  'c:\MyImages\snow.gif'	

	};	
COLOR or FOREGROUN D	STYLE(DATA)=  {COLOR=WHITE};	_____
FLYOVER	STYLE(DATA)=  {FLYOVER='Try it!'};	
FONTFAMILY or  FONT_FACE	STYLE(DATA)=  {FONTFAMILY=COU RIER};	_____
FONTSIZE or  FONT_SIZE	STYLE(DATA)=  {FONTSIZE=2};	_____
FONTSTYLE or  FONT_STYLE	STYLE(DATA)=  {FONTSTYLE=ITALI C};	

FONTWEIGHT or FONT_WEIGHT	STYLE(DATA)=  {FONTWEIGHT=BOLD};	
PREIMAGE or POSTIMAGE	STYLE(DATA)=  {PREIMAGE='SS2.gif'};	
PRETEXT or POSTTEXT	STYLE(DATA)=  {POSTTEXT=' is fun'};	
TEXTALIGN or JUST	STYLE(DATA)=  {TEXTALIGN=RIGHT};	
URL	STYLE(DATA)=  {URL='http://skating.org'};	

## 5.12 Tracing and Selecting Procedure Output



When ODS receives data from a procedure, it combines the data with a table template. Together the data and corresponding table template are called an output object. Many procedures produce just one output object, while others produce several. For most procedures, when you use a BY statement, SAS produces one output object for each BY group. Every output object has a name. You can find the names of output objects by using the ODS TRACE statement. Then you can use an ODS SELECT (or ODS EXCLUDE) statement to choose just the output objects that you want.

**ODS TRACE statement** The ODS TRACE statement tells SAS to print information about output objects in your SAS log. There are two ODS TRACE statements: one to turn on the trace, and one to turn it off. Here is how to use these statements in a program:

```
ODS TRACE ON;  
  the PROC steps you want to trace go here  
RUN;  
ODS TRACE OFF;
```

Notice that the RUN statement comes before the ODS TRACE OFF statement. Unlike most other SAS statements, ODS statements execute immediately—without waiting for a RUN, PROC, or DATA statement. If you put the ODS TRACE OFF statement before the RUN statement, then the trace would turn off before the procedure completes.

**Example** Here are data about varieties of giant tomatoes. Each line of data includes the name of the variety, its color

(red or yellow), the number of days from planting to harvest, and the weight (in pounds) of a typical tomato. Each line of data includes two varieties.

```
Big Zac, red, 80, 5, Delicious, red, 80, 3  
Dinner Plate, red, 90, 2, Goliath, red, 85, 1.5  
Mega Tom, red, 80, 2, Mortgage Lifter, red, 85, 2  
Big Rainbow, yellow, 90, 1.5, Pineapple, yellow, 85, 2
```

The following program creates a permanent SAS data set named GIANT, and then traces PROC MEANS using ODS TRACE ON and ODS TRACE OFF statements:

```
LIBNAME tomatoes 'c:\MySASLib';  
DATA tomatoes.giant;  
  INFILE 'c:\MyRawData\GiantTom.dat' DSD;  
  INPUT Name :$15. Color $ Days Weight @@;  
RUN;  
* Trace PROC MEANS;  
ODS TRACE ON;  
PROC MEANS DATA = tomatoes.giant;  
  BY Color;  
RUN;  
ODS TRACE OFF;
```

If you run this program, you will see the following trace in your SAS log. Because it contains a BY statement, the MEANS procedure produces one output object for each BY group (red and yellow). Notice that these two output objects have the same name, label, and template, but different paths.

---

Output Added:

---

```
Name: Summary  
Label: Summary statistics  
Template: base.summary  
Path: Means.ByGroup1.Summary
```

---

NOTE: The above message was for the following BY group: Color=red

Output Added:

---

Name: Summary  
Label: Summary statistics  
Template: base.summary  
Path: Means.ByGroup2.Summary

---

NOTE: The above message was for the following BY group:

Color=yellow

---

**ODS SELECT statement** Once you know the names of the output objects that the procedure uses, you can use an ODS SELECT (or EXCLUDE) statement to choose just the output objects you want. The general form of an ODS SELECT statement is:

*The PROC step with the output objects you want to select*  
*ODS SELECT output-object-list;*  
*RUN;*

where *output-object-list* is the name, label, or path of one or more output objects separated by spaces. By default, an ODS SELECT statement lasts for only one PROC step, so by placing the SELECT statement after the PROC statement and before the RUN statement, you are sure to capture the correct output. ODS EXCLUDE statements work the same way except you list output objects that you want to eliminate.

**Example** This program runs PROC MEANS again using the GIANT data set, and an ODS SELECT statement to select just the first output object:  
Means.ByGroup1.Summary.

```
* Print only the first BY group;  
PROC MEANS DATA = tomatoes.giant;  
    BY Color;  
ODS SELECT Means.ByGroup1.Summary;
```

RUN;

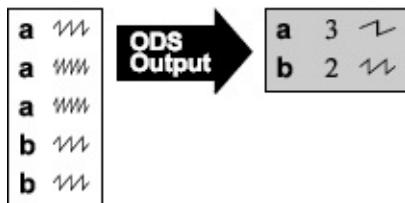
Here are the results containing just the first BY group:

### The MEANS Procedure

Color=red

Variable	N	Mean	Std Dev	Minimum
Days	6	83.33333333	4.0824829	80.0000000
Weight	6	2.58333333	1.2812754	1.5000000

## 5.13 Creating SAS Data Sets from Procedure Output



Sometimes you may want to put the results from a procedure into a SAS data set. Once the results are in a data set, you can merge them with another data set, compute new variables based on the results, or use the results as input for other procedures. Some procedures have OUTPUT statements, or OUT= options, allowing you to save the results as a SAS data set. But with ODS you can save almost any part of procedure output as a SAS data set by sending it to the OUTPUT destination. First you use an ODS TRACE statement (discussed in the previous section) to determine the name of the output object you want. Then you use an ODS OUTPUT statement to send that object to

the OUTPUT destination.

**ODS OUTPUT statement** Here is the general form of a basic ODS OUTPUT statement:

ODS OUTPUT *output-object* = *new-data-set*;

where *output-object* is the name, label, or path of the piece of output you want to save, and *new-data-set* is the name of the SAS data set you want to create.

The ODS OUTPUT statement does not belong to either a DATA or PROC step, but you need to be careful where you put it in your program. The ODS OUTPUT statement opens a SAS data set and waits for the correct procedure output. The data set remains open until the next encounter with the end of a PROC step. Because the ODS OUTPUT statement executes immediately, it will apply to whatever PROC is currently being processed, or it will apply to the next PROC if there is not a current PROC. To ensure that you get the correct output, it is a good idea to put the ODS OUTPUT statement after your PROC statement, and before the next PROC, DATA, or RUN statement.

**Example** This example uses the permanent SAS data set created in the preceding section. The data set contains information about varieties of giant tomatoes including the name of the variety, its color (red or yellow), the number of days from planting to harvest, and the weight (in pounds) of a typical tomato.

To find the name of the desired output object, the following program traces PROC TABULATE using ODS TRACE ON and ODS TRACE OFF statements:

```
LIBNAME tomatoes 'c:\MySASLib';
ODS TRACE ON;
PROC TABULATE DATA = tomatoes.giant;
  CLASS Color;
  VAR Days Weight;
  TABLE Color ALL, (Days Weight) * MEAN;
```

```
RUN;  
ODS TRACE OFF;
```

Here is an excerpt from the SAS log showing the trace produced by PROC TABULATE. PROC TABULATE produces one output object named Table. The names of output objects do not change so you can skip this step if you already know the name of the output object you want to capture.

---

```
Output Added:  
-----  
Name:    Table  
Label:   Table 1  
Data Name: Report  
Path:    Tabulate.Report.Table
```

---

The following program uses an ODS OUTPUT statement to create a temporary SAS data set named TABOUT from the Table output object.

```
PROC TABULATE DATA = tomatoes.giant;  
  CLASS Color;  
  VAR Days Weight;  
  TABLE Color ALL, (Days Weight) * MEAN;  
  TITLE 'Standard TABULATE Output';  
  ODS OUTPUT Table = tabout;  
RUN;
```

Here is the standard tabular result produced by PROC TABULATE:

### Standard TABULATE Output

	Days	
	Mean	

Color	
red	83.33
yellow	87.50
All	84.38

Here is the TABOUT data set created by the ODS OUTPUT statement:

	<b>Color</b>	<b>_TYPE_</b>	<b>_PAGE_</b>	<b>_TABLE_</b>	<b>Days_Mean</b>
1	red	1		1	83.3333
2	yellow	1		1	87.5000
3		0		1	84.3750

# 6

“I usually say, ‘The computer is the dumbest thing on campus. It does exactly what you tell it to; not necessarily what you want. Logic is up to you.’”

NECIA A. BLACK, R.N., PH.D.

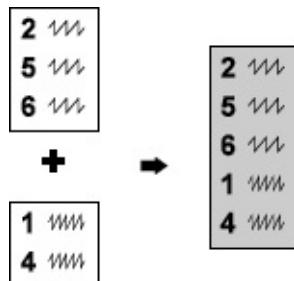
From the SAS-L Listserv,  
May 6, 1994. Reprinted by  
permission of the author.

# CHAPTER 6

## Modifying and Combining SAS Data Sets

- 6.1 Stacking Data Sets Using the SET Statement
- 6.2 Interleaving Data Sets Using the SET Statement
- 6.3 Combining Data Sets Using a One-to-One Match Merge
- 6.4 Combining Data Sets Using a One-to-Many Match Merge
- 6.5 Using PROC SQL to Join Data Sets
- 6.6 Merging Summary Statistics with the Original Data
- 6.7 Combining a Grand Total with the Original Data
- 6.8 Adding Summary Statistics to Data Using PROC SQL
- 6.9 Updating a Master Data Set with Transactions
- 6.10 Using SAS Data Set Options
- 6.11 Tracking and Selecting Observations with the IN= Option
- 6.12 Selecting Observations with the WHERE= Option
- 6.13 Changing Observations to Variables Using PROC TRANSPOSE
- 6.14 Using SAS Automatic Variables

## 6.1 Stacking Data Sets Using the SET Statement



So far in the book, you have seen many examples of using a SET statement to read a single data set. You can also use SET statements to concatenate or stack data sets. This is useful when you want to combine data sets with all or most of the same variables but different observations. For example, you might have data from two different locations or data taken at two separate times, but you need the data together for analysis.

In a DATA step, first specify the name of the new SAS data set in the DATA statement, then list the names of the old data sets you want to combine in the SET statement.

```
DATA new-data-set;  
  SET data-set-1 data-set-n;
```

The number of observations in the new data set will equal the sum of the number of observations in the old data sets. The order of observations is determined by the order of the list of old data sets. If one of the data sets has a variable not contained in the other data sets, then the observations from the other data sets will have missing values for that variable.

**Example** The Fun Times Amusement Park has two entrances where they collect data about their customers. The data file for the south entrance has an S (for south) followed by the customers' Fun Times pass numbers, the sizes of their parties, and their ages. The file for the north

entrance has an N (for north), the same data as the south entrance, plus one more variable for the parking lot where they left their cars (the south entrance has only one lot). The following shows samples of the two data files.

Data for the South entrance:

```
Entrance,PassNumber,PartySize,Age  
S,43,3,27  
S,44,3,24  
S,45,3,2
```

Data for the North entrance:

```
Entrance,PassNumber,PartySize,Age,Lot  
N,21,5,41,1  
N,87,4,33,3  
N,65,2,67,1  
N,66,2,7,1
```

The first two parts of the following program read the comma-delimited data for the south and north entrances into SAS data sets using PROC IMPORT. The third part combines the two SAS data sets using a SET statement. The same DATA step creates a new variable, AmountPaid, which tells how much each customer paid based on their age:

```
PROC IMPORT DATAFILE =  
'c:\MyRawData\South.csv' OUT = southent REPLACE;  
PROC IMPORT DATAFILE =  
'c:\MyRawData\North.csv' OUT = northent REPLACE;  
RUN;  
  
* Create a data set, both, combining northent and  
southent;  
* Create a variable, AmountPaid, based on value of  
variable Age;  
DATA both;  
    SET southent northent;  
    IF Age = . THEN AmountPaid = .;
```

```

ELSE IF Age < 3 THEN AmountPaid = 0;
ELSE IF Age < 65 THEN AmountPaid = 35;
ELSE AmountPaid = 27;
RUN;

```

Here is the SOUTHENT data set:

	<b>Entrance</b>	<b>PassNumber</b>	<b>PartySize</b>
1	S	43	
2	S	44	
3	S	45	

Here is the NORTHENT data set:

	<b>Entrance</b>	<b>PassNumber</b>	<b>PartySize</b>
1	N	21	5
2	N	87	4
3	N	65	2
4	N	66	2

And here is the final data set, BOTH. Notice that this data set has missing values for the variable Lot for all the observations from the south entrance. Because the variable Lot was not in the SOUTHENT data set, SAS assigned missing values to those observations.

	<b>Entrance</b>	<b>PassNumber</b>	<b>PartySize</b>	<b>Age</b>	<b>Lot</b>	
1	S	43	3	27	.	
2	S	44	3	24	.	
3	S	45	3	2	.	
4	N	21	5	41	1	
5	N	87	4	33	3	
6	N	65	2	67	1	
7	N	66	2	7	1	

The following notes appear in the SAS log stating that three observations were read from SOUTHENT, four observations from NORTHENT, and the final data set, BOTH, has seven observations. It is always a good idea to look at the SAS log and make sure that the number of observations and variables in the data sets makes sense.

---

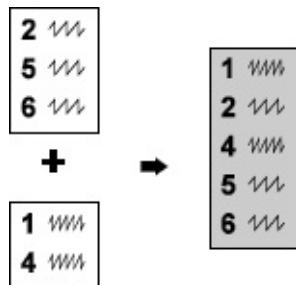
NOTE: There were 3 observations read from the data set  
WORK.SOUTHENT.

NOTE: There were 4 observations read from the data set  
WORK.NORTHENT.

NOTE: The data set WORK.BOTH has 7 observations and 6 variables.

---

## 6.2 Interleaving Data Sets Using the SET Statement



The previous section explained how to stack data sets that have all or most of the same variables but different observations. However, if you have data sets that are already sorted by some important variable, then simply stacking the data sets may unsort the data sets. You could stack the two data sets and then re-sort them using PROC SORT. But if your data sets are already sorted, it is more efficient to preserve that order than to stack and re-sort. All you need to do is use a BY statement with your SET statement. Here's the general form:

```
DATA new-data-set;  
  SET data-set-1 data-set-n;  
    BY variable-list;
```

In a DATA statement, you specify the name of the new SAS data set you want to create. In a SET statement, you list the data sets to be interleaved. Then in a BY statement, you list one or more variables that SAS should use for ordering the observations. The number of observations in the new data set will be equal to the sum of the number of observations in the old data sets. If one of the data sets has a variable that is not contained in the other data sets, then the values of that variable will be set to missing for observations from the other data sets.

Before you can interleave observations, the data sets must be sorted by the BY variables. If one or the other of your

data sets is not already sorted, then use PROC SORT to do the job.

**Example** To show how this is different from stacking data sets, we'll use the amusement park data again. There are two comma-separated data files, one for the south entrance and one for the north. For every customer, the park collects the following data: the entrance (S or N), the customer's Fun Times pass number, size of that customer's party, and age. For customers entering from the north, the data set also includes parking lot number. Notice that the data for the south entrance are already sorted by pass number, but the data for the north entrance are not.

Data for the South entrance:

```
Entrance,PassNumber,PartySize,Age  
S,43,3,27  
S,44,3,24  
S,45,3,2
```

Data for the North entrance:

```
Entrance,PassNumber,PartySize,Age,Lot  
N,21,5,41,1  
N,87,4,33,3  
N,65,2,67,1  
N,66,2,7,1
```

Instead of stacking the two data sets, this program interleaves the data sets by pass number. This program first reads the data for the south entrance and north entrance using PROC IMPORT. Then the program sorts the north entrance data. In the final DATA step, SAS combines the two data sets, SOUTHENT and NORTHENT, creating a new data set named INTERLEAVE. The BY statement tells SAS to combine the data sets by PassNumber:

```
PROC IMPORT DATAFILE =  
'c:\MyRawData\South.csv' OUT = southent REPLACE;  
PROC IMPORT DATAFILE =
```

```

'c:\MyRawData\North.csv' OUT = northent REPLACE;
RUN;
PROC SORT DATA = northent;
    BY PassNumber;
RUN;
* Interleave observations by PassNumber;
DATA interleave;
    SET southent northent;
    BY PassNumber;
RUN;

```

Here is the SOUTHEN data set:

	<b>Entrance</b>	<b>PassNumber</b>	<b>PartySize</b>
1	S	43	
2	S	44	
3	S	45	

Here is the NORTHEN data set after sorting by  
PASSNUMBER:

	<b>Entrance</b>	<b>PassNumber</b>	<b>PartySize</b>
1	N	21	5
2	N	65	2
3	N	66	2
4	N	87	4

---

And here is the data set named INTERLEAVE. Notice that the observations have been interleaved so that the final data set is sorted by PassNumber:

	<b>Entrance</b>	<b>PassNumber</b>	<b>PartySize</b>
1	N	21	5
2	S	43	3
3	S	44	3
4	S	45	3
5	N	65	2
6	N	66	2
7	N	87	4

The following notes appear in the SAS log stating that three observations were read from SOUTHENT, four observations from NORTHENT, and the final data set, INTERLEAVE, has seven observations. It is always a good idea to look at the SAS log and make sure that the number of observations and variables in the data sets makes sense.

---

NOTE: There were 3 observations read from the data set  
WORK.SOUTHENT.

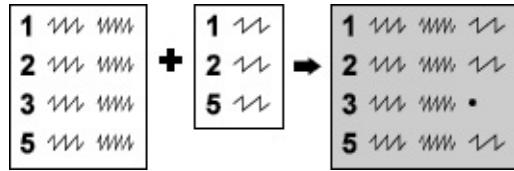
NOTE: There were 4 observations read from the data set  
WORK.NORTHENT.

NOTE: The data set WORK.INTERLEAVE has 7 observations

and 6 variables.

---

## 6.3 Combining Data Sets Using a One-to-One Match Merge



When you want to match observations from one data set with observations from another, you can use the MERGE statement in the DATA step. If you know the two data sets are in EXACTLY the same order, you don't have to have any common variables between the data sets. Typically, however, you will want to have, for matching purposes, a common variable or several variables which taken together uniquely identify each observation. This is important.

Having a common variable to merge by ensures that the observations are properly matched. For example, to merge patient data with billing data, you would use the patient ID as a matching variable. Otherwise, you risk getting Mary Smith's visit to the obstetrician mixed up with Matthew Smith's visit to the optometrist.

Merging SAS data sets is a simple process. First, if the data are not already sorted, use the SORT procedure to sort all data sets by the common variables. Then, in the DATA statement, name the new SAS data set to hold the results and follow with a MERGE statement listing the data sets to be combined. Use a BY statement to indicate the common variables:

```
DATA new-data-set;  
MERGE data-set-1 data-set-n;  
BY variable-list;
```

If you merge two data sets, and they have variables with the

same names—besides the BY variables—then variables from the second data set will overwrite any variables that have the same name in the first data set.

**Example** A Belgian chocolatier keeps track of the number of each type of chocolate sold each day. The code for each chocolate and the number of pieces sold that day are kept in a file. In a separate file she keeps the names and descriptions of each chocolate as well as the code. In order to print the day's sales along with the descriptions of the chocolates, the two files must be merged together using the code as the common variable. Here is a sample of the data:

Sales data		Descriptions data		
Code	Sold	Code	Name	Description
C865	15	A206	Mokka	Coffee buttercream in dark chocolate
K086	9	A536	Walnoot	Walnut halves in dark chocolate
A536	21	B713	Frambozen	Raspberry marzipan in milk chocolate
S163	34	C865	Vanille	Vanilla-flavored rolled in hazelnuts
K014	1	K014	Kroon	Milk chocolate with mint cream
A206	12	K086	Koning	Hazelnut paste in dark chocolate
B713	29	M315	Pyramide	White with dark chocolate trimming
S163	Orbais			Chocolate cream in dark chocolate

The first two parts of the following program read the tab-delimited files for the descriptions and sales data using PROC IMPORT. The descriptions data are already sorted by Code, so we don't need to use PROC SORT. The sales data are not sorted, so a PROC SORT follows the PROC

IMPORT step for the SALES data set. (If you attempt to merge data that are not sorted, SAS will refuse and give you this error message: ERROR: BY variables are not properly sorted.)

```
PROC IMPORT DATAFILE =
  'c:\MyRawData\Chocolate.txt' OUT = names
  REPLACE;
PROC IMPORT DATAFILE =
  'c:\MyRawData\Chocsales.txt' OUT = sales REPLACE;
  RUN;
PROC SORT DATA = sales;
  BY Code;
  RUN;
* Merge data sets by Code;
DATA chocolates;
  MERGE sales names;
  BY Code;
  RUN;
```

The final part of the program creates a data set named CHOCOLATES by merging the SALES data set and the NAMES data set. The common variable Code in the BY statement is used for matching purposes. Here is the final data set after merging:

	<b>Code</b>	<b>Sold</b>	<b>Name</b>	<b>Description</b>
1	A206	12	Mokka	Coffee buttercream in dark chocolate
2	A536	21	Walnoot	Walnut halves in dark chocolate
3	B713	29	Frambozen	Raspberry marzipan in milk chocolate
4	C865	15	Vanille	Vanilla-flavored rolled in hazelnut

5	K014	1	Kroon	Milk chocolate with mint cre
6	K086	9	Koning	Hazelnut paste in dark choc
7	M315	.	Pyramide	White with dark chocolate tr
8	S163	34	Orbais	Chocolate cream in dark choc

Notice that the final data set has a missing value for Sold in the seventh observation. This is because there were no sales for the Pyramide chocolate. All observations from both data sets were included in the final data set whether they had a match or not. In SQL terms, this is called a full outer join. SAS can do all types of joins using the IN= option (Section 6.11), or PROC SQL (Section 6.5).

The following notes appear in the SAS log stating that seven observations were read from SALES, eight observations from NAMES, and the final data set, CHOCOLATES, has eight observations. It is always a good idea to look at the SAS log and make sure that the number of observations and variables in the data sets makes sense.

---

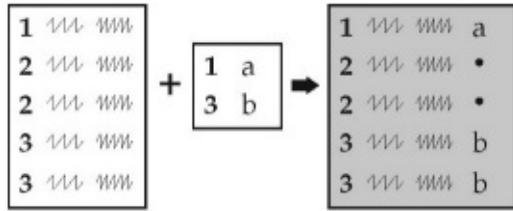
NOTE: There were 7 observations read from the data set  
WORK.SALES.

NOTE: There were 8 observations read from the data set WORK.NAMES.

NOTE: The data set WORK.CHOCOLATES has 8 observations  
and 4 variables.

---

## 6.4 Combining Data Sets Using a One-to-Many Match Merge



Sometimes you need to combine two data sets by matching one observation from one data set with more than one observation in another. Suppose you had data for every state in the U.S. and wanted to combine it with data for every county. This would be a one-to-many match merge because each state observation matches with many county observations.

The statements for a one-to-many match merge are identical to the statements for a one-to-one match merge:

```
DATA new-data-set;
  MERGE data-set-1 data-set-2;
    BY variable-list;
```

The order of the data sets in the MERGE statement does not affect the matching. In other words, a one-to-many merge will match the same observations as a many-to-one merge.

Before you merge two data sets, they must be sorted by one or more common variables. If your data sets are not already sorted in the proper order, then use PROC SORT to do the job.

You cannot do a one-to-many merge without a BY statement. SAS uses the variables listed in the BY statement to decide which observations belong together. Without a BY statement, SAS simply joins together the first observation from each data set, then the second observation from each data set, and so on. In other words, SAS performs a one-to-one unmatched merge, which is probably not what you want.

If you merge two data sets, and they contain variables with the same name—besides the BY variables—then you

should either rename the variables or drop one of the duplicate variables. Otherwise, variables from the second data set may overwrite variables having the same name in the first data set. For example, if you merge two data sets that each contain a variable named BirthDate, then you could rename the variables (perhaps as BirthDate1 and BirthDate2), or you could simply drop BirthDate from one data set. Then the values of BirthDate will not overwrite each other. You can use the RENAME= and DROP= data set options (discussed in Section 6.10) to prevent the overwriting of data values.

**Example** A distributor of athletic shoes is putting all its shoes on sale at 15% to 30% off the regular price. The distributor has two data files, one with information about each type of shoe and one with the discount factors. The first file contains one record for each shoe with values for style, type of exercise (basketball, running, walking, or cross-training), and regular price. The second file contains one record for each type of exercise and its discount. Here are the two raw data files:

<b>Shoes</b>		<b>Discount</b>
<b>data</b>		
<b>data</b>		
Max Flight	running 142.99	b-ball .15
Zip Fit Leather	walking 83.99	c-train .25
Zoom Airborne	running 112.99	running .30
Light Step	walking 73.99	walking .20
Max Step Woven	walking 75.99	
Zip Sneak	c-train 92.99	

To find the sale price, the following program combines the two data files:

```
LIBNAME athshoes 'c:\MySASLib';
DATA athshoes.shoedata;
  INFILE 'c:\MyRawData\Shoe.dat';
    INPUT Style $ 1-15 ExerciseType $ RegularPrice;
RUN;
```

```

PROC SORT DATA = athshoes.shoedata OUT =
regular;
    BY ExerciseType;
RUN;
DATA athshoes.discount;
    INFILE 'c:\MyRawData\Disc.dat';
        INPUT ExerciseType $ Adjustment;
RUN;
* Perform many-to-one match merge;
DATA prices;
    MERGE regular athshoes.discount;
        BY ExerciseType;
        NewPrice = ROUND(RegularPrice - (RegularPrice *
Adjustment), .01);
RUN;

```

The first DATA step reads the shoes data into a SAS data set, then PROC SORT sorts the data by ExerciseType and creates a new temporary data set named REGULAR. The second DATA step reads the price adjustments, creating a permanent data set named DISCOUNT. This data set is already arranged by ExerciseType, so it doesn't have to be sorted. The third DATA step creates a data set named PRICES, merging the first two data sets by ExerciseType, and computes a variable called NewPrice. The final data set looks like this:

	<b>Style</b>	<b>ExerciseType</b>	<b>RegularPric e</b>	<b>Adjustm t</b>
1		b-ball	.	0
2	Zip Sneak	c-train	92.99	0
3	Max Flight	running	142.99	0

4	Zoom Airborne	running	112.99	0
5	Zip Fit Leather	walking	83.99	0
6	Light Step	walking	73.99	0
7	Max Step Woven	walking	75.99	0

Notice that the values for Adjustment from the DISCOUNT data set are repeated for every observation in the REGULAR data set with the same value of ExerciseType. Also, the first observation contains missing values for variables from the REGULAR data set. All observations from both data sets were included in the final data set whether they had a match or not. In SQL, this is called a full outer join. SAS can do all types of joins using the IN= option (Section 6.11) or PROC SQL (Section 6.5).

The following notes appear in the SAS log stating that six observations were read from REGULAR, four observations from DISCOUNT, and the final data set, PRICES, has seven observations.

---

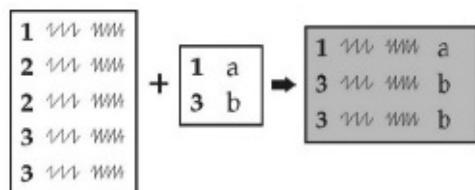
NOTE: There were 6 observations read from the data set  
WORK.REGULAR.

NOTE: There were 4 observations read from the data set  
ATHSHOES.DISCOUNT.

NOTE: The data set WORK.PRICES has 7 observations and 5  
variables.

---

## 6.5 Using PROC SQL to Join Data Sets



There are many different ways that you can join data sets using PROC SQL. If you are familiar with SQL, then you will be glad to know that you can use standard SQL syntax in PROC SQL. It is possible to perform full outer joins (keeping all observations from both data sets), right joins (keeping all matching observations plus all non-matching observations from the data set listed on the right), and left joins (keeping all matching observations plus all non-matching observations from the data set listed on the left). This section describes how to do an inner join keeping only observations that match.

One advantage of using PROC SQL over a DATA step merge, is that the data sets do not need to be sorted before performing the join. This is because initially PROC SQL matches every observation in the first data set with every observation in the second data set (called a Cartesian product), and then determines which observations to keep.

**The WHERE clause** The following is the general syntax for an inner join to create a new data set (keeping all variables) using a WHERE clause. It starts with the PROC SQL statement, followed by several clauses (CREATE TABLE, SELECT, FROM, and WHERE), and ends with a QUIT statement. Notice that only the last clause (WHERE in this case) ends in a semicolon.

```
PROC SQL;
  CREATE TABLE new-data-set AS
    SELECT *
      FROM data-set-1, data-set-2
      WHERE data-set-1.common-variable = data-set-2.common-variable;
    QUIT;
```

The CREATE TABLE clause tells SAS the name of the new data set to create. If you omit this clause, then you will only get a display of the results and not a new data set. The SELECT * clause says to keep all variables from both data sets listed in the FROM clause. If you do not want all

the variables, you can list the ones you want (separated by commas) in place of the asterisk (*). The WHERE clause specifies which observations to keep. Usually, you will have a variable that is common to both data sets to use for matching. In PROC SQL, the variables do not need to have the same names, but they should have some of the same values. Variable names in this case must include the name of the data set and variable name separated by a period.

**The INNER JOIN clause** The following general syntax shows another way to perform an inner join in PROC SQL using the INNER JOIN and ON clauses.

```
PROC SQL;
  CREATE TABLE new-data-set AS
    SELECT *
      FROM data-set-1
        INNER JOIN data-set-2
          ON data-set-1.common-variable = data-set-2.common-variable;
  QUIT;
```

Using this syntax, the ON clause tells SAS which variables to use to match observations, and the INNER JOIN clause tells SAS to keep only observations that appear in both data sets. It doesn't matter if you use the WHERE clause or the INNER JOIN clause syntax, the results are the same.

**Example** This example uses the SAS data sets created in the previous section. A distributor of shoes has two data files. The SHOEDATA data set contains one record for each type of shoe with values for style, type of exercise, and regular price. The DISCOUNT data set contains one record for each type of exercise and its discount. Here are the two SAS data sets:

Shoedata				Discount	

	<b>Style</b>	<b>Exercise Type</b>	<b>Regular Price</b>			<b>Exercise price</b>
1	Max Flight	running	142.99	1		b-ball
2	Zip Fit Leather	walking	83.99	2		c-train
3	Zoom Airborne	running	112.99	3		running
4	Light Step	walking	73.99	4		walking
5	Max Step Woven	walking	75.99			
6	Zip Sneak	c-train	92.99			

The following program joins the two data sets using an inner join:

```

LIBNAME athshoes 'c:\MySASLib';
* Perform an inner join using PROC SQL;
PROC SQL;
  CREATE TABLE prices AS
  SELECT *
    FROM athshoes.shoedata, athshoes.discount
   WHERE shoedata.ExerciseType =
discount.ExerciseType;
QUIT;

```

The PROC SQL step creates a data set named PRICES, joining the SHOEDATA and DISCOUNT data sets and keeping only those observations where ExerciseType is the

same in both data sets. Notice that the SHOEDATA data set is not sorted by exercise type, and it does not need to be sorted before performing the SQL join. The final data set PRICES looks like this:

	<b>Style</b>	<b>ExerciseType</b>	<b>RegularPrice</b>
1	Max Flight	running	142.99
2	Zip Fit Leather	walking	83.99
3	Zoom Airborne	running	112.99
4	Light Step	walking	73.99
5	Max Step Woven	walking	75.99
6	Zip Sneak	c-train	92.99

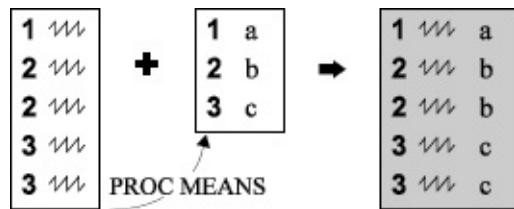
As with the DATA step merge example in the previous section, the values for Adjustment from the DISCOUNT data set are repeated for every observation in the SHOEDATA data set with the same value of ExerciseType. However, the result of the SQL join does not include the observation where ExerciseType is “b-ball” in the DISCOUNT data set because there is no corresponding observation in the SHOEDATA data set. Because the variable ExerciseType is common to both data sets, when you run this program the following message will appear in the SAS log.

---

WARNING: Variable ExerciseType already exists on file  
WORK.PRICES.

---

## 6.6 Merging Summary Statistics with the Original Data



Once in a while you need to combine summary statistics with your data, such as when you want to compare each observation to the group mean, or when you want to calculate a percentage using the group total. To do this, summarize your data using PROC MEANS, and put the results in a new data set. Then merge the summarized data back with the original data using a one-to-many match merge. (You can also use PROC SQL to combine summary statistics with original data. See Section 6.8.)

**Example** A distributor of athletic shoes is considering doing a special promotion for the top selling styles. The vice president of marketing has asked you to produce a report. The report should be divided by type of exercise (running, walking, or cross-training) and show the percentage of sales for each style within its type. For each shoe, the tab-delimited data file contains the style name, type of exercise, and total sales for the last quarter:

Style	ExerciseType	Sales
Max Flight	running	1930
Zip Fit Leather	walking	2250
Zoom Airborne	running	4150
Light Step	walking	1130
Max Step Woven	walking	2230
Zip Sneak	c-train	1190

Here is the program:

```

LIBNAME athshoes 'c:\MySASLib';
PROC IMPORT DATAFILE =
  'c:\MyRawData\Shoesales.txt'
    OUT = athshoes.shoesales REPLACE;
RUN;
PROC SORT DATA = athshoes.shoesales OUT = shoes;
  BY ExerciseType;
RUN;
* Summarize sales by ExerciseType;
PROC MEANS NOPRINT DATA = shoes;
  VAR Sales;
  BY ExerciseType;
  OUTPUT OUT = summarydata SUM(Sales) = Total;
RUN;
* Merge totals with the original data set;
DATA shoessummary;
  MERGE shoes summarydata;
  BY ExerciseType;
  Percent = Sales / Total * 100;
RUN;

PROC PRINT DATA = shoessummary LABEL;
  ID ExerciseType;
  VAR Style Sales Total Percent;
  LABEL Percent = 'Percent By Type';
  TITLE 'Sales Share by Type of Exercise';
RUN;

```

This program is long but straightforward. It starts by reading the tab-delimited file using PROC IMPORT, and it sorts the data with PROC SORT. Then it summarizes the data with PROC MEANS by the variable ExerciseType. The OUTPUT statement tells SAS to create a new data set named SUMMARYDATA (shown below), containing a variable named Total, which equals the sum of the variable Sales. The NOPRINT option tells SAS not to print the standard PROC MEANS report.

	<b>ExerciseType</b>	<b>_TYPE_</b>	<b>_FREQ_</b>
1	c-train		0
2	running		0
3	walking		0

In the last part of the program, the data set SHOES is merged with SUMMARYDATA to make a new data set, SHOESUMMARY. This DATA step computes a new variable called Percent. Then the PROC PRINT writes the final report with percentage of sales by ExerciseType for each shoe style.

### Sales Share by Type of Exercise

<b>ExerciseType</b>	<b>Style</b>	<b>Sales</b>	<b>Total</b>
<b>c-train</b>	Zip Sneak	1190	11
<b>running</b>	Max Flight	1930	60
<b>running</b>	Zoom Airborne	4150	60
<b>walking</b>	Zip Fit Leather	2250	56
<b>walking</b>	Light Step	1130	56

The following notes appear in the SAS log stating that six observations were read from SHOES, three observations from SUMMARYDATA, and the final data set, SHOESUMMARY, has six observations.

---

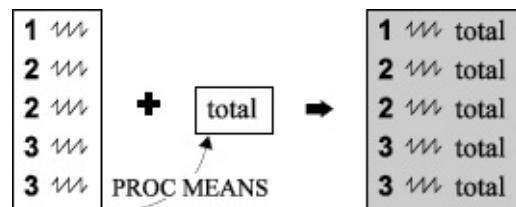
NOTE: There were 6 observations read from the data set  
WORK.SHOES.

NOTE: There were 3 observations read from the data set  
WORK.SUMMARYDATA.

NOTE: The data set WORK.SHOESUMMARY has 6  
observations and 7 variables.

---

## 6.7 Combining a Grand Total with the Original Data



You can use the MEANS procedure to create a data set containing a grand total rather than BY group totals. But you cannot use a MERGE statement to combine a grand total with the original data because there is no common variable to merge by. Luckily, there are other ways to combine grand totals with your original data. You can use PROC SQL (as described in the next section) or you can use two SET statements, like this:

```
DATA new-data-set;  
  IF _N_ = 1 THEN SET summary-data-set;  
  SET original-data-set;
```

In this DATA step, *original-data-set* is the data set with more than one observation (the original data) and *summary-data-set* is the data set with a single observation (the grand total). SAS reads *original-data-set* in a normal SET statement, simply reading the observations in a straightforward way. SAS also reads *summary-data-set* with a SET statement but only in the first iteration of the DATA step—when the SAS automatic variable `_N_` equals 1. (SAS automatic variables are described in more detail in Section 6.14.) SAS then retains the values of variables from *summary-data-set* for all observations in *new-data-set*.

This works because variables that are read with a SET statement are automatically retained. Normally, you don't notice this because the retained values are overwritten by the next observation. But in this case the variables from *summary-data-set* are read once at the first iteration of the DATA step and then retained for all other observations. The effect is similar to a RETAIN statement (discussed in Section 3.15). This technique can be used any time you want to combine a single observation with many observations, without a common variable.

**Example** To show how this is different from merging BY group summary statistics with original data, we'll use the SHOESALES data set created in the previous section. A distributor of athletic shoes is considering doing a special promotion for the top-selling styles. The vice president of marketing asks you to produce a report showing the percentage of total sales for each style. For each style of shoe, the data set contains the style, type of exercise, and sales for the last quarter:

	<b>Style</b>	<b>ExerciseType</b>
1	Max Flight	running

2	Zip Fit Leather	walking
3	Zoom Airborne	running
4	Light Step	walking
5	Max Step Woven	walking
6	Zip Sneak	c-train

Here is the program:

```

LIBNAME athshoes 'c:\MySASLib';
* Output grand total of sales to a data set;
PROC MEANS NOPRINT DATA = athshoes.shoesales;
  VAR Sales;
  OUTPUT OUT = summarydata SUM(Sales) =
GrandTotal;
RUN;
* Combine the grand total with the original data;
DATA shoesummary;
  IF _N_ = 1 THEN SET summarydata;
  SET athshoes.shoesales;
  Percent = Sales / GrandTotal * 100;
RUN;
PROC PRINT DATA = shoesummary;
  ID Style;
  VAR ExerciseType Sales GrandTotal Percent;
  TITLE 'Overall Sales Share';
RUN;

```

This program starts with a PROC MEANS which creates a variable named GrandTotal, which is equal to the sum of Sales. This will be a grand total because there is no BY or CLASS statement. Here is the new SUMMARYDATA data

set:

	<u>TYPE</u>	<u>FREQ</u>	<u>GrandTotal</u>
1		0	6

The second DATA step combines the original data with the grand total using two SET statements and then computes the variable Percent using the grand total data. Here is the result of the PROC PRINT of the final data set SHOESUMMARY:

### Overall Sales Share

<b>Style</b>	<b>ExerciseType</b>	<b>Sales</b>	<b>GrandTotal</b>
<b>Max Flight</b>	running	1930	12000
<b>Zip Fit Leather</b>	walking	2250	12000
<b>Zoom Airborne</b>	running	4150	12000
<b>Light Step</b>	walking	1130	12000
<b>Max Step Woven</b>	walking	2230	12000
<b>Zip Sneak</b>	c-train	1190	12000

The following notes appear in the SAS log stating that one observation was read from SUMMARYDATA, six

observations from SHOESALES, and the final data set, SHOESUMMARY, has six observations.

---

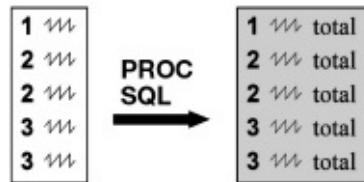
NOTE: There were 1 observations read from the data set WORK.SUMMARYDATA.

NOTE: There were 6 observations read from the data set ATHSHOES.SHOESALES.

NOTE: The data set WORK.SHOESUMMARY has 6 observations and 7 variables.

---

## 6.8 Adding Summary Statistics to Data Using PROC SQL



The previous two sections described how you can add summary statistics to data by first computing the statistics using PROC MEANS, then combining those results with the original data. This section describes how you can use PROC SQL to get the same results. Here is the general form of a PROC SQL to create a new table that includes a summary variable along with the original data. It starts with a PROC SQL statement followed by several clauses (CREATE TABLE, SELECT, FROM, and GROUP BY), and ends with a QUIT statement. Notice that only the last clause (GROUP BY) ends in a semicolon.

```
PROC SQL;  
  CREATE TABLE new-data-set AS  
    SELECT *, summary-statistic(variable-name) AS  
      new-variable-name  
    FROM data-set  
    GROUP BY variable-name;  
  QUIT;
```

The CREATE TABLE clause names the new data set to

create. If you omit this clause, then you will only get a report showing the results and no new data set. The FROM clause specifies the old data set. The GROUP BY clause specifies the name of the variable to use for grouping the statistics. If you omit the GROUP BY clause, then the statistics will be summarized over the entire data set.

The SELECT clause is where you specify which old variables to keep and where you create the new summary variables. You can either keep all variables from the old data set by using an asterisk (*), or list the variables to keep separated by commas. You can compute summary statistics in the SELECT clause including MEAN, SUM, FREQ, MIN, and MAX. To create a summary variable, add a specification to the SELECT clause. For example, to create a new variable named AvgTemp, which is the mean value for the variable Temp, you would add this to the SELECT clause:

```
MEAN(Temp) AS AvgTemp
```

You can also calculate new variables. Simply add the expression to the SELECT statement and give the new variable a name using AS. For example, to create a new variable Minutes from the variable Hours, you would add this expression to the SELECT clause:

```
Hours * 60 AS Minutes
```

**Example** To illustrate how to use PROC SQL to create summarized variables, we'll use the same data as in the previous section about athletic shoe sales:

	<b>Style</b>	<b>ExerciseType</b>
1	Max Flight	running
2	Zip Fit Leather	walking

3	Zoom Airborne	running
4	Light Step	walking
5	Max Step Woven	walking
6	Zip Sneak	c-train

The following program uses PROC SQL to create a new SAS data set named SHOESUMS, which contains two new variables TotalByType and PercentByType.

```

LIBNAME athshoes 'c:\MySASLib';
/*Create summary variables by exercise type;
PROC SQL;
CREATE TABLE shoesums AS
SELECT *, SUM(Sales) AS TotalByType,
      (Sales/SUM(Sales))*100 AS PercentByType
FROM athshoes.shoesales
GROUP BY ExerciseType;
QUIT;
```

The asterisk (*) in the SELECT clause keeps all the variables from the SHOESALES data set. Next, a new variable, TotalByType, is created that is the sum of the old variable Sales. Finally, a new variable, PercentByType, is created using an expression. Because there is a GROUP BY clause, the variable Sales will be summed over unique values of the variable ExerciseType and the data will be ordered by ExerciseType. Here is the new data set SHOESUMS:

	<b>Style</b>	<b>ExerciseType</b>	<b>Sales</b>	<b>TotalByType</b>	P

1	Zip Sneak	c-train	1190	1190
2	Zoom Airborne	running	4150	6080
3	Max Flight	running	1930	6080
4	Max Step Woven	walking	2230	5610
5	Zip Fit Leather	walking	2250	5610
6	Light Step	walking	1130	5610

If you want to create grand totals instead of totals by ExerciseType, then simply omit the GROUP BY clause, as in the following example:

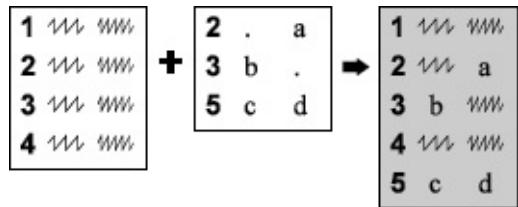
```
*Create summary variables for whole data set;
PROC SQL;
  CREATE TABLE shoetotal AS
    SELECT *, SUM(Sales) AS GrandTotal,
           (Sales/SUM(Sales))*100 AS Percent
    FROM athshoes.shoessales;
QUIT;
```

Here is the data set SHOETOTAL that is created by the above program.

	<b>Style</b>	<b>ExerciseType</b>	<b>Sales</b>	<b>GrandTotal</b>
1	Max Flight	running	1930	12
2	Zip Fit Leather	walking	2250	12

3	Zoom Airborne	running	4150	12
4	Light Step	walking	1130	12
5	Max Step Woven	walking	2230	12
6	Zip Sneak	c-train	1190	12

## 6.9 Updating a Master Data Set with Transactions



The UPDATE statement is used far less than the MERGE statement, but it is just right for those times when you have a master data set that must be updated with bits of new information. A bank account is a good example of this type of transaction-oriented data, since it is regularly updated with credits and debits.

The UPDATE statement is similar to the MERGE statement, because both combine data sets by matching observations on common variables. However, there are critical differences:

- ◆ First, with the UPDATE statement, the resulting master data set always has just one observation for each unique value of the common variables. That way, you don't get a new observation for your bank account every time you deposit a paycheck.
- ◆ Second, missing values in the transaction data set do not overwrite existing values in the master data set.

That way, you are not obliged to enter your address and tax ID number every time you make a withdrawal.

Here is the general form of a DATA step with the UPDATE statement:

```
DATA master-data-set;  
  UPDATE master-data-set transaction-data-set;  
  BY variable-list;
```

Here are a few points to remember about the UPDATE statement. You can specify only two data sets: one master and one transaction. Both data sets must be sorted by their common variables. Also, the values of those BY variables must be unique in the master data set. Using the bank example, you could have many transactions for a single account, but only one observation per account in the master data set.

**Example** A hospital maintains a master database with information about patients. A sample appears below. Each record contains the patient's account number, last name, address, date of birth, sex, insurance code, and the date that the patient's information was last updated.

620135 Smith	234 Aspen St.	12-21-1975	m	CBC
02-16-2005				
645722 Miyamoto	65 3rd Ave.	04-03-1936	f	MCR
05-30-1999				
645739 Jensvold	505 Glendale Ave.	06-15-1960	f	HLT
09-23-2006				
874329 Kazoyan	76-C La Vista	.	.	. MCD 01-15-
2020				

Whenever a patient is admitted to the hospital, the admissions staff check the data for that patient. They create a transaction record for every new patient and for any returning patients whose status has changed. Here are three transactions:

```
620135 . . . . . HLT 06-15-2020
874329 . . . . 04-24-1954 m . 06-15-2020
235777 Harman 5656 Land Way 01-18-2000 f MCD
06-15-2020
```

The first transaction is for a returning patient whose insurance has changed. The second transaction fills in missing information for a returning patient. The last transaction is for a new patient who must be added to the database.

Since master data sets are updated frequently, they are usually saved as permanent SAS data sets. This program creates a master data set named PATIENTMASTER.

```
LIBNAME records 'c:\MySASLib';
DATA records.patientmaster;
INFILE 'c:\MyRawData\Admit.dat';
INPUT Account LastName $ 8-16 Address $ 17-34
      BirthDate MMDDYY10. Sex $ InsCode $ 48-50
      @52 LastUpdate MMDDYY10.:;
FORMAT BirthDate LastUpdate Date9.:;
RUN;
```

The next program reads the transaction data and sorts them with PROC SORT. Then it adds the transactions to PATIENTMASTER with an UPDATE statement. The master data set is already sorted by Account and, therefore, doesn't need to be sorted again:

```
LIBNAME records 'c:\MySASLib';
DATA transactions;
INFILE 'c:\MyRawData\NewAdmit.dat';
INPUT Account LastName $ 8-16 Address $ 17-34
      BirthDate MMDDYY10. Sex $ InsCode $ 48-50
      @52 LastUpdate MMDDYY10.:;
RUN;
PROC SORT DATA = transactions;
BY Account;
RUN;
```

```

* Update patient data with transactions;
DATA records.patientmaster;
  UPDATE records.patientmaster transactions;
  BY Account;
RUN;

```

Here is the PATIENTMASTER data set after updating:

	<b>Accou nt</b>	<b>LastNa me</b>	<b>Address</b>	<b>BirthDate</b>	<b>Se x</b>	<b>InsC e</b>
1	235777	Harman	5656 Land Way	18JAN200 0	f	MCD
2	620135	Smith	234 Aspen St.	21DEC197 5	m	HLT
3	645722	Miyamot o	65 3rd Ave.	03APR193 6	f	MCR
4	645739	Jensvold	505 Glendale Ave.	15JUN196 0	f	HLT
5	874329	Kazoyan	76-C La Vista	24APR195 4	m	MCD

The following notes appear in the SAS log stating that four observations were read from PATIENTMASTER, three observations from TRANSACTIONS, and the final data set, PATIENTMASTER, has five observations.

---

NOTE: There were 4 observations read from the data set  
RECORDS.PATIENTMASTER.

NOTE: There were 3 observations read from the data set  
WORK.TRANSACTIONS.

NOTE: The data set RECORDS.PATIENTMASTER has 5  
observations and 7 variables.

---

## 6.10 Using SAS Data Set Options

In this book, you have already seen a lot of options. It may help to keep them straight if you realize that the SAS language has three basic types of options: system options, statement options, and data set options. System options have the most global influence, followed by statement options, with data set options having the most limited effect.

System options (discussed in Section 1.7) are those that stay in effect for the duration of your job or session. These options affect how SAS operates, and are usually issued when you invoke SAS or via an OPTIONS statement.

System options include the CENTER option, which tells SAS to center all output, and the YEARCUTOFF= option, which tells SAS how to interpret two-digit years.

Statement options appear in individual statements and influence how SAS runs that particular DATA or PROC step. The NOPRINT option in PROC MEANS, for example, tells SAS not to produce a printed report. DATA= is a statement option that tells SAS which data set to use for a procedure. You can use DATA= in any procedure that reads a SAS data set. Without it, SAS defaults to the most recently created data set.

In contrast, data set options affect only how SAS reads or writes an individual data set. You can use data set options in DATA steps (in DATA, SET, MERGE, or UPDATE statements) or in PROC steps (in conjunction with a DATA= statement option). To use a data set option, you simply put it between parentheses directly following the data set name. These are the most frequently used data set options:

KEEP = *variable-list* tells SAS which variables to keep

DROP = *variable-list* tells SAS which variables to drop

RENAME = (*oldvar* = *newvar*) tells SAS to rename certain variables

FIRSTOBS = *n* tells SAS to start reading at observation *n*

OBS = *n* tells SAS to stop reading at observation *n*

LABEL = '*data-set-label*' specifies a descriptive label for a SAS data set

IN = *new-var-name* creates a temporary variable for tracking whether that data set contributed to the current observation (discussed in Section 6.11)

WHERE = *condition* selects observations that meet a specified condition (discussed in Section 6.12)

**Selecting and renaming variables** Here are examples of the KEEP=, DROP=, and RENAME= data set options:

DATA selectedvars;

SET animals (KEEP = Class Species Status);

PROC PRINT DATA = animals (DROP = Habitat);

DATA animalhomes (RENAME = (Class = Type Habitat

```
= Home));  
SET animals;  
PROC PRINT DATA = animals (RENAME = (Class =  
Type Habitat = Home));  
RUN;
```

You could probably get by without these options, but they play an important role in fine tuning SAS programs. Data sets, for example, have a way of accumulating unwanted variables. Dropping unwanted variables will make your program run faster and use less disk space. Likewise, when you read a large data set, you often need only a few variables. By using the KEEP= option, you can avoid reading a lot of variables that you don't intend to use.

The DROP=, KEEP=, and RENAME= options are similar to the DROP, KEEP, and RENAME statements. However, the statements apply to all data sets named in the DATA statement, while the options apply only to the particular data set whose name they follow. Also, the statements are more limited than the options since they can be used only in DATA steps, and apply only to the data set being created. In contrast, the data set options can be used in DATA or PROC steps and can apply to input or output data sets. Please note that these options do not change input data sets; they change only what is read from input data sets.

**Selecting observations by observation number** You can use the FIRSTOBS= and OBS= data set options together to tell SAS which observations to read from a data set. The options in the following statements tell SAS to read just 20 observations:

```
DATA sample;  
SET animals (FIRSTOBS = 101 OBS = 120);  
PROC PRINT DATA = animals (FIRSTOBS = 101 OBS  
= 120);  
RUN;
```

If you use large data sets, you can save development time by testing your programs with a subset of your data with the FIRSTOBS= and OBS= options.

The FIRSTOBS= and OBS= data set options are similar to statement and system options with the same name. The statement options apply only to raw data files being read with an INFILE statement, whereas the data set options apply only to existing SAS data sets that you read in a DATA or PROC step. The system options apply to all files and data sets. If you use similar system and data set options, the data set option will override the system option for that particular data set.

**Labeling SAS data sets** The LABEL= option is somewhat different from other options covered here. All the other options affect your data, but not LABEL=. Instead, LABEL= adds a text string to the descriptor portion of your data set. In this example, SAS creates a data set named RARE, and gives it the label “Endangered Species Data”:

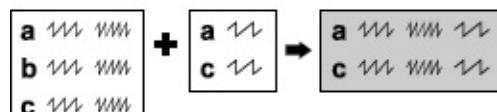
```
DATA rare (LABEL = 'Endangered Species Data');
  SET animals;
  IF Status = 'Endangered';
RUN;
```

The LABEL= data set option is similar to the LABEL statement used in DATA and PROC steps. However, the LABEL statement applies labels to individual variables, while the LABEL= data set option applies a label to an entire data set. Using data set labels is a good habit because it helps to document your work. Data set labels are displayed in the output of PROC CONTENTS.

## 6.11 Tracking and Selecting Observations

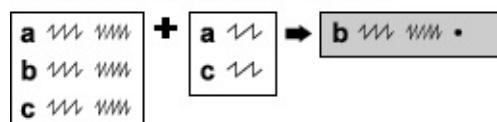
## with the IN= Option

Select matching observations



OR

Select non-matching observations



When you combine two data sets in a DATA step, you can use IN= options to track which of the original data sets contributed to each observation in the new data set. You can think of the IN= option as a sort of tag. Instead of saying “Product of Canada,” the tag says something like “Product of data set one.” Once you have that information, you can use it in many ways, including selecting matching or non-matching observations during a merge.

The IN= data set option can be used any time you read a SAS data set in a DATA step—with SET, MERGE, or UPDATE—but is most often used with MERGE. To use the IN= option, you simply put the option in parentheses directly following the data set you want to track, and specify a name for the IN= variable. The names of IN= variables must start with a letter or underscore; be 32 characters or fewer in length; and contain only letters, numerals, or underscores.

The DATA step below creates a data set named BOTH by merging two data sets named STATE and COUNTY. Then the IN= options create two variables named InState and InCounty:

```
DATA both;
  MERGE state (IN = InState) county (IN = InCounty);
    BY StateName;
  RUN;
```

Unlike most variables, IN= variables are temporary, existing only during the current DATA step. SAS gives the IN= variables a value of 0 or 1. A value of 1 means that data set did contribute to the current observation, and a value of 0 means the data set did not contribute. Suppose that the COUNTY data set above contained no data for Louisiana. (Louisiana has parishes, not counties.) In that case, there would be one observation for Louisiana, which would have a value of 1 for the variable InState and a value of 0 for InCounty because the STATE data set contributed to that observation, but the COUNTY data set did not.

You can use this variable like any other variable in the current DATA step, but it is most often used in subsetting IF or IF-THEN statements such as these:

Subsetting IF InState = 1;  
IF:

IF InCounty = 0;

IF InState = 1 AND InCounty = 1;

IF-THEN: IF InCounty = 1 THEN Origin = 1;

IF InState = 1 THEN State = 'Yes';

**Example** A sporting goods manufacturer wants to find all customers who did not place any orders during the third quarter of the year. The company has two data files, one that contains all customers and one that contains all orders placed during the third quarter. To find customers without orders, you merge the two data sets using the IN= option, and then select customers who had no observations in the orders data set. The customer data file contains the

customer number, name, and address. The orders data file contains the customer number and total price, with one observation for every order placed during the third quarter. Here are samples of the two tab-delimited data files:

**Customer  
data**

**Orders data**

CustNum	Name	Address	CustNum
Total			
101	Murphy's Sports	115 Main St.	
102		562.01	
102	Sun N Ski	2106 Newberry Ave.	
104		254.98	
103	Sports Outfitters	19 Cary Way	
104		1642.00	
104	Cramer & Johnson	4106 Arlington Blvd.	
101		3497.56	
105	Sports Savers	2708 Broadway	
102		385.30	

Here is the program that finds customers who did not place any orders:

```
PROC IMPORT DATAFILE =
  'c:\MyRawData\CustAdd.txt' OUT = customer
  REPLACE;
PROC IMPORT DATAFILE =
  'c:\MyRawData\OrdersQ3.txt' OUT = orders
  REPLACE;
RUN;
PROC SORT DATA = orders;
  BY CustNum;
RUN;
* Combine the data sets using the IN= option;
DATA noorders;
  MERGE customer orders (IN = Recent);
  BY CustNum;
  IF Recent = 0;
```

RUN;

The customer data are already sorted by customer number and so do not need to be sorted with PROC SORT. The orders data, however, are in the order received and must be sorted by customer number before merging. In the DATA step, the IN= option creates a variable named Recent, which equals 1 if the ORDERS data set contributed to that observation and 0 if it did not. Then a subsetting IF statement keeps only the observations where Recent is equal to 0—those observations with no orders data. Notice that there is no IN= option on the CUSTOMER data set. Only one IN= option was needed to identify customers who did not place any orders. Here is the data set NOORDERS:

	<b>CustNum</b>	<b>Name</b>	<b>Address</b>
1	103	Sports Outfitters	19 Cary Way
2	105	Sports Savers	2708 Broadway

The values for the variable Total are missing because these customers did not have observations in the ORDERS data set. The variable Recent does not appear in the output because, as a temporary variable, it was not added to the NOORDERS data set.

The following notes appear in the SAS log stating that five observations were read from CUSTOMER, five observations from ORDERS, and the final data set, NOORDERS, has two observations.

---

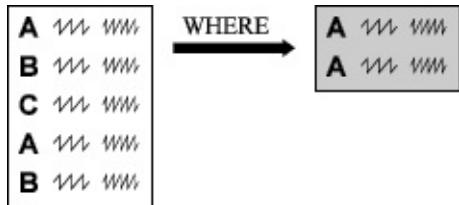
NOTE: There were 5 observations read from the data set  
WORK.CUSTOMER.

NOTE: There were 5 observations read from the data set WORK.ORDERS.

NOTE: The data set WORK.NOORDERS has 2 observations and  
4 variables.

---

## 6.12 Selecting Observations with the WHERE= Option



By this point, you've probably realized that with SAS programming there is usually more than one way to perform any particular task. Of all the things you can do with SAS, subsetting your data probably presents you with the most choices. The idea is simple: you have a data set, but you want to use only part of it. Maybe you have census data for the entire U.S., but you want data only for Arkansas, or for males, or for households with more than 10 people. In any particular case, the best way to subset your data depends on the type of data file you have, and what you want to do after you subset the data. That's why SAS offers you so many ways to do this.

This book has already covered several ways to subset a data set. If your data are in a raw data file, then you can read part of the file using multiple INPUT statements (Section 2.16). If your data are in a SAS data set, you can use a subsetting IF or WHERE statement in a DATA step (Section 3.8), or PROC SQL (Section 3.9). In a DATA step, you can also use OUTPUT statements to control which observations are written to a data set (Section 3.10). If you are using a procedure, you can subset your data using a WHERE statement (Section 4.2). Even with all these ways to subset your data, there is another way worth knowing: the WHERE= data set option.

The WHERE= data set option is the most flexible of all ways to subset data. You can use it in DATA steps or PROC steps, when you read existing data sets and when you write

new data sets. The basic form of a WHERE= data set option is:

WHERE = (*condition*)

Only observations satisfying the condition will be used by SAS. The WHERE= data set option is, not surprisingly, similar to the WHERE statement, and uses the same symbolic and mnemonic operators listed in Section 4.2. To use the WHERE= data set option in a DATA step, you simply put it between parentheses following the name of the data set to which it applies. If used in a SET, MERGE, or UPDATE statement, the WHERE= option applies to the data set that is being read.

DATA gone;

  SET animals (WHERE = (Status = 'Extinct'));

  RUN;

If used in a DATA statement, the WHERE= option applies to the data set that is being written.

DATA uncommon (WHERE = (Status IN ('Endangered',  
'Threatened')));

  SET animals;

  RUN;

If used in a PROC statement, the procedure uses only the observations that satisfy the WHERE= condition.

PROC IMPORT DATAFILE =

  'c:\MyRawData\Wildlife.csv'

  OUT = animals (WHERE = (Class = 'Mammalia'))

  REPLACE;

  RUN;

PROC PRINT DATA = animals (WHERE = (Habitat =  
'Riparian'));

  RUN;

Note that in order to use a WHERE= option with a PROC IMPORT, you must know—ahead of time—the names that SAS will give to your variables.

**Example** The following data contain information about the Seven Summits, the highest mountains on each continent. Each line of data includes the name of a mountain, its continent, and height in meters.

Kilimanjaro	Africa	5895
Vinson Massif	Antarctica	4897
Everest	Asia	8848
Elbrus	Europe	5642
McKinley	North America	6194
Aconcagua	South America	6962
Kosciuszko	Australia	2228

This program reads the data with an INPUT statement, and creates two data sets named TALLPEAKS and AMERICAN. The WHERE= data set options control which observations are included in each data set.

```
*Input the data and create two subsets;
DATA tallpeaks (WHERE = (Height > 6000))
    american (WHERE = (Continent CONTAINS
    ('America')));
    INFILE 'c:\MyRawData\Mountains.dat';
    INPUT Name $1-14 Continent $15-28 Height;
    RUN;
PROC PRINT DATA = tallpeaks;
    TITLE 'Members of the Seven Summits above 6,000
Meters';
    RUN;
PROC PRINT DATA = american;
    TITLE 'Members of the Seven Summits in the
Americas';
    RUN;
```

Here are the results:

### Members of the Seven Summits above 6,000 Meters

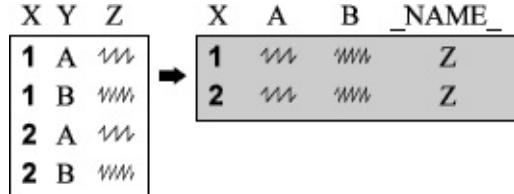
Obs	Name	Continent

	<b>1</b> Everest	Asia
	<b>2</b> McKinley	North America
	<b>3</b> Aconcagua	South America

## Members of the Seven Summits in the Americas

Obs	Name	Continent
<b>1</b>	McKinley	North America
<b>2</b>	Aconcagua	South America

## 6.13 Changing Observations to Variables Using PROC TRANSPOSE



We have already seen ways to combine data sets, create new variables, and sort data. Now, using PROC TRANSPOSE, we will flip data—so get your spatulas ready.

The TRANSPOSE procedure transposes SAS data sets, turning observations into variables or variables into observations. In most cases, to convert observations into variables, you can use the following statements:

```
PROC TRANSPOSE DATA = old-data-set OUT = new-data-set;  
    BY variable-list;  
    ID variable-list;  
    VAR variable-list;
```

In the PROC TRANSPOSE statement, *old-data-set* refers to the SAS data set you want to transpose, and *new-data-set* is the name of the newly transposed data set.

**BY statement** Use the BY statement if you have grouping variables that you want to keep as variables. These variables are included in the transposed data set but are not transposed. The transposed data set will have one observation for each BY level. In the figure above, the variable X is the BY variable. The data set must be sorted by these variables before transposing.

**ID statement** The ID statement names the variable whose formatted values will become the new variable names. If more than one variable is listed, then the values of all variables in the ID statement will be concatenated to form the new variable names. The ID values must occur only once in the data set; or if a BY statement is present, then the values must be unique within BY groups. If the first ID variable is numeric, then the new variable names have an underscore for a prefix (_1 or _2, for example). If you don't use an ID statement, then the new variables will be named COL1, COL2, and so on. In the figure above, the variable Y is the ID variable. Notice how its values are the names of the new variables in the transposed data set.

**VAR statement** The VAR statement names the variables whose values you want to transpose. In the figure above, the variable Z is the VAR variable. SAS creates a new

variable, `_NAME_`, which has as values the names of the variables in the VAR statement. If there is more than one VAR variable, then `_NAME_` will have more than one value.

**Example** Suppose you have the following comma-delimited file about players for minor league baseball teams. You want to look at the relationship between salary and batting average. You have the team name, player's number, the type of data (salary or batting average), and the entry:

```
Team,Player,Type,Entry
Garlics,10,salary,43000
Peaches,8,salary,38000
Garlics,21,salary,51000
Peaches,10,salary,47500
Garlics,10,batavg,.281
Peaches,8,batavg,.252
Garlics,21,batavg,.265
Peaches,10,batavg,.301
```

To look at the relationship, salary and batting average must be variables. The following program reads the comma-delimited file into a SAS data set and sorts the data by team and player. Then the data are transposed using PROC TRANSPOSE.

```
PROC IMPORT DATAFILE =
  'c:\MyRawData\Transpos.csv' OUT = baseball
  REPLACE;
PROC SORT DATA = baseball;
  BY Team Player;
RUN;
* Transpose data so salary and batavg are variables;
PROC TRANSPOSE DATA = baseball OUT = flipped;
  BY Team Player;
  ID Type;
  VAR Entry;
```

RUN;

In the PROC TRANSPOSE step, the BY variables are Team and Player. The BY variables remain in the data set, and define the new observations creating one observation for each combination of team and player. The ID variable is Type, whose values (salary and batavg) will be the new variable names. The variable to be transposed, Entry, is specified in the VAR statement.

Here is the data set BASEBALL after sorting and before transposing:

	<b>Team</b>	<b>Player</b>	<b>Type</b>
1	Garlics		10 salary
2	Garlics		10 batavg
3	Garlics		21 salary
4	Garlics		21 batavg
5	Peaches		8 salary
6	Peaches		8 batavg
7	Peaches		10 salary
8	Peaches		10 batavg

Here is the data set FLIPPED after transposing:

--	--	--	--	--

	<b>Team</b>	<b>Player</b>	<b>_NAME_</b>	<b>salary</b>
1	Garlics	10	Entry	4300
2	Garlics	21	Entry	5100
3	Peaches	8	Entry	3800
4	Peaches	10	Entry	4750

Notice that the old variable name, Entry, appears as a value under the variable **_NAME_** in the transposed data set. The TRANSPOSE procedure automatically generates the **_NAME_** variable, but in this application it is not very meaningful and could be dropped.

The following messages appear in the SAS log stating that eight observations were read from the data set BASEBALL and that the resulting data set, FLIPPED, has four observations.

---

NOTE: There were 8 observations read from the data set  
WORK.BASEBALL.

NOTE: The data set WORK.FLIPPED has 4 observations and 5  
variables.

---

## 6.14 Using SAS Automatic Variables

In addition to the variables that you create in your SAS data set, SAS creates a few more called automatic variables. You don't ordinarily see these variables because they are temporary and are not saved with your data. But they are available in the DATA step, and you can use them just like you use any variable that you create yourself.

**_N_ and _ERROR_** The **_N_** and **_ERROR_** variables

are always available to you in the DATA step. `_N_` indicates the number of times SAS has looped through the DATA step. This is not necessarily equal to the observation number, since a simple subsetting IF statement can change the relationship between observation number and the number of iterations of the DATA step. The `_ERROR_` variable has a value of 1 if there is a data error for that observation and 0 if there isn't. Things that can cause data errors include invalid data (such as characters in a numeric field), conversion errors (like division by zero), and illegal arguments in functions (including log of zero).

**FIRST.variable and LAST.variable** Other automatic variables are available only in special circumstances. The `FIRST.variable` and `LAST.variable` automatic variables are available when you are using a BY statement in a DATA step. The `FIRST.variable` will have a value of 1 when SAS is processing an observation with the first occurrence of a new value for that variable and a value of 0 for the other observations. The `LAST.variable` will have a value of 1 for an observation with the last occurrence of a value for that variable and the value 0 for the other observations.

**Example** Your hometown is having a walk around the town square to raise money for the library. You have the following data: entry number, age group, and finishing time. (Notice that there is more than one observation per line of data.)

```
54 youth 35.5 21 adult 21.6 6 adult 25.8 13 senior  
29.0  
38 senior 40.3 19 youth 39.6 3 adult 19.0 25  
youth 47.3  
11 adult 21.9 8 senior 54.3 41 adult 43.0 32  
youth 38.6
```

The first thing you want to do is create a new variable for overall finishing place and print the results. The first part of the following program reads the raw data, and sorts the data

by finishing time (Time). Then another DATA step creates the new Place variable and gives it the current value of _N_. The PRINT procedure produces the list of finishers:

```
DATA walkers;
  INFILE 'c:\MyRawData\Walk.dat';
    INPUT Entry AgeGroup $ Time @@;
  RUN;
  PROC SORT DATA = walkers;
    BY Time;
  RUN;
  * Create a new variable, Place;
  DATA ordered;
    SET walkers;
    Place = _N_;
  RUN;
  PROC PRINT DATA = ordered;
    ID Place;
    TITLE 'Results of Walk';
  RUN;
  PROC SORT DATA = ordered;
    BY AgeGroup Time;
  RUN;
  * Keep the first observation in each age group;
  DATA winners;
    SET ordered;
    BY AgeGroup;
    IF FIRST.AgeGroup = 1;
  PROC PRINT DATA = winners;
    ID Place;
    TITLE 'Winners in Each Age Group';
  RUN;
```

The second part of this program produces a list of the top finishers in each age category. The ORDERED data set is sorted by AgeGroup and Time. In the DATA step, the SET statement reads the ORDERED data set. The BY statement in the DATA step generates the FIRST.AgeGroup and

LAST.AgeGroup temporary variables. The subsetting IF statement, IF FIRST.AgeGroup = 1, keeps only the first observation in each BY group. Since the Winners data set is sorted by AgeGroup and Time, the first observation in each BY group is the top finisher of that group.

Here are the results of the two PROC PRINTs. The first table shows the data after sorting by Time and including the new variable Place. Because Place is in the ID statement, it appears in the first column. Notice that the _N_ temporary variable does not appear in the results. The second table shows the winners for each age category and their overall place determined in the last DATA step:

## Results of Walk

<b>Place</b>	<b>Entry</b>	<b>AgeGroup</b>
1	3	adult
2	21	adult
3	11	adult
4	6	adult
5	13	senior
6	54	youth
7	32	youth
8	19	youth

<b>9</b>	38	senior
<b>10</b>	41	adult
<b>11</b>	25	youth
<b>12</b>	8	senior

## Winners in Each Age Group

Place	Entry	AgeGroup
<b>1</b>	3	adult
<b>5</b>	13	senior
<b>6</b>	54	youth

7

“Nobody is too old to learn—  
but a lot of people keep putting  
it off.”

WILLIAM O’NEILL

From *The Official Explanations* by Paul Dickson. Copyright 1980 by Delacorte Press. Reprinted by permission of the publisher.

# CHAPTER 7

## Writing Flexible Code with the SAS Macro Facility

- [7.1 Macro Concepts](#)
- [7.2 Substituting Text with Macro Variables](#)
- [7.3 Concatenating Macro Variables with Other Text](#)
- [7.4 Creating Modular Code with Macros](#)
- [7.5 Adding Parameters to Macros](#)
- [7.6 Writing Macros with Conditional Logic](#)
- [7.7 Using %DO Loops in Macros](#)
- [7.8 Writing Data-Driven Programs with CALL SYMPUTX](#)
- [7.9 Writing Data-Driven Programs with PROC SQL](#)
- [7.10 Debugging Macro Errors](#)

### 7.1 Macro Concepts

The SAS macro facility allows you to assign a name to a block of text and then use that name in your program wherever you want to insert the text. The SAS macro facility is sometimes considered an advanced topic relevant only to experienced SAS users. However, even new SAS users would do well to know a little about using macros. Fortunately, the basic concepts are not difficult to understand. This chapter introduces the most commonly used features of the SAS macro language.

Because programs containing macros take longer to write and debug than standard SAS code, you generally won't want macros in short programs that will be run only a few times. But used properly, macros can make the development and maintenance of production programs much easier. They do this in several ways. For example, with macros you can make one small change in your program and have SAS echo that change throughout your program. In addition, macros allow you to write a piece of code once and use it over and over, in the same program or in different programs. You can even store macros in a central location and share them between programs and between programmers. Also, you can make your programs data driven, letting SAS decide what to do based on actual data values.

**The macro processor** When you submit a standard SAS program, SAS compiles and then immediately executes it. But when you write a macro, there is an additional step. Before SAS can compile and execute your program, SAS must pass your macro statements to the macro processor, which "resolves" your macros, generating standard SAS code. Because you are writing a program that writes a program, this is sometimes called meta-programming.



**Macro variables versus macros** The value of a macro variable is a single text string. This text string could be a variable name, a numeral, or any text that you want substituted into your program. The names of macro variables are prefixed with an ampersand (&).

A macro, on the other hand, is a larger piece of a program that may contain complex logic including complete DATA and PROC steps and macro statements such as %DO, %END, and %IF-%THEN/%ELSE. The names of macros are prefixed with a percent sign (%).

When SAS users talk about “macros” they sometimes mean macros, and sometimes they mean macro processing in general. Macro variables are usually called *macro variables*.

**Local versus global** Macro variables can have two kinds of “scope”—either local or global. Generally, a macro variable is local if it is defined inside a macro. A macro variable is generally global if it is defined in “open code”, which is everything outside a macro. You can use a global macro variable anywhere in your program, but you can use a local macro variable only inside its own macro. If you keep this in mind as you write your programs, you will avoid two common errors: trying to use a local macro variable outside its macro and accidentally creating local and global macro variables with the same name. It is possible to force a local macro variable to be global and vice versa, but this is generally not necessary and not covered in this book.

**Turning on the macro processor** Before you can use macros, you must have the MACRO system option turned on. This option is usually turned on by default, but may be turned off, especially on large shared servers, because SAS runs slightly faster when it doesn’t have to bother with checking for macros. If you are not sure whether the MACRO option is on, you can find out by submitting these statements:

```
PROC OPTIONS OPTION = MACRO;  
RUN;
```

Check your SAS log. If you see the option MACRO, then the macro processor is turned on, and you can use it. If you see NOMACRO there, you need to specify the MACRO option at invocation or in a configuration file. Specifying this type of option is system dependent. For details about

how to do this, see the SAS Documentation for your operating environment.

**Avoiding macro errors** There's no question about it, macros can make your head hurt. You can avoid the macro migraine by developing your program in a piecewise fashion. First, write your program in standard SAS code. Then, when it's bug-free, convert it to macro logic by adding one feature at a time. This modular approach to programming is always a good idea, but it's critical with macros.

**Quoting problems** One common mistake people make is to use single quotes around text where double quotes are needed. The macro processor doesn't resolve macros inside single quotation marks. To get around this, use double quotation marks whenever you refer to a macro or macro variable and you want SAS to resolve it. For example, below are two TITLE statements containing a macro variable named &Month. If the value of &Month is January, then the macro processor will substitute January in the title with the double quotation marks, but not the title with single quotation marks.

<b>Original statement</b>	<b>Statement after resolution</b>
TITLE 'Report for &Month';	TITLE 'Report for &Month';
TITLE "Report for &Month";	TITLE "Report for January";

## 7.2 Substituting Text with Macro Variables

Macro variables may be the most straightforward and easy-to-use part of the macro facility, yet if you master only this one feature of macro programming, you will have greatly increased your flexibility as a SAS programmer. Suppose that you have a SAS program that you run once a week. Each time you run it you have to edit the program so it will select data for the correct range of dates and print the correct dates in the title. This process is time-consuming and prone to errors. (What if you accidentally delete a semicolon?!?) Instead, you can use a macro variable to insert the correct date. Then you can have another cup of coffee while someone else, someone who knows very little about SAS, runs this program for you.

When SAS encounters the name of a macro variable, the macro processor simply replaces the name with the value of that macro variable. That value is a character constant that you specify.

**Creating a macro variable with %LET** The simplest way to assign a value to a macro variable is with the %LET statement. The general form of this statement is:

%LET *macro-variable-name* = *value*;

where *macro-variable-name* must be 32 characters or fewer in length; start with a letter or underscore; and contain only letters, numerals, and underscores. *Value* is the text to be substituted for the macro variable name, and can be longer than you are ever likely to need—over 65,000 characters long. The following statements each create a macro variable:

%LET Iterations = 10;

%LET Country = New Zealand;

Notice that unlike an ordinary assignment statement, *value* does not require quotation marks even when it contains

characters. Except for blanks at the beginning and end, which are trimmed, everything between the equal sign and the semicolon becomes part of the value for that macro variable.

**Using a macro variable** To use a macro variable you simply add the ampersand prefix (&) and stick the macro variable name wherever you want its value to be substituted. Keep in mind that the macro processor doesn't look for macros inside *single* quotation marks. To get around this, simply use double quotation marks. The following statements show possible ways to use the macro variables defined above:

```
DO i = 1 to &Iterations;  
CountryName = "&Country";
```

After being resolved by the macro processor, these statements would become:

```
DO i = 1 to 10;  
CountryName = "New Zealand";
```

**Example** A grower of tropical flowers records information about each sale in a raw data file. The data include location, customer ID, date of sale, variety of flower, sale quantity, and sale amount.

```
1 240W 02-07-2020 Ginger 120 960  
1 240W 02-10-2020 Protea 180 1710  
2 356W 02-10-2020 Heliconia 60 720  
2 356W 02-15-2020 Anthurium 300 1050  
2 188R 02-16-2020 Ginger 24 192  
1 240W 02-21-2020 Heliconia 48 600  
1 240W 02-27-2020 Protea 48 456  
2 356W 02-28-2020 Ginger 240 1980
```

Periodically, the grower needs a report about sales of a single variety. The macro variable in this program allows the grower to choose one variety without editing the

WHERE and TITLE statements in the PROC step. Instead, he just types the name of the variety once, in the %LET statement.

```
%LET FlowerType = Ginger;  
* Read the raw data and create a permanent SAS data  
set;  
LIBNAME tropical 'c:\MySASLib';  
DATA tropical.flowersales;  
    INFILE 'c:\MyRawData\TropicalFlowers.dat';  
    INPUT Location CustomerID $4. @8 SaleDate  
MMDDYY10. @19 Variety $9.  
        SaleQuantity SaleAmount;  
    FORMAT SaleDate MMDDYY10.;  
RUN;  
* Print the report using a macro variable;  
PROC PRINT DATA = tropical.flowersales;  
    WHERE Variety = "&FlowerType";  
    FORMAT SaleAmount DOLLAR7.;  
    TITLE "Sales of &FlowerType";  
RUN;
```

The program starts with a %LET statement that creates a macro variable named &FlowerType, assigning to it a value of Ginger. Because &FlowerType is defined outside a macro, it is a global macro variable and can be used anywhere in this program. In this case, the value Ginger is substituted for &FlowerType in a WHERE statement and a TITLE statement. Here are the results:

## Sales of Ginger

Obs	Location	CustomerID	SaleDate	Variety	SaleQuant
1	1240W		02/07/2020	Ginger	12

5	2	188R	02/16/2020	Ginger	2
8	2	356W	02/28/2020	Ginger	24

This is a short program, so using a macro variable didn't save much trouble. However, if you had a program 100 or even 1,000 lines long, a macro variable could be a blessing.

## 7.3 Concatenating Macro Variables with Other Text

The previous section described how you can use macro variables to increase the flexibility of your SAS programs. Macro variables hold pieces of text that you can use later to insert into your program. These pieces of text can be used alone, or they can be combined with other text.

**Combining text with macro variables** When SAS encounters an ampersand (&) embedded in text, it will look for macro variable names starting with the first character after the ampersand until it encounters either a space, a semicolon, another ampersand, or a period. So, if you want to add text before your macro variable, simply concatenate the text with an ampersand and the macro variable name. If you want to add text after the macro variable, then you need to insert a period between the end of the macro variable name and the text. The period signals the end of the macro variable and will not be included in the resolved text. Concatenating two macro variables together does not require a period between the names, because the ampersand for the second macro variable signals the end of the first macro variable.

Here are examples with two macro variables, &Region and

&MyName, defined as follows:

```
%LET Region = West;  
%LET MyName = Sam;
```

**SAS statement before  
resolution**

```
Office =  
"NorthAmerica&Region";
```

```
Office = "&Region.Coast";
```

```
DATA &MyName..Sales;
```

```
DATA  
&MyName&Region.ern_Sales;
```

**SAS statement after**

```
Office = "NorthAme
```

```
Office = "WestCoast
```

```
DATA Sam.Sales;
```

```
DATA SamWestern_
```

**Automatic macro variables** Every time you invoke SAS, the macro processor automatically creates certain macro variables. These macro variables can be used in your programs just like macro variables that you create. Some examples of automatic macro variables are:

<b>Variable name</b>	<b>Exam ple</b>	<b>Description</b>
--------------------------	---------------------	--------------------

&SYSDATE	18MAY E	the character value of the date that the job or session began
----------	------------	---------------------------------------------------------------

&SYSDAY	Wednes day	the day of the week that job or session began
---------	---------------	-----------------------------------------------

**&SYSNO** 312 number of observations in last SA  
BS data set created

**Example** This example uses the FLOWERSALES data set created in the previous section.

	<b>Location</b>	<b>CustomerID</b>	<b>SaleDate</b>	<b>Variety</b>	<b>SaleQuantity</b>
1	1	240W	02/07/2020	Ginger	12
2	1	240W	02/10/2020	Protea	18
3	2	356W	02/10/2020	Heliconia	6
4	2	356W	02/15/2020	Anthurium	30
5	2	188R	02/16/2020	Ginger	2
6	1	240W	02/21/2020	Heliconia	4
7	1	240W	02/27/2020	Protea	4
8	2	356W	02/28/2020	Ginger	24

This program creates a macro variable named &SumVar which, when used with the prefix Sale, determines which variable will be summarized. When the macro variable has the value Quantity, as in this example, then the variable in the VAR statement of the PROC MEANS becomes SaleQuantity. The &SumVar macro variable also appears in the TITLE statement.

```
%LET SumVar = Quantity;  
LIBNAME tropical 'c:\MySASLib';  
* Create RTF file with today's date in the file name;  
ODS RTF PATH = 'c:\MyRTFFiles' FILE =  
"FlowerSales_&SYSDATE..rtf";  
* Summarize the sales for the selected variable;  
PROC MEANS DATA = tropical.flowersales SUM MIN  
MAX MAXDEC=0;  
VAR Sale&SumVar;  
CLASS Variety;  
TITLE "Summary of Sales &SumVar by Variety";  
RUN;  
* Close the RTF file;  
ODS RTF CLOSE;
```

The program creates an RTF file, and the name of the file depends on the date the SAS job or session begins. If the session begins on January 25, 2020, then the filename will be FlowerSales_25JAN20.rtf. Notice that there are two periods in the file specification in the ODS RTF statement. The first period signals the end of the macro variable name &SYSDATE, while the second period becomes part of the filename. Here is what the RTF file looks like opened in Microsoft Word.

The screenshot shows a Microsoft Word document titled "FlowerSales_25\AN20.rtf [Compatibility Mode] - Word". The ribbon tabs are FILE, HOME, INSERT, DESIGN, PAGE LAYOUT, REFERENCES, MAILINGS, REVIEW, VIEW, DESIGN, LAYOUT, and Lora D... . The HOME tab is selected. The font is set to Times, size 11. The table in the center of the page is titled "Analysis Variable : SaleQuantity". It has five columns: Variety, N Obs, Sum, Minimum, and Maximum. The data is as follows:

Variety	N Obs	Sum	Minimum	Maximum
Anthurium	1	300	300	300
Ginger	3	384	24	240
Heliconia	2	108	48	60
Protea	2	228	48	180

PAGE 1 OF 1 30 WORDS

## 7.4 Creating Modular Code with Macros



Anytime that you find yourself writing the same or similar SAS statements over and over, you should consider using a macro. A macro lets you package a piece of bug-free code and use it repeatedly within a single SAS program or in many SAS programs.

You can think of a macro as a kind of sandwich. The %MACRO and %MEND statements are like two slices of bread. Between those slices you can put any statements you want. The general form of a macro is:

```
%MACRO macro-name;  
  macro-text  
%MEND macro-name;
```

The %MACRO statement tells SAS that this is the beginning of a macro, while %MEND marks the end.

*Macro-name* is a name you make up, and can be up to 32 characters in length, start with a letter or underscore, and contain only letters, numerals, and underscores. The *macro-name* in the %MEND statement is optional, but your macros will be easier to debug and maintain if you include it. That way there's no question which %MACRO statement goes with which %MEND. *Macro-text* (also called a macro definition) is a set of SAS statements.

**Invoking a macro** After you have defined a macro, you can invoke it by adding the percent sign prefix to its name, like this:

```
%macro-name
```

A semicolon is not required when invoking a macro, though adding one generally does no harm.

**Example** Using the data from the previous section, this example creates a simple macro. The data include location, customer ID, date of sale, variety of flower, sale quantity, and sale amount.

	<b>Locatio n</b>	<b>CustomerI D</b>	<b>SaleDate</b>	<b>Variety</b>	<b>SaleQuant y</b>
1	1	240W	02/07/202 0	Ginger	12
2	1	240W	02/10/202 0	Protea	18
3	2	356W	02/10/202 0	Heliconia	6

				0		
4	2	356W	02/15/2020	Anthuriu m		30
5	2	188R	02/16/2020	Ginger		2
6	1	240W	02/21/2020	Heliconia		4
7	1	240W	02/27/2020	Protea		4
8	2	356W	02/28/2020	Ginger		24

The following program creates a macro named %Sample to sort the data by SaleQuantity and print the five observations with the largest sales. Then the program invokes the macro.

```

LIBNAME tropical 'c:\MySASLib';

* Macro to print 5 largest sales;
%MACRO Sample;
  PROC SORT DATA = tropical.flowersales OUT =
salesout;
    BY DESCENDING SaleQuantity;
  RUN;
  PROC PRINT DATA = salesout (OBS = 5);
    FORMAT SaleAmount DOLLAR7.:
    TITLE 'Five Largest Sales by Quantity';
  RUN;
%MEND Sample;

* Invoke the macro;

```

%Sample

Here is the output:

## Five Largest Sales by Quantity

Obs	LocationID	CustomerID	SaleDate	Variety	SaleQuantity
1	2356W		02/15/2020	Anthurium	30
2	2356W		02/28/2020	Ginger	24
3	1240W		02/10/2020	Protea	18
4	1240W		02/07/2020	Ginger	12
5	2356W		02/10/2020	Heliconia	10

This macro is fairly limited because it does the same thing every time. To increase the flexibility of macros, combine them with %LET statements or add parameters as described in the next section.

**Macro autocall libraries** The macros in this book are defined and invoked inside a single program, but you can also store macros in a central location, called an autocall library. Macros in an autocall library can be shared by programs and programmers. Basically you save your macros as files in a directory or as members of a partitioned data set (depending on your operating environment), and

use the MAUTOSOURCE and SASAUTOS= system options to tell SAS where to look for macros. Then you can invoke a macro even though the original macro does not appear in your program. For more information see the SAS Documentation.

## 7.5 Adding Parameters to Macros



Macros can save you a lot of trouble, allowing you to write a set of statements once and then use them over and over. However, you usually don't want to repeat exactly the same statements. You may want the same report, but for a different data set, or product, or patient. Parameters allow you to do this.

Parameters are macro variables whose value you set when you invoke a macro. The simplest macros, like the macro in the previous section, have no parameters. To add parameters to a macro, you simply list the macro-variable names between parentheses in the %MACRO statement. Here is one of the possible forms of the parameter-list.

```
%MACRO macro-name (parameter-1= ,parameter-2= ,  
... parameter-n=);  
    macro-text  
%MEND macro-name;
```

For example, a macro named %QuarterlyReport might start like this:

```
%MACRO QuarterlyReport(Quarter=,SalesRep=);
```

This macro has two parameters: &Quarter and &SalesRep. You could invoke the macro with this statement:

```
%QuarterlyReport(Quarter=3,SalesRep=Smith)
```

The SAS macro processor would replace each occurrence of the macro variable &Quarter with the value 3, and it would substitute Smith for &SalesRep.

**Example** Using the tropical flower data again, suppose the grower often needs a report showing sales to an individual customer. The following program defines a macro that lets the grower select sales for a single customer and then sort the results. As before, the data contain the location, customer ID, date of sale, variety of flower, sale quantity, and sale amount.

	<b>Locatio n</b>	<b>CustomerI D</b>	<b>SaleDate</b>	<b>Variety</b>	<b>SaleQuan tity</b>
1	1	240W	02/07/2020	Ginger	12
2	1	240W	02/10/2020	Protea	18
3	2	356W	02/10/2020	Heliconia	6
4	2	356W	02/15/2020	Anthuriu m	30
5	2	188R	02/16/2020	Ginger	2
6	1	240W	02/21/2020	Heliconia	4
7	1	240W	02/27/2020	Protea	4

The following program defines a macro named %Select, and then invokes the macro twice. This macro sorts and prints the FLOWERSALES data set, using parameters to create two macro variables named &Customer and &SortVar.

```

LIBNAME tropical 'c:\MySASLib';

* Macro with parameters;
%MACRO Select(Customer=,SortVar=);
  PROC SORT DATA = tropical.flowersales OUT =
salesout;
    BY &SortVar;
    WHERE CustomerID = "&Customer";
  RUN;
  PROC PRINT DATA = salesout;
    FORMAT SaleAmount DOLLAR7.;
    TITLE1 "Orders for Customer Number
&Customer";
    TITLE2 "Sorted by &SortVar";
  RUN;
%MEND Select;

```

```

*Invoke the macro;
%Select(Customer = 356W, SortVar = SaleQuantity)
%Select(Customer = 240W, SortVar = Variety)

```

Here is the output:

## Orders for Customer Number 356W Sorted by SaleQuantity

Obs	Location	CustomerID	SaleDate	Variety	SaleQuantity

<b>1</b>	2356W	02/10/2020	Heliconia	1
<b>2</b>	2356W	02/28/2020	Ginger	2
<b>3</b>	2356W	02/15/2020	Anthurium	3

## Orders for Customer Number 240W Sorted by Variety

Obs	LocationID	CustomerID	SaleDate	Variety	SaleQuantity
<b>1</b>	1	240W	02/07/2020	Ginger	12
<b>2</b>	1	240W	02/21/2020	Heliconia	4
<b>3</b>	1	240W	02/10/2020	Protea	18
<b>4</b>	1	240W	02/27/2020	Protea	4

## 7.6 Writing Macros with Conditional Logic

Combining macros and macro variables gives you a lot of flexibility, but you can increase that flexibility even more

by adding macro statements such as %IF. Fortunately, many macro statements have parallel statements in standard SAS code so they should feel familiar. Here are the general forms of the statements used for conditional logic in macros:

```
%IF condition %THEN action;  
%ELSE %IF condition %THEN action;  
%ELSE action;  
  
%IF condition %THEN %DO;  
  SAS statements  
%END;
```

The %IF-%THEN/%ELSE statements must be used inside a macro. The %IF-%THEN-%DO-%END statements can be used inside a macro, or starting with SAS 9.4M5, outside a macro in open SAS code as long as no other %IF-%THEN statements are nested within the %DO-%END group.

You may be wondering why anyone needs these statements. Why not just use the standard IF-THEN? You may indeed use standard IF-THEN statements in your macros, but you will use them for different actions. %IF statements can contain actions that standard IF statements can't contain, such as complete DATA or PROC steps and even other macro statements. The %IF-%THEN statements don't appear in the standard SAS code generated by your macro. Remember you are writing a program that writes a program.

For example, you could combine conditional logic with the &SYSDAY automatic variable, like this:

```
%IF &SYSDAY = Tuesday %THEN %LET Country =  
Belgium;  
%ELSE %LET Country = France;
```

If you run the program on Tuesday, the macro processor resolves the statements to:

%LET Country = Belgium;

If you run the program on any other day, then the macro processor resolves the statements to:

%LET Country = France;

**Example** Using the tropical flower data again, this example shows a macro with conditional logic. The grower wants to print one report on Monday and a different report on Tuesday.

	<b>Locatio n</b>	<b>CustomerI D</b>	<b>SaleDate</b>	<b>Variety</b>	<b>SaleQuant y</b>
1	1	240W	02/07/2020	Ginger	12
2	1	240W	02/10/2020	Protea	18
3	2	356W	02/10/2020	Heliconia	6
4	2	356W	02/15/2020	Anthuriu m	30
5	2	188R	02/16/2020	Ginger	2
6	1	240W	02/21/2020	Heliconia	4
7	1	240W	02/27/2020	Protea	4
8	2	356W	02/28/2020	Ginger	24

Here is the program which uses the &SYSDAY automatic macro variable to control which report is generated:

```
LIBNAME tropical 'c:\MySASLib';

*This macro selects which report to run based on the
day of the week;
%MACRO DailyReports;
  %IF &SYSDAY = Monday %THEN %DO;
    PROC PRINT DATA = tropical.flowersales;
      FORMAT SaleAmount DOLLAR7.;
      TITLE 'Monday Report: Current Flower Sales';
    RUN;
  %END;
  %ELSE %IF &SYSDAY = Tuesday %THEN %DO;
    PROC MEANS DATA = tropical.flowersales MEAN
    MIN MAX MAXDEC = 2;
      CLASS Variety;
      VAR SaleQuantity;
      TITLE 'Tuesday Report: Summary of Flower
Sales';
    RUN;
  %END;
  %MEND DailyReports;

%DailyReports
```

When the program is submitted on Tuesday, the macro processor will write this program:

```
LIBNAME tropical 'c:\MySASLib';
PROC MEANS DATA = tropical.flowersales MEAN
MIN MAX MAXDEC = 2;
  CLASS Variety;
  VAR SaleQuantity;
  TITLE 'Tuesday Report: Summary of Flower Sales';
```

RUN;

If you run this program on Tuesday the output will look like this:

## Tuesday Report: Summary of Flower Sales

### The MEANS Procedure

Analysis Variable: SaleC			
Variety	N Obs	Mean	Std Dev
Anthurium	1	300.00	
Ginger	3	128.00	
Heliconia	2	54.00	
Protea	2	114.00	

If you run this program on Monday, you will get the PROC PRINT output. If you run this program on another day of the week, such as Wednesday, you will get no errors, but also no output.

## 7.7 Using %DO Loops in Macros

Like the iterative DO group in the DATA step, %DO loops in macros can be useful anytime you find yourself writing the same, or similar, blocks of code over and over. The difference is that with using %DO loops in macros, you can include all types of SAS statements in the loop, not just DATA step statements. The iterative %DO statement can be used only inside a macro and not in open SAS code. Here is

the general form of the statements used for iterative processing in macros:

```
%DO macro-variable-name = start-value %TO stop-value;  
    macro-text  
%END;
```

When the macro variable name is defined in the %DO statement, it does not start with an ampersand (&). But if you want to use the macro variable elsewhere in your macro, then, in those places, you need to include the ampersand. The start and stop values must be integers, or expressions or macro variables that resolve to integers. By default, the %DO loop will increment by one. If you want to increment by something other than one, then add a %BY:

```
%DO macro-variable-name = start-value %TO stop-value %BY increment;  
    macro-text  
%END;
```

It is important to remember that, as with other macro statements, the %DO and %END statements do not become part of the program that SAS executes.

For example, suppose that you have SAS data sets for ten years named SALES2011 through SALES2020 and you want to print each one. You could write ten separate PROC PRINTs, or you could include a %DO loop inside a macro:

```
%MACRO PrintYears;  
    %DO Year = 2011 %TO 2020;  
        PROC PRINT DATA = sales&Year;  
        RUN;  
    %END;  
%MEND PrintYears;
```

**Example** Using the tropical flower data again, this example shows how %DO loops in macros can be useful. For each location, the grower wants to calculate the sum of

the variables SaleQuantity and SaleAmount and then store the results in two data sets, one for each location.

	<b>Location</b>	<b>CustomerID</b>	<b>SaleDate</b>	<b>Variety</b>	<b>SaleQuantity</b>
1	1	240W	02/07/2020	Ginger	12
2	1	240W	02/10/2020	Protea	18
3	2	356W	02/10/2020	Heliconia	6
4	2	356W	02/15/2020	Anthurium	30
5	2	188R	02/16/2020	Ginger	2
6	1	240W	02/21/2020	Heliconia	4
7	1	240W	02/27/2020	Protea	4
8	2	356W	02/28/2020	Ginger	24

Here is the program:

```
LIBNAME tropical 'c:\MySASLib';
```

*This macro creates summary data sets for each value of

```

location;
%MACRO MeanSales;
%DO Loc = 1 %TO 2;
PROC MEANS DATA = tropical.flowersales
NOPRINT;
WHERE Location = &Loc;
VAR SaleQuantity SaleAmount;
OUTPUT OUT = salesloc&Loc
SUM(SaleQuantity SaleAmount) = TotalQuantity
TotalAmount;
RUN;
%END;
%MEND MeanSales;

```

### %MeanSales

The %DO loop creates a macro variable named &Loc that takes on the values of 1 and 2. These values are then substituted in the WHERE statement and again in the name of the output data set for the PROC MEANS. The macro processor will write this program, which contains two PROC MEANS:

```

PROC MEANS DATA = tropical.flowersales
NOPRINT;
WHERE Location = 1;
VAR SaleQuantity SaleAmount;
OUTPUT OUT = salesloc1
SUM(SaleQuantity SaleAmount) = TotalQuantity
TotalAmount;
RUN;
PROC MEANS DATA = tropical.flowersales
NOPRINT;
WHERE Location = 2;
VAR SaleQuantity SaleAmount;
OUTPUT OUT = salesloc2
SUM(SaleQuantity SaleAmount) = TotalQuantity

```

```
TotalAmount;  
RUN;
```

Here is the data set SALESLOC1:

	<b>_TYPE_</b>	<b>_FREQ_</b>	<b>TotalQuantity</b>	<b>TotalAmount</b>
1	0	4		396

And here is the data set SALESLOC2:

	<b>_TYPE_</b>	<b>_FREQ_</b>	<b>TotalQuantity</b>	<b>TotalAmount</b>
1	0	4		624

## 7.8 Writing Data-Driven Programs with CALL SYMPUTX



When you submit a SAS program containing macros it goes first to the macro processor which generates standard SAS code from the macro references. Then SAS compiles and executes your program. Not until execution—the final

stage—does SAS see any actual data values. This is the tricky part of writing data-driven programs: SAS doesn't know the values of your data until the execution phase, and by that time it is ordinarily too late. However, there are ways to have your digital cake and eat it too. These include CALL SYMPUTX (described here) and PROC SQL (see the next section).

CALL SYMPUTX takes a value from a DATA step and assigns it to a macro variable. You can then use this macro variable in later steps (in other words, not the same DATA step where you created the macro variable). To assign a value to a single macro variable, you use CALL SYMPUTX with this general form:

```
CALL SYMPUTX("macro-variable-name", value);
```

where *macro-variable-name*, enclosed in quotation marks, is the name of a macro variable, either new or old, and *value* is the value you want to assign to that macro variable. *Value* can be the name of a variable whose value SAS will use, or it can be a constant value enclosed in quotation marks. CALL SYMPUTX strips leading and trailing blanks from *value*, while the older CALL SYMPUT does not.

CALL SYMPUTX is often used in IF-THEN statements such as this:

```
IF Age >= 18 THEN CALL SYMPUTX("Status",  
"Adult");  
ELSE CALL SYMPUTX("Status", "Minor");
```

These statements create a macro variable named &Status and assign to it a value of Adult or Minor depending on the variable Age. The following CALL SYMPUTX uses a variable as its *value*:

```
IF TotalSales > 1000000 THEN CALL  
SYMPUTX("BestSeller", BookTitle);
```

This statement tells SAS to create a macro variable named &BestSeller, which is equal to the value of the variable

BookTitle when TotalSales exceed 1,000,000.

**Example** Here again are the flower sales data used in the previous sections.

	<b>Location</b>	<b>CustomerID</b>	<b>SaleDate</b>	<b>Variety</b>	<b>SaleQuantity</b>
1	1	240W	02/07/2020	Ginger	12
2	1	240W	02/10/2020	Protea	18
3	2	356W	02/10/2020	Heliconia	6
4	2	356W	02/15/2020	Anthurium	30
5	2	188R	02/16/2020	Ginger	2
6	1	240W	02/21/2020	Heliconia	4
7	1	240W	02/27/2020	Protea	4
8	2	356W	02/28/2020	Ginger	24

In this example, the grower wants a program that will find the customer with the single largest order in dollars, and print all the orders for that customer.

```

LIBNAME tropical 'c:\MySASLib';

* Sort the data by descending SaleAmount and save to a
temporary data set;
PROC SORT DATA = tropical.flowersales OUT =
salessorted;
    BY DESCENDING SaleAmount;
RUN;

* Find biggest order and pass the customer id to a macro
variable;
DATA _NULL_;
    SET salessorted ;
    IF _N_ = 1 THEN CALL
SYMPUTX("SelectedCustomer",CustomerID);
    STOP;
RUN;

PROC PRINT DATA = tropical.flowersales;
    WHERE CustomerID = "&SelectedCustomer";
    FORMAT SaleAmount DOLLAR7.;
    TITLE "Customer &SelectedCustomer Had the Single
Largest Order";
RUN;

```

This program has several steps, but each step is fairly simple. First, PROC SORT sorts the data by descending SaleAmount and saves the data to a temporary SAS data set SALESSORTED. That way, the largest single order will be the first observation in the newly sorted data set.

The DATA step then uses CALL SYMPUTX to assign the value of the variable CustomerID to the macro variable &SelectedCustomer when _N_ equals 1 (the first iteration of the DATA step). Since that is all we need from this DATA step, we can use the STOP statement to tell SAS to end this DATA step. The STOP statement is not necessary, but it is efficient because it prevents SAS from reading the

remaining observations for no reason.

When SAS reaches the RUN statement, SAS knows that the DATA step has ended so SAS executes the DATA step. At this point the macro variable &SelectedCustomer has the value 356W (the customer ID with the largest single order in dollars) and can be used in the PROC PRINT. The output looks like this:

### Customer 356W Had the Single Largest Order

Obs	LocationID	CustomerID	SaleDate	Variety	SaleQuantity
3	2	356W	02/10/2020	Heliconia	1
4	2	356W	02/15/2020	Anthurium	30
8	2	356W	02/28/2020	Ginger	2

## 7.9 Writing Data-Driven Programs with PROC SQL



In the previous section we described how to write data-driven programs by creating macro variables in DATA steps using CALL SYMPUTX. But that is not the only way to

write data-driven programs. You can also use the INTO clause in the SQL procedure to create macro variables, and then use those macro variables later in your program.

Here is the general form of PROC SQL to create a single macro variable from a variable in a data set:

```
PROC SQL NOPRINT;
  SELECT variable
    INTO :macro-variable-name
    FROM data-set-name;
  QUIT;
```

where *variable* is the name of the variable whose value you want to store in a macro variable and *macro-variable-name* is the name of the macro variable you want to create. Note that the *macro-variable-name* is preceded with a colon (:) here. The FROM clause tells which SAS data set to use. If the result of the PROC SQL has more than one observation, then the first value of the variable will be stored in the macro variable. For this reason, you may want to limit the result of the PROC SQL to just one observation by using a summary function in the SELECT clause (as described in Section 6.8), or by using subsetting clauses such as WHERE (Section 3.9) or HAVING. The HAVING clause is similar to WHERE except you can include summary functions in the HAVING clause but not in WHERE.

For example, if you want the macro variable named &Entries to contain the number of observations from the data set CONTEST, you could use these SELECT and INTO clauses:

```
SELECT COUNT(*)
  INTO :Entries
  FROM contest;
```

Or, suppose that you have weather data and you want to put the date of the coldest day (variable RecordedDate) into a macro variable named &ColdestDate. To do this, you could use the HAVING clause and the MIN summary function to

select the observation with the minimum value of the variable TempF:

```
SELECT RecordedDate  
INTO :ColdestDate  
FROM weatherdata  
HAVING TempF = MIN(TempF);
```

**Example** Here again are the flower sales data. As with the example from the previous section, the grower wants a program that will find the customer with the single largest order in dollars, and print all the orders for that customer.

	<b>Location</b>	<b>CustomerID</b>	<b>SaleDate</b>	<b>Variety</b>	<b>SaleQuantity</b>
1	1	240W	02/07/2020	Ginger	12
2	1	240W	02/10/2020	Protea	18
3	2	356W	02/10/2020	Heliconia	6
4	2	356W	02/15/2020	Anthurium	30
5	2	188R	02/16/2020	Ginger	2
6	1	240W	02/21/2020	Heliconia	4

7	1	240W	02/27/2020	Protea	4
8	2	356W	02/28/2020	Ginger	24

This program reproduces the results from the previous section using PROC SQL instead of CALL SYMPUTX.

```

LIBNAME tropical 'c:\MySASLib';
*Find customer with largest single sale amount;
PROC SQL NOPRINT;
  SELECT CustomerID
    INTO :SelectedCustomer
    FROM tropical.flowersales
   HAVING SaleAmount = MAX(SaleAmount);
QUIT;
PROC PRINT DATA = tropical.flowersales;
  WHERE CustomerID = "&SelectedCustomer";
  FORMAT SaleAmount DOLLAR7.;
  TITLE "Customer &SelectedCustomer Had the Single
Largest Order";
RUN;

```

This program starts with a PROC SQL and the NOPRINT option since we only want to create a macro variable and do not want SQL to produce a report. The INTO clause creates a macro variable named &SelectedCustomer from the value of the variable, CustomerID, listed in the SELECT clause. The FROM clause specifies the input data set. The HAVING clause keeps the observation where SaleAmount is equal to the maximum value for SaleAmount in the data set. This ensures that the value of the macro variable &SelectedCustomer will be the customer ID having the highest value of the variable

SaleAmount.

As in the previous section, the macro variable &SelectedCustomer is then used in the PROC PRINT to print all the orders for the customer ID with the largest single order in dollars. The output looks like this:

### Customer 356W Had the Single Largest Order

Obs	LocationID	CustomerID	SaleDate	Variety	SaleQuantity
3	2	356W	02/10/2020	Heliconia	1
4	2	356W	02/15/2020	Anthurium	30
8	2	356W	02/28/2020	Ginger	24

## 7.10 Debugging Macro Errors

Many people find that writing macros is not that hard. Debugging them, however, is another matter. This section covers techniques to ease the debugging process.

**Avoiding macro errors** As much as possible, develop your program in standard SAS code first. Then, when it is bug-free, add the macro logic one feature at a time. Add your %MACRO and %MEND statements. When that's working, add your macro variables, one at a time, and so on, until your macro is complete and bug-free. Also, always

remember to use double quotes around text that contains macro variables as macro variables are not resolved inside single quotes.

**Listing macro variables** You can get a listing in the SAS log of all macro variables you have created along with their values and scope (local or global) by including the following in your program:

```
%PUT _USER_;
```

This statement can be very useful for debugging long, or complicated macros and can appear anywhere in your program.

**System options for debugging macros** These five system options affect the kinds of messages SAS writes in your log. The default settings appear in bold.

MERROR |  
NOMERROR

When this option is on, SAS will issue a warning if you invoke a macro that SAS cannot find.

SERROR |  
NOSERROR

When this option is on, SAS will issue a warning if you use a macro variable that SAS cannot find.

MLOGIC |  
**NOMLOGIC**

When this option is on, SAS print in your log details about the execution of macros.

MPRINT |  
NOMPRINT

When this option is on, SAS print in your log the standard SAS code generated by macros.

SYMBOLGEN |

When this option is on, SAS print

NOSYMBOLGEN      in your log the values of macro variables.

While you want the MERROR and SERROR options to be on at all times, you will probably want to turn on MLOGIC, MPRINT, and SYMBOLGEN one at a time and only while you are debugging since they tend to make your log hard to read. To turn them on (or off), use the OPTIONS statement, for example:

```
OPTIONS MPRINT NOSYMBOLGEN NOMLOGIC;
```

**MERROR message** If SAS has trouble finding a macro, and the MERROR option is on, then SAS will print this message:

WARNING: Apparent invocation of macro SAMPL not resolved.

Check for a misspelled macro name.

**SERROR message** If SAS has trouble resolving a macro variable in open code, and the SERROR option is on, then SAS will print this message:

WARNING: Apparent symbolic reference FLOWER not resolved.

Check for a misspelled macro variable name. If the name is spelled right, then the scope may be wrong. Check to see if you are using a local variable outside of its macro. See Section 7.1 for definitions of local and global macro variables.

**MLOGIC messages** When the MLOGIC option is on, SAS prints messages in your log describing the actions of the macro processor. Here is a macro named %Sample:

```
%MACRO Sample(FlowerType=);  
PROC PRINT DATA = tropical.flowersales;  
WHERE Variety = "&FlowerType";
```

```
RUN;  
%MEND Sample;  
%Sample(FlowerType = Anthurium)
```

If you run %Sample with the MLOGIC option, your log will look like this:

---

```
24 OPTIONS MLOGIC;  
25 %Sample(FlowerType=Anthurium)  
MLOGIC(SAMPLE): Beginning execution.  
MLOGIC(SAMPLE): Parameter FLOWERTYPE has value Anthurium  
MLOGIC(SAMPLE): Ending execution.
```

**MPRINT messages** When the MPRINT option is on, SAS prints messages in your log showing the SAS statements generated by your macro. If you run %Sample with the MPRINT option, your log will look like this:

---

```
36 OPTIONS MPRINT;  
37 %Sample(FlowerType=Anthurium)  
MPRINT(SAMPLE): PROC PRINT DATA = tropical.flowersales;  
MPRINT(SAMPLE): WHERE Variety = "Anthurium";  
MPRINT(SAMPLE): RUN;
```

**SYMBOLGEN messages** When the SYMBOLGEN option is on, SAS prints messages in your log showing the value of each macro variable after resolution. If you run %Sample with the SYMBOLGEN option, your log will look like this:

---

```
30 OPTIONS SYMBOLGEN;  
31 %Sample(FlowerType=Anthurium)  
SYMBOLGEN: Macro variable FLOWERTYPE resolves to  
Anthurium
```

---

# 8

“Graphs reveal discoveries as  
the bud unfolds the flower.”

Henry D. Hubbard

From Graphic Presentation  
by Willard Cope Brinton,  
1939.

# CHAPTER 8

## Visualizing Your Data

- [8.1 Concepts of ODS Graphics](#)
- [8.2 Creating Bar Charts with PROC SGPlot](#)
- [8.3 Creating Histograms and Density Curves with PROC SGPlot](#)
- [8.4 Creating Box Plots with PROC SGPlot](#)
- [8.5 Creating Scatter Plots with PROC SGPlot](#)
- [8.6 Creating Series Plots with PROC SGPlot](#)
- [8.7 Creating Fitted Curves with PROC SGPlot](#)
- [8.8 Controlling Axes and Reference Lines in PROC SGPlot](#)
- [8.9 Controlling Legends and Insets in PROC SGPlot](#)
- [8.10 Customizing Graph Attributes in PROC SGPlot](#)
- [8.11 Creating Paneled Graphs with PROC SGPanel](#)
- [8.12 Specifying Image Properties and Saving Graphics Output](#)

### 8.1 Concepts of ODS Graphics

ODS Graphics is designed to give you high-quality graphs with a minimum of effort. By its nature, ODS graphics is a big topic. You have many choices to make about types of

graphs, file formats, and options for color, line type, tick marks, and so on. This chapter covers the basic statements and options to get you started.

As you might expect, ODS Graphics is an extension of the Output Delivery System, but instead of creating tabular output, ODS Graphics creates graphs, and it produces them using the same destinations and styles as ODS tabular output. ODS Graphics is different from SAS/GPGRAPH, which is licensed separately. Starting with SAS 9.3, ODS Graphics is part of Base SAS, so you do not need to license any additional products.

**Using ODS Graphics in statistical procedures** Over 80 statistical procedures have the ability to produce graphs using ODS Graphics. When you run these procedures, ODS Graphics will produce graphs that are specially designed for that type of analysis. Starting with SAS 9.3, ODS Graphics is turned on by default when you run SAS interactively in Microsoft Windows and UNIX. ODS Graphics is turned off by default when you run in batch mode or in other operating environments. To turn this feature on, insert this statement into your program before any statistical procedures:

ODS GRAPHICS ON;

Statistical procedures that support ODS Graphics will then create appropriate graphs.

You do not need to turn ODS Graphics off, but if you want to turn it off (either to make your programs run faster, or simply because you do not want the graphs) use this statement:

ODS GRAPHICS OFF;

Note that the ODS GRAPHICS statement does not open a destination (like ODS HTML or ODS PDF statements do). You open and close ODS destinations; you turn ODS GRAPHICS on or off.

**Using ODS Graphics for stand-alone graphs** ODS Graphics also includes a family of procedures designed to create stand-alone graphs (graphs that are not embedded in the output of a statistical procedure). The SGLOT and SGPANEL procedures are two of these. Because these procedures always produce graphs, you do not need to specify the ODS GRAPHICS ON statement even in batch mode. However, there may still be times when you want to use the ODS GRAPHICS statement to specify graphics options. (See Section 8.12.)

The SGLOT procedure creates single-celled graphs while SGPANEL creates multicelled graphs based on classification variables. The various types of graphs fall into four general categories:

Category	Types of Graphs
Basic plots	band, block, bubble, fringe, h low, needle, scatter, series, spli vector
Fit and confidence plots	ellipse, loess, penalized B-sp regression
Distribution plots	box, density, and histogram
Categorization plots	bar, dot, line, and waterfall

You can overlay multiple graphs as long as combining them makes sense. This chapter covers the most common types of graphs. Other graphs use similar syntax and options.

**ODS destinations** Unless you specify otherwise, your

graphs will be rendered in your default destination. HTML is the default destination for SAS Studio. HTML is also the default destination for SAS Enterprise Guide starting with release 8.1 (earlier versions use SASREPORT). Likewise, HTML is the default destination for the SAS windowing environment in Microsoft Windows and UNIX starting with SAS 9.3 (earlier versions use the LISTING destination). If you run in batch or in other operating environments, the default destination is LISTING. See Chapter 5 for information about controlling destinations.

**Saving graphs** Also starting with SAS 9.3, graphs are written in your WORK library and will therefore be deleted when you exit SAS. This is good because it prevents your disks from becoming cluttered with old graphs. For information about how to save graphs, see Section 8.12.

**Styles for graphs** ODS style templates control the overall appearance of your output. You can use the same style templates for graphs as for tabular output. However, some styles are better suited to statistical graphics than others. The following table lists styles that are recommended for graphical results:

Desired Output	Style Name	Default for Destination
Color	ANALYSI S	HTML
	HTMLBL UE	
		LISTING (graphs only)

LISTING

PDF, PS

PEARL

RTF

RTF

STATISTI  
CAL

Gray scale      JOURNAL

Black and  
white          JOURNAL  
                  2

You can change the default style for some destinations using menus in SAS Studio, SAS Enterprise Guide, or the SAS windowing environment. You can also specify a style for your graphs using the STYLE= option in the ODS statement for a destination. For example, to produce a gray-scale graph in the LISTING destination (and save it in a default location) you would use this statement:

ODS LISTING STYLE = JOURNAL;

For the LISTING destination, the STYLE= option applies only to graphical output; tabular output is still rendered as plain text. Also keep in mind that every destination has a default style associated with it, so if you change the destination for a graph, its appearance may change too. For more about specifying image properties and saving graphs, See Section 8.12.

## 8.2 Creating Bar Charts with PROC SGLOT

Bar charts show the distribution of a categorical variable where the length of each bar is proportional to the number of observations in that category. To create a chart with vertical bars, use the SGLOT procedure with a VBAR statement with this general form:

```
PROC SGLOT;  
  VBAR variable-name / options;
```

For horizontal bars, replace the keyword VBAR with HBAR. Possible options include:

ALPHA = <i>n</i>	specifies the level for confidence limits. The value of <i>n</i> must be between 0 (100% confidence) and (0% confidence). The default is 0.05 (95% confidence limits).
BARWIDTH = <i>n</i>	sets the width of bars. Values range from 0 to 1 with a default of 0.8.
DATALABEL = <i>variable-name</i>	displays a label for each bar. If you specify a variable name, then the values of that variable will be used. Otherwise, SAS will calculate appropriate values.
DISCRETEOFFSET = <i>n</i>	offsets bars from midpoints, which is useful for overlaying bar charts. The

value must be between –0.5 (left) and +0.5 (right). The default is 0 (no offset).

LIMITSTAT =  
*statistic*

specifies the type of limit lines to show. Possible values are CLM, STDDEV (standard deviation), or STDERR (standard error). You must specify a RESPONSE= option and STAT=MEAN. To display limits without using the GROUP= option, you must also specify GROUPDISPLAY=CLUSTER.

MISSING

includes a bar for missing values.

GROUP = *variable-name*

specifies a variable used to group data.

GROUPDISPLAY =  
*type*

specifies how to display grouped bars, either STACK (the default) or CLUSTER.

RESPONSE =  
*variable-name*

specifies a numeric variable to be summarized.

STAT = *statistic*

specifies a statistic, either FREQ, MEAN, MEDIAN, PERCENT, or SUM. FREQ is the default if there is no response variable. SUM is the default when you specify a response variable.

**TRANSPARENCY =** specifies the degree of transparency for the bars. The value of  $n$  must be between 0 (the default) and 1, with 0 being completely transparent and 1 being completely opaque.

**Example** A chocolate manufacturer is considering whether to add four new varieties of chocolate to its line of products. The company asked volunteers to taste the new flavors. The data contain each person's age group (A for adult, C for child) followed by their favorite flavor (80%Cacao, Earl Grey, Ginger, or Pear). Notice that each line of data contains six responses.

```
A Pear A 80%Cacao A EarlGrey C 80%Cacao A Ginger  
C Pear  
C 80%Cacao C Pear C Pear A EarlGrey A 80%Cacao C  
80%Cacao  
A Ginger A Pear C EarlGrey C 80%Cacao A 80%Cacao  
A EarlGrey  
A 80%Cacao C Pear C Pear A 80%Cacao C Pear C  
80%Cacao
```

The following program reads the raw data and creates a user-defined format. Then PROC SGLOT creates a bar chart using the GROUP= and GROUPDISPLAY= options.

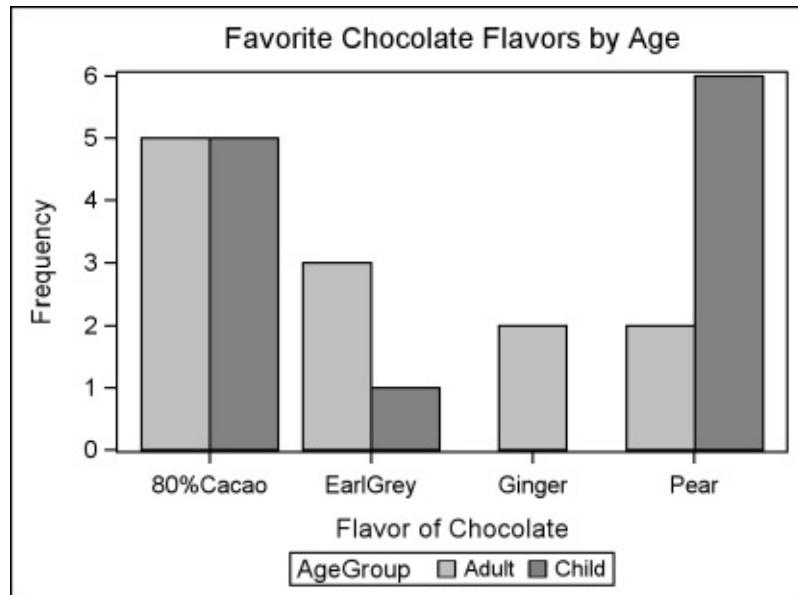
```
DATA chocolate;  
INFILE 'c:\MyRawData\Choc.dat';  
INPUT AgeGroup $ FavoriteFlavor $ @@;  
RUN;  
PROC FORMAT;  
  VALUE $AgeGp 'A' = 'Adult' 'C' = 'Child';  
RUN;  
* Bar chart for favorite flavor;  
PROC SGLOT DATA = chocolate;  
  VBAR FavoriteFlavor / GROUP = AgeGroup  
  GROUPDISPLAY = CLUSTER;
```

```

FORMAT AgeGroup $AgeGp.;
LABEL FavoriteFlavor = 'Flavor of Chocolate';
TITLE 'Favorite Chocolate Flavors by Age';
RUN;

```

This chart has clustered bars showing the number of respondents in each age group who chose each flavor. The LABEL statement replaced the name of the variable FavoriteFlavor with the words "Flavor of Chocolate" in the X-axis label. The FORMAT statement replaced the data values (A and C) with more descriptive values (Adult and Child) in the legend.



## 8.3 Creating Histograms and Density Curves with PROC SGPlot

The bar charts in the preceding section show the distribution of categorical data. To show the distribution of continuous data, you can use histograms (or box plots, which are described in the next section). In a histogram, the data are divided into discrete intervals called bins. Each bin is represented by a rectangle, which makes histograms look similar to bar charts. However, bar charts typically have a

gap between the bars while histograms do not.

**Histograms** To create a histogram, use PROC SGLOT and a HISTOGRAM statement with this general form:

HISTOGRAM *variable-name* / *options*;

Possible options include:

BINSTART = *n* specifies the midpoint for the first bin.

BINWIDTH = *n* specifies the bin width (in units of the horizontal axis). SAS determines the number of bins. This option is ignored if you specify the NBINS= option.

GROUP = *variable-name* specifies a variable used to group the data.

NBINS = *n* specifies the number of bins. SAS determines the bin width.

SCALE = *scaling-type* specifies the scale for the vertical axis, either PERCENT (the default), COUNT, or PROPORTION.

SHOWBINS places tick marks at the midpoints of the bins. By default, tick marks are placed at regular intervals based on minimum and maximum values.

TRANSPARENCY = *n* specifies the degree of transparency for the histogram. The value of *n* must be between 0 (the default) and 1, with

being completely transparent and 0 completely opaque.

**Density curves** You can also plot density curves for your data. The general form of a DENSITY statement is:

DENSITY *variable-name / options*;

Common options are:

GROUP = *variable-name* specifies a variable used to group the data.

TYPE = *distribution-type* specifies the type of distribution curve, either NORMAL (the default) or KERNEL.

TRANSPARENCY = *n* specifies the degree of transparency for the density curve. The value of *n* must be between 0 (the default) and 1, with 1 being completely transparent and 0 completely opaque.

The HISTOGRAM and DENSITY statements can be used together. Keep in mind that when you overlay graphs, the order of statements is important because the second graph will be drawn on top of the first and could hide it.

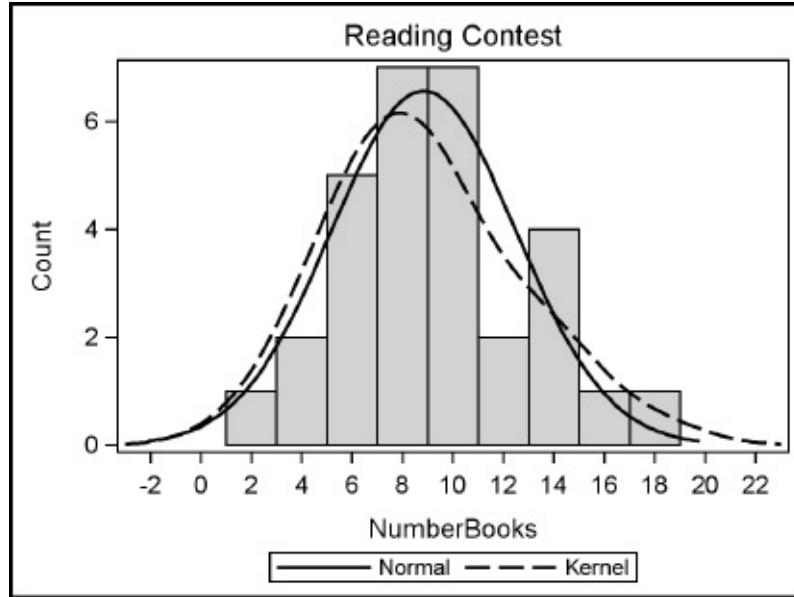
**Example** A fourth grade class has a competition to see who can read the most books in one month. For each student, the teacher records the student's name and the number of books read. Notice that each line of data includes six students.

Bella 4 Anthony 9 Joe 10 Chris 6 Beth 5 Daniel 2  
David 7 Emily 7 Josh 7 Will 9 Olivia 7 Matt 8  
Maddy 8 Sam 13 Jessica 6 Jose 6 Mia 12 Elliott 8  
Tyler 15 Lauren 10 Cate 14 Ava 11 Mary 9 Eric 10  
Megan 13 Michael 9 John 18 Alex 5 Cody 11 Amy 4

The DATA step below reads the raw data from a file named Reading.dat. Then an SGPLOT procedure creates a histogram for the number of books. The bins will have a width of two, the horizontal axis will have a tick mark at the center of each bin, and the vertical axis will show the count (in this case, the number of students). Two density distributions will be overlaid on the histogram: the normal distribution and the kernel density estimate.

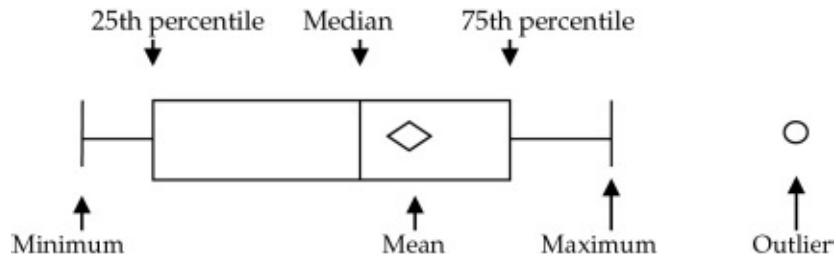
```
DATA contest;  
  INFILE 'c:\MyRawData\Reading.dat';  
  INPUT Name $ NumberBooks @@;  
RUN;  
PROC SGPLOT DATA = contest;  
  HISTOGRAM NumberBooks / BINWIDTH = 2  
  SHOWBINS SCALE = COUNT;  
  DENSITY NumberBooks;  
  DENSITY NumberBooks / TYPE = KERNEL;  
  TITLE 'Reading Contest';  
RUN;
```

Here is the graph of books read (shown in the JOURNAL style):



## 8.4 Creating Box Plots with PROC SGPlot

Like histograms, box plots show the distribution of continuous data. This type of graph is also called a box-and-whisker plot because of the way it looks. Every part of a box plot tells you something about the distribution of your data.



The ends of the box indicate the 25th and 75th percentiles (also called the interquartile range). The line inside the box indicates the 50th percentile (the median), and the marker indicates the mean. By default, the whiskers cannot be longer than 1.5 times the length of the box. Any points beyond the whiskers are considered outliers and are marked with circles. If you specify the EXTREME option, then the

whiskers will extend the entire range.

To create a vertical box plot, use PROC SGLOT and a VBOX statement, like this:

```
VBOX variable-name / options;
```

For horizontal box plots, replace the keyword VBOX with HBOX. Possible options include:

CATEGORY = *variable-name* specifies a categorical variable. A separate box plot will be created for each value of the categorical variable.

EXTREME specifies that the whiskers show the true minimum and maximum values. Outliers will not be identified.

GROUP = *variable-name* specifies a second categorical variable. One box plot will be created for each value of this variable within the categorical variable.

MISSING includes a box for missing values in the group or category variable.

TRANSPARENCY = *n* specifies the degree of transparency of the box plot. The value of *n* must be between 0 and 1, with 0 being completely transparent and 1 being completely opaque.

**Example** A small town sponsors an annual bicycle

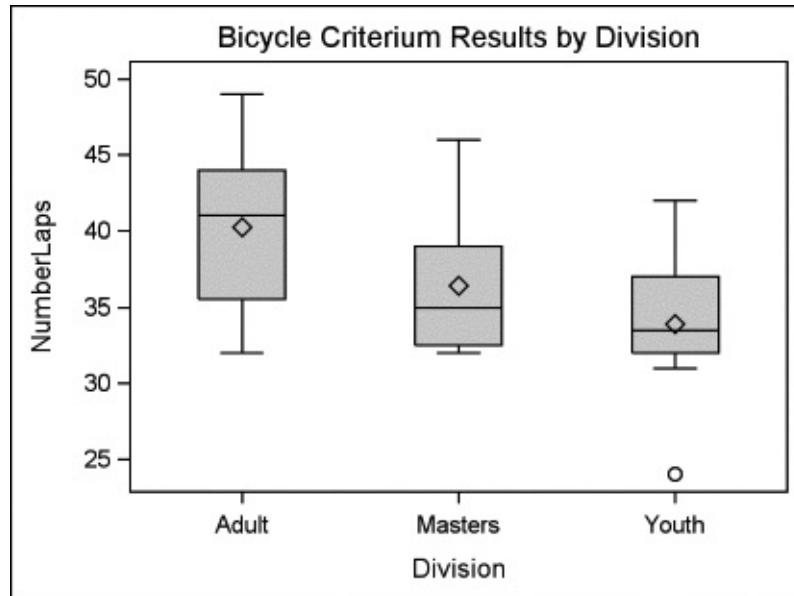
criterium. That's a race where bicyclists go round and round a loop. The racers compete in three divisions: Youth, Adult, and Masters. The data include each bicyclist's division and the number of laps they completed in one hour. Notice that each line of data contains results for five competitors.

```
Adult 44 Adult 33 Youth 33 Masters 38 Adult 40  
Masters 32 Youth 32 Youth 38 Youth 33 Adult 47  
Masters 37 Masters 46 Youth 34 Adult 42 Youth 24  
Masters 33 Adult 44 Youth 35 Adult 49 Adult 38  
Adult 39 Adult 42 Adult 32 Youth 42 Youth 31  
Masters 33 Adult 33 Masters 32 Youth 37 Masters 40
```

The DATA step below reads the raw data from a file named Criterium.dat. Then an SGPlot procedure creates vertical box plots of the number of laps. The CATEGORY= option tells SAS to create a separate box plot for each division.

```
DATA bikerace;  
  INFILE 'c:\MyRawData\Criterium.dat';  
  INPUT Division $ NumberLaps @@;  
  RUN;  
  * Create box plot;  
  PROC SGPLOT DATA = bikerace;  
    VBOX NumberLaps / CATEGORY = Division;  
    TITLE 'Bicycle Criterium Results by Division';  
  RUN;
```

Here is the box plot showing the number of laps by division:



## 8.5 Creating Scatter Plots with PROC SGPlot

Scatter plots are an effective way to show the relationship between two continuous variables. For experimental data, the independent variable is traditionally assigned to the horizontal axis while the dependent variable is assigned to the vertical axis. To create scatter plots, use PROC SGPlot and a SCATTER statement, like this:

```
SCATTER X=horizontal-variable Y=vertical-variable /  
options;
```

Possible options include:

DATALABEL =  
*variable-name*

displays a label for each data point. If you specify a variable name, the values of that variable will be used as labels. If you do not specify a variable name, then the values of the Y variable will be used.

GROUP = <i>variable-name</i>	specifies a variable to be used for grouping data.
JITTER	offsets the data markers slightly when multiple observations have the same response value.
NOMISSINGGROUP	specifies that observations with missing values for the group variable should not be included.
TRANSPARENCY = <i>n</i>	specifies the degree of transparency for the markers. The value of <i>n</i> must be between 0 (the default) and 1, with 1 being completely transparent and 0 completely opaque.

**Example** To illustrate the use of scatter plots, here are data about birds. For each species, there are four variables: name, type (S for songbirds or R for raptors), length in cm (from tip of beak to tip of tail), and wingspan in cm. Note that each line of data includes several birds.

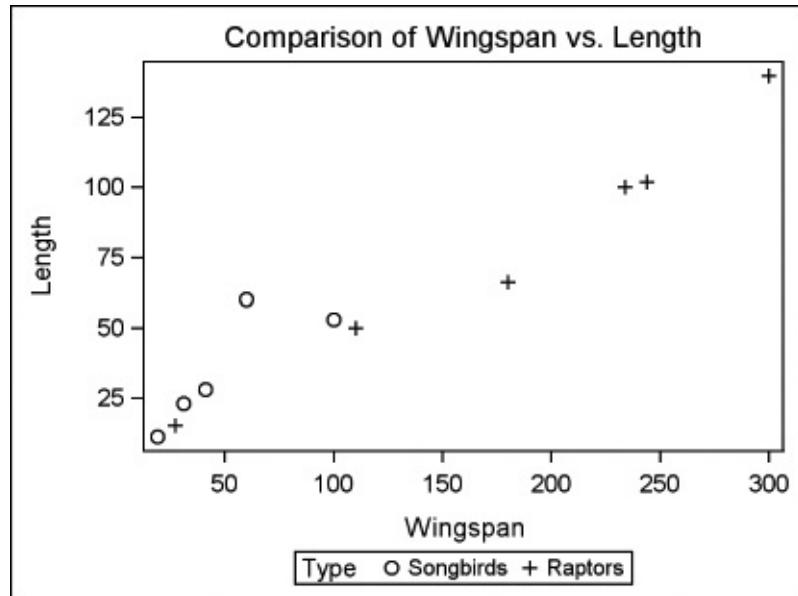
Robin	S	28	41	Bald Eagle	R	102	244	Barn
Owl	R	50	110					
Osprey	R	66	180	Cardinal	S	23	31	
Goldfinch	S	11	19					
Golden Eagle	R	100	234	Crow	S	53	100	
Magpie	S	60	60					
Elf Owl	R	15	27	Condor	R	140	300	

The following program creates a permanent SAS data set named WINGS in the MySASLib directory on the C drive.

Then the program reads the data and produces a scatter plot grouped by type. Since the values S and R are not very descriptive, PROC FORMAT is used to create a user-defined format that is then specified in a FORMAT statement in the SGPlot procedure to change S to Songbirds and R to Raptors.

```
LIBNAME flight 'c:\MySASLib';
DATA flight.wings;
    INFILE 'c:\MyRawData\Birds.dat';
    INPUT Name $12. Type $ Length Wingspan @@;
RUN;
* Plot Wingspan by Length;
PROC FORMAT;
    VALUE $birdtype
        'S' = 'Songbirds'
        'R' = 'Raptors';
RUN;
PROC SGPLOT DATA = flight.wings;
    SCATTER X = Wingspan Y = Length / GROUP =
Type;
    FORMAT Type $birdtype. ;
    TITLE 'Comparison of Wingspan vs. Length';
RUN;
```

Here is the scatter plot (shown in the JOURNAL style):



## 8.6 Creating Series Plots with PROC SGPlot

A series plot is similar to a scatter plot except that instead of marking each data point, SAS connects the data points with a line. Series plots make sense whenever data must be displayed in a particular order. Dates and times of any kind are good candidates for series plots. To create a series plot, use PROC SGPlot and a SERIES statement with this general form:

```
SERIES X = horizontal-variable Y = vertical-variable /  
options;
```

Possible options include:

CURVELABEL = '*text-string*'

adds a label for the curve. I specify a text string, then SA label from the Y variable.

DATALABEL = *variable-name*

displays a label for each da you specify a variable name,

that variable will be used as you do not specify a variable for the values of the Y variable.

GROUP = *variable-name*

specifies a variable to be used for grouping the data. A separate plot is created for each unique value of the grouping variable.

MARKERS

adds a marker for each data point.

NOMISSINGGROUP

specifies that observations with missing values for the group variable will not be included.

TRANSPARENCY = *n*

specifies the degree of transparency of the plot line. The value of *n* must be between 0 (the default) and 1, where 0 being completely transparent and 1 being completely opaque.

Note that SAS will connect the points in the order in which they appear in the data set. To have the points connected properly, your data must be sorted by the horizontal variable. If your data are not already sorted, then use PROC SORT before plotting your data.

**Example** A technical writer collects data about her use of electricity for one day. Each hour she checks her meter and records the number of kilowatt hours used since the last reading. The data include the time (on a 24-hour clock) and the number of kilowatt hours. Note that each line of data contains six readings.

0 .22 1 .15 2 .17 3 .18 4 .19 5 .23

```

6 .5 7 .63 8 .61 9 .6 10 .48 11 .45
12 .44 13 .44 14 .39 15 .35 16 .42 17 .47
18 .7 19 .66 20 .7 21 .69 22 .6 23 .4

```

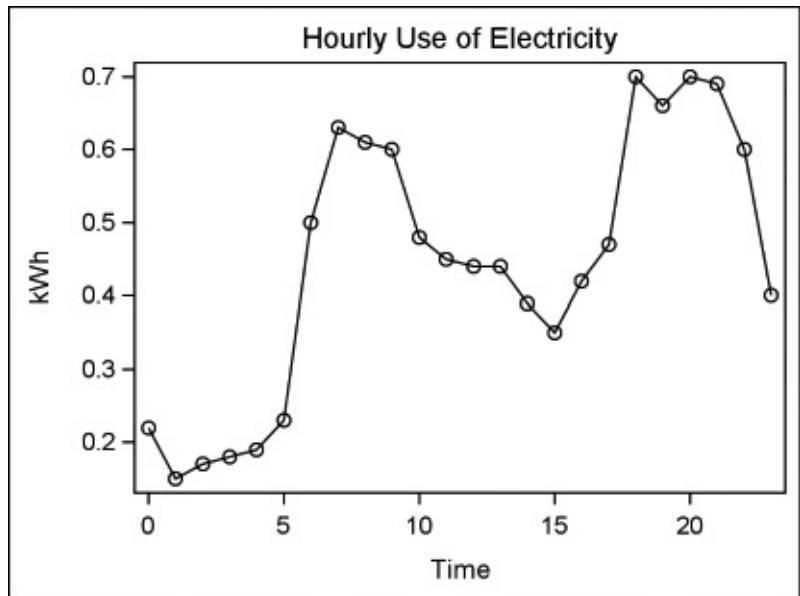
She could use a scatter plot to display these data, but since one of the variables is time, using a series plot makes more sense. The following program reads the data from a raw data file named Hourly.dat and creates a SAS data set named ELECTRICITY. Then the program creates a series plot of time versus kilowatt hours. The MARKERS option tells SAS to add a marker for each data point on the line.

```

DATA electricity;
INFILE 'c:\MyRawData\Hourly.dat';
INPUT Time kWh @@;
RUN;
* Plot temperatures by time;
PROC SGPLOT DATA = electricity;
SERIES X = Time Y = kWh / MARKERS;
TITLE 'Hourly Use of Electricity';
RUN;

```

The plot looks like this:



Note that the data did not need to be sorted because they were already ordered by the variable on the X-axis (Time).

## 8.7 Creating Fitted Curves with PROC SGPLOT

Scatter plots show the relationship between two variables. One way to explore that relationship further is to plot a fitted curve. The SGLOT procedure produces several kinds of fitted curves including regression lines, loess curves, and penalized B-spline curves. To create any of these types of fitted curves, use a statement with this general form:

*statement-name* **X** = *horizontal-variable* **Y** = *vertical-variable* / *options*;

Where the *statement-name* can be:

REG	regression line or curve
LOESS	loess curve
PBSPLINE	penalized B-spline curve

Options for fitted curves include:

ALPHA = $n$	specifies the level for the confidence limits. The value of $n$ must be between 0 (100% confidence) and 1 (0% confidence). The default is 0.05 (95% confidence limits).
CLI	adds prediction limits for individual predicted values (for REG and

	PBSPLINE only).
CLM	adds confidence limits for mean predicted values.
CURVELABEL = 'text'	adds a label for the curve. If you do not specify a text string, then SAS uses the label from the Y variable.
GROUP = <i>variable-name</i>	specifies a variable to be used for grouping the data. A separate line is created for each unique value of the grouping variable.
NOLEGCLI	removes the legend entry for the CLI band.
NOLEGCLM	removes the legend entry for the CLM band.
NOLEGFIT	removes the legend entry for the fit curve.
NOMARKERS	removes markers for data points.
CLMTRANSPAREN CY = <i>n</i>	specifies the degree of transparency for the confidence limits. The value of <i>n</i> must be between 0 (the default) and 1, with 1 being completely transparent and 0 completely opaque.
TRANSPARENCY =	specifies the degree of transparency.

*n*

for the plot line. The value of *n* must be between 0 (the default) and 1, with 1 being completely transparent and 0 completely opaque.

Each type of fitted curve offers additional options for controlling the parameters for the interpolation. See the SAS Documentation for more information.

**Example** A runner decides that she would like to improve her time in the 1500 meter run. In order to track her progress, she records her best time each week. The data values are the week (1 to 28) and time in seconds. Note that each line of data contains several observations.

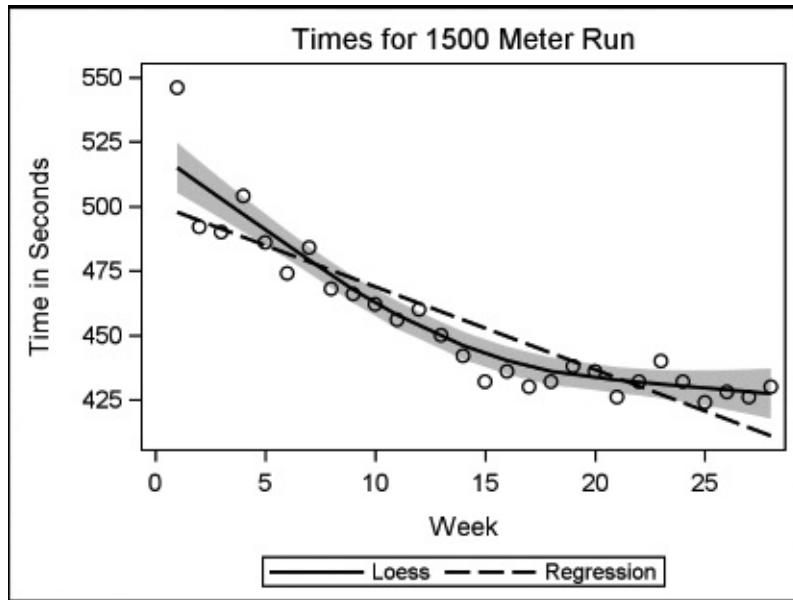
```
1 546 2 492 3 490 4 504 5 486 6 474 7 484  
8 468 9 466 10 462 11 456 12 460 13 450 14 442  
15 432 16 436 17 430 18 432 19 438 20 436 21 426  
22 432 23 440 24 432 25 424 26 428 27 426 28 430
```

This program reads the data and overlays two fitted curves for the times: a loess plot and a regression plot. The CLM option in the LOESS statement generates the 95% confidence limit band for the mean predicted values, while the NOLEGCLM option tells SAS not to include the confidence limit band in the legend.

```
DATA weekly1500;  
  INFILE 'C:\MyRawData\Weekly1500.dat';  
  INPUT Week Time @@;  
RUN;  
PROC SGPLOT DATA = weekly1500;  
  LOESS X = Week Y = Time / NOMARKERS CLM  
  NOLEGCLM;  
  REG X = Week Y = Time;  
  LABEL Time = 'Time in Seconds';
```

```
TITLE 'Times for 1500 Meter Run';  
RUN;
```

Here is the plot of run times (shown in the JOURNAL style):



This graph shows the data points with both a regression line and a loess curve. Because the NOMARKERS option was included in the LOESS statement, the data points are plotted only once. The confidence limits for mean predicted values, based on the loess fit, are shown by the gray band surrounding the loess line.

## 8.8 Controlling Axes and Reference Lines in PROC SGPlot

Statements like VBAR and LOESS tell SAS the type of graph to create. However, the SGPlot procedure also has supporting statements that allow you to control other features of your graph, such as axes and reference lines.

**Axes** To specify options for the horizontal axis, use a statement with this general form:

```
XAXIS options;
```

For the vertical axis, replace the keyword XAXIS with YAXIS. Options include:

GRID creates a line at each tick mark on the axis.

LABEL = '*text-string*' specifies a text string enclosed in quotes to be used as the label for the axis. You can also use an ordinary LABEL statement, but a label specified using an AXIS statement will override one from any other source. If there is no variable label, then SAS uses the variable name.

TYPE = *axis-type* specifies the type of axis. DISCRETE is the default for character variables. LINEAR is the default for numeric variables. TIME is the default for variables that have date, time, or datetime formats associated with them. LOG specifies a logarithmic scale.

VALUES = (*values-list*) specifies values for tick marks on axes. Values must be enclosed in parentheses, and can be specified either as a list (0 5 10 15 20) or a range (0 TO 20 BY 5).

**Reference lines** Adding reference lines to a graph shows which points are above or below important levels. To add a horizontal or vertical reference line, use a REFLINE statement.

## REFLINE *values / options*;

The values are the points at which the reference lines should be drawn. You can specify values as a list, 0 5 10 15 20; a range, 0 TO 20 BY 5; or the name of a variable whose values will be used. Options include:

AXIS = *axis*

specifies the axis that contains the reference line values, either X or Y. The default is the Y-axis.

LABEL = (*label-list*)

specifies one or more text strings (each enclosed in quotes and separated by spaces) to be used as labels for the reference lines.

TRANSPARENCY  
Y = *n*

specifies the degree of transparency for the reference line. The value of *n* must be between 0 (the default) and 1, with 1 being completely transparent and 0 completely opaque.

If the REFLINE statement comes before any plot statements, then the line will be drawn behind the plot elements. If it comes afterward, then the line will be drawn in front of the plot elements.

**Example** This example compares average high temperatures in three cities: International Falls, Minnesota; Raleigh, North Carolina; and Yuma, Arizona. The variables are month and the high temperatures for each city. Temperatures are in Fahrenheit. There are three months in each line.

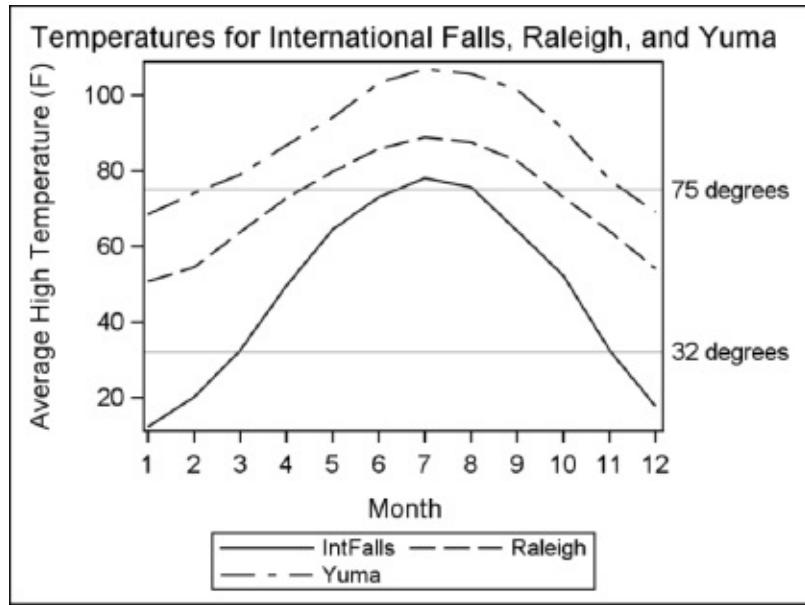
```
1 12.2 50.7 68.5 2 20.1 54.5 74.1 3 32.4 63.7 79.0
4 49.6 72.7 86.7 5 64.4 79.7 94.1 6 73.0 85.8 103.1
```

```
7 78.1 88.7 106.9 8 75.6 87.4 105.6 9 64.0 82.6  
101.5  
10 52.2 72.9 91.0 11 32.5 63.9 77.5 12 17.8 54.1 68.9
```

The following program has three SERIES statements, one for each city. Reference lines will be drawn at 32 and 75 degrees. In this data set, the variable Month is simply a number (1–12) rather than a SAS date value. In order to avoid having values of month, such as 3.5, the axis type has been set to DISCRETE using an XAXIS statement. A YAXIS statement specifies an axis label.

```
DATA cities;  
  INFILE 'c:\MyRawData\ThreeCities.dat';  
  INPUT Month IntFalls Raleigh Yuma @@;  
  RUN;  
  * Plot average high and low temperatures by city;  
  PROC SGPLOT DATA = cities;  
    SERIES X = Month Y = IntFalls;  
    SERIES X = Month Y = Raleigh;  
    SERIES X = Month Y = Yuma;  
    REFLINE 32 75 / LABEL = ('32 degrees' '75 degrees')  
    TRANSPARENCY = 0.5;  
    XAXIS TYPE = DISCRETE;  
    YAXIS LABEL = 'Average High Temperature (F)';  
    TITLE 'Temperatures for International Falls, Raleigh,  
    and Yuma';  
  RUN;
```

Here is the plot of temperatures (shown in the JOURNAL style):



## 8.9 Controlling Legends and Insets in PROC SGPlot

The SGPLOT procedure generates legends automatically for your plots when appropriate. This is great because then you don't have to think about them. But sometimes you may want to remove the legend, or move it to a different place, or add a note or comment of your own.

**Changing legends** You can change many aspects of a legend using the KEYLEGEND statement with this general form:

`KEYLEGEND / options;`

Options for legends include:

`ACROSS = n` specifies the number of columns in the legend.

`DOWN = n` specifies the number of rows in the legend.

**LOCATION** = *value* specifies the location for the legend, either IN the axis area or OUTSIDE (the default)

**NOBORDER** removes the border around the legend

**POSITION** = *value* specifies the position of the legend, either TOPLEFT, TOPRIGHT, BOTTOM (the default), BOTTOMLEFT, BOTTOMRIGHT, LEFT, or RIGHT.

**Removing legends** Sometimes you don't want a legend. To remove it, simply add the option NOAUTOLEGEND to the PROC SGLOT statement.

`PROC SGLOT DATA = data-set NOAUTOLEGEND;`

If you have both a KEYLEGEND statement and the NOAUTOLEGEND option, then the NOAUTOLEGEND option will be ignored.

**Adding insets** To place text in the axis area use an INSET statement with this general form:

`INSET 'text-string-1' 'text-string-2' ... 'text-string-n' /  
options;`

If you have more than one text string, then the strings will be placed one below the other. Options for insets include:

**BORDER** adds a border.

**POSITION** = *value* specifies the position of the inset, either TOPLEFT, TOPRIGHT, BOTTOM (the default), BOTTOMLEFT, BOTTOMRIGHT, LEFT, or RIGHT.

**Example** This example uses the permanent SAS data set created in Section 8.5. The data are about birds. For each species, there are four variables: name, type (S for songbirds or R for raptors), length in cm (from tip of beak to tip of tail), and wingspan in cm.

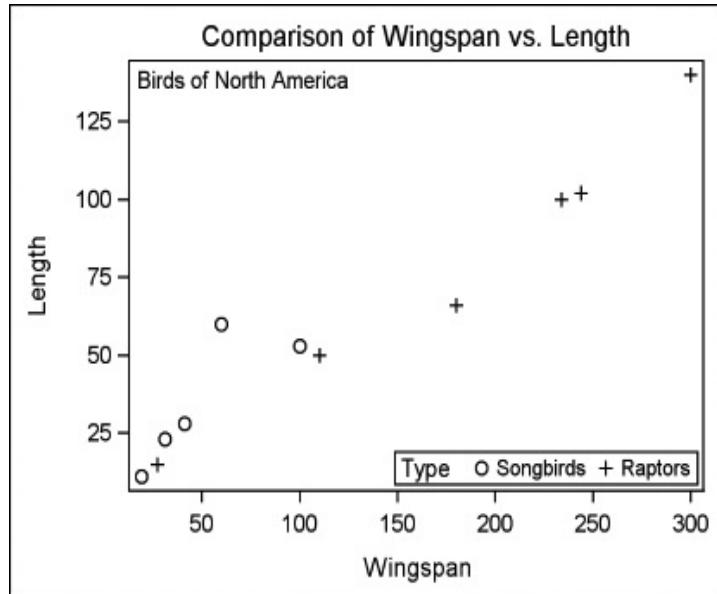
In this program, a SCATTER statement plots wingspan versus length. The KEYLEGEND statement specifies that the legend should be located inside the graph area in the bottom right corner. The INSET statement places a note in the top left corner of the graph.

```
LIBNAME flight 'c:\MySASLib';
* Plot Wingspan by Length;
PROC FORMAT;
  VALUE $birdtype
    'S' = 'Songbirds'
    'R' = 'Raptors';
RUN;

PROC SGPLOT DATA = flight.wings;
  SCATTER X = Wingspan Y = Length / GROUP =
Type;
  KEYLEGEND / LOCATION =
INSIDE POSITION =
BOTTOMRIGHT;
  INSET 'Birds of North America' /
POSITION = TOPLEFT;
  FORMAT Type $birdtype.;
  TITLE 'Comparison of Wingspan vs. Length';
RUN;
```

Here is the plot with an inset and a new legend (shown in

the JOURNAL style):



## 8.10 Customizing Graph Attributes in PROC SGPlot

When you create graphs, you want them to be attractive and easy to read. That's why SAS has style templates that have been designed specifically for use with graphs (listed in Section 8.1). Still, there may be times when you want stars instead of circles, or thicker lines, or a different color. Fortunately, the SGPlot procedure includes options for controlling graph attributes. To use these options, put them after a slash at the end of a basic plot statement. For example, this SCATTER statement tells SAS to use the STAR symbol for markers:

```
SCATTER X = Score Y = HoursOfStudy /  
MARKERATTRS = (SYMBOL = STAR);
```

There are many options for controlling graph attributes. Some common ones are:

FILLATTRS = ( <i>attribute</i> = <i>value</i> )	specifies the appearance of filled area. Attributes include COLOR=.
LABELATTRS = ( <i>attribute</i> = <i>value</i> )	specifies the appearance of axis labels. Attributes include COLOR=, SIZE=, STYLE=, and WEIGHT=.
LINEATTRS = ( <i>attribute</i> = <i>value</i> )	specifies the appearance of line. Attributes are COLOR=, PATTERN=, and THICKNESS=.
MARKERATTRS = ( <i>attribute</i> = <i>value</i> )	specifies the appearance of marker. Attributes include COLOR=, SIZE=, and SYMBOL=.
VALUEATTRS = ( <i>attribute</i> = <i>value</i> )	specifies the appearance of axis tick labels. Attributes include COLOR=, SIZE=, STYLE=, and WEIGHT=.

Each attribute has many possible values. Here are just a few:

Attribute	Possible Values
COLOR	RGB notation such as #FF0000 (red) or named colors like red, blue, green, etc.

=	values such as RED, plus many others
PATTER	SOLID, DASH, SHORTDASH, LONGDASH,
N=	DOT, DASHDASHDOT, or DASHDOTDOT
SIZE=	numbers with the units CM, IN, MM, PCT, PT, or PX (the default)
STYLE=	ITALIC or NORMAL (the default)
SYMBOL	CIRCLE, CIRCLEFILLED, DIAMOND,
L=	DIAMONDFILLED, PLUS, SQUARE,
	SQUAREFILLED, STAR, STARFILLED,
	TRIANGLE, or TRIANGLEFILLED
THICKNESS=	numbers with the units CM, IN, MM, PCT, PT, or PX (the default)
WEIGHT	BOLD or NORMAL
T=	

Of course, not all types of plots support all graph attributes. For example, you cannot use FILLATTRS= with a scatter plot because scatter plots don't have any filled areas. There are additional graph attributes. For a complete list, check the SAS Documentation.

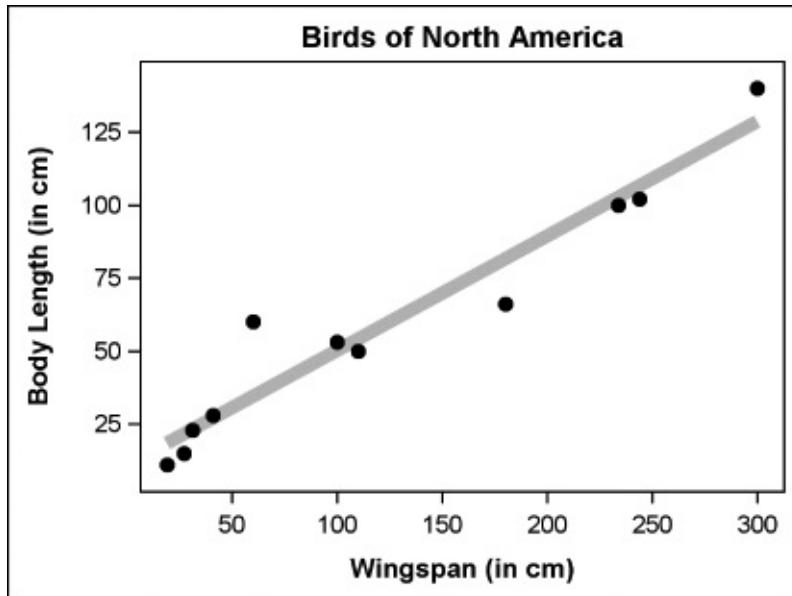
**Example** This example uses the permanent SAS data set created in Section 8.5. The data are about birds. For each species, there are four variables: name, type (S for songbirds or R for raptors), length in cm (from tip of beak to tip of tail), and wingspan in cm.

The following program produces two plots. First, a REG

statement plots a regression line for wingspan versus length with a line that is 2mm thick and 75% transparent. Then a SCATTER statement plots the data points using a filled circle 2mm in size. The axis labels and title are bold.

```
LIBNAME flight 'c:\MySASLib';
* Plot Wingspan by Length;
PROC SGPlot DATA = flight.wings
NOAUTOLEGEND;
    REG X = Wingspan Y = Length /
        LINEATTRS = (THICKNESS = 2MM)
TRANSPARENCY = .75;
    SCATTER X = Wingspan Y = Length /
        MARKERATTRS = (SYMBOL = CIRCLEFILLED
SIZE = 2MM);
    TITLE BOLD 'Birds of North America';
    XAXIS LABEL = 'Wingspan (in cm)' LABELATTRS
= (WEIGHT = BOLD);
    YAXIS LABEL = 'Body Length (in cm)'
LABELATTRS = (WEIGHT = BOLD);
RUN;
```

Here is the graph with new attributes:



## 8.11 Creating Paneled Graphs with PROC SGPANEL

The SGPANEL procedure is a close cousin of the SGPlot procedure. The SGPANEL procedure produces nearly all the same types of graphs as the SGPlot procedure, but while SGPlot produces single-celled graphs, SGPANEL can produce multicelled graphs. PROC SGPANEL produces a separate cell for each combination of values of the classification variables that you specify. Each of those cells uses the same variables on their X- and Y-axes.

The syntax for PROC SGPANEL is almost identical to PROC SGPlot, so it is easy to convert one to the other by making just a couple of changes to your code. You simply replace the keyword SGPlot with SGPANEL and add a PANELBY statement, like this:

```
PROC SGPANEL;
  PANELBY variable-list / options;
  plot-statement;
```

The PANELBY statement must appear before any statements that create plots. Possible options include:

COLUMNS = *n* specifies the number of columns in the panel.

MISSING specifies that observations with missing values for the PANELBY variable should be included.

NOVARNAM E removes the variable name from cell headings.

NOHEADERB ORDER removes the border from the cell headings.

ROWS = *n* specifies the number of rows in the panel.

SPACING = *n* specifies the number of pixels between rows and columns in the panel. The default is 0.

UNISCALE = *value* specifies which axes will share the same range of values. Possible values are COLUMN, ROW, and ALL (the default).

Instead of XAXIS and YAXIS statements, the SGPMANL procedure uses COLAXIS and ROWAXIS statements to control axes. See Section 8.8 for axis options.

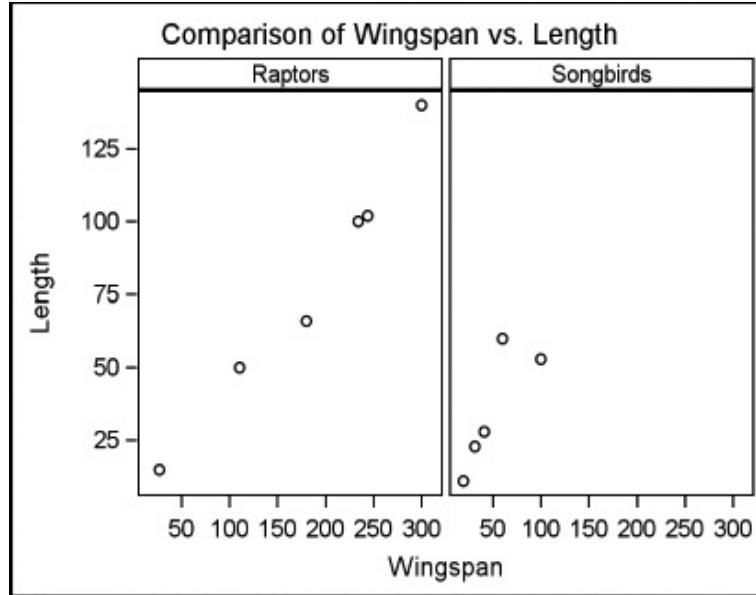
**Example** This example uses the permanent SAS data set created in Section 8.5. The data are about birds. For each species, there are four variables: name, type (S for

songbirds or R for raptors), length in cm (from tip of beak to tip of tail), and wingspan in cm.

The following program produces a paneled plot. This plot is similar to the grouped plot shown in Section 8.5. In that graph, data for songbirds and raptors are overlaid in a single cell. In this example, the two groups are plotted in separate cells. The NOVARNAME option removes the word "Type=" from the column headings, and the SPACING= option inserts a little space between the two cells. This example also uses PROC FORMAT to create a user-defined format for the variable Type so that the column headings are words instead of the coded values R and S.

```
LIBNAME flight 'c:\MySASLib';
* Plot Wingspan by Length;
PROC FORMAT;
  VALUE $birdtype
    'S' = 'Songbirds'
    'R' = 'Raptors';
RUN;
PROC SGPANEL DATA = flight.wings;
  PANELBY Type / NOVARNAME SPACING = 5;
  SCATTER X = Wingspan Y = Length;
  FORMAT Type $birdtype.:
  TITLE 'Comparison of Wingspan vs. Length';
RUN;
```

Here is the graph with a panel for each type of bird:



Note that you could have used a standard BY statement with PROC SGPlot instead of a PANELBY statement with PROC SGPANEL, but then SAS would have produced two completely separate graphs (one for raptors and another for songbirds) instead of two cells within a single graph. Also, when you use a BY statement, the data must be presorted by the values of the BY variables, but when you use a PANELBY statement, the data do not need to be sorted.

## 8.12 Specifying Image Properties and Saving Graphics Output

If you are writing a paper or creating a presentation, you may need to access individual graphs. You may be able to simply copy and paste images when you view them in SAS, and you can always save or download your results in formats like HTML, PDF, or RTF. Sometimes that may be all you need. However, at other times you may want to specify the properties of your graphs or save them in separate graphics files for later use.

**Specifying properties of images** To specify properties for your images, use the ODS GRAPHICS statement with this general form:

ODS GRAPHICS / *options*;

Options include:

HEIGHT = *n* specifies the image height in CM, IN, MM, PT, or PX.

IMAGENAME = '*filename*' specifies the base filename for the image. The default name for an image file is the name of its ODS output object. See Section 5.12 for a discussion of output objects.

OUTPUTFMT = *file-type* specifies the graph format. The default varies by destination. Possible values include BMP, GIF, JPEG, PDF, PNG, PS, SVG, TIFF, and many others.

RESET resets options to their defaults.

WIDTH = *n* specifies the image width in CM, IN, MM, PT, or PX.

In most cases, the default size for graphs is 640 pixels wide by 480 pixels high. If you specify only one dimension (width but not height, or vice versa), then SAS will adjust the other dimension to maintain a default aspect ratio of 4:3.

When you save image files, SAS will append numerals to the end of the image name. For example, if you specify an

image name of Final, then your files will be named Final, Final1, Final2, and so on. If you rerun your code, SAS will, by default, continue counting so that the new files will not overwrite the old. Specifying the RESET option before the IMAGENAME= option tells SAS to start over each time.

**Saving graphical output** LISTING is a good destination for capturing individual graphs since it offers the most image formats and saves images in separate files. To create stand-alone graphs, use an ODS LISTING statement with a GPATH= option, like this:

ODS LISTING GPATH = '*path*' *options*;

where *path* is the location where your image should be saved. Options include:

IMAGE_DPI = *n* specifies the image resolution. The default is 96.

STYLE = *style-name* specifies a style template. See Section 8.1 for a list of possible styles.

This statement would save ODS Graphics images in individual files in a folder named MyGraphs on the C drive using the STATISTICAL style and 300 dots per inch:

ODS LISTING GPATH = 'c:\MyGraphs' STYLE = STATISTICAL IMAGE_DPI = 300;

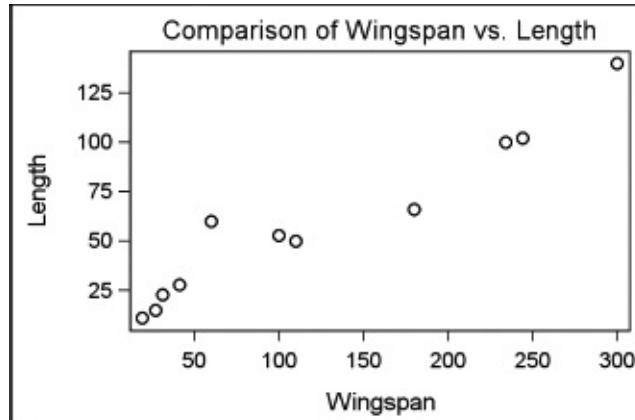
**Example** This example uses the permanent SAS data set created in Section 8.5. The data are about birds. For each species, there are four variables: name, type (S for songbirds or R for raptors), length in cm (from tip of beak to tip of tail), and wingspan in cm.

The following program produces a scatter plot, and sends it to the ODS LISTING destination using the JOURNAL

style. This graph will be saved in the MyGraphs directory on the C drive, be named BirdGraph, be in BMP format, and be two inches high and three inches wide. The final ODS GRAPHICS statement resets all the image properties back to the default values so that any subsequent graphs you generate will be normal.

```
LIBNAME flight 'c:\MySASLib';
* Create BMP image of Wingspan by Length;
ODS LISTING GPATH = 'c:\MyGraphs' STYLE =
JOURNAL;
ODS GRAPHICS / RESET IMAGENAME =
'BirdGraph' OUTPUTFMT = BMP
HEIGHT = 2IN WIDTH = 3IN;
PROC SGPLOT DATA = flight.wings;
SCATTER X = Wingspan Y = Length;
TITLE 'Comparison of Wingspan vs. Length';
RUN;
ODS GRAPHICS / RESET;
```

Here is the new graph:



Note that the file named BirdGraph will not open automatically, but you can navigate to the image file in your operating environment and open it using a viewer designed for that type of file.

# 9

“33½% of the mice used in the experiment were cured by the test drug; 33½% of the test population were unaffected by the drug and remained in a moribund condition; the third mouse got away.”

ERWIN NETER

From “How to Write a Scientific Paper” by Robert A. Day, ASM News, vol. 41, no. 7, pp 486-494, July 1975. Reprinted by permission of publisher and author. Also appears in *How to Write and Publish a Scientific Paper* 4th edition by Robert A. Day, copyright 1994 by Oryx Press.

# CHAPTER 9

## Using Basic Statistical Procedures

- 9.1 Examining the Distribution of Data with PROC UNIVARIATE
- 9.2 Creating Statistical Graphics with PROC UNIVARIATE
- 9.3 Producing Statistics with PROC MEANS
- 9.4 Testing Means with PROC TTEST
- 9.5 Creating Statistical Graphics with PROC TTEST
- 9.6 Testing Categorical Data with PROC FREQ
- 9.7 Creating Statistical Graphics with PROC FREQ
- 9.8 Examining Correlations with PROC CORR
- 9.9 Creating Statistical Graphics with PROC CORR
- 9.10 Using PROC REG for Simple Regression Analysis
- 9.11 Creating Statistical Graphics with PROC REG
- 9.12 Using PROC ANOVA for One-Way Analysis of Variance
- 9.13 Reading the Output of PROC ANOVA

- 9.1 Examining the Distribution of Data with PROC UNIVARIATE

When you are doing statistical analysis, you usually have a goal in mind, a question that you are trying to answer, or a hypothesis you want to test. But before you jump into statistical tests, it is a good idea to pause and do a little exploration. A good procedure to use at this point is PROC UNIVARIATE.

PROC UNIVARIATE, which is part of Base SAS, produces statistics and graphs describing the distribution of a single variable. The statistics include the mean, median, mode, standard deviation, skewness, and kurtosis.

Using PROC UNIVARIATE is fairly simple. After the PROC statement, you specify one or more numeric variables in a VAR statement:

```
PROC UNIVARIATE;  
  VAR variable-list;
```

Without a VAR statement, SAS will calculate statistics for all numeric variables in your data set. You can specify other options in the PROC statement, if you want, such as NORMAL, which produces tests of normality:

```
PROC UNIVARIATE NORMAL;
```

**Example** The following data consist of test scores from a statistics class. Each line contains scores for 10 students.

```
56 78 84 73 90 44 76 87 92 75  
85 67 90 84 74 64 73 78 69 56  
87 73 100 54 81 78 69 64 73 65
```

This program reads the data from a file called Scores.dat and then runs PROC UNIVARIATE:

```
LIBNAME f2020 'c:\MySASLib';  
DATA f2020.statclass;  
  INFILE 'c:\MyRawData\Scores.dat';  
  INPUT Score @@;  
RUN;  
*Summarize data using PROC UNIVARIATE;
```

```

PROC UNIVARIATE DATA = f2020.statclass;
  VAR Score;
  TITLE;
  RUN;

```

The output appears on the next page. The output starts with basic information about your distribution: number of observations (N), mean, and standard deviation. Skewness indicates how asymmetrical the distribution is (whether it is more spread out on one side), while kurtosis indicates how flat or peaked the distribution is. The normal distribution has values of 0 for both skewness and kurtosis. Other sections of the output contain three measures of central tendency: mean, median, and mode; tests of the hypothesis that the population mean is 0; quantiles; and extreme observations (in case you have outliers).

## The UNIVARIATE Procedure Variable: Score

Moments		
N	30	Sum Weights
Mean	74.6333333	Sum Observations
Std Deviation	12.5848385	Variance
Skewness	-0.3495061	Kurtosis
Uncorrected SS	171697	Corrected SS
Coeff Variation	16.8622222	Std Error Mean

Basic Statistical Measures		
Location		Variability
Mean	74.63333	Std Deviation
Median	74.50000	Variance
Mode	73.00000	Range
		Interquartile Range

Tests for Location: Mu0=0			
Test	Statistic	p-value	Method
Student's t	t	32.48223	Pr >  t
Sign	M	15	Pr >=  M
Signed Rank	S	232.5	Pr >=  S

Quantiles (Definition 5)	
Quantile	

<b>100% Max</b>	
99%	
95%	
90%	
<b>75% Q3</b>	
<b>50% Median</b>	
<b>25% Q1</b>	
10%	
5%	
1%	
<b>0% Min</b>	

Extreme Observations			
Lowest		Highest	
	Value	Obs	Value
	44	6	87

	54	24	90
	56	20	90
	56	1	92
	64	28	100

## 9.2 Creating Statistical Graphics with PROC UNIVARIATE

The UNIVARIATE procedure can produce several graphs that are useful for data exploration. For example, histograms are a good way to visualize the distribution of a variable, while probability and quantile-quantile plots can show how the data compare to theoretical distributions. To produce these graphs, include the desired plot request statement. Here is the general form of PROC UNIVARIATE with plot requests:

```
PROC UNIVARIATE;
  VAR variable-list;
  Plot-request variable-list / options;
```

**Plot requests** The following graphs can be requested with PROC UNIVARIATE:

CDFPLOT cumulative distribution function plot

HISTOGRAM histogram

PPPLOT probability-probability plot

PROBPLOT probability plot

## QQPLOT      quantile-quantile plot

If no variable list is specified, then plots will be produced for all variables in the VAR statement. If there is no VAR statement or variable list, then plots will be produced for all numeric variables.

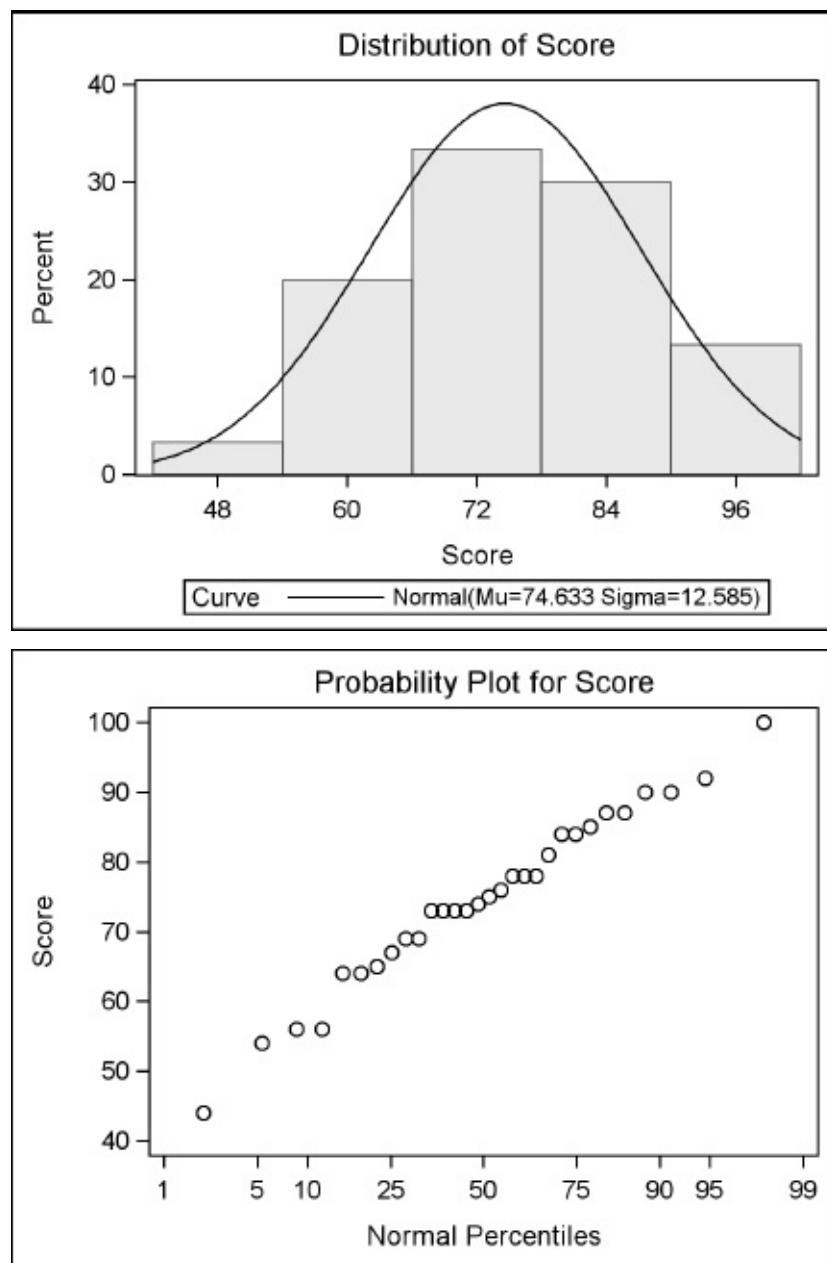
**Plot options** The CDFPLOT and HISTOGRAM plots show the distribution of the specified variable. To overlay a curve showing a standard distribution, specify the desired distribution with a plot option. Available distribution options include: BETA, EXPONENTIAL, GAMMA, LOGNORMAL, NORMAL, and WEIBULL. The PPLOT, PROBPLOT, and QQPLOT statements use the normal distribution as the default. If you would like to use a different distribution, specify it with a plot option. For example, to create a probability plot of the variable Score using the exponential distribution, use the following:

```
PROBPLOT Score / EXPONENTIAL;
```

**Example** This example uses the data from the previous section, which consist of test scores from 30 students in a statistics class. The following program creates a histogram of the Score variable with the normal distribution overlaid, and a probability plot using the normal distribution:

```
LIBNAME f2020 'c:\MySASLib';
*Create histogram and probability plot for variable
Score;
PROC UNIVARIATE DATA = f2020.statclass;
  VAR Score;
  HISTOGRAM Score / NORMAL;
  PROBPLOT Score;
  TITLE;
  RUN;
```

Here are the two plots: a histogram and a probability plot. (The NORMAL option in the HISTOGRAM statement produces additional tabular results that are not shown.) The relatively linear pattern formed by the points in the probability plot indicate that the data are closely matched to the normal distribution.



### 9.3 Producing Statistics with PROC

## MEANS

Most of the descriptive statistics that you produce with PROC UNIVARIATE you can also produce with PROC MEANS. PROC UNIVARIATE is useful when you want an in-depth statistical analysis of the data distribution. But if you know you want only a few statistics, then PROC MEANS is a better way to go. With PROC MEANS you can ask for just the statistics you want. The MEANS procedure does not produce any ODS Graphics.

The MEANS procedure requires only one statement:

PROC MEANS *statistic-keywords*;

If you do not include any statistic keywords, then MEANS will produce the mean, the number of nonmissing values, the standard deviation, the minimum value, and the maximum value for each numeric variable. The following table shows statistics you can request. (Some statistics have two names; the alternate name is shown in parentheses.) If you add any statistic keywords in the PROC MEANS statement, then MEANS will no longer produce the default statistics—you must request all the statistics you want.

CLM	two-sided confidence limits	RAN	range GE
CSS	corrected sum of squares	SKE	skewness WN ESS
CV	coefficient of variation	STD	standard deviation DEV
KURTO	kurtosis	STD	standard error of th

SIS		ERR	mean
LCLM	lower confidence limit	SUM	sum
MAX	maximum value	SUM	sum of weighted variables
		T	
MEAN	mean	UCL	upper confidence limit
MIN	minimum value	USS	uncorrected sum of squares variance
MODE	mode	VAR	variance
N	number of nonmissing values	PROBT	probability for Student's <i>t</i>
NMISS	number of missing values	T	Student's <i>t</i>
MEDIAN(P50)	median	Q3(P75)	75% quantile
Q1(P25)	25% quantile	P5	5% quantile
P1	1% quantile	P90	90% quantile
P10	10% quantile	P99	99% quantile

P95            95% quantile

**Confidence limits** The default confidence level for the confidence limits is .05 or 95%. If you want a different confidence level, then request it with the ALPHA= option in the PROC MEANS statement. For example, if you want 90% confidence limits, then specify ALPHA=.10 along with the CLM option. Then the PROC MEANS statement would look like this:

```
PROC MEANS ALPHA = .10 CLM;
```

**VAR statement** By default, PROC MEANS will produce statistics for all numeric variables in your data set. If you do not want all the variables, then specify the ones you want in the VAR statement. Here is the general form of the MEANS procedure with the VAR statement:

```
PROC MEANS options;  
  VAR variable-list;
```

**Example** Your friend is an aspiring author of children's books. To increase her chances of getting her books published, she wants to know how many pages her books should have. At the local library, she counts the number of pages in a random selection of children's picture books. Here are the data:

```
34 30 29 32 52 25 24 27 31 29  
24 26 30 30 30 29 21 30 25 28  
28 28 29 38 28 29 24 24 29 31  
30 27 45 30 22 16 29 14 16 29  
32 20 20 15 28 28 29 31 29 36
```

To determine the average number of pages in children's picture books, use the MEANS procedure. PROC MEANS

can also produce the median number of pages, as well as the 90% confidence limits. Here is the program that will read the data and produce the desired statistics:

```
DATA booklengths;  
  INFILE 'c:\MyRawData\Picbooks.dat';  
  INPUT NumberOfPages @@;  
RUN;  
*Produce summary statistics;  
PROC MEANS DATA = booklengths N MEAN  
  MEDIAN CLM ALPHA = .10;  
  TITLE 'Summary of Picture Book Lengths';  
RUN;
```

Here are the results of the MEANS procedure:

### **Summary of Picture Book Lengths**

#### **The MEANS Procedure**

Analysis Variable : NumberOfPages				
N	Mean	Median	Lower 90% CL for Mean	Upper 90% CL for Mean
50	28.0000000	29.0000000	26.4419136	29.5580864

The average number of pages in the children's books sampled was 28. The median value of 29 says that half the books sampled had 29 pages or fewer. The confidence limits tell us that we are 90% certain that the true population mean (all children's picture books) falls between 26.44 and 29.56 pages. From this analysis your friend concludes that she should make her books between 26 and 30 pages long to maximize her chances of getting published (of course subject matter and writing style might also help).

## 9.4 Testing Means with PROC

### TTEST

As you would expect from its name, the TTEST procedure, which is part of SAS/STAT software, computes  $t$  tests. You use  $t$  tests when you want to compare means. Suppose, for example, that a statistics instructor selected a random sample of students to have extra tutoring. She could test whether the true mean of their scores was above a specific level (called a one-sample test), she could compare the scores of students who had tutoring to those who did not (a two independent sample test), and she could compare the scores of students before and after tutoring (a paired test). PROC TTEST performs all these types of tests.

**One sample comparisons** To compute a  $t$  test for a single mean, you list that variable in a VAR statement. SAS will test whether the mean is significantly different from  $H_0$ , a specified null value. The default value of  $H_0$  is zero. You can specify a different value using the H0= option.

```
PROC TTEST H0 = n options;  
    VAR variable;
```

**Two independent sample comparisons** To compare two independent groups, you use a CLASS and a VAR statement. In the CLASS statement, you list the variable that distinguishes the two groups. In the VAR statement, you list the response variable.

```
PROC TTEST options;  
    CLASS variable;  
    VAR variable;
```

**Paired comparisons** When the variables that you are comparing are paired, you use a PAIRED statement. The simplest form of this statement lists the two variables to be compared, separated by an asterisk.

```
PROC TTEST options;
```

PAIRED *variable1* * *variable2*;

**Options** Here are a few of the options available:

ALPHA = *n* specifies the level for the confidence limits. The value of *n* must be between 0 (100% confidence) and 1 (0% confidence). The default is 0.05 (95% confidence limits).

CI = *type* specifies the type of confidence interval for the standard deviation. If you don't specify this option, then by default the value of *type* is EQUAL, which produces an equal-tailed confidence interval. Other possible values are UMPU for an interval based on the uniformly most powerful unbiased test, and NONE to request no confidence interval for the standard deviation.

H0 = *n* requests a test of the hypothesis  $H_0 = n$ . The default value is 0.

NOBYVAR moves the names of the variables from the title to the output table.

SIDES = *type* specifies whether the *p*-value and confidence interval are one or two-tailed. Possible values of *type* are 2 (the default) for two-tailed, L (for a lower one-sided test) or U (for an upper one-sided test).

**Example** The following data give the finishing times for semifinal and final races of the women's 50-meter freestyle

swim. Each swimmer's initials are followed by their final time and semifinal time in seconds. Each line of data contains times for four swimmers.

```
RK 24.05 24.07 AH 24.28 24.45 MV 24.39 24.50 BS  
24.46 24.57  
FH 24.47 24.63 TA 24.61 24.71 JH 24.62 24.68 AV  
24.69 24.64
```

The following program reads the raw data and uses a paired *t* test to test the mean difference between the semifinal and final times:

```
LIBNAME results 'c:\MySASLib';  
DATA results.swim;  
  INFILE 'c:\MyRawData\Olympic50mSwim.dat';  
  INPUT Swimmer $ FinalTime SemiFinalTime @@;  
  RUN;  
*Perform paired t test for semifinal and final times;  
PROC TTEST DATA = results.swim;  
  TITLE '50m Freestyle Semifinal vs. Final Results';  
  PAIRED SemiFinalTime * FinalTime;  
  RUN;
```

Here are the tabular results produced by PROC TTEST. Graphical results are shown in the next section.

## 50m Freestyle Semifinal vs. Final Results

### The TTEST Procedure

Difference: SemiFinalTime - FinalTime

N	Mean	Std Dev	Std Err	Minimum
8	0.0850	0.0731	0.0258	-0.0500

Mean	95% CL Mean	Std Dev	95%
0.0850	0.0239	0.1461	0.0731 0.04

DF	t Value
7	3.29

In this example, the mean difference between each swimmer's semifinal time and their final time is 0.0850 seconds. The *t* test shows significant evidence ( $DF=7$ ,  $t=3.29$ ,  $p = 0.0133$ ) of a difference between the mean semifinal and final times.

## 9.5 Creating Statistical Graphics with PROC TTEST

The TTEST procedure uses ODS Graphics to produce several plots that help you visualize your data including histograms, box plots, and Q-Q plots. Many plots are generated by default, but you can control which plots are created using the PLOTS option in the PROC TTEST statement. Here is the general form of the PROC TTEST statement with plot options:

```
PROC TTEST PLOTS = (plot-request-list);
```

**Plot requests** The plots available to you depend on the type of comparison you request. Here are plots you can request for one sample, two sample, and paired  $t$  tests:

ALL

all

appropriate plots

BOXPLOT

box plots

HISTOGRAM

histograms  
overlaid with normal and  
kernel density curves

INTERVALPLOT

plots of  
confidence interval of  
means

NONE

no plots

QQPLOT

normal  
quantile-quantile (Q-Q) plot

**SUMMARYPLOT** one plot  
that includes both  
histograms and box plots

The following plots are also available for paired  $t$  tests:

**AGREEMENTPLOT** agreement  
 $t$  plots

**PROFILESPLOT** profiles plot

**Excluding automatic plots** By default the QQPLOT and SUMMARYPLOT plots are generated automatically for one sample, two sample and paired  $t$  tests. For paired  $t$  tests, the AGREEMENTPLOT and PROFILESPLOT are also generated by default. If you choose specific plots in the plot-list, the default plots will still be created unless you add the ONLY global option:

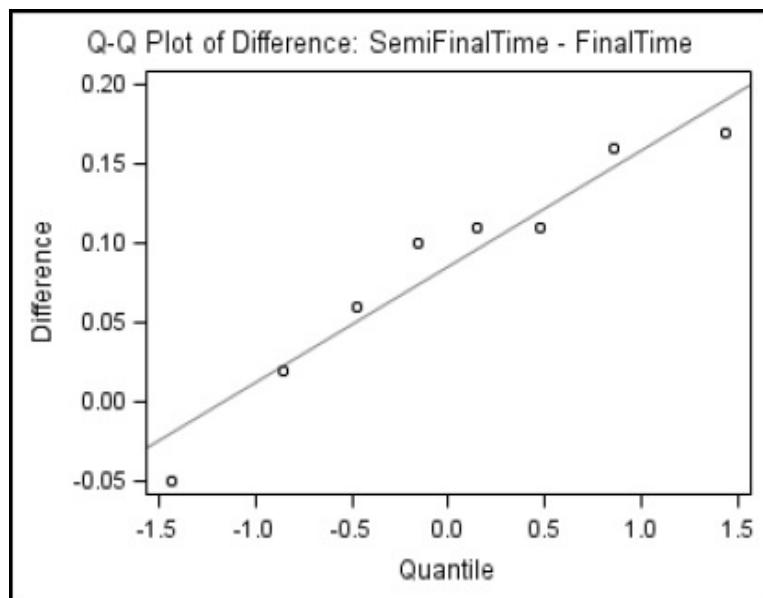
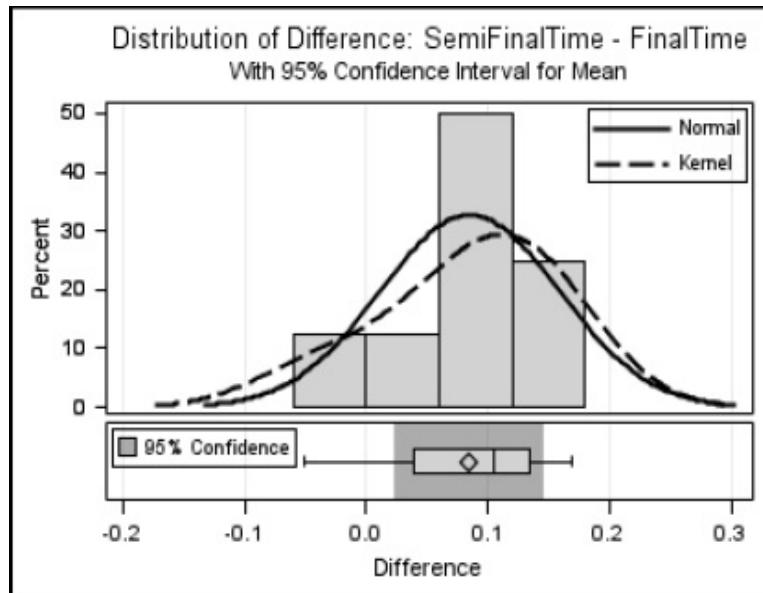
PROC TTEST PLOTS(ONLY) = (*plot-request-list*);

**Example** This example uses the data from the previous section, which gives the finishing times for semifinal (SemiFinalTime) and final races (FinalTime) of the women's 50-meter freestyle swim. The following program uses a paired  $t$  test to test the mean difference between the semifinal and final times, and requests just the Summary and Q-Q plots.

```
LIBNAME results 'c:\MySASLib';
*Produce just the Summary and QQ plots;
PROC TTEST DATA = results.swim PLOTS(ONLY) =
(SUMMARYPLOT QQPLOT);
TITLE '50m Freestyle Semifinal vs. Final Results';
PAIRED SemiFinalTime * FinalTime;
```

RUN;

Here are the results for the Q-Q and Summary plots. The tabular results for the paired *t* test were shown in the preceding section.



## 9.6 Testing Categorical Data with PROC FREQ

PROC FREQ, which is part of Base SAS, produces many statistics for categorical data. The best known of these is chi-square. One of the most common uses of PROC FREQ is to test the hypothesis of no association between two variables. Another use is to compute measures of association, which indicate the strength of the relationship between the variables. The general form of PROC FREQ is:

```
PROC FREQ;  
  TABLES variable-combinations / options;
```

**Options** Here are a few of the statistical options that you can request:

AGR tests and measures of classification agreement,  
EE including McNemar's test, Bowker's test,  
Cochran's Q test, and kappa statistics

CHIS chi-square tests of independence and measures of  
Q association

CL confidence limits for measures of association

CMH Cochran-Mantel-Haenszel statistics, typically for  
stratified two-way tables

EXA Fisher's exact test for tables larger than 2X2  
CT

MEA measures of association, including Pearson and

SURE Spearman correlation coefficients, gamma,  
S Kendall's tau-*b*, Stuart's tau-*c*, Somer's D, lambda  
odds ratios, risk ratios, and confidence intervals

RELR relative risk measures for 2X2 tables  
ISK

TRE Cochran-Armitage test for trend  
ND

**Example** One day your neighbor, who rides the bus to work, complains that the regular bus is usually late. He says the express bus is usually on time. Realizing that this is categorical data, you decide to test whether there really is a relationship between the type of bus and arriving on time. You collect data for type of bus (E for express or R for regular) and promptness (L for late or O for on time). Each line of data contains several observations.

```
E O E L E L R O E O E O E O R L R O R L R O E O R  
L E O R L R O E O  
E O R L E L E O R L E O R L E O R L E O R O E L E  
O E O E O E O E L  
E O E O R L R L R O R L E L E O R L R O E O E O E  
O E L R O R L
```

The following program reads the raw data and runs PROC FREQ with the CHISQ option:

```
LIBNAME trans 'c:\MySASLib';  
DATA trans.bus;  
  INFILE 'c:\MyRawData\Bus.dat';  
  INPUT BusType $ OnTimeOrLate $ @@;  
RUN;  
*Perform chi-square analysis on bus data;  
PROC FREQ DATA = trans.bus;  
  TABLES BusType * OnTimeOrLate / CHISQ;
```

```
TITLE;  
RUN;
```

Here are the results showing that the regular bus is late 61.90% of the time, while the express bus is late only 24.14% of the time. Assuming that bus type and arrival time are independent, the probability of obtaining a chi-square this large or larger by chance alone is 0.0071. So the data do support the idea that there is an association between type of bus and arrival time. The Fisher's exact test provides the same conclusion with a *p*-value of 0.0097.

## The FREQ Procedure

Table of BusType by OnTimeOrLate			
BusType	OnTimeOrLate		
	Frequency	L	O
	Percent		
	Row Pct		
	Col Pct		
E	7 14.00 24.14 35.00	22 44.00 75.86 73.33	
R	13 26.00 61.90 65.00	8 16.00 38.10 26.67	

<b>Total</b>	20	30
	40.00	60.00

## Statistics for Table of BusType by OnTimeOrLate

	<b>Statistic</b>	D F	Valu e	Pro b	
	<b>Chi-Square</b>	1 6	7.238 1	0.007 1	
	<b>Likelihood Ratio Chi-Square</b>	1 4	7.336 4	0.006 8	
	<b>Continuity Adj. Chi-Square</b>	1 5	5.750 5	0.016 5	
	<b>Mantel-Haenszel Chi-Square</b>	1 9	7.093 7	0.007 7	
	<b>Phi Coefficient</b>		-0.38 05		
	<b>Contingency Coefficient</b>		0.355 6		
	<b>Cramer's V</b>		-0.38		

<b>Fisher's Exact Test</b>	
<b>Cell (1,1) Frequency (F)</b>	
<b>Left-sided Pr &lt;= F</b>	
<b>Right-sided Pr &gt;= F</b>	
<b>Table Probability (P)</b>	
<b>Two-sided Pr &lt;= P</b>	

**Sample Size = 50**

## 9.7 Creating Statistical Graphics with PROC FREQ

The FREQ procedure uses ODS Graphics to produce

several plots that help you visualize your data including frequency plots, odds ratio plots, agreement plots, deviation plots, and two types of plots with Kappa statistics and confidence limits. Here is the general form of PROC FREQ with plot options:

```
PROC FREQ;  
  TABLES variable-combinations / options PLOTS =  
    (plot-list);
```

**Plot requests** The plots available to you depend on the type of table you request. For example, the DEVIATIONPLOT is available only for one-way tables when using the CHISQ option in the TABLES statement. Here are the plots that you can request along with the required option, if any, and type of table request:

<b>Plot Name</b>	<b>Table type</b>	<b>Required option statement</b>
AGREEPLOT	two-way	AGREE
CUMFREQPLOT	one-way	
DEVIATIONPLOT	one-way	CHISQ
FREQPLOT	any request	
KAPPAPLOT	three-way	AGREE
ODDSRATIOPLOT	hx2x2 T	MEASURES or RE
RELREISKPLOT	hx2x2	MEASURES or RE

RISKDIFFPLOT hx2x2 RISKDIFF

WTKAPPAPLOT hxr_r(r>2) AGREE

To produce a CUMFREQPLOT or a FREQPLOT, you must specify it in the PLOTS= option in the TABLES statement. Otherwise, if you do not specify any plots in the TABLE statement, then all plots associated with the table type you request will be produced by default.

**Plot options** Many options are available that control the look of the plots generated. For a complete list of options, see the SAS Documentation. For example, the FREQPLOT has options for controlling the layout of the plots for two-way tables. By default, the bars are grouped vertically. To group the bars horizontally, use the following:

```
TABLES variable1 * variable2 / PLOTS =  
    FREQPLOT(TWOWAY = GROUPTHORIZONTAL);
```

To stack the bars, use the TWOWAY=STACKED option.

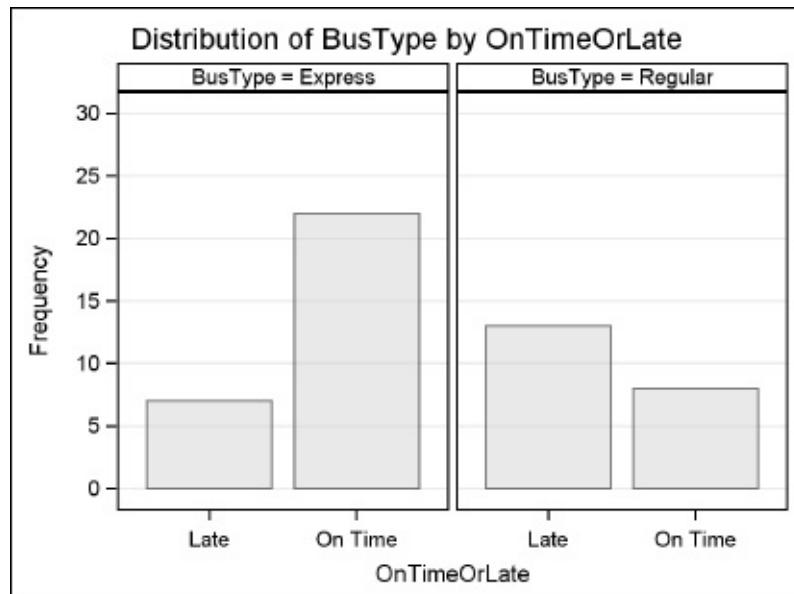
**Example** This example uses the same data as the previous section about promptness of buses. The data are for type of bus (E for express or R for regular) and promptness (L for late or O for on time). The following program uses PROC FREQ to request a two-way frequency table. The PLOTS=FREQPLOT option in the TABLES statement produces a frequency plot. Adding the TWOWAY=GROUPTHORIZONTAL option to FREQPLOT produces bars that are grouped horizontally instead of vertically. The FORMAT procedure creates formats that are applied to the BusType and OnTimeOrLate variables using a FORMAT statement in the FREQ procedure. This gives more descriptive labels to the plot.

```

LIBNAME trans 'c:\MySASLib';
PROC FORMAT;
  VALUE $type 'R'='Regular'
    'E'='Express';
  VALUE $late 'O'='On Time'
    'L'='Late';
RUN;
*Produce frequency plot grouped horizontally;
PROC FREQ DATA = trans.bus;
  TABLES BusType * OnTimeOrLate /
PLOTS=FREQPLOT(TWOWAY=GROUPHORIZONTAL);
  FORMAT BusType $Type. OnTimeOrLate $Late. ;
RUN;

```

Here is the plot. The tabular portion of the output, the frequency table, was shown in the previous section.



## 9.8 Examining Correlations with PROC CORR

The CORR procedure, which is included with Base SAS,

computes correlations. A correlation coefficient measures the strength of the linear relationship between two variables. If two variables are completely unrelated, they will have a correlation of 0. If two variables have a perfect linear relationship, they will have a correlation of 1.0 or –1.0. In real life, correlations fall somewhere between these numbers. The basic syntax for PROC CORR is rather simple:

```
PROC CORR;  
RUN;
```

By default, SAS will compute correlations between all possible pairs of the numeric variables. You can add the VAR and WITH statements to specify variables:

```
VAR variable-list;  
WITH variable-list;
```

Variables listed in the VAR statement appear across the top of the table of correlations, while variables listed in the WITH statement appear down the side of the table. If you use a VAR statement but no WITH statement, then the variables will appear both across the top and down the side.

By default, PROC CORR computes Pearson product-moment correlation coefficients. You can add options to the PROC statement to request nonparametric correlation coefficients. The SPEARMAN option in the statement below tells SAS to compute Spearman's rank correlations instead of Pearson's correlations:

```
PROC CORR SPEARMAN;
```

Other options include HOEFFDING (for Hoeffding's D statistic) and KENDALL (for Kendall's tau-*b* coefficient).

**Example** Each student in a statistics class recorded three values: test score, the number of hours spent watching videos in the week prior to the test, and the number of hours spent exercising during the same week. Here are the raw data:

56	6	2	78	7	4	84	5	5	73	4	0	90	3	4
44	9	0	76	5	1	87	3	3	92	2	7	75	8	3
85	1	6	67	4	2	90	5	5	84	6	5	74	5	2
64	4	1	73	0	5	78	5	2	69	6	1	56	7	1
87	8	4	73	8	3	100	0	6	54	8	0	81	5	4
78	5	2	69	4	1	64	7	1	73	7	3	65	6	2

Notice that each line contains data for five students. The following program reads the raw data from a file called Exercise.dat, and then uses PROC CORR to compute the correlations:

```

LIBNAME mylib 'c:\MySASLib';
DATA mylib.class;
  INFILE 'c:\MyRawData\Exercise.dat';
    INPUT Score Videos Exercise @@;
  RUN;

*Perform correlation analysis;
PROC CORR DATA = mylib.class;
  VAR Videos Exercise;
  WITH Score;
  TITLE 'Correlations for Test Scores';
  TITLE2 'With Hours of Videos and Exercise';
  RUN;

```

Here is the report from PROC CORR:

## **Correlations for Test Scores With Hours of Videos and Exercise**

---

### **The CORR Procedure**

---

1 With Variables:	Score
-------------------	-------

**2 Variables:**

Videos Exercise

Simple Statistics					
Variable	N	Mean	Std Dev	Sum	Minimum
Score	30	74.63333	12.58484	2239	44.00000
Videos	30	5.10000	2.33932	153.00000	
Exercise	30	2.83333	1.94906	85.00000	

① Pearson Correlation Coefficients, N = 30 ②		
		Videos
Score	① -0.55390 ② 0.0015	

This report starts with descriptive statistics for each variable, and then lists the correlation matrix, which contains: ① correlation coefficients (in this case, Pearson), and ② the probability of getting a larger absolute value for each correlation assuming that the population correlation is zero.

In this example, both hours of videos and hours of exercise are correlated with test score, but exercise is positively correlated while videos is negatively correlated. This means students who watched more videos tended to have lower scores, while the students who spent more time exercising tended to have higher scores.

## 9.9 Creating Statistical Graphics with PROC CORR

The CORR procedure evaluates the strength of the linear relationship between pairs of variables. The tabular output gives the correlation coefficients and other simple statistics, and using ODS Graphics you can also visualize the relationship. Plots are not generated by default, so you need to specify the desired plots using the PLOTS= option. Here is the general form of PROC CORR with the PLOTS option:

```
PROC CORR PLOTS = (plot-list);  
  VAR variable-list;  
  WITH variable-list;
```

**Plot requests** The CORR procedure can create two types of plots:

SCATTER scatter plots for pairs of variables overlaid w or confidence ellipses

MATRIX matrix of scatter plots for all variables

**Plot options** By default, the scatter plots include prediction ellipses for new observations. If you want confidence ellipses for means, then specify the ELLIPSE=CONFIDENCE option on the scatter plot:

```
PROC CORR PLOTS = SCATTER(ELLIPSE =
CONFIDENCE);
```

If you do not want any ellipses on your scatter plots, then use the ELLIPSE=NONE option.

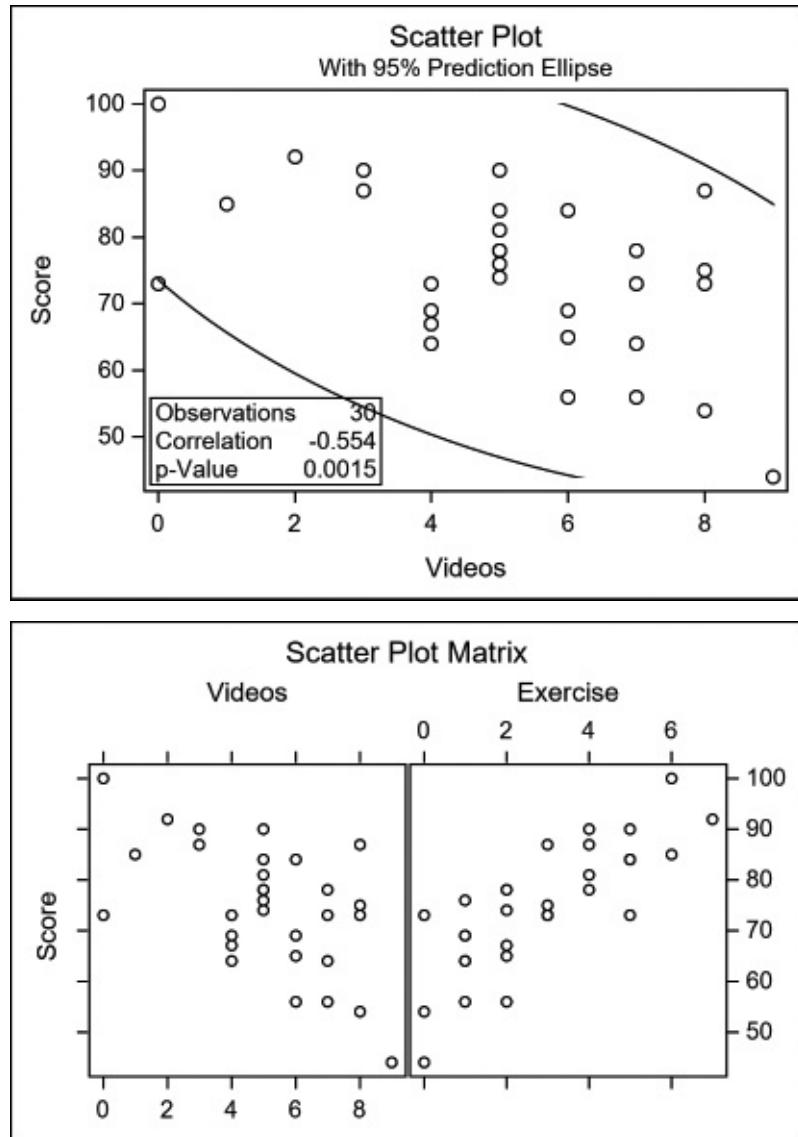
If you do not have a WITH statement, then matrix plots will show a symmetrical plot with all variable combinations appearing twice. By default the diagonal cells in the matrix will be empty. If you use the HISTOGRAM option for the matrix plot, then a histogram will be produced for each variable and displayed along the diagonal.

```
PROC CORR PLOTS = MATRIX(HISTOGRAM);
```

**Example** This example uses the data from the previous section about students in a statistics class. For each student, we have the test score (Score), the number of hours spent watching videos in the week prior to the test (Videos), and the number of hours spent exercising during the same week (Exercise). The following program uses the same PROC CORR statements shown in the previous section except both the scatter and matrix plots are requested:

```
LIBNAME mylib 'c:\MySASLib';
*Generate scatter and matrix plots for correlation
analysis;
PROC CORR DATA = mylib.class PLOTS =
(SCATTER MATRIX);
  VAR Videos Exercise;
  WITH Score;
  TITLE 'Correlations for Test Scores';
  TITLE2 'With Hours of Videos and Exercise';
RUN;
```

The program produces three plots: a scatter plot for Score by Videos, a scatter plot for Score by Exercise (not shown), and the Scatter Plot Matrix.



## 9.10 Using PROC REG for Simple Regression Analysis

The REG procedure fits linear regression models by least squares and is one of many SAS procedures that perform regression analysis. PROC REG is part of SAS/STAT, which is licensed separately from Base SAS. We will show an example of a simple regression analysis using continuous numeric variables with only one regressor

variable. However, PROC REG is capable of analyzing models with many regressor variables using a variety of model-selection methods, including stepwise regression, forward selection, and backward elimination. Other procedures in SAS/STAT will perform non-linear and logistic regression. In SAS/ETS, you will find procedures for time series analysis. If you are unsure about what type of analysis you need, or are unfamiliar with basic statistical principles, we recommend that you seek advice from a trained statistician, or consult a good statistical textbook.

The REG procedure has only two required statements. It must start with the PROC REG statement and have a MODEL statement specifying the analysis model. Here is the general form:

```
PROC REG;  
    MODEL dependent = independent;
```

In the MODEL statement, the dependent variable is listed on the left side of the equal sign and the independent, or regressor, variable is listed on the right.

**Example** At your young neighbor's T-ball game (that's where the players hit the ball from the top of a tee instead of having the ball pitched to them), he said to you, "You can tell how far they'll hit the ball by how tall they are." To give him a little practical lesson in statistics, you decide to test his hypothesis. You gather data from 30 players, measuring their height in inches and their longest of three hits in feet. The following are the data. Notice that data for several players are listed on one line:

50	110	49	135	48	129	53	150	48	124	50	143	51
126	45	107										
53	146	50	154	47	136	52	144	47	124	50	133	50
128	50	118										
48	135	47	129	45	126	48	118	45	121	53	142	46
122	47	119										
51	134	49	130	46	132	51	144	50	132	50	131	

The following program reads the data and performs the regression analysis. In the MODEL statement, Distance is the dependent variable, and Height is the independent variable.

```
LIBNAME tball 'c:\MySASLib';
DATA tball.hits;
  INFILE 'c:\MyRawData\Baseball.dat';
    INPUT Height Distance @@;
RUN;
* Perform regression analysis;
PROC REG DATA = tball.hits;
  MODEL Distance = Height;
  TITLE 'Results of Regression Analysis';
RUN;
```

The REG procedure produces tabular results and several graphs by default. Only the tabular results are shown here; see the next section for an example showing the graphical results. The first section of the tabular output is the analysis of variance section, which gives information about how well the model fits the data:

## Results of Regression Analysis

---

**The REG Procedure**  
**Model: MODEL1**  
**Dependent Variable: Distance**

Number of Observations Read	3 0
Number of Observations Used	3 0

Analysis of Variance				
Source	① DF	Sum of Squares	② Mean Square	③ F value
Model	1	1365.50831	1365.50831	
Error	28	2268.35836	81.01280	
Corrected Total	29	3633.86667		

⑤ Root MSE	9.00071	R-Squared
Dependent Mean	130.73333	⑦ Adjusted R-Sq
⑥ Coeff Var	6.88479	

① DF degrees of freedom associated with the source

② Mean Square mean square (sum of squares divided by the degrees of freedom)

③ F value  $F$  value for testing the null hypothesis (all parameters equal zero except intercept)

**④ Pr > F** significance probability or *p-value*

**⑤ Root MSE** root mean square error

**⑥ Coeff Var** the coefficient of variation

**⑦ Adj R-sq** the R-square value adjusted for degrees of freedom

The parameter estimates follow the analysis of variance section and give the parameters for each term in the model, including the intercept:

Parameter Estimates				
Variable	<b>① DF</b>	Parameter Estimate	Standard Error	
Intercept	1	-11.00859	34.56363	
Height	1	2.89466	0.70506	

degrees of freedom for the variables

**① DF**

**② t Value** *t* test for the parameter equal to 0

③ Pr > |t| two-tailed significance probab

From the parameter estimates you can construct the regression equation:

$$\text{Distance} = -11.00859 + (2.89466 * \text{Height})$$

In this example, the distance the ball was hit did increase with the player's height. The slope of the model is significant ( $p= 0.0003$ ) but the relationship is not very strong (R-square = 0.3758). Perhaps age or years of experience are better predictors of how far the ball will go.

## 9.11 Creating Statistical Graphics with PROC REG

There are many plots that are useful for visualizing the results of regression analysis and for assessing how well the model fits the data. PROC REG uses ODS Graphics to produce many such plots, including a diagnostic panel that contains up to nine plots in one figure. Some plots are produced automatically while others need to be specified. Here is the general form of PROC REG with the PLOTS option:

```
PROC REG PLOTS(options) = (plot-request-list);  
MODEL dependent = independent;
```

**Plot requests** Here are some plots that you can request for simple linear regression:

FITPLOT

scatter plot with regression line and confidence and prediction bands

RESIDUALS

residuals plotted against independent

	variable
DIAGNOSTICS	diagnostics panel including all of the following plots
COOKSD	Cook's <i>D</i> statistic by observation number
OBSERVEDBYPR EDICTED	dependent variable by predicted value
QQPLOT	normal quantile plot of residuals
RESIDUALBYPR EDICTED	residuals by predicted values
RESIDUALHISTO GRAM	histogram of residuals
RFPLOT	residual fit plot
RSTUDENTBYLE VERAGE	studentized residuals by leverage
RSTUDENTBYPR EDICTED	studentized residuals by predicted values

**Excluding automatic plots** By default, the RESIDUALS and DIAGNOSTICS plots are generated automatically. Additional plots may also be produced by default depending on the type of model. For example, a FITPLOT is automatically generated when there is one

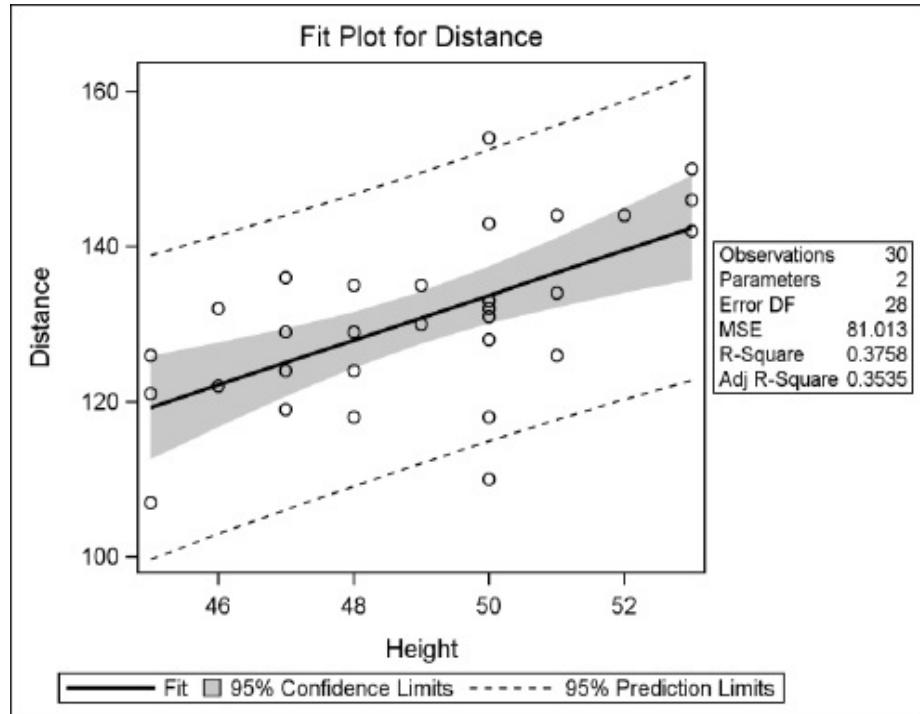
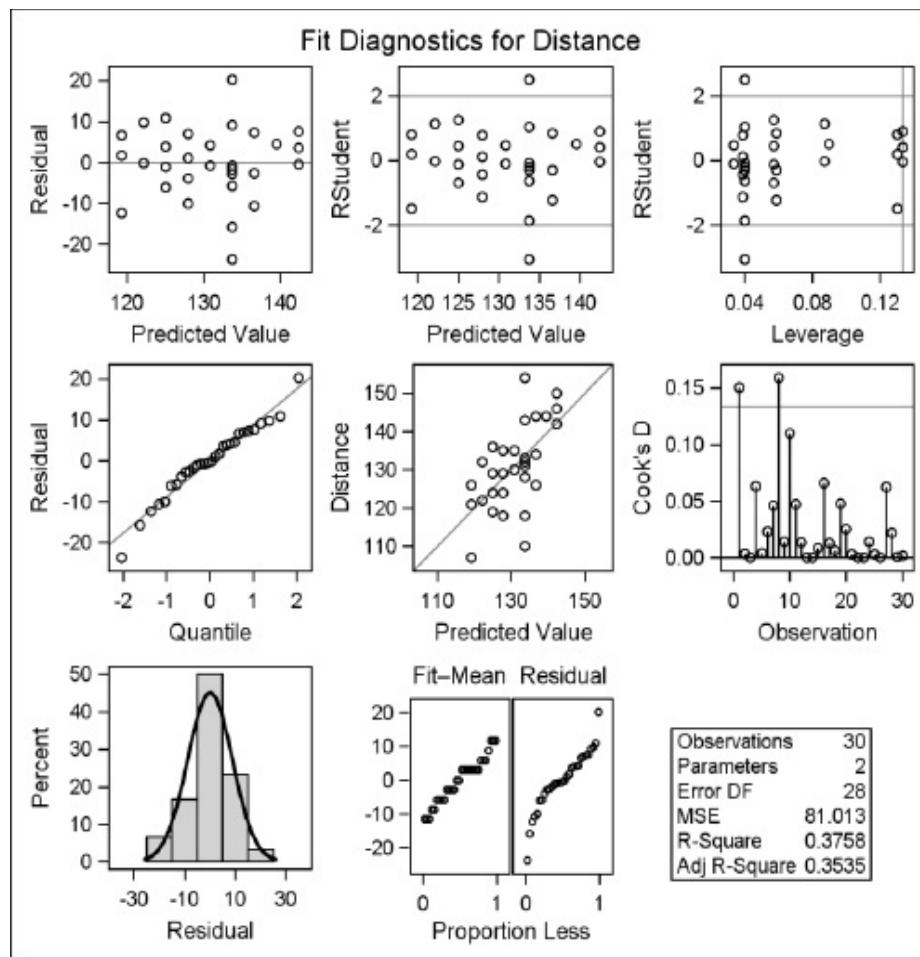
regressor variable. If you choose specific plots in the plot-request-list, the default plots will still be created unless you use the ONLY global option:

```
PROC REG PLOTS(ONLY) = (plot-request-list);
```

**Example** The following example uses the same data as the previous section about T-ball players. The data are the player's height in inches (Height) and their longest of three hits in feet (Distance). The following program performs the regression analysis as in the previous section. However, in this program only the FITPLOT and DIAGNOSTICS plots are requested:

```
LIBNAME tball 'c:\MySASLib';
*Limit plots generated to diagnostics panel and fit plot;
PROC REG DATA = tball.hits PLOTS(ONLY) =
(DIAGNOSTICS FITPLOT);
  MODEL Distance = Height;
  TITLE 'Results of Regression Analysis';
  RUN;
```

Here are the results for the Fit Diagnostics panel and the Fit Plot. The tabular results are not shown here, but are the same as in the previous section.



## 9.12 Using PROC ANOVA for One-Way Analysis of Variance

The ANOVA procedure is one of many in SAS that perform analysis of variance. PROC ANOVA is part of SAS/STAT, which is licensed separately from Base SAS. PROC ANOVA is specifically designed for balanced data—data where there are equal numbers of observations in each combination of the classification factors. An exception is for one-way analysis of variance where the data do not need to be balanced. If you are not doing one-way analysis of variance and your data are not balanced, then you should use the GLM procedure, whose statements are almost identical to those of PROC ANOVA. Although we are discussing only simple one-way analysis of variance in this section, PROC ANOVA can handle multiple classification variables and models that include nested and crossed effects as well as repeated measures. If you are unsure of the appropriate analysis for your data, or are unfamiliar with basic statistical principles, we recommend that you seek advice from a trained statistician or consult a good statistical textbook.

The ANOVA procedure has two required statements: the CLASS and MODEL statements. Here is the general form of the ANOVA procedure:

```
PROC ANOVA;  
  CLASS variable-list;  
  MODEL dependent = effects;
```

The CLASS statement must come before the MODEL statement and defines the classification variables. For one-way analysis of variance, only one variable is listed. The MODEL statement defines the dependent variable and the effects. For one-way analysis of variance, the effect is the

classification variable.

As you might expect, there are many optional statements for PROC ANOVA. One of the most useful is the MEANS statement, which calculates means of the dependent variable for any of the main effects in the MODEL statement. In addition, the MEANS statement can perform several types of multiple-comparison tests, including Bonferroni  $t$  tests (BON), Duncan's multiple-range test (DUNCAN), Scheffe's multiple-comparison procedure (SCHEFFE), pairwise  $t$  tests (T), and Tukey's studentized range test (TUKEY). The MEANS statement has the following general form:

MEANS *effects / options*;

The effects can be any effect in the MODEL statement, and options include the name of the desired multiple-comparison test (DUNCAN for example).

**Example** Your friend's daughter plays basketball on a team that travels throughout the state. She complains that it seems like the girls from the other regions in the state are all taller than the girls from her region. You decide to test her hypothesis by getting the heights for a sample of girls from the four regions and performing one-way analysis of variance to see if there are any differences. Each data line includes region and height for eight girls:

```
West 65 West 58 West 63 West 57 West 61 West 53  
West 56 West 66  
West 55 West 56 West 65 West 54 West 55 West 62  
West 55 West 58  
East 65 East 55 East 57 East 66 East 59 East 63 East 58  
East 57  
East 58 East 63 East 61 East 62 East 58 East 57 East 65  
East 57  
South 63 South 63 South 68 South 56 South 60 South 65  
South 64 South 62  
South 59 South 67 South 59 South 65 South 66 South 67
```

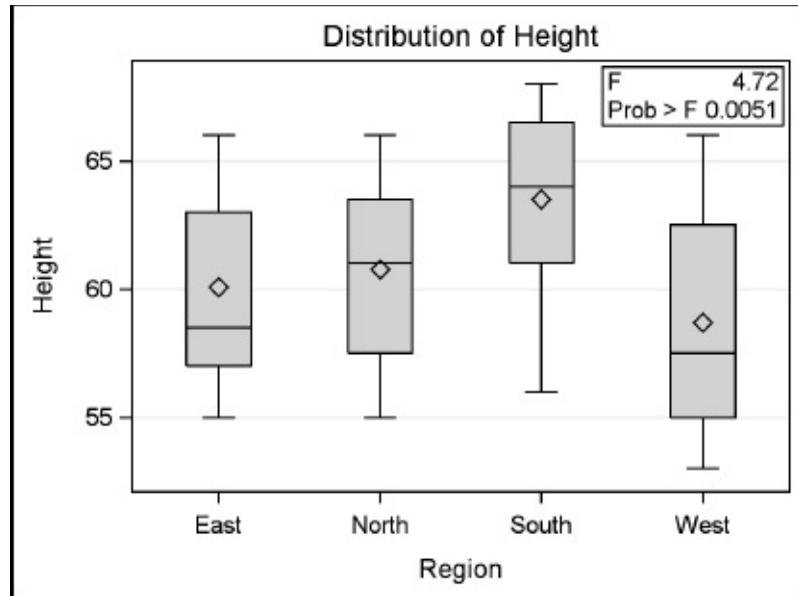
```
South 64 South 68
North 63 North 65 North 58 North 55 North 57 North 66
North 59 North 61
North 65 North 56 North 57 North 63 North 61 North 60
North 64 North 62
```

You want to know which, if any, regions have taller girls than the rest, so you use the MEANS statement in your program and choose Scheffe's multiple-comparison method to compare the means. Here is the program to read the data and perform the analysis of variance:

```
DATA heights;
  INFILE 'c:\MyRawData\GirlHeights.dat';
  INPUT Region $ Height @@;
  RUN;
  * Use ANOVA to run one-way analysis of variance;
  PROC ANOVA DATA = heights;
    CLASS Region;
    MODEL Height = Region;
    MEANS Region / SCHEFFE;
    TITLE "Girls' Heights from Four Regions";
    RUN;
```

In this case, Region is the classification variable and also the effect in the MODEL statement. Height is the dependent variable. The MEANS statement will produce means of the girls' heights for each region, and the SCHEFFE option will test which regions are different from the others.

Here is the box plot that is created automatically. The small  $p$ -value ( $p=0.0051$ ) indicates that at least two of the four regions differ in mean height. The tabular output is shown and discussed in the next section.



## 9.13 Reading the Output of PROC ANOVA

The tabular output from PROC ANOVA has at least two parts. First, ANOVA produces a table giving information about the classification variables: number of levels, values, and number of observations. Next, it produces an analysis of variance table. If you use optional statements like MEANS, then their output will follow.

The example from the previous section used the following PROC ANOVA statements:

```
PROC ANOVA DATA = heights;
  CLASS Region;
  MODEL Height = Region;
  MEANS Region / SCHEFFE;
  TITLE "Girls' Heights from Four Regions";
  RUN;
```

The graph produced by the ANOVA procedure is shown in the previous section. The first page of the tabular output (shown below) gives information about the classification variable Region. It has four levels with values East, North, South, and West; and there are 64 observations.

### Girls' Heights from Four Regions

---

## The ANOVA Procedure

Class Level Information		
Class	Levels	Values
Region	4	Ea

Number of Observations Read
Number of Observations Used

The second part of the output is the analysis of variance table:

### Girls' Heights from Four Regions

## The ANOVA Procedure

Dependent Variable: Height

<b>① Source</b>	<b>② DF</b>	<b>③ Sum of Squares</b>	<b>④ Mean Square</b>	<b>⑤ F</b>
<b>Model</b>	3	196.625000	65.541667	
<b>Error</b>	60	833.375000	13.889583	
<b>Corrected Total</b>	63	1030.000000		

<b>⑦ R-Square</b>	<b>⑧ Coeff Var</b>	<b>⑨ Root MSE</b>	
0.190898	6.134771	3.726873	

Source	DF	Anova SS	Mean Square	F
<b>Region</b>	3	196.625000	65.541667	

Highlights of the output are:

- ① Source source of variation
- ② DF degrees of freedom for the model, error, ai

<b>③ Sum of Squares</b>	sum of squares for the portion attributed to model, error, and total
<b>④ Mean Square</b>	mean square (sum of squares divided by the degrees of freedom)
<b>⑤ F Value</b>	F value (mean square for model divided by mean square for error)
<b>⑥ Pr &gt; F</b>	significance probability associated with the F statistic
<b>⑦ R-Square</b>	R-square
<b>⑧ Coeff Var</b>	coefficient of variation
<b>⑨ Root MSE</b>	root mean square error
<b>⑩ Height Mean</b>	mean of the dependent variable

Because the effect of Region is significant ( $p = .0051$ ), we conclude that there are differences in the mean heights of girls from the four regions. The SCHEFFE option in the MEANS statement compares the mean heights between the regions. The following results show that your friend's daughter is partially correct—one region (South) has taller girls than her region (West) but no other two regions differ significantly in mean height. Older versions of SAS may display a table instead of a graph. If you prefer a table, then add the LINESTABLE option to the MEANS statement.

### Girls' Heights from Four Regions

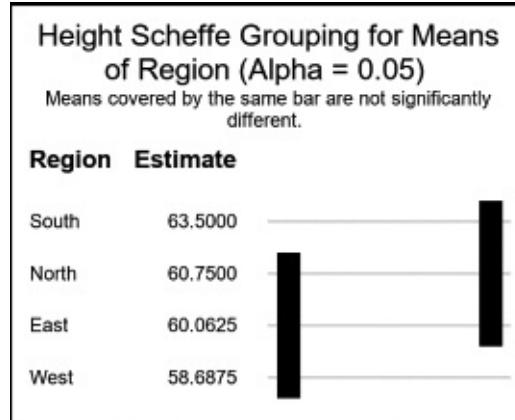
---

## The ANOVA Procedure

### Scheffe's Test for Height

Note: This test controls the Type I experimentwise error rate.

Alpha
Error Degrees of Freedom
Error Mean Square
Critical Value of F
Minimum Significant Difference



10

“When we try to pick out anything by itself, we find it hitched to everything in the Universe.”

JOHN MUIR

From *My First Summer in the Sierra* by John Muir. Public domain.

# CHAPTER 10

## Exporting Your Data

- [10.1 Methods for Exporting Your Data](#)
- [10.2 Writing Delimited Files with the EXPORT Procedure](#)
- [10.3 Writing Delimited Files Using ODS](#)
- [10.4 Writing Microsoft Excel Files with the EXPORT Procedure](#)
- [10.5 Writing Microsoft Excel Files Using ODS](#)
- [10.6 Writing Raw Data Files with the DATA Step](#)

### 10.1 Methods for Exporting Your Data



In our ever increasingly complex world, people often need to transfer data from one application to another. Fortunately, SAS gives you many options for doing this. The types of files that you can create and the methods available for creating those files depend on how you run SAS, what operating environment you are using, and whether you have SAS/ACCESS software.

Methods for exporting data to other applications fall into

these general categories:

- ◆ creating delimited or text files that the other software can read
- ◆ creating files in formats like HTML or XML that the other software can read
- ◆ writing the data in the other software's native format

**Creating delimited and text files** No matter what your environment, you can always create delimited or text files and most software has the ability to read these types of data files.

- ◆ The DATA step, discussed in Section 10.6, gives you the most control over the format of your files, but requires the most steps.
- ◆ The EXPORT procedure, discussed in Section 10.2, is easy to use, but you have less control over the result.
- ◆ SAS Studio, SAS Enterprise Guide, and the SAS windowing environment each have point-and-click methods for creating delimited files from SAS data sets. These methods are not covered in this book, but with a little exploration you will find various menus, icons, or tasks for exporting data.
- ◆ The Output Delivery System (ODS), discussed in Section 10.3, can create comma-separated values (CSV) files from any procedure output, and a simple PROC PRINT will produce a reasonable file for importing data into other programs.

**Creating HTML and XML files** Using ODS, you can create HTML and XML files from any procedure output. Many applications can read data in these types of files. Although we do not cover creating HTML and XML files in this chapter, the general method is the same as creating

CSV files. Creating HTML files is discussed in Section 5.2.

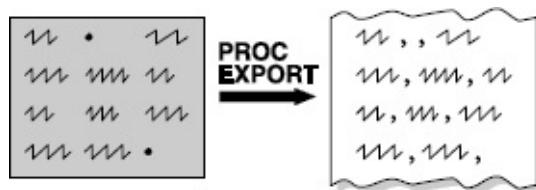
**Creating files in native formats** There are several methods for creating files in the native format of other software applications. Not all methods are available for all software applications, and some methods depend on what SAS software products you have installed, which operating environment, and what version of SAS you are using. We do not attempt to cover all methods, so see the SAS Documentation for complete information.

- ◆ The EXPORT procedure, discussed in Section 10.4, can create PC files in many formats including Microsoft Excel, Microsoft Access, dBase, Paradox, SPSS, Stata, and JMP. However, for most of these formats you need SAS/ACCESS Interface to PC Files. Support for JMP files is now included in Base SAS. This method is available only for the Windows or UNIX operating systems.
- ◆ The EXPORT procedure can also create PC files via the PC Files Server. You may have a PC Files Server application running in your SAS environment. The PC Files Server facilitates sharing of some PC files between different Windows computers, between Windows and UNIX computers, or even on a single Windows computer. The PC Files Server application requires that SAS/ACCESS Interface to PC Files be installed. This method is not covered in this book.
- ◆ SAS Studio, SAS Enterprise Guide, and the SAS windowing environment each have point-and-click methods for creating several native file formats from SAS data sets. For most formats you need SAS/ACCESS Interface to PC Files. These methods are not covered in this book, but with a little exploration you will find various menus, icons, or tasks for exporting data.
- ◆ The Output Delivery System(ODS), discussed in

Section 10.5, allows you to create Microsoft Excel files using the ODS EXCEL destination. The ODS EXCEL destination gives you more control over the appearance of the Excel file than other exporting methods discussed here.

- ◆ With the XLSX LIBNAME engine, discussed in Section 2.5, you can read from and write to an Excel file as if it were a SAS data set. This engine works for files created by any version of Microsoft Excel 2007 or later on the Windows or UNIX operating environments. You must have SAS 9.4M2 or higher and SAS/ACCESS Interface to PC Files software.
- ◆ For database management systems like ORACLE, DB2, INGRES, MYSQL, and Sybase, there are SAS/ACCESS products that allow you to create files in the native formats of these applications. These products are not discussed in this book.

## 10.2 Writing Delimited Files with the EXPORT Procedure



If you want to create a generic data file that most software can read, then creating a delimited text file may be the way to go. The EXPORT procedure allows you to create data files with any delimiter you choose. The resulting text file will have one delimiter between each data value, and each observation will occupy one line of text.

**The EXPORT procedure** The general form of PROC EXPORT is:

```
PROC EXPORT DATA = data-set OUTFILE = 'filename'
```

**REPLACE;**

where *data-set* is the SAS data set you want to export, and *filename* is the name you make up for the output data file. The REPLACE option tells SAS to replace the file if it already exists. The following statement tells SAS to read a temporary SAS data set named HOTELS and write a comma-delimited file named Hotels.csv in a directory named MyRawData on the C drive (Windows):

```
PROC EXPORT DATA = hotels OUTFILE =
  'c:\MyRawData\Hotels.csv' REPLACE;
  RUN;
```

SAS uses the last part of the filename, called the file extension, to decide what type of file to create. You can also specify the file type by adding the DBMS= option to the PROC EXPORT statement. The following table shows the filename extensions and DBMS identifiers that are available for delimited files. If you specify the DBMS= option, then it takes precedence over the file extension.

Type of file	Extension	DBMS Identif
Comma-delimited	.csv	CSV
Tab-delimited	.txt	TAB
Space-delimited		DLM

Notice that for space-delimited files, there is no standard extension so you must use the DBMS= option. The following statement, containing the DBMS= option, tells SAS to create a space-delimited file named Hotels.dat.

```
PROC EXPORT DATA = hotels OUTFILE =
```

```
'c:\MyRawData\Hotels.dat'  
DBMS = DLM REPLACE;  
RUN;
```

If you want to create a file with a delimiter other than a comma, tab, or space, then you can add the DELIMITER statement. If you use the DELIMITER statement, then it does not matter which file extension you use, or which DBMS identifier you specify, the file will have the delimiter that you specify in the DELIMITER statement. For example, the following would produce a file, Hotels.txt, that has the ampersand (&) as the delimiter:

```
PROC EXPORT DATA = hotels OUTFILE =  
'c:\MyRawData\Hotels.txt'  
DBMS = DLM REPLACE;  
DELIMITER='&';  
RUN;
```

**Example** A travel company has the following tab-delimited file containing information about golf courses in Hawaii. For each golf course the file includes its name, number of holes, par, yardage, and greens fees.

CourseName	Holes	Par	Yardage	GreenFees
Pukalani	18	72	6945	89.00
Kahili	18	72	6570	119.00
Maui Nui Blue	18	71	6404	69.00
Waiehu Municipal	18	72	6330	63.00
Wailea Gold	18	72		250.00
Dunes at Maui Lani	18	72	6841	105.00
Kaanapali	18	71	6555	144.00

The following program uses the IMPORT procedure to read the data file and create a permanent SAS data set named GOLF in the MySASLib directory on the C drive (Windows).

```
LIBNAME travel 'c:\MySASLib';
```

```
PROC IMPORT DATAFILE = 'c:\MyRawData\Golf.txt'  
OUT = travel.golf REPLACE;  
RUN;
```

Now, suppose that a customer wants the data in a comma-delimited file so they can import the data into their software. The following program writes a plain text, comma-delimited file:

```
LIBNAME travel 'c:\MySASLib';  
*Create comma-delimited file;  
PROC EXPORT DATA = travel.golf OUTFILE =  
'c:\MyRawData\Golf.csv' REPLACE;  
RUN;
```

Because the name of the output file ends with .csv and there is no DELIMITER statement, SAS will write a comma-delimited file. If you run this program, your log will contain the following notes:

---

```
NOTE: 8 records were written to the file  
'c:\MyRawData\Golf.csv'.
```

```
NOTE: There were 7 observations read from the data set  
TRAVEL.GOLF.
```

---

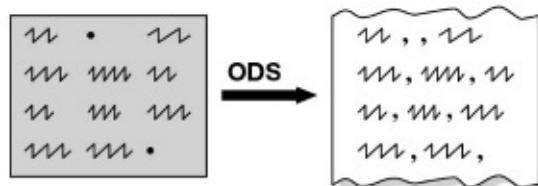
Notice that while the SAS data set contained seven observations, SAS wrote eight records. The extra record contains the variable names. Here is the comma-delimited text file created by the EXPORT procedure.

```
CourseName,Holes,Par,Yardage,GreenFees  
Pukalani,18,72,6945,89  
Kahili,18,72,6570,119  
Maui Nui Blue,18,71,6404,69  
Waiehu Municipal,18,72,6330,63  
Wailea Gold,18,72,,250  
Dunes at Maui Lani,18,72,6841,105  
Kaanapali,18,71,6555,144
```

Any format that you have assigned to variables in the SAS data set will be used by PROC EXPORT when creating the

delimited file. If you want to change a format, use a FORMAT statement (discussed in Section 4.6) in a DATA step before running PROC EXPORT.

### 10.3 Writing Delimited Files Using ODS



ODS is a powerful tool for creating all sorts of output formats. Some of the output formats that ODS can create are useful for transferring data and results from SAS to other applications. Many applications can read data that are in comma-separated values (CSV) format, and the great thing is that you can use this method in any operating environment and it's included in Base SAS.

You can use ODS CSV to export data from most procedures, and PROC PRINT is a good way to get a file with a listing of your data. By default, SAS will insert a period for any missing numeric data. If you would rather have SAS print nothing for missing numeric data, then you can use the MISSING=' ' system option. Also, by default, PROC PRINT includes observation numbers. If you don't want observation numbers in your output file, then use the NOOBS option in the PROC PRINT statement.

The CSV destination puts commas between data values and double quotation marks around character values. The double quotation marks allow data values to contain commas. To create a CSV file containing your data, use the following ODS and PROC PRINT statements:

```
OPTIONS MISSING = ";
ODS CSV FILE = 'filename.csv';
PROC PRINT DATA = data-set-name NOOBS;
RUN;
```

ODS CSV CLOSE;

where *filename.csv* is the name of the CSV file that you are creating, and *data-set-name* is the name of the SAS data set you want to export. The CSV output destination does not include titles or footnotes. If you want titles and footnotes to appear in the CSV file, then use the CSVALL output destination instead of CSV.

**Example** This example uses the permanent SAS data set, GOLF (created in the previous section), which has information about golf courses in Hawaii.

	<b>CourseName</b>	<b>Holes</b>	<b>Par</b>	<b>Yardage</b>
1	Pukalani	18	72	6945
2	Kahili	18	72	6570
3	Maui Nui Blue	18	71	6404
4	Waiehu Municipal	18	72	6330
5	Wailea Gold	18	72	.
6	Dunes at Maui Lani	18	72	6841
7	Kaanapali	18	71	6555

The following program uses ODS to create two CSV files. The first file, GolfInfo.csv, contains the results of PROC PRINT which gives a listing of

the data. The second file, GolfMeans.csv, captures the results of a PROC MEANS.

```
LIBNAME travel 'c:\MySASLib';
*Create comma-delimited file from PROC PRINT
results;
OPTIONS MISSING = ";
ODS CSV FILE = 'c:\MyRawData\GolfInfo.csv';
PROC PRINT DATA = travel.golf NOOBS;
RUN;
ODS CSV CLOSE;

*Create comma-delimited file from PROC MEANS
results;
ODS CSV FILE='c:\MyRawData\GolfMeans.csv';
PROC MEANS DATA = travel.golf MAXDEC = 0
MEAN MAX;
CLASS par;
VAR Yardage GreenFees;
RUN;
ODS CSV CLOSE;
```

Here is the comma-delimited text file, GolfInfo.csv, created from the PROC PRINT:

```
"CourseName","Holes","Par","Yardage","GreenFees"
"Pukalani",18,72,6945,89
"Kahili",18,72,6570,119
"Maui Nui Blue",18,71,6404,69
"Waiehu Municipal",18,72,6330,63
"Wailea Gold",18,72,,250
"Dunes at Maui Lani",18,72,6841,105
"Kaanapali",18,71,6555,144
```

Here is the comma delimited text file, GolfMeans.csv, created from the PROC MEANS:

```
"Par","N
Obs","Variable","Mean","Maximum","Variable","Mean"
,"Maximum"
```

```
"71","2","Yardage",6480,6555,"GreenFees",107,144  
"72","5","Yardage",6672,6945,"GreenFees",125,250
```

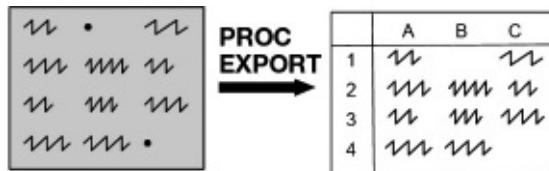
The following notes appear in the SAS log showing that seven observations were read from the GOLF data set, and both CSV files were successfully written:

---

```
NOTE: Writing CSV Body file: c:\MyRawData\GolfInfo.csv  
NOTE: There were 7 observations read from the data set TRAVEL.GOLF.  
NOTE: Writing CSV Body file: c:\MyRawData\GolfMeans.csv  
NOTE: There were 7 observations read from the data set  
TRAVEL.GOLF.
```

---

## 10.4 Writing Microsoft Excel Files with the EXPORT Procedure



If you are using the Windows or UNIX operating environment, and you have SAS/ACCESS Interface to PC Files, then you can use the EXPORT procedure to create Microsoft Excel files.

Here is the general form of PROC EXPORT for writing Excel files:

```
PROC EXPORT DATA = data-set OUTFILE =  
'filename' DBMS = identifier  
REPLACE;
```

where *data-set* is the SAS data set that you want to export, and *filename* is the name that you make up for the output data file. The DBMS= option tells SAS what type of Excel file to create. The REPLACE option tells SAS to replace the file if it already exists.

**DBMS identifiers** There are several DBMS identifiers you can use to create Excel files. Three commonly used identifiers are EXCEL, XLS, and XLSX. The EXCEL identifier is available only on Windows. The XLS identifier creates older style files (.xls extension) and is available on Windows and UNIX. The XLSX identifier creates newer style files (.xlsx extension) and is available on both Windows and UNIX. Not all of these identifiers may work for you if your Windows computer has a mixture of 64-bit and 32-bit applications. In addition, some computer configurations may require that a PC Files Server be installed. The PC Files Server uses the EXCELCS identifier. See the SAS Documentation for more information. The following statement, containing the DBMS= option, tells SAS to create an Excel file named Hotels.xlsx.

```
PROC EXPORT DATA = hotels OUTFILE =
  'c:\MyExcel\Hotels.xlsx'
  DBMS = XLSX REPLACE;
RUN;
```

**Naming sheets** By default, the name of the Microsoft Excel sheet will be the same as the name of the SAS data set. If you want the sheet to have a different name, then specify it in the SHEET= statement. Special characters in sheet names will be converted to underscores. The following statement creates a sheet named Golf_Hotels:

```
SHEET = 'Golf/Hotels';
```

You can create Excel files with multiple sheets by submitting multiple EXPORT procedures, specifying the same filename in the OUTFILE= option, but using a different name in the SHEET= statement. If a sheet by that name already exists in the file, it will not be overwritten unless you also specify the REPLACE option.

**Example** This example uses the SAS data set created in Section 10.2, which contains information about Hawaiian golf courses.

	<b>CourseName</b>	<b>Holes</b>	<b>Par</b>	<b>Yardage</b>
1	Pukalani	18	72	6945
2	Kahili	18	72	6570
3	Maui Nui Blue	18	71	6404
4	Waiehu Municipal	18	72	6330
5	Wailea Gold	18	72	.
6	Dunes at Maui Lani	18	72	6841
7	Kaanapali	18	71	6555

Suppose that your office mate needs that information, but she wants it in a Microsoft Excel file. The following program writes a Microsoft Excel file from the SAS data set GOLF:

```
LIBNAME travel 'c:\MySASLib';
* Create Microsoft Excel file';
PROC EXPORT DATA = travel.golf OUTFILE =
'c:\MyExcel\Golf.xlsx'
DBMS = XLSX REPLACE;
RUN;
```

Here is what the Microsoft Excel file looks like. Notice that the name of the sheet is the same as the name of the SAS

data set.

A	B	C	D	E	F
1	CourseName	Holes	Par	Yardage	GreenFees
2	Pukalani	18	72	6945	89
3	Kahili	18	72	6570	119
4	Maui Nui Blue	18	71	6404	69
5	Waiehu Municipal	18	72	6330	63
6	Wailea Gold	18	72		250
7	Dunes at Maui Lani	18	72	6841	105
8	Kaanapali	18	71	6555	144

If you have user-defined formats that have been associated with variables, only the unformatted values will be exported to Excel. The FORMAT statement is not supported in the EXPORT procedure.

If you run this program, your log will contain the following notes.

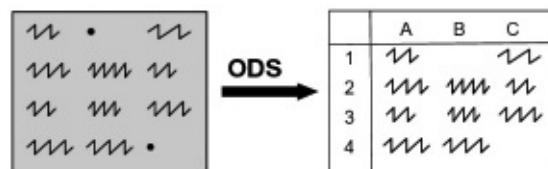
---

NOTE: The export data set has 7 observations and 5 variables.

NOTE: "c:\MyExcel\Golf.xlsx" file was successfully created.

---

## 10.5 Writing Microsoft Excel Files Using ODS



In Section 10.3 we showed how to create comma-separated values (CSV) files using ODS. You can read CSV files in Microsoft Excel, but using the ODS EXCEL destination you can create native Excel files, and you have a lot of control over what those files look like in Excel. The EXCEL destination is included in Base SAS and can be

used in any operating environment (for z/OS it works only under HFS file systems).

You can use ODS EXCEL to export data from most procedures, and PROC PRINT is a good way to get a file with a listing of your data. By default, SAS will insert a period for any missing numeric data. If you would rather have nothing for missing numeric data, then use the MISSING=' ' system option. Also, by default, PROC PRINT includes observation numbers. If you don't want observation numbers in your output file, then use the NOOBS option in the PROC PRINT statement. To create an Excel file containing your data, use the following ODS and PROC PRINT statements:

```
OPTIONS MISSING = '';
ODS EXCEL FILE = 'filename.xlsx';
PROC PRINT DATA = data-set-name NOOBS;
RUN;
ODS EXCEL CLOSE;
```

where *filename.xlsx* is the name of the Excel file that you are creating. To create an Excel file from other procedure results, insert the desired PROC statements instead of the PROC PRINT. If you want, you can specify a style for your Excel file, by adding the STYLE= option to the ODS EXCEL statement.

There are several options you can add to the ODS EXCEL statement to control what the Excel file looks like. One of these options is OPTIONS, which itself has numerous suboptions. Suboptions are listed in parentheses after the keyword OPTIONS. For example, by default, titles are not included in the worksheet. The following statement uses the SHEET_NAME= suboption to specify a name for the Excel worksheet and the EMBEDDED_TITLES= suboption to embed titles in the worksheet:

```
ODS EXCEL OPTIONS(SHEET_NAME = 'Golf Data'
EMBEDDED_TITLES = 'ON');
```

The AUTOFILTER= suboption allows you to set up

filtering for specified columns in the worksheet. Filtering in Excel allows you to easily sort your data and exclude specific rows in the worksheet. For example, the following statement turns on filtering for columns 2 through 10 of the Golf Data worksheet:

```
ODS EXCEL OPTIONS(SHEET_NAME = 'Golf Data'  
AUTOFILTER = '2-10');
```

Other suboptions allow you to control things like column width, freezing of headers, starting cells for the output, and when to create a new sheet.

**Example** This example uses the permanent SAS data set, GOLF (created in Section 10.2), which has information about golf courses in Hawaii. The following program uses ODS to create an Excel file, golfinfo.xlsx, which contains two worksheets. The first worksheet contains the results of a PROC PRINT, while the second contains the results of a PROC MEANS.

```
LIBNAME travel 'c:\MySASLib';  
*Create one Excel file with two worksheets;  
OPTIONS MISSING = ":";  
ODS EXCEL FILE = 'c:\MyExcel\golfinfo.xlsx';  
ODS EXCEL OPTIONS(SHEET_NAME = 'GolfData'  
AUTOFILTER = '1-5');  
PROC PRINT DATA = travel.golf NOOBS;  
RUN;  
ODS NOPROCTITLE;  
ODS EXCEL OPTIONS(SHEET_NAME = 'GolfMeans'  
AUTOFILTER = 'NONE');  
PROC MEANS DATA = travel.golf MAXDEC = 0  
MEAN MIN MAX;  
CLASS Par;  
VAR Yardage GreenFees;  
RUN;  
ODS EXCEL CLOSE;
```

Notice that the program contains several ODS

statements. The first ODS statement specifies the output file. The second ODS statement, which is in effect for the PROC PRINT, tells SAS to name the worksheet GolfData, and to set up filters for columns 1 through 5. The ODS NOPROCTITLE statement prevents the name of the next procedure (MEANS) from appearing in the first row of the worksheet. The fourth ODS statement, which is in effect for the PROC MEANS, tells SAS to name the worksheet GolfMeans and not to include the filters. The final ODS statement closes the Excel file.

Here is the first worksheet, GolfData, created by the PROC PRINT. Note the filters on columns A through E (1-5). These filters allow you to easily sort and filter your data in Excel.

	A	B	C	D	E	F
1	CourseName	Hol	F	Yarda	GreenFe	
2	Pukalani	18	72	6945	89	
3	Kahili	18	72	6570	119	
4	Maui Nui Blue	18	71	6404	69	
5	Wailehu Municipal	18	72	6330	63	
6	Wailea Gold	18	72		250	
7	Dunes at Maui Lani	18	72	6841	105	
8	Kaanapali	18	71	6555	144	
9						

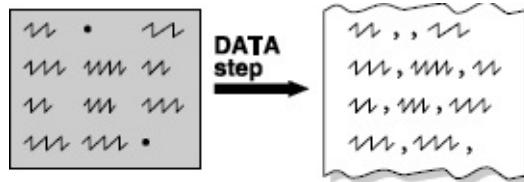
GolfData    GolfMeans    (+) : ▲ ▼ ▶

Here is the second worksheet, GolfMeans, created by the PROC MEANS.

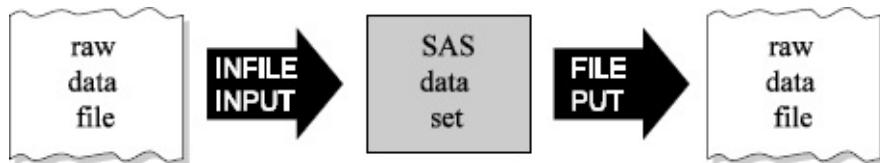
	A	B	C	D	E	F	G
1	Par	Obs	N Variable	Mean	Minimum	Maximum	
2	71	2	Yardage	6480	6404	6555	
			GreenFees	107	69	144	
3	72	5	Yardage	6672	6330	6945	
			GreenFees	125	63	250	

GolfData    GolfMeans    (+) : ▲ ▼ ▶

## 10.6 Writing Raw Data Files with the DATA Step



When you need total control over the contents and format of raw data files that you are creating, then the DATA step is the way to go. Using FILE and PUT statements in the DATA step, you can write almost any form of raw data file.



You can write raw data the same way that you read raw data in a DATA step, with just a few changes. Instead of naming the external file in an INFILE statement, you name it in a FILE statement. Instead of reading variables with an INPUT statement, you write them with a PUT statement. In other words, you use INFILE and INPUT statements to get raw data into SAS, and FILE and PUT statements to get raw data out.

PUT statements can be in list, column, or formatted style, just like INPUT statements, but since SAS already knows whether a variable is numeric or character, you don't have to put a \$ after character variables. If you use list style PUT statements, SAS will automatically put one space between each variable, creating a space-delimited file. To write files with other delimiters, use a list-style PUT statement and the DSD and DLM= options in your FILE statement. These options are covered in more detail in Section 2.18. Here is the general form of the FILE statement with the DSD and DLM= options:

```
FILE 'file-specification' DSD DLM = 'delimiter';
```

If you use column or formatted styles of PUT statements, SAS will put the variables wherever you specify. You can control spacing with the same pointer controls that INPUT statements use: @*n* to move to column *n*, +*n* to move *n* columns, / to skip to the next line, #*n* to skip to line *n*, and the trailing @ to hold the current line. In addition to printing variables, you can insert a text string by simply enclosing it in quotation marks.

**Example** This example uses the same SAS data set created in Section 10.2 containing information about Hawaiian golf courses.

	<b>CourseName</b>	<b>Holes</b>	<b>Par</b>	<b>Yardage</b>
1	Pukalani	18	72	6945
2	Kahili	18	72	6570
3	Maui Nui Blue	18	71	6404
4	Waiehu Municipal	18	72	6330
5	Wailea Gold	18	72	.
6	Dunes at Maui Lani	18	72	6841
7	Kaanapali	18	71	6555

This example shows how much more control you have when using the DATA step as opposed to PROC EXPORT or ODS. Suppose you want to put the data in a raw data

file, but with only three variables, in a new order, and with dollar signs added to the variable GreenFees. The following program reads the SAS data set and writes a raw data file using FILE and PUT statements:

```
LIBNAME travel 'c:\MySASLib';
*Create raw data file using FILE and PUT statements;
DATA _NULL_;
  SET travel.golf;
  FILE 'c:\MyRawData\Newfile.dat';
  PUT CourseName 'Golf Course' @32 GreenFees
DOLLAR7.2 @40 'Par ' Par;
RUN;
```

The word _NULL_ appears in the DATA statement instead of a SAS data set name. You could put a data set name there, but _NULL_ is a special keyword that tells SAS not to bother making a new SAS data set. By not writing a new SAS data set, you save computer resources.

The SET statement simply tells SAS to read the permanent SAS data set GOLF. The FILE statement tells SAS the name of the output file you want to create. Then the PUT statement tells SAS what to write and where. The PUT statement contains two quoted strings, “Golf Course” and “Par ” which SAS inserts in the raw data file. The PUT statement also tells SAS exactly where to place the data values for each variable using the @ column pointer, and to use the DOLLAR7.2 format to write the values for the GreenFees variable.

If you run this program, your log will contain the following note telling how many records were written to the output file:

---

NOTE: 7 records were written to the file  
'c:\MyRawData\Newfile.dat'.

---

Here is the text file, Newfile.dat, created in the DATA _NULL_ step:

Pukalani Golf Course	\$89.00 Par 72
Kahili Golf Course	\$119.00 Par 72
Maui Nui Blue Golf Course	\$69.00 Par 71
Waiehu Municipal Golf Course	\$63.00 Par 72
Wailea Gold Golf Course	\$250.00 Par 72
Dunes at Maui Lani Golf Course	\$105.00 Par 72
Kaanapali Golf Course	\$144.00 Par 71

11

“Problems that go away by themselves come back by themselves.”

MARCY E. DAVIS

From *The Official Explanations* by Paul Dickson. Copyright 1980 by Delacorte Press. Reprinted by permission of the publisher.

# CHAPTER 11

## Debugging Your SAS Programs

- [11.1 Writing SAS Programs That Work](#)
- [11.2 Fixing Programs That Don't Work](#)
- [11.3 Searching for the Missing Semicolon](#)
- [11.4 Note: INPUT Statement Reached Past the End of a Line](#)
- [11.5 Note: Lost Card](#)
- [11.6 Note: Invalid Data](#)
- [11.7 Note: Missing Values Were Generated](#)
- [11.8 Note: Numeric Values Have Been Converted to Character \(or Vice Versa\)](#)
- [11.9 DATA Step Produces Wrong Results but No Error Message](#)
- [11.10 Error: Invalid Option, Error: The Option Is Not Recognized, or Error: Statement Is Not Valid](#)
- [11.11 Note: Variable Is Uninitialized or Error: Variable Not Found](#)
- [11.12 SAS Truncates a Character Variable](#)
- [11.13 Saving Memory or Disk Space](#)

### 11.1 Writing SAS Programs That Work

It's not always easy to write a program that works the first time you run it. Even experienced SAS programmers will tell you it's a delightful surprise when their programs run

on the first try. The longer and more complicated the program, the more likely it is to have syntax or logic errors. But don't despair, there are a few guidelines you can follow that can make your programs run correctly sooner and help you discover errors more easily.

**Make programs easy to read** One simple thing that you can do is develop the habit of writing programs in a neat and consistent manner. Programs that are easy to read are easier to debug and will save you time in the long run. For easier programming, follow these guidelines:

- ◆ Put only one SAS statement on a line. SAS allows you to put as many statements on a line as you want, which may save some space in your program, but the saved space is rarely worth the sacrifice in readability.
- ◆ Use indentation to show the different parts of the program. Indent all statements within the DATA and PROC steps. This way you can tell at a glance how many DATA and PROC steps there are in a program and which statement belongs to which step. It's also helpful to further indent any statements between a DO statement and its END statement.
- ◆ Use comment statements generously to document your programs. This takes some discipline but is important, especially if anyone else is likely to read or use your program. Everyone has a different programming style, and it is often impossible to figure out what someone else's program is doing and why. Comment statements take the mystery out of the program.

**Syntax sensitive editors** SAS program editors color code your programs as you write them. SAS keywords appear in one color, variables in another. All text within quotation marks appears in the same color, so it is immediately obvious when you forget to close your

quotation marks. Similarly, missing semicolons are much easier to discover because the colors in your program are not right. Catching errors as you type them can be a real time saver.

**Test each part of the program** You can increase your programming efficiency tremendously by making sure each part of your program is working before moving on to the next part. If you were building a house, you would make sure the foundation was level and square before putting up the walls. You would test the plumbing before finishing the bathroom. You are required to have each stage of the house inspected before moving on to the next. The same should be done for your SAS program. But you don't have to wait for the inspector to come out; you can do it yourself.

If you are reading data from a file, make sure that it has been read correctly before moving on. Sometimes, even though there are no errors or even suspicious notes in your SAS log, the SAS data set is not correct. This could happen because SAS did not read the data the way you imagined (after all it does what you say, not what you're thinking) or because the data had some peculiarities you did not realize. For example, a researcher who received two data files from Taiwan wanted to merge them together by date. She could not figure out why they refused to merge correctly until she examined both data sets and realized one of the files used Taiwanese dates, which are offset by 11 years.

It's a good habit to look at all the SAS data sets you create in a program at least once to make sure they are correct. As with reading raw data files, sometimes merging and setting data sets can produce the wrong result even though there were no error messages.

**Test programs with small data sets** Sometimes it's not practical to test your program with your entire data set.

If your data files are very large, it may take a long time for your programs to run. In these cases, you can test your program with a subset of your data.

If you are reading data from a file, you can use the OBS= option in the INFILE statement to tell SAS to stop reading when it gets to that line in the file. This way you can read only the first 50 or 100 lines of data, or however many it takes to get a good representation of your data. The following statement will read only the first 100 lines of the raw data file Mydata.dat:

```
INFILE 'Mydata.dat' OBS = 100;
```

You can also use the FIRSTOBS= option to start reading from the middle of the data file. So, if the first 100 data lines are not a good representation of your data but 101 through 200 are, you can use the following statement to read just those lines:

```
INFILE 'Mydata.dat' FIRSTOBS = 101 OBS = 200;
```

Here, FIRSTOBS= and OBS= relate to the records of raw data in the file. These do not necessarily correspond to the observations in the SAS data set created. If, for example, you are reading two records for each observation, then you would need to read 200 records to get 100 observations.

If you are reading a SAS data set instead of a raw data file, you can use the OBS= and FIRSTOBS= data set options in the SET, MERGE, or UPDATE statements (discussed in Section 6.10). This controls which observations are processed in the DATA step. For example, the following DATA step will read the first 50 observations in the CATS data set. Note that when reading SAS data sets, OBS= and FIRSTOBS= truly do correspond to the observations and not to lines of raw data:

```
DATA sampleofcats;  
  SET cats (OBS = 50);  
  RUN;
```

**Test with representative data** Using OBS= and FIRSTOBS= is an easy way to test your programs, but sometimes it is difficult to get a good representation of your data this way. You may need to create a small test data set by extracting representative parts of the larger data set. Or, you may want to make up representative data for testing purposes. Making up data has the advantage that you can simplify the data and make sure you have every possible combination of values to test.

Sometimes you may want to make up data and write a small program just to test one aspect of your larger program. This can be extremely useful for narrowing down possible sources of errors in a large, complicated program.

## 11.2 Fixing Programs That Don't Work



In spite of your best efforts, sometimes programs just don't work. More often than not, programs don't run the first time. Even with simple programs it is easy to forget a semicolon or misspell a keyword—everyone does sometimes. If your program doesn't work, the source of the problem may be obvious, like an error message with the offending part of your program underlined, or not so obvious as when you have no errors but still don't have the expected results. Whatever the problem, here are a few guidelines you can follow to help fix your program.

**Read the SAS log** The SAS log has a wealth of information about your program. In addition to listing the program statements, it tells you things like how many lines were read from your raw data file and what the minimum and maximum line lengths were. It states the number of observations and variables in each SAS data set you create.

Information like this may seem inconsequential at first but can be very helpful in finding the source of your errors.

The SAS log has three types of messages about your program: errors, warnings, and notes.

**Errors** These are hard to ignore. Not only do they come up in red on your screen, but your program will not run with errors. Usually, errors are some kind of syntax or spelling mistake. The following log shows the error messages when you accidentally add a slash between the PROC PRINT and DATA= keywords. SAS underlines the problem (the slash) and tells you there is a syntax error. Sometimes SAS tells you what it expected and this can be very revealing.

---

```
1 PROC PRINT / DATA=one;
-
22
200
ERROR 22-322: Syntax error, expecting one of the following: :,  
BLANKLINE, CONTENTS,  
DATA, DOUBLE, GRANDTOTAL_LABEL,  
GRANDTOT_LABEL, GRAND_LABEL,  
GTOTAL_LABEL, GTOT_LABEL, HEADING, LABEL, N,  
NOOBS, NOSUMLABEL,  
OBS, ROUND, ROWS, SPLIT, STYLE, SUMLABEL, UNIFORM,  
WIDTH.  
ERROR 200-322: The symbol is not recognized and will be  
ignored.
```

---

The location of the error is usually easy to find because it is underlined, but the source of the error can be tricky. Sometimes what is wrong is not what is underlined but something else earlier in the program.

**Warnings** These are less serious than errors because your program will run with warnings. But beware, a warning may mean that SAS has done something you have not intended. For example, SAS will attempt to correct the

spelling of certain keywords. If you misspell INPUT as IMPUT, you will get the following message in your log:

---

WARNING 1-322: Assuming the symbol INPUT was misspelled as IMPUT.

---

Usually, you would think, “SAS is so smart—it knows what I meant to say,” but occasionally that may not be what you meant at all. Make sure that you know what all the warnings are about and that you agree with them.

**Notes** These are less straightforward than either warnings or errors. Sometimes, notes just give you information, like telling you the execution time of each step in your program. But sometimes notes can indicate a problem. Suppose, for example, that you find the following note in your SAS log:

---

NOTE: SAS went to a new line when INPUT statement reached past the end of a line.

---

This could mean that SAS did exactly what you wanted, or it could indicate a problem with your program or your data. Make sure that you know what each note means and why it is there.

**Start at the beginning** Whenever you read the SAS log, start at the beginning. This might seem like a ridiculous statement—why wouldn’t you start at the beginning? Well, often when you run a SAS program, the SAS log rolls by in the Log window. So when the program is finished, you are left looking at the end of the log. If you happen to see an error at the end of the log, it is natural to try to fix that error first—the first one you see. Avoid this temptation. Often errors at the end of the log are caused by earlier ones. If you fix the first error, often most or all of the other errors will disappear. If your lawnmower is out of gas and won’t start, it’s probably better to add gas before trying to figure out why it won’t start. The same logic applies to debugging SAS programs; fixing one problem will often fix others.



**Look for common mistakes first** More often than not there is a simple reason why your program doesn't work. Look for simple reasons before trying to find something more complicated. The remainder of this chapter consists of sections discussing common errors encountered in SAS programming. When you see this little bug in the upper right corner of a section, you'll know that the material deals with how to debug your program.

Sometimes error messages just don't make any sense. For example, you may get an error message saying the INPUT statement is not valid. This doesn't make much sense because you know INPUT is a valid SAS statement. In cases like these, look for missing semicolons in the statements before the error. If SAS has underlined an item, be sure to look not only at the underlined item but also at the previous few statements.

Finally, if you just can't figure out why you are not getting the results you expect, make sure to take a close look at any new SAS data sets you create. This can really help you discover errors in your logic, and sometimes uncover surprising details about your data.

**Check your syntax** If you have large data sets, you may want to check for syntax errors in your program before processing your data. To do this, add the following line to your program and submit it in the usual way:

```
OPTIONS OBS=0 NOREREPLACE;
```

The OBS=0 option tells SAS not to process any data, while the NOREREPLACE option tells SAS not to replace existing SAS data sets with empty ones. Once you know your syntax is correct, you can resubmit your program without the OPTIONS statement in batch mode, or submit the

following if you are using the SAS interactively.

```
OPTIONS OBS=MAX REPLACE;
```

Note that this syntax check will not uncover any errors related to your data or logic.



### 11.3 Searching for the Missing Semicolon



Missing semicolons are the most common source of errors in SAS programs. For whatever reason, we humans can't seem to remember to put a semicolon at the end of all our statements. (Maybe we all have rebellious right pinkies —who knows.) This is unfortunate because, while it is easy to forget the semicolon, it is not always easy to find the missing semicolon. The error messages produced are often misleading, making it difficult to find the error.

SAS reads statements from one semicolon to the next without regard to the layout of the program. If you leave off a semicolon, you in effect concatenate two SAS statements. Then SAS gets confused because it seems as though you are missing statements, or it tries to interpret entire statements as options in the previous statement. This can produce some very puzzling messages. So, if you get an error message that just doesn't make sense, look for missing semicolons.

**Example** The following program is missing a semicolon on the comment statement before the DATA statement:

```
* Read the data file ToadJump.dat using list input  
DATA toads;  
    INFILE 'c:\MyRawData\ToadJump.dat';  
    INPUT ToadName $ Weight Jump1 Jump2 Jump3;  
    RUN;
```

Here is the SAS log after the program has run:

---

```
1 * Read the data file ToadJump.dat using list input  
2 DATA toads;  
3     INFILE 'c:\MyRawData\ToadJump.dat';  
-----  
180  
ERROR 180-322: Statement is not valid or it is used out of proper order.  
4     INPUT ToadName $ Weight Jump1 Jump2 Jump3;  
-----  
180  
ERROR 180-322: Statement is not valid or it is used out of proper order.  
5 RUN;
```

---

In this case, DATA toads becomes part of the comment statement. Because there is now no DATA statement, SAS underlines the INFILE and INPUT keywords and says, “Hey, these statements are in the wrong place; they have to be part of a DATA step.” This doesn’t make much sense to you because you know INFILE and INPUT are valid statements, and you did put them in a DATA step (or so you thought). That’s when you should suspect a missing semicolon.

**Example** The next example shows the same program, but now the semicolon is missing from the DATA statement. The INFILE statement becomes part of the DATA statement, and SAS tries to create a SAS data set named INFILE. SAS also tries to interpret the filename,

'c:\MyRawData\ToadJump.dat' as a SAS data set name, but the .dat extension is not valid for SAS data sets. It also gives you an error saying that there is no DATALINES or INFILE statement. In addition, you get some warnings about data sets being incomplete. This is a good example of how one simple mistake can produce a lot of confusing messages:

---

```
30 * Read the data file ToadJump.dat using list input;
31 DATA toads
32 INFILE 'c:\MyRawData\ToadJump.dat';
33 INPUT ToadName $ Weight Jump1 Jump2 Jump3;
34 RUN;
ERROR: No DATALINES or INFILE statement.
ERROR: Extension for physical file name 'C:\MyRawData\ToadJump.dat'
does
not correspond to a valid member type.
NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set WORK.TOAD$ may be incomplete. When this step
was
stopped there were 0 observations and 5 variables.
WARNING: Data set WORK.TOAD$ was not replaced because this step was
stopped.
WARNING: The data set WORK.INFILE may be incomplete. When this step
was
stopped there were 0 observations and 5 variables.
```

---

Missing semicolons can produce a variety of error messages. Usually, the messages say that either a statement is not valid, or an option or parameter is not valid or recognized. Sometimes you don't get an error message, but the results are still not right.

**The DATASTMTCHK system option** Some missing semicolons, such as the one in the last example, are easier to find if you use the DATASTMTCHK system option. This option controls which names you can use for SAS data sets in a DATA statement. By default, it is set so that you cannot use the words: MERGE, RETAIN, SET, or

UPDATE as SAS data set names. This prevents you from accidentally overwriting an existing data set just because you forgot a semicolon at the end of a DATA statement. You can make all SAS keywords invalid SAS data set names by setting the DATASTMTCHK option to ALLKEYWORDS. The partial log below again shows a missing semicolon at the end of the DATA statement, but this time DATASTMTCHK is set to ALLKEYWORDS:

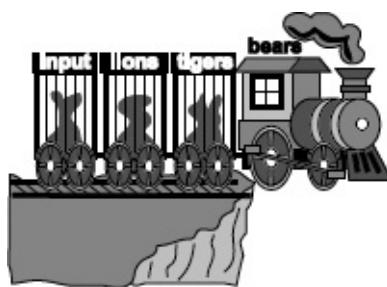
---

```
35 OPTIONS DATASTMTCHK=ALLKEYWORDS;
36 * Read the data file ToadJump.dat using list input;
37 DATA toads
38 INFILE 'C:\MyRawData\ToadJump.dat';
-----
57
ERROR 57-185: INFILE is not allowed in the DATA statement when option
DATASTMTCHK=ALLKEYWORDS. Check for a missing
semicolon in
      the DATA statement, or use DATASTMTCHK=NONE.
39 INPUT ToadName $ Weight Jump1 Jump2 Jump3;
40 RUN;
```

---



## 11.4 Note: INPUT Statement Reached Past the End of a Line



The note “SAS went to a new line when INPUT statement reached past the end of a line” is rather innocent looking,

but its presence can indicate a problem. This note often goes unnoticed. It doesn't come up in red lettering. It doesn't cause your program to stop. But look for it in your SAS log because it is a common note that usually means there is a problem.

This note means that as SAS was reading raw data, it got to the end of the data line before it read values for all the variables in your INPUT statement. When this happens, SAS goes by default to the next line of data to get values for the remaining variables. Sometimes this is exactly what you want SAS to do, but if it's not, take a good look at your SAS log and output to be sure you know why this is happening.

Check the note in your SAS log that tells you the number of lines SAS read from the data file and the number of observations in the SAS data set. If you have fewer observations than lines read, and you planned to have one observation per line, then you know you have a problem. Taking a close look at your data set can be very helpful in determining the source of the problem.

**Example** This example shows what can happen if you are using list input, and don't have periods for missing values. The following data come from a middle school long jump competition, where the student's number is followed by the distances for each of three jumps. When a student was disqualified for a jump, no entry was made for that jump:

```
10 3.5 3.7  
12 4.3 4.0 3.9  
23 3.8 3.9 4.0  
15 2.1 2.3  
16 4.0 2.2  
28 3.5 3.6 4.2
```

Here is the SAS log from a program that reads the raw data using list input:

---

```
1 DATA jumps;
2  INFILE 'c:\MyRawData\LongJump.dat';
3  INPUT StudentNumber Jump1 Jump2 Jump3;
4 RUN;
```

NOTE: The infile 'c:\MyRawData\LongJump.dat' is:

File Name=c:\MyRawData\LongJump.dat,  
RECFM=V,LRECL=256

① NOTE: 6 records were read from the infile  
'c:\MyRawData\LongJump.dat'.

The minimum record length was 10.

The maximum record length was 14.

③ NOTE: SAS went to a new line when INPUT statement reached past the  
end of a line.

② NOTE: The data set WORK.JUMPS has 4 observations and 4 variables.

NOTE: DATA statement used (Total process time):

real time 0.37 seconds

---

① Notice that six records were read from the raw data file.

② But there are only four observations in the SAS data set.

③ The note, "...INPUT statement reached past...", should  
alert you that there may be a problem.

If you look at the data set, you can see that there is a  
problem. The numbers don't look correct. (Can a person  
jump 16 meters?)

	<b>StudentNumber</b>	<b>Jump1</b>	<b>Jump2</b>
1	10	3.5	3.1
2	23	3.8	3.9
3	15	2.1	2.3
4	28	3.5	3.6

Here, SAS went to a new line when you didn't want it to. To fix this problem, the simplest thing to do is use the MISSOVER option in the INFILE statement. MISSOVER instructs SAS to assign missing values instead of going to the next line when it runs out of data. The INFILE statement would look like this:

```
INFILE 'c:\MyRawData\LongJump.dat' MISSOVER;
```

**Possible causes** Other reasons for receiving a note informing you that the INPUT statement reached past the end of the line include:

- ◆ You planned for SAS to go to the next data line when it ran out of data.
- ◆ Blank lines in your data file, usually at the beginning or end, can cause this note. Look at the minimum line length in the SAS log. If it is zero, then you have blank lines. Edit out the blank lines and rerun your program.
- ◆ If you are using list input and you do not have a space between every data value, you can get this note. For example, if you try to read the following data using list input, SAS will run out of data for the Gilroy Garlics because there is no space between the 15 and the 1035. SAS will read it as one number, then read the 12 where it should have been reading the 1035, and so on. To correct this problem, either add a space between the two numbers, or use column or formatted input.

Columbia Peaches	35 67 1 10 2 1
Gilroy Garlics	151035 12 11 7 6
Sacramento Tomatoes	124 85 15 4 9 1

- ◆ If you have some data lines that are shorter than the rest, and you are using column or formatted input, this can cause a problem. If you try to read a name, for example, in columns 60 through 70 when some of

the names extend only to column 68, and you didn't add spaces at the end of the line to fill it out to column 70, then SAS will go to the next line to read the name. To avoid this problem, use the TRUNCOVER option in the INFILE statement (discussed in Section 2.17). For example:

```
INFILE 'c:\MyRawData\Addresses.dat'  
TRUNCOVER;
```



## 11.5 Note: Lost Card



Lost card? You thought you were writing SAS programs, not playing a card game. This note makes more sense if you remember that computer programs and data used to be punched out on computer cards. A lost card means that SAS was expecting another line (or card) of data and didn't find it.

If you are reading multiple lines of raw data for each observation, then a lost card could mean you have missing or duplicate lines of data. If you are reading two data lines for each observation, then SAS will expect an even number of lines in the data file. If you have an odd number, then you will get the lost-card message. It can often be difficult to locate the missing or duplicate lines, especially with large data files. Printing or viewing the SAS data set as

well as careful proofreading of the data file can be helpful in identifying problem areas.

**Example** The following example shows what can happen if you have a missing line of data. The data values are the normal high and low temperatures and the record high and low for the month of July for each city, but the last city is missing a data line:

```
Nome AK
55 44
88 29
Miami FL
90 75
97 65
Raleigh NC
88 68
```

Here is the SAS log from a program which reads the data, three lines per observation:

---

```
1 DATA highlow;
2  INFILE 'c:\MyRawData\Temps1.dat';
3  INPUT City $ State $ / NormalHigh NormalLow / RecordHigh
RecordLow;
NOTE: The infile 'c:\MyRawData\Temps1.dat' is:
      File Name=c:\MyRawData\Temps1.dat,
      RECFM=V,LRECL=256
NOTE: LOST CARD.
      City=Raleigh State=NC NormalHigh=88 NormalLow=68 RecordHigh=.
      RecordLow=.
      _ERROR_=1 _N_=3
NOTE: 8 records were read from the infile 'c:\MyRawData\Temps1.dat'.
      The minimum record length was 5.
      The maximum record length was 10.
NOTE: The data set WORK.HIGHLOW has 2 observations and 6 variables.
NOTE: DATA statement used (Total process time):
      real time      0.03 seconds
      cpu time       0.03 seconds
```

---

In this case, you get the lost-card note, and SAS prints the data values that it read for the observation with the missing data. You can see from the log that SAS read eight records from the file but the SAS data set has only two observations. The incomplete observation was not included.

**Example** Often you get other messages along with the lost-card note. The invalid-data note is a common by-product of the lost card. If the second line were missing from the temperature data, then you would get invalid data because SAS would try to read Miami FL as the record high and low for Nome AK.

```
Nome AK  
88 29  
Miami FL  
90 75  
97 65  
Raleigh NC  
88 68  
105 50
```

Here is the SAS log showing the invalid-data note:

---

```
NOTE: Invalid data for RecordHigh in line 3 1-5.  
NOTE: Invalid data for RecordLow in line 3 7-8.  
RULE:  ----+---1---+---2---+---3---+---4---+---5---+---6---+  
3      Miami FL  
City=Nome State=AK NormalHigh=88 NormalLow=29 RecordHigh=.  
RecordLow=.  
_ERROR_=1 _N_=1  
NOTE: LOST CARD.
```

---

**Example** Along with the lost-card note, it is common to get a note indicating that the INPUT statement reached past the end of a line. If you forgot the last number in the file, as in the following example, then you would get these two notes together:

```
Nome AK
```

```
55 44  
88 29  
Miami FL  
90 75  
97 65  
Raleigh NC  
88 68  
105
```

When a program uses list input, SAS will try to go to the next line to get the data for the last variable. Since there isn't another line of data, you get the lost-card note.

---

```
NOTE: LOST CARD.  
City=Raleigh State=NC NormalHigh=88 NormalLow=68 RecordHigh=105  
RecordLow=._ERROR_=1 _N_=3  
NOTE: 9 records were read from the infile  
      'c:\MyRawData\Temps3.dat'.  
      The minimum record length was 3.  
      The maximum record length was 10.  
NOTE: SAS went to a new line when INPUT statement reached past the end  
      of  
      a line.  
NOTE: The data set WORK.HIGHLOW has 2 observations and 6  
      variables.
```

---



## 11.6 Note: Invalid Data

The typical new SAS user, upon seeing the invalid-data note, will ignore it, hoping perhaps that it will simply go away by itself. That's rather ironic considering that the message is explicit and easy to interpret once you know how to read it.

**Interpreting the message** The invalid-data note

appears when SAS is unable to read from a raw data file because the data are inconsistent with the INPUT statement. This note almost always indicates a problem. For example, one common mistake is typing in the letter O instead of the number 0. If the variable is numeric, then SAS is unable to interpret the letter O. In response, SAS does two things; it sets the value of this variable to missing and prints out a message like this for the problematic observation:

- 
- ① NOTE: Invalid data for IDNumber in line 8 1-5.
  - ② RULE:---+---1---+---2---+---3---+---4---+---5---+---6---+
  - ③ 8 007 James Bond SA341
  - ④ IDNumber=. Name=James Bond class="SA" Q1=3 Q2=4  
Q3=1 _ERROR_=1 _N_=8
- 

- ① The first line tells you where the problem occurred. Specifically, it states the name of the variable SAS got stuck on and the line number and columns of the raw data file that SAS was trying to read. In this example, the error occurred while SAS was trying to read a variable named IDNumber from columns 1 through 5 in line 8 of the input file.
- ② The next line is a ruler with columns as the increments. The numeral 1 marks the 10th column, 2 marks the 20th column, and so on. Below the ruler, SAS dumps the actual line of raw data so you can see the little troublemaker for yourself. Using the ruler as a guide, you can count over to the column in question. At this point you can compare the actual raw data to your INPUT statement, and the error is usually obvious. The value of IDNumber should be zero-zero-seven, but looking at the line of actual data you can see that a careless typist has typed zero-letter O-seven. Such an error may seem minor to you, but you'll soon learn that computers are hopelessly persnickety.

- ③ As if this weren't enough, SAS prints more information: the value of each variable for that observation as SAS read it. In this case, you can see that IDNumber equals missing, Name equals James Bond, and so on. Two automatic variables appear at the end of the line: _ERROR_ and _N_. The _ERROR_ variable has a value of 1 if there is a data error for that observation, and 0 if there is not. In an invalid-data note, _ERROR_ always equals 1. The automatic variable _N_ is the number of times SAS has looped through the DATA step.

**Unprintable characters** Occasionally, invalid data contain unprintable characters. In these cases, SAS shows you the raw data in hexadecimal format.

NOTE: Invalid data for IDNumber in line 10 1-5.

RULE: -----+-----1-----+-----2-----+-----3-----+-----4-----+-----5-----+-----6-----+

- ① CHAR .. Indiana Jones PI83.

2

2

IdNumber=. Name=Indiana Jones class="PI" Q1=8 Q2=3 Q3=.  
ERROR =1 N =10

- ① As before, SAS prints the line of raw data that contains the invalid data.
  - ② Directly below the line of raw data, SAS prints two lines containing the hexadecimal equivalent of the data. You needn't understand hexadecimal values to be able to read this. SAS prints the data this way because the normal 10 numerals and 26 letters don't provide enough values to represent all computer symbols uniquely. Hexadecimal uses two characters to represent each symbol. To read hexadecimal, take

a digit from the first line (labeled ZONE) together with the corresponding digit from the second line (labeled NUMR). In this case, a tab slipped into column 2 and appears as a harmless-looking period in the line of data. In hexadecimal, however, the tab appears as 09, while a real period in column 1 is 2E in hexadecimal. (In z/OS the hexadecimal representation of a tab is 05.)

**Possible causes** Common reasons for receiving the invalid-data note include the following:

- ◆ character values in a field that should be numeric (including using the letter O instead of the numeral zero)
- ◆ forgetting to specify that a variable is character (SAS assumes it is numeric)
- ◆ incorrect column specifications producing embedded spaces in numeric data
- ◆ list-style input with two periods in a row and no space in between
- ◆ missing data not marked with a period for list-style input causing SAS to read the next data value
- ◆ special characters such as tab, carriage-return-line-feed, or form-feed in numeric data
- ◆ using the wrong informat such as MMDDYY. instead of DDMMYY.
- ◆ invalid dates (such as September 31) read with a date informat

## Double question mark informat

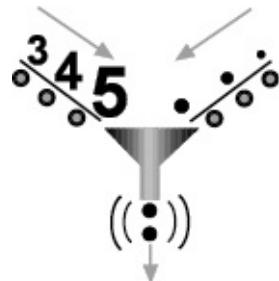
**modifier** Sometimes you have invalid data, and there is nothing you can do about it. You know the data are bad, and you just want SAS to go ahead and set those values to missing without filling your log with notes. At those times, you can use the ?? informat modifier. The ?? informat modifier suppresses the invalid-data note, and prevents the

automatic variable `_ERROR_` from being set to 1. Just insert the two question marks after the name of the problematic variable and before any informat or column specifications. For example, to prevent the preceding invalid-data notes for the variable `IdNumber`, you would add `??` to the `INPUT` statement, like this:

```
INPUT IdNumber ?? 1-5 Name $ 6-18 Class $ 20-21 Q1  
22 Q2 23 Q3 24;
```



## 11.7 Note: Missing Values Were Generated



The missing-values note appears when SAS is unable to compute the value of a variable because of preexisting missing values in your data. This is not necessarily a problem. It is possible that your data contain legitimate missing values and that setting a new variable to missing is a desirable response. But it is also possible that the missing values result from an error and that you need to fix your program or your data. A good rule is to think of the missing-values note as a flag telling you to check for an error.

**Example** Here are data from a toad-jumping contest including the toad's name, weight, and the distance jumped in each of three trials:

Lucky 2.3 1.9 . 3.0  
Spot 4.6 2.5 3.1 .5  
Toadzilla 7.1 .. 3.8  
Hop 4.5 3.2 1.9 2.6  
Noisy 3.8 1.3 1.8 1.5  
Winner 5.7 ...

Notice that several of the toads have missing values for one or more jumps. To compute the average distance jumped, the program in the following SAS log reads the raw data, adds together the values for the three jumps, and divides by three:

---

```
1 DATA toads;
2 LENGTH ToadName $ 9;
3 INFILE 'c:\MyRawData\ToadJump.dat';
4 INPUT ToadName Weight Jump1 Jump2 Jump3;
5 AverageJump = (Jump1 + Jump2 + Jump3) / 3;
6 RUN;
```

NOTE: The infile 'c:\MyRawData\ToadJump.dat' is:

Filename=c:\MyRawData\ToadJump.dat,  
RECFM=V,LRECL=32767, File Size (bytes)=125

NOTE: 6 records were read from the infile 'c:\MyRawData\ToadJump.dat'.

The minimum record length was 16.

The maximum record length was 21.

NOTE: ① Missing values were generated as a result of performing an operation on missing values.

② Each place is given by: (Number of times) at (Line):(Column)  
3 at 5:25

NOTE: The data set WORK.TOADDS has 6 observations and 6 variables.

---

Because of missing values in the data, SAS was unable to compute AverageJump for some of the toads. In response, SAS printed the missing-values note, which has two parts:

- ① The first part of the note says that SAS was forced to set some values to missing.
- ② The second part is a bit more cryptic. SAS lists the number of times values were set to missing. This

generally corresponds to the number of observations that generated missing values, unless the problem occurs within a DO loop. Next, SAS states where in the program it encountered the problem. In the preceding example, SAS set three values to missing: at line 5, column 25. Looking at the program, you can see that line 5 is the line that calculates AverageJump, and column 25 contains the first plus sign. Looking at the raw data, you can see that three observations have missing values for Jump1, Jump2, or Jump3. Those observations are the three times mentioned in the missing-values note.

**Finding the missing values** In this case, it was easy to find the observations with missing values. But if you had a data set with hundreds, or millions, of observations, then you couldn't just glance at the data. In that case, you could subset the problematic observations with a subsetting IF statement, like this:

```
DATA missing;  
  SET toads;  
  IF AverageJump = .;  
RUN;
```

Once you have selected just the observations that have missing values, you can examine them more closely. Here are the observations with missing values for AverageJump:

	ToadName	Weight	Jump1	Jump2	Jump3	A
1	Lucky	2.3	1.9	.	.	3.0
2	Toadzilla	7.1	.	.	.	3.8
3	Winner	5.7	.	.	.	.

**Using the SUM and MEAN functions** You may be able to circumvent this problem when you are computing a sum or mean by using the SUM or MEAN function instead of an arithmetic expression. In the preceding program, you could remove this line:

```
AverageJump = (Jump1 + Jump2 + Jump3) / 3;
```

and substitute this line in its place:

```
AverageJump = MEAN(Jump1, Jump2, Jump3);
```

The SUM and MEAN functions ignore missing values by using only nonmissing values in the computation. In this example, you would still get the missing-values note for one toad, Winner, because it had missing values for all three jumps.



## 11.8 Note: Numeric Values Have Been Converted to Character (or Vice Versa)

Even with only two data types, numeric and character, SAS programmers sometimes get their variables mixed up. When you accidentally mix numeric and character variables, SAS tries to fix your program by converting variables from numeric to character or vice versa, as needed. Programmers sometimes ignore this problem, but that is not a good idea. If you ignore this message, it may come back to haunt you as you find new incompatibilities resulting from the fix. If, indeed, a variable needs to be converted, you should do it yourself, explicitly, so you know what your variables are doing.

**Example** To show how SAS handles this kind of incompatibility, here are data about a class. Each line of

data contains a student's ID number, name, and scores on two tests.

```
110 Linda 53 60
203 Derek 72 64
105 Kathy 98 82
224 Michael 80 55
```

The instructor runs the following program to read the data and create a permanent SAS data set named SCORES.

```
LIBNAME students 'c:\MySASLib';
DATA students.scores;
INFILE 'c:\MyRawData\TestScores.dat';
INPUT StudentID Name $ Score1 Score2 $;
RUN;
```

After creating the permanent SAS data set, the instructor runs a program to compute the total score and substring the first digit of StudentID. (Students in section 1 of the class have IDs starting with 1 while students in section 2 have IDs starting with 2.) Here is the log from the program:

---

```
2 DATA grades;
3   SET students.scores;
4   TotalScore = Score1 + Score2;
5   Class = SUBSTR(StudentID,1,1);
6   RUN;
NOTE: Character values have been converted to numeric values at the places
      given by:(Line):(Column).
4:26
NOTE: Numeric values have been converted to character values at the places
      given by:(Line):(Column).
5:19
NOTE: There were 4 observations read from the data set
      STUDENTS.SCORES.
NOTE: The data set WORK.GRADES has 4 observations and 6 variables.
NOTE: DATA statement used (Total process time):
      real time      0.04 seconds
      cpu time       0.04
```

---

This program produces two values-have-been-converted notes. The first conversion occurred in line 4, column 26. Looking at line 4 of the log, you can see that the variable name Score2 appears in column 26. Score2 was accidentally input as a character variable, so SAS had to convert it to numeric before adding it to Score1 to compute TotalScore.

The second conversion occurred in line 5, column 19. Looking at line 5 of the log, you can see that the variable StudentID appears in column 19. StudentID was input as a numeric variable, but the SUBSTR function requires character variables, so SAS was forced to convert StudentID to character.

**Converting variables** You could go back and input the raw data with the correct types, but sometimes that's just not practical. Instead, you can convert the variables from one type to another. To convert variables from character to numeric, you use the INPUT function. To convert from numeric to character, you use the PUT function. Most often, you would use these functions in an assignment statement with the following syntax:

### Character to Numeric

### Numeric to Char

*newvar* = INPUT(*oldvar*, *informat*);      *newvar* = PUT(*oldvar*,

These two slightly eccentric functions are first cousins of the PUT and INPUT statements. Just as an INPUT statement uses informats, the INPUT function uses informats; and just as PUT statements use formats, the PUT function uses formats. These functions can be confusing because they are similar but different. In the case of the INPUT function, the informat must be the type you are converting to—numeric. In contrast, the format for the PUT

function must be the type you are converting from —numeric. To convert the troublesome variables in the preceding program, you would use these statements:

### Character to Numeric

```
NewScore2 = INPUT(Score2, 2.);
```

### Numeric to Char

```
NewID = PUT(Stu
```

Here is a log showing the program with the statements to convert Score2 and StudentID. This version of the program runs without any suspicious messages:

---

```
7  DATA grades;
8  SET students.scores;
9  NewScore2 = INPUT(Score2, 2.);
10 TotalScore = Score1 + NewScore2;
11 NewID = PUT(StudentID,3.);
12 Class = SUBSTR(NewID,1,1);
13 RUN;
```

```
NOTE: There were 4 observations read from the data set
STUDENTS.SCORES.
```

```
NOTE: The data set WORK.GRADES has 4 observations and 8 variables.
```

```
NOTE: DATA statement used (Total process time):
```

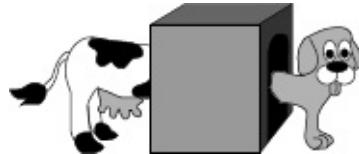
real time	0.03 seconds
cpu time	0.03 seconds

---

Note that this section is about converting variables from numeric to character or vice versa, but you can also use the PUT function to change one character value to another character value. When you do that, both *oldvar* and *newvar* would be character variables, and the format would be a character format. See Section 4.14 for more about using the PUT function.



## 11.9 DATA Step Produces Wrong Results but No Error Message



Some of the hardest errors to debug aren't errors at all, at least not to SAS. If you do complex programming, you may write a DATA step that runs just fine—without any errors or suspicious notes—but the DATA step produces the wrong results. The more complex your programs are, the more likely you are to get this kind of error. Sometimes it seems like a DATA step is a black box. You know what goes in, and you know what comes out, but what happens in the middle is a mystery. This problem is actually a logic error; somewhere along the way, SAS got the wrong instruction.

**Example** Here is a program that illustrates this problem and how to debug it. The raw data below contain information from a class. For each student there are three scores from tests, and one score from homework:

```
Linda 53 60 66 42  
Derek 72 64 56 32  
Kathy 98 82 100 48  
Michael 80 55 95 50
```

This program is supposed to select students whose average score is below 70, but it doesn't work. Here is the log from the wayward program:

---

```
1 * Keep only students with mean below 70;  
2 DATA lowscore;  
3  INFILE 'c:\MyRawData\Class.dat';  
4  INPUT Name $ Score1 Score2 Score3 Homework;  
5  Homework = Homework * 2;  
6  AverageScore = MEAN(Score1 + Score2 + Score3 + Homework);
```

```
7    IF AverageScore < 70;  
8  RUN;  
NOTE: The infile 'c:\MyRawData\Class.dat' is:  
      File Name=c:\MyRawData\Class.dat,  
      RECFM=V,LRECL=256  
NOTE: 4 records were read from the infile 'c:\MyRawData\Class.dat'.  
      The minimum record length was 20.  
      The maximum record length was 20.  
NOTE: The data set WORK.LOWSCORE has 0 observations and  
      6 variables.
```

---

First, the DATA step reads the raw data from a file named Class.dat. The highest possible score on homework is 50. To make the homework count the same as a test, the program doubles the value of Homework. Then the program computes the mean of the three test scores and Homework, and subsets the data by selecting only observations with a mean score below 70. Unfortunately, something went wrong. The LOWSCORE data set contains no observations. A glance at the raw data confirms that there should be students whose mean scores are below 70.

**Using the PUT and PUTLOG statements to debug** To debug a problem like this, you have to figure out exactly what is happening inside the DATA step. A good way to do this—especially if your DATA step is long and complex—is with PUT or PUTLOG statements. Elsewhere in this book, PUT statements are used along with FILE statements to write raw data files and custom reports. If you use a PUT statement without a FILE statement, then SAS writes in the SAS log. PUTLOG statements are the same except that they always write to the log even when you have a FILE statement. PUT and PUTLOG statements can take many forms, but for debugging, a handy style is:

```
PUTLOG _ALL_;
```

SAS will print all the variables in your data set. If you have a lot of variables, you can print just the relevant ones this

way:

```
PUTLOG variable-1= variable-2= ... variable-n=;
```

The DATA step below is identical to the one shown earlier except that a PUTLOG statement was added. In a longer DATA step, you might choose to have PUTLOG statements at several points. In this case, one will suffice. This PUTLOG statement is placed before the subsetting IF, since in this particular program the subsetting IF eliminates all observations:

---

```
9 * Keep only students with mean below 70;
10 DATA lowsore;
11  INFILE 'c:\MyRawData\Class.dat';
12  INPUT Name $ Score1 Score2 Score3 Homework;
13  Homework = Homework * 2;
14  AverageScore = MEAN(Score1 + Score2 + Score3 + Homework);
15  PUTLOG Name= Score1= Score2= Score3= Homework=
AverageScore=;
16  IF AverageScore < 70;
17  RUN;
```

NOTE: The infile 'c:\MyRawData\Class.dat' is:

```
FILE NAME=c:\MyRawData\Class.dat,
RECFM=V,LRECL=256
```

```
Name=Linda Score1=53 Score2=60 Score3=66 Homework=84
```

```
AverageScore=263
```

```
Name=Derek Score1=72 Score2=64 Score3=56 Homework=64
```

```
AverageScore=256
```

```
Name=Kathy Score1=98 Score2=82 Score3=100 Homework=96
```

```
AverageScore=376
```

```
Name=Michael Score1=80 Score2=55 Score3=95 Homework=100
```

```
AverageScore=330
```

NOTE: 4 records were read from the infile 'c:\MyRawData\Class.dat'.

The minimum record length was 20.

The maximum record length was 20.

NOTE: The data set WORK.LOWSCORE has 0 observations and  
6 variables.

---

Looking at the log, you can see the result of the PUTLOG statement. The data listed in the middle of the log show that

the variables are being input properly, and the variable Homework is being adjusted properly. However, something is wrong with the values of AverageScore; they are much too high. There is a syntax error in the line that computes AverageScore. Instead of commas separating the three score variables in the MEAN function, there are plus signs. Since functions can contain arithmetic expressions, SAS simply added the four variables together, as instructed, and computed the mean of a single number. That's why the values of AverageScore are all above 70.

Depending on the interface you use, you may have another option for finding logic errors: an interactive DATA step debugger. DATA step debuggers allow you to step through your code examining data values at various points inside a DATA step. For more information, check the SAS Documentation for your interface.



## 11.10 Error: Invalid Option, Error: The Option Is Not Recognized, or Error: Statement Is Not Valid

If SAS cannot make sense out of one of your statements, it stops executing the current DATA or PROC step and prints one of these messages:

ERROR 22-7: Invalid option name.

ERROR 202-322: The option or parameter is not recognized and will be ignored.

ERROR 180-322: Statement is not valid or it is used out of proper order.

The invalid-option message and its cousin, the option-is-

not-recognized message, tell you that you have a valid statement, but SAS can't make sense out of an apparent option. The statement-is-not-valid message, on the other hand, means that SAS can't understand the statement at all. Thankfully, with all three messages SAS underlines the point at which it got confused so you know where to look for the problem.

**Example** The SAS log below contains an invalid option:

---

```
1  DATA scores (ROP = Score1);
---
22
ERROR 22-7: Invalid option name ROP.

2  INFILE 'c:\MyRawData\Class.dat';
3  INPUT Name $ Score1 Score2 Score3 Homework;
4  RUN;

NOTE: The SAS System stopped processing this step because of errors.
NOTE: DATA statement used (Total process time):
      real time      0.03 seconds
      cpu time      0.00 seconds
```

---

In this DATA step, the word DROP was misspelled as ROP. Since SAS cannot interpret this, it underlines the word ROP, prints the invalid-option message, and stops processing the DATA step.

**Example** The following log contains an option-is-not-recognized message:

---

```
5  PROC PRINT
6  VAR Score2;
---
22
202
ERROR 22-322: Syntax error, expecting one of the following: ;
BLANKLINE, CONTENTS,
      DATA, DOUBLE, GRANDTOTAL_LABEL,
GRANDTOT_LABEL, GRAND_LABEL,
      GTOTAL_LABEL, GTOT_LABEL, HEADING, LABEL, N,
```

```
NOOBS, NOSUMLABEL,  
      OBS, ROUND, ROWS, SPLIT, STYLE, SUMLABEL, UNIFORM,  
      WIDTH.  
ERROR 202-322: The option or parameter is not recognized and will be  
ignored.  
7  RUN;  
NOTE: The SAS System stopped processing this step because of errors.  
NOTE: PROCEDURE PRINT used (Total process time):  
      real time      0.25 seconds  
      cpu time      0.09 seconds
```

---

SAS underlined the VAR statement. This message may seem puzzling since VAR is not an option, but a statement, and a valid statement at that. But if you look at the previous statement, you will see that the PROC statement is missing one of those pesky semicolons. As a result, SAS tried to interpret the words VAR and Score2 as options in the PROC statement. Since no options exist with those names, SAS stopped processing the step and printed the option-is-not-recognized message. SAS also printed the syntax-error message listing all the valid options for a PROC PRINT statement.

**Example** Here is a log with the statement-is-not-valid message:

```
8  PROC PRINT;  
9   SET class;  
---  
180  
ERROR 180-322: Statement is not valid or it is used out of proper order.  
10  RUN;  
NOTE: The SAS System stopped processing this step because of errors.  
NOTE: PROCEDURE PRINT used (Total process time):  
      real time      0.01 seconds  
      cpu time      0.01 seconds
```

---

In this case, a SET statement was used in a PROC step.

Since SET statements can be used only in DATA steps, SAS underlines the word SET and prints the statement-is-not-valid message.

**Possible causes** Generally, with these error messages, the cause of the problem is easy to detect. You should check the underlined item and the previous statement for possible errors. Possible causes include the following:

- ◆ a misspelled keyword
- ◆ a missing semicolon
- ◆ a DATA step statement in a PROC step (or vice versa)
- ◆ a RUN statement in the middle of a DATA or PROC step (this does not cause errors for some procedures)
- ◆ the correct option with the wrong statement
- ◆ an unmatched quotation mark
- ◆ an unmatched comment



## 11.11 Note: Variable Is Uninitialized or Error: Variable Not Found

If you find one of these messages in your SAS log, then SAS is telling you that the variable named in the message (Temp in this case) does not exist:

NOTE: Variable Temp is uninitialized.

WARNING: Variable Temp not found.

ERROR: Variable Temp not found.

Generally, the first time that you get one of these messages, it is quite a shock. You may be sure that the variable does

exist. After all, you remember creating it. Fortunately, the problem is usually easy to fix once you understand what SAS is telling you.

If the problem happens in a DATA step, then SAS prints the variable-is-uninitialized note, initializes the variable, and continues to execute your program. Normally, variables are initialized when they are read (via an INPUT, SET, MERGE, or UPDATE statement) or when they are created via an assignment statement. If you use a variable for the first time in a way that does not assign a value to the variable (such as on the right side of an assignment statement, in the condition of an IF statement, or in a DROP or KEEP option), then SAS tries to fix the problem by assigning a value of missing to the variable for all observations. This is very generous of SAS, but it almost never fixes the problem, since you probably don't want the variable to have missing values for all observations.

When the problem happens in a PROC step, the results are more grave. If the error occurs in a critical statement, such as a VAR statement, then SAS prints the variable-not-found error and does not execute the step. If the error occurs in a less critical statement, such as a LABEL statement, then SAS prints the variable-not-found warning message and attempts to run the step.

**Example** Here is the log from a program with missing-variable problems in both a DATA and a PROC step:

---

```
1  DATA highscores (KEEP = Name Total);
2  INFILE 'c:\MyRawData\TestScores.dat';
3  INPUT StudentID Name $ Score1 Score2;
4  IF Scor1 > 90;
5  Total = Score1 + Score2;
6  RUN;

NOTE: Variable Scor1 is uninitialized.
NOTE: The data set WORK.HIGHSCORES has 0 observations and 2
variables.
NOTE: DATA statement used (Total process time):
```

real time	0.04 seconds
cpu time	0.03 seconds

7

---

```
8 PROC PRINT DATA = highscores;
9   VAR Name Score2 Total;
ERROR: Variable SCORE2 not found.
10 RUN;
NOTE: The SAS System stopped processing this step because of errors.
NOTE: PROCEDURE PRINT used (Total process time):
      real time      0.03 seconds
      cpu time      0.01 seconds
```

---

In this DATA step, the INPUT statement reads four variables: StudentID, Name, Score1, and Score2. But a misspelling in the subsetting IF statement causes SAS to initialize a new variable named Scor1. Because Scor1 has missing values, none of the observations satisfies the subsetting IF, and the data set HIGHSCORES is left with zero observations.

In the PROC PRINT, the VAR statement requests three variables: Name, Score2, and Total. Score2 did exist but was dropped from the data set by the KEEP= option in the DATA statement. That KEEP= option kept only two variables, Name and Total. As a result, SAS prints the variable-not-found error message, and does not execute the PROC PRINT.

**Possible causes** Common ways to “lose” variables include the following:

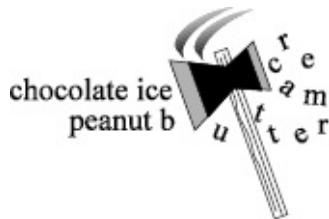
- ◆ misspelling a variable name
- ◆ using a variable that was dropped at some earlier time
- ◆ using the wrong data set
- ◆ committing a logic error, such as using a variable

before it is created

If the source of the problem is not immediately obvious, a look at the properties of the data set can often help you figure out what is going on. You can examine the properties of a data set using the Properties window or PROC CONTENTS. Both of these give you information about what is in a SAS data set, including variable names. To open a Properties window, right-click the icon for a data set and select Properties from the pop-up menu. PROC CONTENTS is covered in Section 2.3.



## 11.12 SAS Truncates a Character Variable



Sometimes you may notice that some, or all, of the values of a character variable are truncated. You may be expecting “peanut butter” and get “peanut b” or “chocolate ice cream” and get “chocolate ice.” This usually happens when you use IF statements to create a new character variable, or when you are using list-style input and you have values longer than eight characters. All character variables have a fixed length determined by one of the following methods.

**INPUT statement** If you are using an INPUT statement with list-style input, then the length defaults to 8. If you are using column or formatted input, then the length is determined by the number of columns, or the informat. Here are examples of INPUT statements that read a

variable named Food, and the resulting lengths:

<b>INPUT statement</b>	<b>Length of Food</b>
INPUT Food \$;	8
INPUT Food \$ 1-10;	1 0
INPUT Food \$15.;	1 5

**Assignment statement** If you are creating the variable in an assignment statement, then the length is determined by the first occurrence of the new variable name. For example, the following program creates a variable, Status, whose values depend on the value of the variable Temperature:

```
DATA summer;  
  SET temps;  
  IF Temperature > 100 THEN Status = 'Hot';  
  ELSE Status = 'Cold';  
RUN;
```

Because the word Hot has three characters and this is the first time the variable Status is used, SAS gives Status a length of 3. Any other values for this variable would be truncated to three characters (Col instead of Cold, for example).

**LENGTH statement** The LENGTH statement in a DATA step defines variable lengths and, if it comes before an INPUT or assignment statement, will override either of the previous two methods of determining length. The

following LENGTH statement sets the length of the variable Status to 4 and the variable Food to 15:

```
LENGTH Status $4 Food $15;
```

**ATTRIB statement** You can also assign variable lengths in an ATTRIB statement in a DATA step where you can associate formats, informats, labels, and lengths with variables in a single statement. Always place the LENGTH option before a FORMAT option in an ATTRIB statement to ensure that the variables are assigned proper lengths. For example, the following statement creates a character variable named Status with a length of 4 and the label Hot or Cold:

```
ATTRIB Status LENGTH = $4 LABEL = 'Hot or Cold';
```

**Example** This example shows what can happen if you let SAS determine the length of a character variable (in this case, using an assignment statement). Here are data for a consumer survey of car color preferences. Age is followed by sex (coded as 1 for male and 2 for female), annual income, and preferred car color (yellow, gray, blue, or white):

```
Age,Sex,Income,Color  
19,1,28000,Y  
45,1,130000,G  
72,2,70000,B  
31,1,88000,Y  
58,2,166000,W
```

In the following program, a series of IF-THEN/ELSE statements create a variable named AgeGroup based on the value of Age.

```
DATA carsurvey;  
  INFILE 'c:\MyRawData\Cars.csv' DLM = ','  
  FIRSTOBS = 2;  
  INPUT Age Sex Income Color $;  
  IF Age < 20 THEN AgeGroup = 'Teen';
```

```

ELSE IF Age < 65 THEN AgeGroup = 'Adult';
ELSE AgeGroup = 'Senior';
RUN;

```

Here is the CARSURVEY data set. Notice that the values of AgeGroup are truncated to four characters—the number of characters in Teen.

	<b>Age</b>	<b>Sex</b>	<b>Income</b>	<b>Color</b>	<b>A</b>
1	19	1	28000	Y	Teen
2	45	1	130000	G	Adul
3	72	2	70000	B	Seni
4	31	1	88000	Y	Adul
5	58	2	166000	W	Adul

Adding a LENGTH statement to the DATA step will eliminate the truncation problem:

```

DATA carsurvey;
INFILE 'c:\MyRawData\Cars.csv' DLM = ','
FIRSTOBS = 2;
INPUT Age Sex Income Color $;
LENGTH AgeGroup $6;
IF Age < 20 THEN AgeGroup = 'Teen';
ELSE IF Age < 65 THEN AgeGroup = 'Adult';
ELSE AgeGroup = 'Senior';
RUN;

```

Here is the new data set with untruncated values for AgeGroup:

---

	<b>Age</b>	<b>Sex</b>	<b>Income</b>	<b>Color</b>	<b>A</b>
1	19	1	28000	Y	Teen
2	45	1	130000	G	Adult
3	72	2	70000	B	Senior
4	31	1	88000	Y	Adult
5	58	2	166000	W	Adult

## 11.13 Saving Memory or Disk Space

What do you do when you finally get your program working, and it seems to take forever to run, or—even worse—you get a message that your computer is out of memory or disk space? Well, you could petition to buy a more powerful computer, which isn’t really such a bad idea, but there are a few things you can try before resorting to spending money. Because this issue depends on your operating environment, it is not possible to cover everything you might be able to do in this section. However, this section describes a few universal actions you can take to remedy the situation.

It is helpful, in trying to solve the problem, to know why it happens. Usually, when you run out of memory, it’s when you are doing some pretty intensive computations or sorting data sets with lots of variables. The GLM procedure (General Linear Models), for example, can use lots of memory when your model is complicated and there are many levels for each classification variable. You run out of

disk space because SAS uses disk space to store all its temporary working files, including temporary SAS data sets, and the SAS log and output. If you are creating many large temporary SAS data sets during the course of a SAS session, this can quickly fill up your disk space.

**Deleting unneeded data sets** SAS automatically deletes data sets in the WORK library when you your job or session ends. But if you have large temporary data sets, you can free up disk space by deleting them as soon as you are finished with them. To do this, use PROC DELETE with this general form:

PROC DELETE DATA = *data-set-name*;

**Reducing the storage size of variables** If your files are too big, one thing you can do is decrease the number of bytes needed to store individual variables. This can also help memory problems that arise when sorting data sets with character data. Since all numbers are expanded to the fullest precision (eight bytes) while SAS is processing data, changing storage requirements for numeric data will not help memory problems. If memory or disk space is at a premium, you can usually find some variables that require fewer bytes.

For character data, each character requires one byte of storage. The length of a character variable is determined when it is created. If you are using list input, then by default, variables are given a length of eight. If your data are only one character long, Y or N for example, then you are using eight times the storage space you actually need. You can use a LENGTH or ATTRIB statement in a DATA step to change the variable's length. For example, the following gives the character variable Answer a length of one byte:

LENGTH Answer \$1;

Be sure to put the LENGTH or ATTRIB statement before

any other statements that refer to the variable.

If you are running out of disk space, in addition to shortening the lengths of character variables, you may also be able to decrease the lengths of numeric variables.

Numeric data are a little trickier than character when it comes to length. All numbers can be safely stored in eight bytes, and that's why eight is the default. Some numbers can be safely stored in fewer bytes, but which numbers depends on your operating environment. Check the SAS Documentation for your operating environment to determine the length and precision of numeric variables. Under Windows and UNIX, for example, you can safely store integers up to 8,192 in three bytes. In general, if your numbers contain decimal values, then you must use eight bytes. If you have small integer values, then you can use four bytes (in some operating environments two or three bytes). Use the LENGTH statement to change the lengths of data:

```
LENGTH Tigers 4;
```

This statement changes the length of the numeric variable Tigers to four bytes. If your numbers are categorical, like 1 for male and 2 for female, then you can read them as character data with a length of 1 and save even more space.

**Reducing the number of observations** If you are going to use only a fraction of your data in a DATA step, then subset as soon as possible using a subsetting IF statement, WHERE statement, or WHERE= data set option. If you are using a procedure, you may be able to skip a DATA step entirely by subsetting in the procedure using a WHERE statement or WHERE= data set option. This PROC PRINT, for example, uses a WHERE statement to subset:

```
PROC PRINT DATA = survey;
  WHERE Sex = 'female';
  RUN;
```

**Reducing the number of variables** If you need only a few of the variables in your data set, then use the KEEP= (or DROP=) data set option (Section 6.10). For example, if you had a data set containing information about all the zoo animals, but you wanted to look at only the lions and tigers, then you could use the following statements:

```
DATA partial;  
  SET zooanimals (KEEP = Lions Tigers);  
  RUN;
```

**Compressing data sets** It is also possible to compress SAS data sets. Compressing may save space if your data have many repeated values. But beware, compressing can in some cases actually increase the size of your data set. Fortunately, SAS prints a message in your log telling you the change in size of your data sets. You can turn on compression by using either the COMPRESS=YES system option, or the COMPRESS=YES data set option. Use the system option if you want all the SAS data sets that you create to be compressed. Use the data set option when you want to control which SAS data sets to compress. For example:

```
DATA compressedzooanimals (COMPRESS = YES);  
  SET zooanimals;  
  RUN;
```

**Memory** If memory is your problem, then do what you can to eliminate other programs that are using your computer's memory. If you are using an interactive environment to run your SAS programs, try running in batch mode instead. Also, see the SAS Documentation for your operating environment for potential ways to make more memory available on your system.

If you have tried all of the above, and you are still running out of memory or disk space, then you can always try finding a more powerful computer. One of the nice things about SAS is that the language is the same for all operating

environments. To move your program to another operating environment, you would need to change only a few statements like INFILE or LIBNAME, which deal directly with the operating environment.

# **Index**

## **Special Characters**

_ALL_ variable name list 88  
    in PUT statements 308-309  
_CHARACTER_ variable name list 88-89  
_ERROR_ automatic variable 198  
    invalid data message 302-303  
_FREQ_ variable in MEANS procedure 114-115  
_N_ automatic variable 198-199  
    invalid data message 302-303  
_NAME_ variable, TRANSPOSE procedure 196-197  
_NULL_ data set name 111, 288-289  
_NUMERIC_ variable name list 88-89  
_PAGE_ keyword in PUT statements 110-111  
_TYPE_ variable, MEANS procedure 114-115  
; semicolon 2  
    missing 296-297  
: colon modifier 42-43  
! comparison operator 67, 96  
?? informat modifier 303  
@ line-hold specifier 48-49, 288  
    compared to @@ 49  
@@ line-hold specifier 46-47  
    compared to @ 49  
@'character' column pointer 48-49  
@n column pointer 40-41, 288-289  
* ; comments 3  
/ line pointer 44-45, 288  
/* */ comments 3  
& ampersand modifier 42-43  
& comparison operator 67, 96  
& macro variable prefix 202  
&SYSDATE macro variable 206-207

&SYSDAY macro variable 206  
&SYSNOBS macro variable 206  
#n line pointer 44-45, 288  
% in LIKE comparisons 96-97  
% macro prefix 202  
%DO statements 212-215  
%ELSE statement 212-213  
%END statement 212-215  
%IF-%THEN statements 212-213  
%LET statement 204-207  
%MACRO statement 208-211  
%MEND statement 208-209  
%PUT statement 220  
^= comparison operator 66, 96  
+n column pointer 37  
<= comparison operator 66, 96  
< comparison operator 66-67, 96  
= comparison operator 66-67, 96-97  
=: comparison operator 66  
=* comparison operator 96-97  
> = comparison operator 66, 96  
> comparison operator 66-67, 96  
| comparison operator 67, 96  
|| concatenation operator 63  
~= comparison operator 66, 96  
¬= comparison operator 82, 102  
\$CHARw. informat 38-39  
\$UPCASEw. format 106-107  
\$UPCASEw. informat 38-39  
\$w. format 106-107  
\$w. informat 38-39

## A

ACROSS usage option 132, 134-135  
ACROSS= option in KEYLEGEND statement 240  
AFTER location in REPORT procedure 136-137  
age, calculating 80-81

AGREE option in FREQ procedure 260, 262  
AGREEMENTPLOT option in TTEST procedure 258  
AGREEPLOT option in FREQ procedure 262  
ALL keyword in TABULATE procedure 124  
ALL option in TTEST procedure 258  
ALPHA= option  
    fitted curves 236  
    MEANS procedure 254-255  
    TTEST procedure 256  
    VBAR or HBAR statement 226  
analysis of variance 272-275  
ANALYSIS usage option 132-133  
AND operator 66-67, 96  
annotation in graphics 240-241  
ANOVA procedure 272-275  
ANYALNUM function 62-63  
ANYALPHA function 62-63  
ANYDIGIT function 62-63  
ANYDTDTEw. informat 15, 38-39, 82-83  
ANYSPACE function 62-63  
arithmetic operators 58-59  
ARRAY statement 86-87  
ASCII files 18, 30  
ASCII sort order 100  
assignment statements 58-59  
    dates 80-81  
    functions 60-63  
ATTRIB statement 314  
attributes, style  
    PRINT procedure 156-157  
    REPORT procedure 158-159  
    Table of 164-165  
    TABULATE procedure 160-161  
autocall library, macro 209  
autoexec file or process flow 14  
automatic variables  
    _ERROR_ 198

_N_ 198-199  
FIRST.byvariable 198-199  
LAST.byvariable 198-199  
macro 206-207  
axes, controlling in graphs 238-239  
AXIS= option in REFLINE statement 238

## B

BACKGROUND style attribute 164-165  
BACKGROUNDCOLOR style attribute 164-165  
BACKGROUNDIMAGE style attribute 164-165  
bar charts 226-227, 262-263  
BARWIDTH= option in VBAR or HBAR statement 226  
batch mode 11  
BCOLOR= option in TITLE statement 154-155  
BEFORE location in REPORT procedure 136  
BESTw. format 106-107  
BETA option for distribution plots 252  
BETWEEN AND operator 96  
BINSTART= option in HISTOGRAM statement 228  
BINWIDTH= option in HISTOGRAM statement 228-229  
BMP image format 246-247  
BODY= option in ODS HTML statement 146-147  
BODYTITLE option in ODS RTF statement 148-149  
BOLD option in TITLE statement 154-155  
bolding in graphics 242-243  
BON option in ANOVA procedure 272  
Bonferroni t tests 272  
BORDER option in INSET option 240  
BOTTOMMARGIN= system option 15  
Bowker's test 264  
box plots 230-231, 272-273  
BOX= option in TABULATE procedure 126-127  
BOXPLOT option in TTEST procedure 258  
BREAK statement in REPORT procedure 136-137  
BY groups, definition 98  
BY statement 94

compared to PANELBY 245  
FIRST.byvariable 198-199  
LAST.byvariable 198-199  
MERGE statement 176-179  
PRINT procedure 102-103  
SET statement 174-175  
SGPLOT procedure 245  
SORT procedure 98-101  
TRANSPOSE procedure 196-197  
UPDATE statement 188-189

## BY variables

definition 98  
FIRST. and LAST. 198-199  
sorting 98-101

## C

calculated variables in SQL procedure 72-73  
CALL SYMPUTX 216-217  
capitalization in SAS programs xiii, 3, 5  
CARDS statement 30  
CAT function 62-63  
CATEGORY= option HBOX or VBOX statement 230-231  
CATS function 62-63  
CATX function 62-63  
CDFPLOT statement in UNIVARIATE procedure 252  
CENTER system option 15  
CHAR option in REPORT procedure 140-141  
character data  
    converting to numeric 306-307  
    definition 4  
    formats 106-107  
    functions 62-63  
    informats 38-39  
    length 314-315  
    sorting 100-101  
    truncation error 314-315  
character-values-converted note 306-307  
charts, bar 226-227

chi-square statistic with FREQ procedure 260-262  
CHISQ option in FREQ procedure 260-262  
CI= option in TTEST procedure 256  
CLASS statement  
    ANOVA procedure 272-275  
    MEANS procedure 112-113  
    STYLE= option in TABULATE procedure 160  
    TABULATE procedure 122-129  
    TTEST procedure 256  
CLASSLEV statement in TABULATE procedure 160  
CLI option in fitted curves 236  
CLM option in fitted curves 236-237  
CLM option in MEANS procedure 254-255  
CLMTRANSPARENCY= option in fitted curves 236  
CLOSE option  
    ODS HTML statement 146-147  
    ODS LISTING statement 152-153  
    ODS PDF statement 150-151  
    ODS RTF statement 148-149  
Cochran-Armitage test 260  
Cochran-Mantel-Haenszel statistics 260  
Cochran's Q test 264  
coded data, custom formats 108-109  
coefficient of variation  
    ANOVA procedure 274-275  
    MEANS procedure 254  
    REG procedure 268  
COLAXIS statement 244  
collating sequence 100  
colon informat modifier 42-43  
color  
    graph attributes 242  
    PRINT procedure 156-157  
    REPORT procedure 158-159  
    style attributes 164-165  
    style templates 145  
    TABULATE procedure 160-161

COLOR style attribute 164-165  
COLOR= option for graph attributes 242  
COLOR= option in TITLE statement 154  
COLUMN location in STYLE= option 158-159  
column pointers  
    @n 40-41, 288-289  
    +n 37  
COLUMN statement in REPORT procedure 130-139  
column-style input 34-35  
columns of data, definition 4  
COLUMNS= option  
    ODS PDF statement 150  
    ODS RTF statement 148-149  
    PANELBY statement 244  
combining SAS data sets  
    concatenating data sets 172-173  
    grand total with original data 184-185  
    interleaving data sets 174-175  
    merging summary statistics 182-183  
    one observation with many 184-185  
    one-to-many match merge 178-183  
    one-to-one match merge 176-177  
    selecting observations during a merge 192-193  
    stacking data sets 172-173  
    updating a master data set 188-189  
    using SQL procedure 180-181  
commas  
    reading comma-delimited data 28-29, 52-53  
    reading numbers containing commas 36, 38-39  
    writing comma-delimited data 280-283  
    writing numbers containing commas 106-107  
COMMAw.d format 106-107  
COMMAw.d informat 38-39  
COMMAXw.d informat 38-39  
Comments 3  
comparison operators 66-67, 96-97  
compile and execute phases 216

COMPRESS function 62-63  
COMPRESS= data set option 317  
COMPUTE statement in REPORT procedure 140-141  
COMPUTED usage option 132, 140-141  
concatenating SAS data sets 172-173  
concatenation  
    function 62-63  
    operator, || 63  
conditional statements  
    macro 212-213  
    standard 66-71  
confidence limits 254-260  
    plotting 236-237  
CONFIDENCE option in CORR procedure 266  
constants  
    ASCII 52  
    character 58  
    date 80-81  
    hexadecimal 52  
    numeric 58  
CONTAINS operator 96  
CONTENTS procedure 22-23  
    debugging programs 313  
    POSITION option 88  
converting character to numeric and vice versa 306-307  
COOKSD option in REG procedure 270  
CORR procedure 264-267  
correlations 264-265  
counts, frequency 116-121, 124-125, 138-139  
CREATE TABLE clause in SQL procedure 72-73, 180-181, 186-187  
CROSSLIST option in FREQ procedure 118  
crosstabulations 118-123  
CSS option in MEANS procedure 254  
CSV destination 282-283  
CSV files  
    reading 28-29, 52-53  
    writing 280-283

CSV value in the DBMS= option  
    EXPORT procedure 280  
    IMPORT procedure 28  
CSVALL destination 282  
CUMFREQPLOT option in FREQ procedure 262  
cumulative distribution function plots 252  
cumulative totals  
    FREQ procedure 116-119  
    sum statement in DATA step 84-85  
CURVELABEL= option  
    fitted curves 236  
    SERIES statement 234  
custom formats, FORMAT procedure 108-109  
CV option in MEANS procedure 254

## D

DATA_NULL_  
    writing custom reports 110-111  
    writing raw data files 288-289  
data dictionary 22-23  
data engines 19  
Data Grid 18  
DATA location in STYLE= option 156-157  
data set options  
    compared to statement options 190-191  
    compared to system options 190-191  
    COMPRESS= 317  
    DROP= 190-191, 317  
    FIRSTOBS= 190-191, 293  
    IN= 190-193  
    KEEP= 190-191, 317  
    OBS= 190-191, 293  
    RENAME= 190-191  
data sets, SAS  
    changing observations to variables 196-197  
    combining a grand total with data 184-185  
    combining one observation with many 184-185

compressing 317  
concatenating 172-173  
contents of 22-23  
creating 20-21  
creating from procedure output 168-169  
definition 4  
deleting 316  
interleaving data sets 174-175  
inverting, TRANSPOSE procedure 196-197  
joining using SQL procedure 180-181  
LABEL= data set option 190-191  
merging summary statistics 182-187  
merging, one-to-many 178-183  
merging, one-to-one 176-177  
modifying a single data set 56-57  
names 5, 20-21  
options 190-195  
permanent 20-21  
printing 102-103  
reading a single data set 56-57  
saving 20-21  
saving summary statistics to 114-115, 168-169  
selecting observations during a merge 192-193  
size 5  
sorting 98-101  
stacking data sets 172-173  
subsetting IF statement 70-71  
subsetting OUTPUT statement 74-75  
subsetting using SQL procedure 72-73  
subsetting WHERE statement 70, 96-97  
subsetting WHERE= data set option 190, 194-195  
temporary versus permanent 20-21  
updating a master data set 188-189  
WORK library 20-21  
writing multiple data sets 74-75

DATA statement 6-7, 20-21  
_NULL_ data set name 110-111, 288-289

- multiple data sets 74-75
- permanent data sets 20-21
- DATA step 6-9, 20-21
  - built-in loop 8-9
  - combining SAS data sets 172-193
  - creating and modifying variables 58-59
  - definition 6
  - reading a single SAS data set 56-57
  - reading raw data files 18-19, 30-53
  - writing raw data files 288-289
  - wrong results, no message 308-309
- data types 4
  - assignment statements 58-59
  - converting, character to numeric 306-307
  - converting, numeric to character 306-307
- data, reading 18-21, 24-53
  - column style 34-35
  - comma-separated values 28-29, 52-53
  - delimited data 28-29, 52-53
  - Excel files 24-27
  - internal 30
  - messy data 42-43
  - methods for getting into SAS 18-19
  - missing data at end of line 51
  - mixing input styles 40-41
  - multiple lines of data per observation 44-45
  - multiple observations per line of data 46-47, 77
  - non-standard format 36-37
  - part of a data file 48-49, 293
  - PC files 24-25
  - skipping lines of raw data 44-45, 50
  - skipping over variables 34-35
  - space-delimited 32-33
  - variable length records 51
  - variable length values 42-43
- data, writing 278-283
  - delimited 280-283, 288-289

methods 278-279  
PC files 284-287  
procedure results 282-283, 286-287  
raw data 280-281, 282-283, 288-289  
DATA= option 94, 190-191  
DATAFILE= option in IMPORT procedure 24-25, 28-29  
DATALABEL= option  
    SCATTER statement 232  
    SERIES statement 234  
    VBAR or HBAR statement 226  
DATALINES statement 30  
DATAROWS= statement in the IMPORT procedure 28-29  
DATASTMCHK= system option 15, 297  
DATE system option 15  
DATEJUL function 64-65, 82-83  
dates 80-83  
    automatic macro variables 206-207  
    constants 80-81  
    converting dates 64-65, 80-81  
    definition of a SAS date 80  
    formats, table of 82-83, 106-107  
    functions, table of 64-65, 82-83  
    informats, table of 38-39, 82-83  
    Julian dates 82-83  
    printing current date on output 15  
    reading raw data with 15, 36-37  
    setting default century 14-15, 80  
DATESTYLE= system option 15  
DATETIMEw. informat 38-39  
DATETIMEw.d format 106-107  
DATEw. format 82-83, 106-107  
DATEw. informat 38-39, 82-83  
DAY function 61, 64-65, 82-83  
DBMS= option  
    EXPORT procedure 280-281, 284-285  
    IMPORT procedure 24-25, 28-29  
DDMMYYw. informat 38-39, 82-83

debugging SAS programs 292-317

- INPUT reached past end of line 298-299, 301
- invalid data 301-303
- invalid option 310-311
- lost card 300-301
- macros 220-221
- missing semicolon 296-297
- missing values were generated 304-305
- option not recognized 310-311
- out of memory or disk space 316-317
- statement not valid 310-311
- truncation of character data 314-315
- values have been converted 306-307
- variable not found 312-313
- variable uninitialized 312-313
- wrong results, no message 308-309

decimal places

- printing data 104-105
- reading data 36-37

DEFINE statement in REPORT procedure 132-135, 140-141

DELETE procedure 316

DELETE statement 70-71

deleting

- data sets 316
- observations 70-73
- variables 190-191

delimited data

- reading 28-29, 32-33, 52-53
- writing 280-283

DELIMITER= option

- FILE statements 288
- INFILE statements 52-53

DELIMITER= statement in IMPORT procedure 28-29

density curves 228-229

DENSITY statement in SGLOT procedure 228-229

DESCENDING option in SORT procedure 98

descriptive statistics 112-139, 250-255

descriptor portion of data sets 5, 22-23  
destinations, output  
    CSV 144  
    EXCEL 144  
    for graphics 225, 246-247  
    HTML 144-147  
    LISTING 144, 152-153, 225  
    OUTPUT 144  
    PDF 144, 150-151  
    POWERPOINT 144  
    PS 144  
    RTF 144-145, 148-149  
    SASREPORT 144  
    WORD 144  
detail report 102-105, 130-131  
DEVIATIONPLOT option in FREQ procedure 262  
DIAGNOSTICS option in REG procedure 270-271  
dictionary, data 22-23  
dimensions in TABULATE procedure 122-125  
dimensions of graph images 246  
DISCRETE option in XAXIS or YAXIS statement 238-239  
DISCRETEOFFSET= option in VBAR or HBAR statement 226  
disk space, running out of 316-317  
Display Manager 10-11  
DISPLAY usage option 132  
dividing  
    data file 48-49  
    SAS data set 74-75, 194-195  
DLM value in the DBMS= option  
    EXPORT procedure 280  
    IMPORT procedure 28  
DLM= option  
    FILE statements 288  
    INFILE statements 52-53  
DLMSTR= option in INFILE statements 52  
DO statement 66-67  
    arrays 86-87

- iterative 78-79
- loops 78-79
  - with OUTPUT statement 76
- DO UNTIL statement 78-79
- DO WHILE statement 78-79
- documenting
  - data sets 5, 22-23, 190-191
  - programs 3
- dollar signs
  - printing data 106-107
  - reading data 36, 38-39
- DOLLARw.d format 106-107
- DOWN= option in KEYLEGEND statement 240
- DROP= data set option 190-191, 317
- DSD option
  - FILE statements 288
  - INFILE statements 52-53
- DTDATEw. format 106-107
- DUNCAN option in ANOVA procedure 272
- Duncan's multiple range test 276
- duplicate observations, eliminating 98-99
- DUPOUT= option in SORT procedure 98

## **E**

- EBCDIC sort order 100
- editor, syntax sensitive 293
- ELLIPSE= option in CORR procedure 266
- ELSE statement 68-69
- END statement 66-67
- ENDCOMP statement in REPORT procedure 140-141
- engines, data 19
  - Excel files 26-27
- entering data in SAS data sets 18
- Enterprise Guide, SAS 10
- EQ comparison operator 66, 96
- equations, assignment statements 58-59
- errors

avoiding errors 292-293  
    fixing errors 294-295  
    INPUT reached past end of line 298-299, 301  
    invalid data 302-303  
    invalid option 310-311  
    lost card 300-301  
    missing semicolon 296-297  
    missing values were generated 304-305  
    option not recognized 310-311  
    out of memory or disk space 316-317  
    statement not valid 310-311  
    truncation of character data 314-315  
    values have been converted 306-307  
    variable not found 312-313  
    variable uninitialized 312-313  
    wrong results, no message 308-309

EURDFDDw. format 82-83, 106-107  
EUROXw.d format 106-107  
Ew. format 106-107  
EXACT option in FREQ procedure 260  
EXCEL destination 144, 286-287  
EXCEL engine 19  
Excel files  
    reading 24-27  
    writing 284-287  
excluding output objects 167  
executing SAS programs  
    methods 10-11  
EXPONENTIAL option in distribution plots 252  
EXPORT procedure  
    delimited files 280-281  
    Excel files 284-285  
exporting data 278-283  
    delimited files 280-283, 288-289  
    methods 278-279  
    PC files 284-287  
    procedure results 282-283, 286-287

raw data files 280-283, 288-289  
to other software 284  
expressions  
    mathematical 58-59  
    using dates 80-81  
    using functions 60-61  
external data 30-31  
EXTREME option in HBOX or VBOX statement 230

## F

F value  
    ANOVA procedure 274-275  
    REG procedure 268

FILE statement  
    DLM= option 288  
    DSD option 288  
    PRINT option 110-111  
    writing raw data files 288-289  
    writing reports 110-111

FILE= option  
    ODS HTML statement 146-147  
    ODS LISTING statement 152-153  
    ODS PDF statement 150-151  
    ODS RTF statement 148-149

FILLATTRS= option for graph attributes 242

FIRST.byvariable 198-199

FIRSTOBS= option  
    data set option 190-191, 293  
    INFILE statement 50, 293

Fisher's exact test 264

fit plots 270-271

FITPLOT option in REG procedure 270-271

fitted curves 236-237

flat files 18, 30

FLYOVER style attribute 164-165

font

    graph attributes 242-243

style attributes 164-165  
titles 154-155

FONT_FACE style attribute 164-165  
FONT_SIZE style attribute 164-165  
FONT_STYLE style attribute 164-165  
FONT_WEIGHT style attribute 164-165  
FONT= option in TITLE statement 154-155  
FONTFAMILY style attribute 164-165  
FONTSIZE style attribute 164-165  
FONTSTYLE style attribute 164-165  
FONTWEIGHT style attribute 164-165  
FOOTNOTE statement 94-95, 154-155  
FOREGROUND style attribute 164-165  
FORMAT procedure 108-109  
    grouping with 120-121  
    with SGLOT procedure 227  
    with TABULATE procedure 128-129

FORMAT statement 104-105  
    DATA step compared to PROC step 104

FORMAT= option in TABULATE procedure 126-127

formats

- ATTRIB statement 314
- dates 81-83
- FORMAT statement 104-105
  - grouping with 120-121
  - input formats 36-39
  - table of 106-107
  - use 104-105
  - user-defined 108-109, 120-121
- formatted style input 36-37
- free formatted style input 32-33

FREQ procedure 116-119, 121, 260-263  
FREQPLOT option in FREQ procedure 262-263  
frequency tables 116-125, 138-139, 260-261  
FROM clause in SQL procedure 72-73, 180-181, 186-187  
functions

- dates 80-83

INPUT function 307  
PUT function 307  
table of 62-65, 82-83  
use 60-61

## G

gamma 260  
GAMMA option in distribution plots 252  
GE comparison operator 66, 96  
generating data  
    DO and OUTPUT statements 78-79  
GETNAMES= statement 24, 28-29  
GIF image format 246  
global macro variables 202-203  
global statements  
    OPTIONS 14-15  
    RUN 6  
    TITLE and FOOTNOTE 94  
GPATH= option in ODS statement 247  
GRANDTOTAL location in STYLE= option 156  
graphics, ODS 224-225  
    ANOVA procedure 272-273  
    CORR procedure 266-267  
    FREQ procedure 262-263  
    image formats 246  
    image properties 246-247  
    insets 240-241  
    legends 240-241  
    REG procedure 270-271  
    saving graphs 246-247  
    SGPANEL procedure 244-245  
    SGPLOT procedure 226-237  
    style attributes 242-243  
    TTEST procedure 258-259  
    UNIVARIATE procedure 252-253  
grayscale style for graphics 225  
GRID option in XAXIS or YAXIS statement 238

GROUP BY clause in SQL procedure 186-187  
GROUP usage option 132, 134-135  
GROUP= option  
    DENSITY statement 228  
    fitted curves 236  
    HBOX or VBOX statement 230  
    HISTOGRAM statement 228  
    SCATTER statement 232-233  
    SERIES statement 234  
    VBAR statement 226-227  
GROUPDISPLAY= option in VBAR or HBAR statement 226-227  
GROUPTHORIZONTAL option in FREQ procedure 262-263  
grouping observations  
    BY statement 94  
    FREQ procedure 116-119, 121  
    IF-THEN/ELSE statements 68-69  
    MEANS procedure 112-113  
    PUT function 120-121  
    REPORT procedure 132-135  
    SORT procedure 98-101  
    TABULATE procedure 122-123  
    trafficlighting 162-163  
    user-defined format 120-121, 162-163  
GT comparison operator 66, 96  
GUESSINGROWS= statement in the IMPORT procedure 28-29

## H

H0= option in TTEST procedure 256  
HAVING clause in SQL procedure 218-219  
HBAR statement in SGLOT procedure 226  
HBOX statement in SGLOT procedure 230  
HEADER location in STYLE= option 156-158  
headers  
    changing in TABULATE output 128-129  
    reading raw data 50  
    specifying style for 156-161  
HEIGHT= option

ODS GRAPHICS statement 246-247  
TITLE statement 154-155  
hexadecimal constants 52  
HIGH keyword in FORMAT procedure 108-109  
HISTOGRAM option  
    CORR procedure 266  
    TTEST procedure 258  
HISTOGRAM statement  
    SGPLOT procedure 228-229  
    UNIVARIATE procedure 252-253  
histograms 228-229, 252-253, 258, 266  
HOEFFDING option in CORR procedure 264  
HTML output 144-147  
hypertext links, style attribute 164-165  
HyperText Markup Language 144-147

## I

ID statement  
    PRINT procedure 102  
    STYLE= option in PRINT procedure 156-157  
    TRANSPOSE procedure 196-197  
IF statement, subsetting 70-71  
IF-THEN statements 66-67  
IF-THEN/ELSE statements 68-69  
IMAGE_DPI= option in ODS LISTING statement 247  
IMAGENAME= option in ODS GRAPHICS statement 246-247  
images  
    inserting in output 164-165  
    saving 246-247  
IMPORT procedure 19  
    delimited files 28-29  
    PC files 24-25  
    variable names 90-91  
    WHERE= data set option 194-195  
importing data  
    delimited 28-29  
    Excel files 26-27

from other software 24-25  
methods 18-19  
PC files 24-25

IN operator 96  
in-stream data 30  
IN= data set option 190-193  
indention in SAS programs 3  
INDEX function 62-63  
INFILE statement 30-31  
    DELIMITER= option 52-53  
    DLM= option 52-53  
    DSD option 52-53  
    examples by operating environment 30-31  
    FIRSTOBS= option 50, 293  
    MISSOVER option 51, 299  
    OBS= option 50, 293  
    TRUNCOVER option 51, 299

informats

- ATTRIB statement 314
- colon modifier 42-43
- dates 80-83
- invalid data 302-303
- table of 38-39
- use 36-37

INNER JOIN clause in SQL procedure 180-181

input formats 36-41

INPUT function 307

INPUT reached past end of line

- message in log 298-299, 301

INPUT statement

- column style 34-35
- data with embedded blanks 34-35
- delimited data 52-53
- formatted style 36-39
- free formatted 32-33
- list style 32-33
- mixing input styles 40-41

- multiple INPUT statements 48-49, 77
- multiple lines per observation 44-45
- multiple observations per line 46-47
- reading blanks as missing 34-35
- reading non-standard data 36-37
- reading part of a raw data file 48-49
- skipping lines of raw data 44-45
- skipping over variables 34-35
- space-delimited 32-33

INSET statement in SGLOT procedure 240-241

INT function 60, 64-65

integer data

- data types 4
- truncating decimal places 64-65

interleaving SAS data sets 174-175

internal data 30

internet browser, creating files for 146-147

INTERVALPLOT option in TTEST procedure 258

INTO clause in SQL procedure 218-219

invalid data message in log 302-303

- lost card note 301

invalid option message in log 310-311

inverting data sets 196-197

IS NOT MISSING operator 96

ITALIC option in TITLE statement 154-155

italics, explanation of usage xiii

iterative logic 78-79, 86-87

- in macros 214-215

## J

JITTER option in SCATTER statement 232

JMP files

- reading 19
- writing 278

joining SAS data sets 176-181

JPEG image format 246

Julian dates 82-83

JULIANw. format 82-83, 106-107  
JULIANw. informat 38-39, 82-83  
JUST style attribute 159, 161, 164-165  
justification  
    character variables 62-63  
    output 15  
    style attributes 159, 161, 164-165  
    titles and footnotes 154-155  
JUSTIFY= option in TITLE statement 154-155

## K

kappa statistics 260  
KAPPAPLOT option in FREQ procedure 262  
KEEP= data set option 190-191, 317  
KENDALL option in CORR procedure 264  
Kendall's tau-b 264, 268  
kernel density plot 228-229  
KERNEL option in DENSITY statement 228-229  
KEYLEGEND statement in SGPlot procedure 240-241  
kurtosis  
    MEANS procedure 254  
    UNIVARIATE procedure 250-251  
KURTOSIS option in MEANS procedure 254

## L

LABEL option in PRINT procedure 102  
LABEL statement 95  
    SGPLOT procedure 227  
    TABULATE procedure 128  
LABEL= option  
    data set option 190-191  
    REFLINE statement 238-239  
    XAXIS or YAXIS statement 238-239  
LABELATTRS= option for graph attributes 242-243  
labels  
    ATTRIB statement 314  
    data set 22-23, 190-191

value 108-109  
variable 22-23, 95  
lambda 260  
LAST.byvariable 198-199  
LCLM option in MEANS procedure 254  
LE comparison operator 66  
LEFT function 62-63  
LEFTMARGIN= system option 15  
legends for graphs 240-241  
length of a variable 23, 68-69, 314-317  
LENGTH statement  
    character data 68-69, 314-317  
    numeric data 316-317  
LENGTH= option in REPORT procedure 140-141  
LIBNAME engines 19  
    Excel files 26-27  
LIBNAME statement 19, 21  
    Excel files 26-27  
library, SAS data 5, 20-21  
librefs 20-21  
LIKE comparison operator 96-97  
LIMITSTAT= option in VBAR or HBAR statement 226  
line plots 234-235  
line pointers  
    / 44-45, 288  
    #n 44-45, 288  
line-hold specifiers  
    @ compared to @@ 49  
    @, trailing 48-49, 288  
    @@, double trailing 46-47  
line, graph attributes 242-243  
LINEAR option in XAXIS or YAXIS statement 238  
LINEATTRS= option for graph attributes 242-243  
LINESTABLE option in MEANS statement 275  
LINGUISTIC sort option 100-101  
links, style attributes for hypertext 164-165  
Linux

direct referencing of SAS data sets 20-21  
INFILE statement 31  
LIBNAME statement 20-21  
LIST option in FREQ procedure 118  
list style input 32-33  
LISTING output 144, 152-153, 225, 246  
lists, variable names 88-89  
local macro variables 202-203  
LOCATION= option in KEYLEGEND statement 240-241  
locations in STYLE= option 156-159  
loess curves 236-237  
LOESS statement in SGPlot procedure 236-237  
LOG function 60, 64-65  
LOG option in XAXIS or YAXIS statement 238  
log, SAS 12-13  
    errors, warnings, and notes 294-295  
    writing in with PUT statements 308-309  
LOG10 function 64-65  
logarithmic functions 64-65  
logical operators 66-67, 96-97  
LOGNORMAL option in distribution plots 252  
loop  
    DATA step, built-in 8-9  
    DO loop 78-79, 86-87  
lost card note in log 300-301  
LOW keyword in FORMAT procedure 108  
LT comparison operator 66, 96

## M

MACRO system option 203  
macro variables, creating  
    using %LET 204-205  
    using CALL SYMPUTX 216-217  
    using SQL procedure 218-219  
macros 202-221  
    &SYSDATE macro variable 206-207  
    &SYSDAY macro variable 206

&SYSNOBS macro variable 206  
%DO loops 214-215  
%DO statements 212-213  
%ELSE statement 212-213  
%END statement 212-215  
%IF-%THEN statements 212-213  
%LET statement 204-207  
%MACRO statement 208-211  
%MEND statement 208-209  
%THEN statement 212-213  
autocall libraries 209  
automatic macro variables 206-207  
CALL SYMPUTX 216-217  
concepts 202-203  
debugging errors 220-221  
invoking 208  
iterative %DO loop 214-215  
local versus global variables 202-203  
MACRO system option 203  
macro variables, definition 202  
macro variables, listing 220  
MERROR system option 220-221  
MLOGIC system option 220-221  
MPRINT system option 220-221  
parameters 210-211  
quotation marks 203, 220  
SAS macro processor 202-203  
SError system option 220-221  
SYMBOLGEN system option 220-221  
MARKERATTRS= option for graph attributes 242-243  
MARKERS option in SERIES statement 234-235  
master data set definition 188  
match merging  
  IN= data set option 190-193  
  one-to-many match merge 178-183  
  one-to-one match merge 176-177, 180-181  
  summary statistics 182-185

mathematical expressions 58-59  
MATRIX option in CORR procedure 266-267  
MAX function 64-65  
MAX keyword  
    REPORT procedure 138  
    TABULATE procedure 124  
MAX option in MEANS procedure 112  
MAXDEC= option in MEANS procedure 112-113  
maximum value  
    across observations 64-65  
    across variables 84-85, 112-113, 198-199  
    FIRST. and LAST. byvariable 198-199  
    HBOX or VBOX statement 230-231  
    MAX function 64-65  
    MEANS procedure 112-113  
    REPORT procedure 138  
    RETAIN statement 84-85  
    TABULATE procedure 124  
    UNIVARIATE procedure 251  
McNemar's test 264  
MDY function 60, 64-65, 82-83  
MEAN function 61, 64-65  
    missing data 305  
MEAN keyword  
    REPORT procedure 138-139  
    TABULATE procedure 124-125  
MEAN option in MEANS procedure 112  
mean square  
    ANOVA procedure 274-275  
    REG procedure 268  
means  
    HBOX or VBOX statement 230-231  
    MEAN function 61, 64-65  
    MEANS procedure 112-113  
    multiple comparisons 272-275  
    pairwise comparisons 256-259  
    REPORT procedure 138-139

TABULATE procedure 124-125  
testing 256-259  
UNIVARIATE procedure 251  
MEANS procedure 112-115, 182-185, 254-255  
MEANS statement, ANOVA procedure 272-275  
MEASURES option in FREQ procedure 260, 262  
median  
    HBOX or VBOX statement 230-231  
    MEANS procedure 112, 254-255  
    REPORT procedure 138  
    TABULATE procedure 124  
    UNIVARIATE procedure 250-251  
MEDIAN option in MEANS procedure 112, 254-255  
member, SAS data set 5  
memory, running out 316-317  
MERGE statement 176-183  
    BY statement 176-183  
    IN= data set option 190-193  
    one-to-many match merge 178-183  
    one-to-one match merge 176-177  
    summary statistics 182-183  
MERROR system option 220-221  
messy raw data, reading 42-43  
Microsoft Excel files  
    reading 24-27  
    writing 284-287  
MIN function 64-65  
MIN keyword  
    REPORT procedure 138  
    TABULATE procedure 124  
MIN option in MEANS procedure 112  
minimum value  
    across observations 64-65  
    across variables 84-85, 112-113, 198-199  
    FIRST. and LAST. byvariable 198-199  
    HBOX or VBOX statement 230-231  
    MEANS procedure 112-113

MIN function 64-65  
REPORT procedure 138  
RETAIN statement 84-85  
TABULATE procedure 124  
UNIVARIATE procedure 251  
missing data values 5  
    assignment statements 59, 304-305  
    end of raw data line 51  
    finding number 112-113, 116-119  
    IF-THEN statements 68-69  
    match merge 177  
    MEANS procedure 112  
    reading blanks as 34-35  
    REPORT procedure 132-133  
    SET statement 173  
    SORT procedure 98-99  
    TABULATE statement 122  
    UPDATE statement 188-189  
MISSING option  
    FREQ procedure 116, 118-119  
    HBOX or VBOX statement 230  
    MEANS procedure 112  
    PANELBY statement 244  
    REPORT procedure 132-133  
    TABULATE procedure 122  
    VAR or HBAR statement 226  
missing semicolon 296-297  
missing values generated note 59, 304-305  
MISSING= system option 15, 282-283, 286-287  
MISSOVER option in INFILE statements 51, 299  
MISSPRINT option in FREQ procedure 116  
MISSTEXT= option in TABULATE procedure 126-127  
MIXED= statement in the IMPORT procedure 25  
MLOGIC system option 220-221  
MMDDYYw. format 82-83, 106-107  
MMDDYYw. informat 38-39, 82-83  
mode of a variable

- MEANS procedure 112, 254
- REPORT procedure 138
- TABULATE procedure 124
- UNIVARIATE procedure 250-251
- MODEL statement
  - ANOVA procedure 272-274
  - REG procedure 268-269
- modes of running SAS 10-11
- modifying SAS data sets
  - joining using SQL procedure 180-181
  - MERGE statement 176-183
  - SET statement 56-57, 172-173, 184-185
  - UPDATE statement 188-189
- MONTH function 64-65, 82-83
- MPRINT system option 220-221
- multiple comparisons 272-275
- multiple lines per observation, reading 44-45
- multiple observations per line, reading 46-47

## N

- N function 64-65
- N keyword
  - REPORT procedure 138-139
  - TABULATE procedure 124
- N option in MEANS procedure 112, 254-255
- name literals 90-91
- names for
  - data sets 5, 20
  - formats 108
  - librefs 20-21
  - macro variables 204
  - macros 208
  - variable lists 88-89
  - variables 5, 90-91
- NBINS= option in HISTOGRAM statement 228
- NE comparison operator 66, 96
- NMISS function 64-65

NMISS keyword  
    REPORT procedure 138  
    TABULATE procedure 124

NMISS option in MEANS procedure 112

NOBORDER option in KEYLEGEND statement 240

NOBYVAR option in TTEST procedure 256

NOCENTER system option 15

NOCOL option in FREQ procedure 118-119, 121

NOCUM option in FREQ procedure 116

NODATE system option 15

NODUPKEY option in SORT procedure 98-99

NOHEADERBORDER option in PANELBY statement 244

NOLEGCLI option in fitted curves 236

NOLEGCLM option in fitted curves 236-237

NOLEGFIT option in fitted curves 236

NOMARKERS option in fitted curves 236-237

NOMISSINGGROUP option  
    SCATTER statement 232  
    SERIES statement 234

NONE option  
    CORR procedure 266  
    TTEST procedure 258

NONUMBER system option 15

NOOBS option in PRINT procedure 102

NOPERCENT option in FREQ procedure 116, 118-119, 121

NOPRINT option  
    FREQ procedure 116  
    MEANS procedure 114-115

normal density plot 228-229

NORMAL option  
    CDFPLOT statement 252  
    DENSITY statement 228-229  
    HISTOGRAM statement 252-253  
    UNIVARIATE procedure 250

normality test 250

NOROW option in FREQ procedure 118-119, 121

notes in SAS log 12-13, 294-295

INPUT reached past end line 298-299, 301  
invalid data 302-303  
lost card 300-301  
missing values were generated 59, 304-305  
values have been converted 306-307  
variable uninitialized 312-313  
NOVARNAME option in PANELBY statement 244-245  
NUMBER system option 15  
numbering observations, _N_ variable 198-199  
numeric data  
    commas, reading 36, 38-39  
    commas, writing 106-107  
    converting to character 306-307  
    definition 4  
    formats 106-107  
    functions 64-65  
    informats 38-39  
    length 316-317  
    reading non-standard 36-37  
    reading standard 32-35  
numeric values converted note 306-307  
NUMERIC_COLLATION= suboption in SORT procedure 100-101

## O

OBS location in STYLE= option 156  
OBS= option  
    data set option 190-191, 293  
    INFILE statements 50, 293  
observations  
    changing to variables 196-197  
    combining single observation with many 184-185  
    creating a numbering variable 198-199  
    definition 4  
    deleting in DATA step 70-71  
    deleting using SQL procedure 72-73  
    duplicate, eliminating 98-99  
    grouping in procedures 120-121

grouping with IF-THEN/ELSE 68-69  
interleaving 174-175  
joining using SQL procedure 180-181  
making several from one 76-79  
merging 176-179  
printing 102-103  
reading multiple lines per observation 44-45  
reading multiple observations per line 46-47  
sorting 98-101  
subsetting DELETE statements 70-71  
subsetting FIRSTOBS= option 190-191  
subsetting IF statement 70-71  
subsetting IN= data set option 190-193  
subsetting OBS= option 190-191  
subsetting OUTPUT statements 74-75  
subsetting using SQL procedure 72-73  
subsetting WHERE statements 70, 96-97  
subsetting WHERE= data set option 190, 194-195  
tracking with IN= data set option 190-193  
updating 188-189  
OBSERVEDBYPREDICTED option in REG procedure 270  
OBSHEADER location in STYLE= option 156  
odds ratios 260-261  
ODDSRATIOPLOT option in FREQ procedure 262  
ODS 144-165, 224-247  
ODS CSV statement 282-283  
ODS EXCEL statement 286-287  
ODS EXCLUDE statement 167  
ODS Graphics 145, 224-225

- ANOVA procedure 272-273
- CORR procedure 266-267
- FREQ procedure 262-263
- image properties 246-247
- insets 240-241
- legends 240-241
- REG procedure 270-271
- saving graphs 246-247

SGPANEL procedure 244-245  
SGPLOT procedure 226-237  
style attributes 242-243  
TTEST procedure 258-259  
UNIVARIATE procedure 252-253  
ODS GRAPHICS statement 224, 246-247  
ODS HTML statement 146-147  
ODS LISTING statement 152-153, 225, 246  
ODS NOPROCTITLE statement 146-151  
ODS OUTPUT statement 168-169  
ODS PDF statement 150-151  
ODS RTF statement 148-149  
ODS SELECT statement 167  
ODS TRACE statement 166-167  
ON clause in SQL procedure 180-181  
OnDemand for Academics 11  
one-to-many match merge 178-183  
one-to-one match merge 176-177  
one-way frequency table 116-117  
operators  
    arithmetic 58-59  
    comparison 66-67, 96-97  
    logical 66-67, 96-97  
option not recognized error in log 310-311  
options  
    comparison of types of options 190-191  
    data set 190-195  
    system 14-15  
OPTIONS procedure 14  
    OPTION= option 203  
OPTIONS statement 14-15  
    macro debugging options 220-221  
OR operator 66-67, 96  
ORDER usage option 132-133  
ordering observations 98-101  
ORIENTATION= system option 15  
OS X operating environment 11

OTHER keyword FORMAT procedure 108  
out of disk space message 316-317  
out of memory message 316-317  
OUT= option  
    FREQ procedure 116  
    IMPORT procedure 24-25, 28-29  
    MEANS procedure 114-115  
    SORT procedure 98-99  
OUTFILE= option in EXPORT procedure 280-281, 284-285  
outliers 230-231, 250-251  
output  
    centering 15  
    changing appearance of data in 81-83, 104-109  
    creating SAS data sets from 168-169  
    customizing with STYLE= option 144-151  
    footnotes 94-95, 154-155  
    graphics 224-225  
    HTML 146-147, 225  
    labels 95  
    LISTING 152-153, 225  
    PDF 150-151, 225  
    RTF 148-149, 225  
    text 152-153  
    titles 94-95, 154-155  
    titles, removing 94-95, 146-151  
Output Delivery System 144-165, 224-247  
OUTPUT destination 144, 168-169  
output object 145, 166-169  
OUTPUT statement 9  
    DATA step 74-77  
    DO statement 76, 78-79  
    MEANS procedure 114-115  
    multiple observations from one 76-79  
    writing multiple data sets 74-75  
OUTPUTFMT= option in ODS GRAPHICS statement 246-247

## P

P1 option in MEANS procedure 254  
P5 option in MEANS procedure 254  
P10 option in MEANS procedure 254  
P25 option in MEANS procedure 254  
P50 option in MEANS procedure 254  
P75 option in MEANS procedure 254  
P90 option in MEANS procedure 254  
P95 option in MEANS procedure 254  
P99 option in MEANS procedure 254  
page breaks  
    ODS output 148-151  
    PUT statement 110-111  
PAGE option in REPORT procedure 136  
PAGENO= system option 15  
PAIRED statement in TTEST procedure 256-257  
pairwise t test 256-259  
PANELBY statement 244-245  
parameter estimates 269  
parameters, macro 210-211  
PATH= option  
    ODS HTML statement 146-147  
    ODS RTF statement 148-149  
PATTERN= option for graph attributes 242  
PBSPLINE statement in SGLOT procedure 236  
PC files  
    reading 24, 26-27  
    writing 284-287  
PC Files Server 284  
PCTN keyword  
    REPORT procedure 138  
    TABULATE procedure 124  
PCTSUM keyword  
    REPORT procedure 138  
    TABULATE procedure 124  
PDF output 144, 150-151  
PEARL style template 145, 150-151  
Pearson coefficient 260, 264-265

percentages

- calculating in DATA step 182-183
- FREQ procedure 116-119
- REPORT procedure 138
- TABULATE procedure 124

percentiles

- HBOX or VBOX statement 230-231
  - MEANS procedure 254
  - REPORT procedure 138
  - UNIVARIATE procedure 251
- PERCENTw. informat 38-39
- PERCENTw.d format 106-107
- permanent SAS data sets 20-21
- plots 224-253, 258-259, 262-263
- PLOTS= option
  - CORR procedure 266-267
  - FREQ procedure 262-263
  - REG procedure 270-271
  - TTEST procedure 258-259

PNG image format 246

pointers

- @n column pointer 40-41, 288-289
  - / line pointer 44-45, 110-111, 288
  - #n line pointer 44-45, 288
  - +n column pointer 36-37
  - INPUT statements 36-37, 40-45
  - PUT statements 110-111, 288-289
- POSITION option in CONTENTS procedure 88
- POSITION= option in INSET or KEYLEGEND statement 240-241
- POSTIMAGE style attribute 164-165
- PostScript output 144, 246
- POSTTEXT style attribute 164-165
- POWERPOINT destination 144
- PPLOT statement in UNIVARIATE procedure 252
- precedence, mathematical rules 58
- predicted values in regression 268-269
- PREIMAGE style attribute 164-165

PRETEXT style attribute 164-165  
PRIMARY option in SORT procedure 100-101  
print formats 104-109  
    user-defined 108-109  
PRINT option in FILE statements 110-111  
PRINT procedure 102-103  
    STYLE= option 156-157  
printed values, changing appearance 104-105  
probability plot 252-253  
probability-probability plot 252  
PROBPLOT statement in UNIVARIATE procedure 252-253  
PROBT option in MEANS procedure 254  
PROC ANOVA 272-275  
PROC CONTENTS 22-23  
    for debugging programs 313  
    POSITION option 88  
PROC CORR 264-267  
PROC DELETE 316  
PROC EXPORT  
    delimited files 280-281  
    PC files 284-285  
PROC FORMAT 108-109  
    with SGPlot procedure 227  
    with TABULATE procedure 128-129  
PROC FREQ 116-119, 121, 260-263  
PROC IMPORT 19  
    delimited files 28-29  
    PC files 24-25  
    variable names 90-91  
    WHERE= data set option 194-195  
PROC MEANS 112-115, 182-183, 254-255  
PROC OPTIONS 14, 203  
PROC PRINT 102-103  
    STYLE= option 156-157  
PROC REG 268-271  
PROC REPORT 130-141  
    STYLE= option 158-159

PROC SGANEL 224, 244-245  
PROC SGPLOT 224, 226-243  
PROC SORT 98-101  
PROC SQL  
    calculated variables 72-73  
    combining SAS data sets 180-181  
    GROUP BY clause 186-187  
    HAVING clause 218-219  
    INNER JOIN clause 180-181  
    INTO clause 218-219  
    ON clause 180-181  
    SELECT clause 72-73, 186-187  
    subsetting data using 72-73  
    summarizing variables 186-187, 218-219  
    WHERE clause 72-73, 180-181  
PROC statement 6-7, 94  
    DATA= option 94  
PROC step  
    common statements and options 94-95  
    definition 6-7  
PROC SUMMARY 115  
PROC TABULATE 122-129  
    CLASSLEV statement 160  
    STYLE= option 160-161  
PROC TRANSPOSE 196-197  
PROC TTEST 256-259  
PROC UNIVARIATE 250-253  
procedures  
    common statements and options 94-95  
    definition 6-7  
    title, removing 146-153  
PROFILESPLOT option in TTEST procedure 258  
PROPCASE function 62-63  
PS destination 144-145  
PS image format 246  
PUT function 120-121, 307  
PUT statement

_ALL_ variable name list 308-309  
_PAGE_ keyword 110-111  
debugging with 308-309  
writing a raw data file 288-289  
writing in SAS log 308-309  
writing reports 110-111  
PUTLOG statement 308-309

## **Q**

Q1 option in MEANS procedure 254  
Q3 option in MEANS procedure 254  
QQPLOT option  
    REG procedure 270  
    TTEST procedure 258-259  
QQPLOT statement in UNIVARIATE procedure 252  
QTR function 64-65, 82-83  
quantile-quantile plot 252, 258-259  
quantiles 230-231, 250-254  
question mark informat modifier, double 303  
QUIT statement 7, 72-73  
quotation marks  
    FOOTNOTE statements 94-95  
    in macros 203-204  
    reading delimited data with 28-29, 52-53  
    TITLE statements 94-95

## **R**

R-square  
    ANOVA procedure 274-275  
    REG procedure 268-269  
RAND function 64-65  
random numbers, generating 64-65  
RANGE option in MEANS procedure 112, 254  
RANGE= statement in the IMPORT procedure 24  
RBREAK statement in REPORT procedure 136-137  
reading data 18-21, 28-53  
    column style 34-35

comma-separated values 28-29, 52-53  
delimited data 28-29, 52-53  
Excel files 26-27  
internal 30  
messy data 42-43  
methods for getting into SAS 18-19  
missing data at end of line 51  
mixing input styles 40-41  
multiple lines of data per observation 44-45  
multiple observations per line of data 46-47, 77  
non-standard format 36-37  
part of a data file 48-49, 293  
PC files 24, 26-27  
skipping lines of raw data 44-45, 50  
skipping over variables 34-35  
space-delimited 32-33  
variable length records 51  
variable length values 42-43  
reading SAS data sets 20-21  
    a single data set 56-57  
    concatenating data sets 172-173  
    interleaving data sets 174-175  
    merging summary statistics 182-185  
    one-to-many match merge 178-183  
    one-to-one match merge 176-177, 180-181  
    stacking data sets 172-173  
    updating a master data set 188-189  
record length of raw data files 31  
REFLINE statement 238-239  
REG procedure 268-271  
REG statement in SGLOT procedure 236-237  
regression 268-269  
    lines, plotting 236-237  
relative risk measures 260  
RELRISK option in FREQ procedure 260, 262  
RELRISKPLOT option in FREQ procedure 262  
RENAME= data set option 190-191

REPLACE option  
  EXPORT procedure 280-281, 284-285  
  IMPORT procedure 24-25, 28-29

REPORT procedure 130-141  
  STYLE= option 158-159

reports  
  controlling style of 144-145, 156-165  
  PRINT procedure 102-109  
  REPORT procedure 130-141  
  TABULATE procedure 122-129  
  writing custom 110-111

RESET option in ODS GRAPHICS statement 246-247

RESIDUALHISTOGRAM option in REG procedure 270

RESIDUALS option in REG procedure 270

RESPONSE= option in VBAR statement 226

results  
  centering 15  
  changing appearance of data in 81-83, 104-109  
  creating SAS data sets from 168-169  
  customizing with STYLE= option 144-151  
  footnotes 94-95, 154-155  
  graphics 224-225  
  HTML 146-147, 225  
  labels 95  
  LISTING 152-153, 225  
  PDF 150-151, 225  
  RTF 148-149, 225  
  titles 94-95, 154-155  
    titles, removing 94-95, 146-151

RETAIN statement 9, 84-85

RFPLOT option in REG procedure 270

RIGHTMARGIN= system option 15

risk ratios 260

RISKDIFFPLOT option in FREQ procedure 262

ROUND function 64-65

ROW=FLOAT option in TABULATE procedure 128-129

ROWAXIS statement 244

rows of data, definition 4  
ROWS= option in PANELBY statement 244  
RSTUDENTBYLEVERAGE option in REG procedure 270  
RSTUDENTBYPREDICTED option in REG procedure 270  
RTF  
    output 144-145, 148-149  
    style template 145, 148-149  
RULE, invalid data message 302-303  
RUN statement 6  
    CALL SYMPUTX 216-217  
running SAS programs, methods 10-11

## S

SAS automatic variables  
    _ERROR_ 198  
    _N_ 198-199  
FIRST.byvariable 198-199  
LAST.byvariable 198-199  
macro 206-207  
SAS data library 20-21  
SAS data sets  
    accessing 18-19  
    changing observations to variables 196-197  
    combining a grand total with data 184-185  
    combining one observation with many 184-185  
    compressing 317  
    concatenating 172-173  
    contents of 22-23  
    creating 20-21  
    creating from procedure output 168-169  
    definition 4  
    deleting 316  
    descriptor 5, 22-23  
    interleaving data sets 174-175  
    inverting, TRANSPOSE procedure 196-197  
    joining using SQL procedure 180-181  
    LABEL= data set option 190-191

library names 5  
member names 5  
merging summary statistics 182-187  
merging, one-to-many 178-183  
merging, one-to-one 176-177  
modifying a single data set 56-57  
names 5, 20-21  
options 190-195  
permanent 20-21  
printing 102-103  
reading a single data set 56-57  
SASHELP library 20, 22  
SASUSER library 20  
saving 20-21  
saving summary statistics to 114-115, 168-169  
selecting observations during a merge 192-193  
size 5  
sorting 98-101  
stacking data sets 172-173  
subsetting IF statement 70-71  
subsetting OUTPUT statement 74-75  
subsetting using SQL procedure 72-73  
subsetting WHERE statement 70, 96-97  
subsetting WHERE= data set option 190, 194-195  
temporary 20  
temporary versus permanent 20-21  
updating a master data set 188-189  
WORK library 20  
writing multiple data sets 74-75

SAS dates 80-83  
automatic macro variables 206-207  
constants 80-81  
converting dates 64-65, 80-83  
definition of a SAS date 80  
formats, table of 82-83, 106-107  
functions, table of 64-65, 82-83  
informats, table of 38-39, 82-83

- Julian dates 82-83
  - printing current date on output 15
  - reading raw data with 15, 36-37
  - setting default century 13-14, 80
- SAS Enterprise Guide 10, 18
- SAS functions
  - dates 80-83
  - INPUT function 307
  - PUT function 307
  - table 62-65
  - use 60-61
- SAS Institute x
- SAS language rules 2-3
- SAS listing 144, 152-153, 225, 246
- SAS log 12-13
  - errors, warnings, and notes 294-295
  - writing in log with PUT statements 308-309
- SAS macro processor 202-203
- SAS names, rules for 5
- SAS OnDemand for Academics 10-11
- SAS programs
  - capitalization xiii, 3, 5
  - comments 3
  - data driven 216-219
  - debugging 292-317
  - definition 2
  - documenting 3
  - finding missing semicolons 296-297
  - fixing 294-295
  - indentation xiii, 3
  - major parts 6-7
  - submitting 10-11
  - testing 292-293, 295
- SAS Studio 10-11
- SAS University Edition 10-11
- SAS windowing environment 11
- SAS, modes of running 10-11

SAS/ACCESS 19, 24-25  
SASDATE option in ODS RTF statement 148  
SASHELP library 20, 22  
SASREPORT destination 144  
SASUSER library 20  
saving images 246-247  
saving SAS data sets 20-21  
Scalable Vector Graphics image format 246  
SCALE = option in HISTOGRAM statement 228-229  
SCATTER option in CORR procedure 266-267  
scatter plots 232-233, 266-267  
SCATTER statement in SGPlot procedure 232-233  
SCHEFFE option in ANOVA procedure 272-275  
Scheffe's multiple-comparisons 276-279  
scientific notation  
    format for writing 106-107  
    reading data with 34, 36  
SELECT clause in SQL procedure 72-73, 180-181, 186-187  
selecting observations  
    DELETE statements 70-71  
    IF statements 70-71  
    IN= data set option 192-193  
    INPUT statements 48-49  
    OUTPUT statement 74-75  
    reading raw data file 48-49  
    saving memory and disk space 317  
    SQL procedure 72-73  
    WHERE statement 70, 96-97  
    WHERE= data set option 194-195  
selecting output objects 167  
semicolon 2  
    missing 296-297  
sequential files 18, 30  
series plots 234-235  
SERIES statement in SGPlot procedure 234-235  
SError system option 220-221  
SET statement

BY statement 174-175  
combining grand total with data 184-185  
combining one observation with many 184-185  
concatenating data sets 172-173  
interleaving data sets 174-175  
modifying single data set 56-57  
multiple SET statements 184-185  
reading single data set 56-57  
stacking data sets 172-173

SGPANEL procedure 224, 244-245  
SGPLOT procedure 224, 226-243  
sharing data with other software 18-19  
SHEET= statement in the IMPORT procedure 24  
SHOWBINS option in HISTOGRAM statement 228-229  
SIDES= option in TTEST procedure 256  
size  
    data sets 5, 22-23  
    footnotes 154-155  
    graphics images 246  
    titles 154-155  
    variables 5, 22-23, 68-69, 314-317  
SIZE= option for graph attributes 242-243  
skewness  
    MEANS procedure 254  
    UNIVARIATE procedure 251  
SKEWNESS option in MEANS procedure 254  
skipping over variables at input 34-35  
Somer's D 264  
SORT procedure 98-101  
SORTSEQ= option in SORT procedure 100-101  
Soundex comparisons 96-97  
space-delimited raw data  
    reading 28-29  
    writing 280-281  
SPACING= option in PANELBY statement 244-245  
SPANROWS option in REPORT procedure 158-159  
Spearman coefficient 260, 264

SPEARMAN option in CORR procedure 264  
splitting  
  data file 48-49  
  SAS data set 74-75, 194-195  
SPSS files data engine 19  
SQL procedure  
  calculated variables 72-73  
  combining SAS data sets 180-181  
  GROUP BY clause 186-187  
  HAVING clause 218-219  
  INNER JOIN clause 180-181  
  INTO clause 218-219  
  ON clause 180-181  
  SELECT clause 72-73, 186-187  
  subsetting data using 72-73  
  summarizing variables 186-187, 218-219  
  WHERE clause 72-73, 180-181  
STACKED option in FREQ procedure 262  
stacking SAS data sets 172-173  
standard deviation  
  MEANS procedure 112-113, 254  
  REPORT procedure 138  
  TABULATE procedure 124  
  UNIVARIATE procedure 250-251  
standard error  
  MEANS procedure 254  
  REG procedure 269  
STARTPAGE= option  
  ODS PDF statement 150-151  
  ODS RTF statement 148-149  
STAT= option in VBAR statement 226  
Stata files  
  reading 19  
  writing 278  
statement not valid error in log 310-311  
statement options compared to data set options 190-191  
statistics

analysis of variance 272-275  
categorical data 260-261  
correlations 264-265  
descriptive 112-139, 250-255  
multiple comparisons 272-275  
output data set, MEANS procedure 114-115  
regression 268-269  
t test 256-259, 272  
STD keyword in REPORT procedure 138  
STDDEV option  
    MEANS procedure 112, 254  
    TABULATE procedure 124  
STDERR option in MEANS procedure 254  
STIMERw. informat 38-39  
STOP statement 7, 217  
STRENGTH= suboption in SORT procedure 100-101  
strings, character 4, 58-59  
Stuart's tau-c 264  
student's t 258  
style attributes  
    for graphics 242-243  
    PRINT procedure 156-157  
    REPORT procedure 158-159  
    table of 164-165  
    TABULATE procedure 160-161  
style templates 144-145  
    for graphics 225, 242-243, 247  
STYLE= option  
    graph attributes 242  
    graphics 247  
    ODS HTML statement 146-147  
    ODS LISTING statement 225, 247  
    ODS PDF statement 150-151  
    ODS RTF statement 148-149  
    PRINT procedure 156-157  
    REPORT procedure 158-159  
    TABULATE procedure 160-161

- trafficlighting 162-163
  - user-defined formats 162-163
- submitting SAS programs, methods 10-11
- subsetting observations
  - DELETE statements 70-71
  - IF statements 70-71
  - IN= data set option 192-193
  - INPUT statements 48-49
  - OUTPUT statement 74-75
  - reading raw data file 48-49
  - saving memory and disk space 317
  - SQL procedure 72-73
  - WHERE clause in SQL procedure 72-73
  - WHERE statement 70, 96-97
  - WHERE= data set option 194-195
- SUBSTR function 62-63
- subtotals
  - PRINT procedure 102-103
  - REPORT procedure 136-137
- SUM function 64-65, 305
- SUM keyword
  - REPORT procedure 138
  - TABULATE procedure 124
- sum of squares
  - ANOVA procedure 274-275
  - MEANS procedure 254
  - REG procedure 268
- SUM option in MEANS procedure 112, 254
- SUM statement in PRINT procedure 102-103
- sum statements, DATA step 84-85
- SUMMARIZE option in REPORT procedure 136-137
- SUMMARY location in STYLE= option 158
- SUMMARY procedure 115
- summary statistics
  - combining with data 182-187
  - MEANS procedure 112-113, 254-255
  - REPORT procedure 130-139

saving in SAS data set 114-115  
TABULATE procedure 122-125  
UNIVARIATE procedure 250-251  
SUMMARYPLOT option in TTEST procedure 258-259  
sums  
    across observations 84-85, 102-103, 112-115, 124-129  
    across variables 58-59, 64-65, 305  
    combining with data 182-187  
    controlling style in PRINT procedure 156  
    MEANS procedure 112-115  
    REPORT procedure 130-141  
    SUM function 64-65, 305  
    SUM keyword in TABULATE procedure 124  
    SUM option in MEANS procedure 112, 254  
    sum statement in DATA step 84-85  
    SUM statement in PRINT procedure 102-103  
    TABULATE procedure 124-129  
SUMWGT option in MEANS procedure 254  
SVG image format 246  
SYMBOL= option for graph attributes 242-243  
SYMBOLGEN system option 220-221  
SYMPUTX, CALL 216-217  
syntax of SAS programs 2-3  
syntax-sensitive editor 10-11, 293  
syntax, checking 295  
system options 14-15  
    BOTTOMMARGIN= 15  
    CENTER/NOCENTER 15  
    compared to data set options 190-191  
    DATASTMTCHK= 15, 297  
    DATE/NODATE 15  
    DATESTYLE= 15, 80  
    LEFTMARGIN= 15  
    MACRO 203  
    MERROR 220-221  
    MISSING= 15, 282-283, 286-287  
    MLOGIC 220-221

MPRINT 220-221  
NUMBER/NONUMBER 15  
ORIENTATION= 15  
PAGENO= 15  
RIGHTMARGIN= 15  
SError 220-221  
SYMBOLGEN 220-221  
TOPMARGIN= 15  
VALIDVARNAME= 5, 15, 90-91  
YEARCUTOFF= 15, 80

## T

T option

ANOVA procedure 272  
MEANS procedure 254

t tests

ANOVA procedure 272  
MEANS procedure 254  
TTEST procedure 256-259

TAB value in the DBMS= option

EXPORT procedure 280  
IMPORT procedure 28

tab-delimited data

reading 28-29, 52-53  
writing 280-281

TABLE statement in TABULATE procedure 122-129

STYLE= option 160-161

table templates 144-145

tables of data, definition 4

TABLES statement in FREQ procedure 116-119, 260-261

TABULATE procedure 122-129

CLASSLEV statement 160  
STYLE= option 160-161

templates 144-145

temporary SAS data sets 20-21, 316

text files

reading 18, 30

writing 144, 152-153  
text, adding to graphs 240-241  
TEXTALIGN style attribute 164-165  
THEN keyword 66-69  
THICKNESS= option for graph attributes 242-243  
TIFF image format 246  
time data  
    formats 106-107  
    informats 38-39  
TIME option in XAXIS or YAXIS statement 238  
TIMEw. informat 38-39  
TIMEw.d format 106-107  
Title, removing procedure name 146-151  
TITLE statement 94-95, 154-155  
TODAY function 64-65, 80-83  
TOPMARGIN= system option 15  
TOTAL location in STYLE= option 156  
totals  
    across observations 84-85, 102-103, 112-115, 124-129  
    across variables 58-59, 64-65, 305  
    combining with data 182-187  
    controlling style in PRINT procedure 156  
    MEANS procedure 112-115  
    REPORT procedure 130-141  
    SUM function 64-65, 305  
    SUM keyword in TABULATE procedure 124  
    SUM option in MEANS procedure 112, 254  
    sum statement in DATA step 84-85  
    SUM statement in PRINT procedure 102-103  
    TABULATE procedure 124-129  
tracing output objects 166-167  
tracking observations IN= data set option 192-193  
trafficlighting 162-163  
trailing @ 48-49, 288  
trailing @@ 46-47  
transaction-oriented data 188-189  
TRANSLATE function 62-63

TRANSPARENCY= option  
  DENSITY statement 228  
  HBOX or VBOX statement 230  
  HISTOGRAM statement 228  
  LOESS statement 236  
  PBSPLINE statement 236  
  REFLINE statement 238-239  
  REG statement 236  
  SCATTER statement 232  
  SERIES statement 234  
  VBAR statement 226

TRANSPOSE procedure 196-197  
transposing data with OUTPUT statement 76-77

TRANWRD function 62-63

TREND option in FREQ procedure 260

TRIM function 62-63

truncation of character data 68-69, 314-315

TRUNCOVER option on INFILE statement 51, 299

TTEST procedure 256-259

TUKEY option in ANOVA procedure 272

Tukey's studentized range test 276

two-way frequency table 118-119, 260-261

TWOWAY= option in FREQ procedure 262-263

type of variable 4, 23

TYPE= option  
  DENSITY statement 228-229  
  XAXIS or YAXIS statement 238-239

## U

UCLM option in MEANS procedure 254

uninitialized variables 312-313

UNISCALE= option in PANELBY statement 244

UNIVARIATE procedure 250-253

University Edition 11

UNIX  
  direct referencing of SAS data sets 20-21  
  INFILE statement 31

LIBNAME statement 20-21  
UPCASE function 61-63  
UPDATE statement 188-189  
URL style attribute 164-165  
usage options in REPORT procedure 132-133  
user-defined formats 108-109  
    grouping with 120-121  
    trafficlighting 162-163  
    with TABULATE procedure 128-129  
USS option in MEANS procedure 254

## V

VALIDVARNAME= system option 5, 15, 90-91  
VALUE statement FORMAT procedure 108-109  
VALUEATTRS= option for graph attributes 242  
VALUES= option in XAXIS or YAXIS statement 238  
VAR option in MEANS procedure 254  
VAR statement  
    CORR procedure 264-265  
    MEANS procedure 112-113, 254-255  
    PRINT procedure 102-103  
    STYLE= option in PRINT procedure 156-157  
    STYLE= option in TABULATE procedure 160  
    TABULATE procedure 124-129  
    TRANSPOSE procedure 196-197  
    TTEST procedure 256  
    UNIVARIATE procedure 250  
variable length records, reading 51  
variable length values, reading 42-43  
variable name lists  
    _ALL_ 88-89, 308-309  
    _CHARACTER_ 88-89  
    _NUMERIC_ 88-89  
    name prefix 88  
    name ranges 88-89  
    numbered ranges 88-89  
variable not found error in log 312-313

variable uninitialized note in log 312-313

variables

- arrays 86-87
- automatic 198-199
- automatic macro 206-207
- changing to observations 196-197
- creating a grouping variable 68-69
- creating in REPORT procedure 140-141
- creating with assignment statements 58-59
- definition 4
- dropping 190-191
- keeping 190-191
- labels 22-23, 95
- length 22-23, 68-69, 314-317
- lists 88-89
- means 112-113
- names 5, 90-91
- printing 102-103
- renaming 190-191
- retaining values between observations 84-85
- skipping when reading raw data 34
- type 4, 23
- uninitialized 312-313

variance with MEANS procedure 254

VBAR statement in SGLOT procedure 226-227

VBOX statement in SGLOT procedure 230-231

vector graphics, scalable 246

Viewable window 18

## W

w.d format 106-107

w.d informat 38-39

warnings in SAS log 294

Web, creating files for 146-147

WEEKDATEw. format 82-83, 106-107

WEEKDAY function 64-65, 82-83

WEIBULL option in distribution plots 252

WEIGHT= option for graph attributes 242-243  
WHERE clause in SQL procedure 72-73, 180-181  
WHERE statement  
    DATA steps 70  
    procedures 96-97  
WHERE= data set option 190, 194-195  
WIDTH= option in ODS GRAPHICS statement 246-247  
windowing environment, SAS 10-11  
Windows operating environment  
    direct referencing of SAS data sets 20-21  
    INFILE statement 31  
    LIBNAME statement 20-21  
WITH statement in CORR procedure 264-265  
WORD destination 144  
WORDDATEw. format 82-83, 106-107  
WORK library 20-21, 316  
writing data 278-283  
    delimited 280-283, 288-289  
    methods 278-279  
    PC files 278-279, 284-285  
    raw data 280-283, 288-289  
writing SAS data sets  
    DATA step 6-7  
    multiple data sets 74-75  
    permanent data sets 20-21  
WTKAPPAPLOT option in FREQ procedure 262

## X

XAXIS statement in SGPLOT procedure 238-239  
XLSX LIBNAME engine 26-27

## Y

YAXIS statement in SGPLOT procedure 238-239  
YEAR function 64-65, 82-83  
YEARCUTOFF= system option 15, 80  
YRDIF function 64-65, 80-83

**Z**

z/OS

direct referencing of SAS data sets 20-21

INFILE statement 31

LIBNAME statement 20-21



# Ready to take your SAS® and JMP® skills up a notch?



Be among the first to know about new books,  
special events, and exclusive discounts.

[support.sas.com/newbooks](http://support.sas.com/newbooks)

Share your expertise. Write a book with SAS.

[support.sas.com/publish](http://support.sas.com/publish)

 [sas.com/books](http://sas.com/books)  
*for additional books and resources.*

  
THE POWER TO KNOW.™

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.  
Other brand and product names are trademarks of their respective companies. © 2017 SAS Institute Inc. All rights reserved. M1580359 US 5/217

# Contents

1. Acknowledgments
2. Introducing SAS Software
3. About This Book
4. About These Authors
5. Chapter 1 Getting Started Using SAS Software

1. 1.1 The SAS Language
  2. 1.2 SAS Data Sets
  3. 1.3 DATA and PROC Steps
  4. 1.4 The DATA Step's Built-in Loop
  5. 1.5 Choosing a Method for Running SAS
  6. 1.6 Reading the SAS Log
  7. 1.7 Using SAS System Options
6. Chapter 2 Accessing Your Data

1. 2.1 Methods for Getting Your Data into SAS
2. 2.2 SAS Data Libraries and Data Sets
3. 2.3 Listing the Contents of a SAS Data Set
4. 2.4 Reading Excel Files with the IMPORT Procedure
5. 2.5 Accessing Excel Files Using the XLSX LIBNAME Engine
6. 2.6 Reading Delimited Files with the IMPORT Procedure
7. 2.7 Telling SAS Where to Find Your Raw Data
8. 2.8 Reading Raw Data Separated by Spaces
9. 2.9 Reading Raw Data Arranged in Columns
10. 2.10 Reading Raw Data Not in Standard Format
11. 2.11 Selected Informats
12. 2.12 Mixing Input Styles

13. [2.13 Reading Messy Raw Data](#)
14. [2.14 Reading Multiple Lines of Raw Data per Observation](#)
15. [2.15 Reading Multiple Observations per Line of Raw Data](#)
16. [2.16 Reading Part of a Raw Data File](#)
17. [2.17 Controlling Input with Options in the INFILE Statement](#)
18. [2.18 Reading Delimited Files with the DATA Step](#)

## 7. Chapter 3 Working with Your Data

1. [3.1 Using the DATA Step to Modify Data](#)
2. [3.2 Creating and Modifying Variables](#)
3. [3.3 Using SAS Functions](#)
4. [3.4 Selected SAS Character Functions](#)
5. [3.5 Selected SAS Numeric Functions](#)
6. [3.6 Using IF-THEN and DO Statements](#)
7. [3.7 Grouping Observations with IF-](#)

## THEN/ELSE Statements

8. 3.8 Subsetting Your Data in a DATA Step
9. 3.9 Subsetting Your Data Using PROC SQL
10. 3.10 Writing Multiple Data Sets Using OUTPUT Statements
11. 3.11 Making Several Observations from One Using OUTPUT Statements
12. 3.12 Using Iterative DO, DO WHILE, and DO UNTIL Statements
13. 3.13 Working with SAS Dates
14. 3.14 Selected Date Informats, Functions, and Formats
15. 3.15 Using RETAIN and Sum Statements
16. 3.16 Simplifying Programs with Arrays
17. 3.17 Using Shortcuts for Lists of Variable Names
18. 3.18 Using Variable Names with

## Special Characters

### 8. Chapter 4 Sorting, Printing, and Summarizing Your Data

1. 4.1 Using SAS Procedures
2. 4.2 Subsetting in Procedures with the WHERE Statement
3. 4.3 Sorting Your Data with PROC SORT
4. 4.4 Changing the Sort Order for Character Data
5. 4.5 Printing Your Data with PROC PRINT
6. 4.6 Changing the Appearance of DataValues with Formats
7. 4.7 Selected Standard Formats
8. 4.8 Creating Your Own Formats with PROC FORMAT
9. 4.9 Writing a Report to a Text File
10. 4.10 Summarizing Your Data Using PROC MEANS

11. 4.11 Writing Summary Statistics to a SAS Data Set
12. 4.12 Producing One-Way Frequencies with PROC FREQ
13. 4.13 Producing Crosstabulations with PROC FREQ
14. 4.14 Grouping Data with User-Defined Formats
15. 4.15 Producing Tabular Reports with PROC TABULATE
16. 4.16 Adding Statistics to PROC TABULATE Output
17. 4.17 Enhancing the Appearance of PROC TABULATE Output
18. 4.18 Changing Headers in PROC TABULATE Output
19. 4.19 Producing Simple Output with PROC REPORT
20. 4.20 Using DEFINE Statements in PROC REPORT
21. 4.21 Creating Summary Reports with

## PROC REPORT

22. 4.22 Adding Summary Breaks to PROC REPORT Output
23. 4.23 Adding Statistics to PROC REPORT Output
24. 4.24 Adding Computed Variables to PROC REPORT Output
9. Chapter 5 Enhancing Your Output with ODS
  1. 5.1 Concepts of the Output Delivery System
  2. 5.2 Creating HTML Output
  3. 5.3 Creating RTF Output
  4. 5.4 Creating PDF Output
  5. 5.5 Creating Text Output
  6. 5.6 Customizing Titles and Footnotes
  7. 5.7 Customizing PROC PRINT with the STYLE= Option
  8. 5.8 Customizing PROC REPORT with the STYLE= Option
  9. 5.9 Customizing PROC TABULATE

## with the STYLE= Option

10. 5.10 Adding Trafficlighting to Your Output
  11. 5.11 Selected Style Attributes
  12. 5.12 Tracing and Selecting Procedure Output
  13. 5.13 Creating SAS Data Sets from Procedure Output
10. Chapter 6 Modifying and Combining SAS Data Sets

1. 6.1 Stacking Data Sets Using the SET Statement
2. 6.2 Interleaving Data Sets Using the SET Statement
3. 6.3 Combining Data Sets Using a One-to-One Match Merge
4. 6.4 Combining Data Sets Using a One-to-Many Match Merge
5. 6.5 Using PROC SQL to Join Data Sets

6. 6.6 Merging Summary Statistics with the Original Data
7. 6.7 Combining a Grand Total with the Original Data
8. 6.8 Adding Summary Statistics to Data Using PROC SQL
9. 6.9 Updating a Master Data Set with Transactions
10. 6.10 Using SAS Data Set Options
11. 6.11 Tracking and Selecting Observations with the IN= Option
12. 6.12 Selecting Observations with the WHERE= Option
13. 6.13 Changing Observations to Variables Using PROC TRANSPOSE
14. 6.14 Using SAS Automatic Variables
11. Chapter 7 Writing Flexible Code with the SAS Macro Facility

1. 7.1 Macro Concepts
2. 7.2 Substituting Text with Macro

## Variables

3. 7.3 Concatenating Macro Variables with Other Text
4. 7.4 Creating Modular Code with Macros
5. 7.5 Adding Parameters to Macros
6. 7.6 Writing Macros with Conditional Logic
7. 7.7 Using %DO Loops in Macros
8. 7.8 Writing Data-Driven Programs with CALL SYMPUTX
9. 7.9 Writing Data-Driven Programs with PROC SQL
10. 7.10 Debugging Macro Errors
12. Chapter 8 Visualizing Your Data

1. 8.1 Concepts of ODS Graphics
2. 8.2 Creating Bar Charts with PROC SGLOT
3. 8.3 Creating Histograms and Density Curves with PROC SGLOT

4. 8.4 Creating Box Plots with PROC SGLOT
5. 8.5 Creating Scatter Plots with PROC SGLOT
6. 8.6 Creating Series Plots with PROC SGLOT
7. 8.7 Creating Fitted Curves with PROC SGLOT
8. 8.8 Controlling Axes and Reference Lines in PROC SGLOT
9. 8.9 Controlling Legends and Insets in PROC SGLOT
10. 8.10 Customizing Graph Attributes in PROC SGLOT
11. 8.11 Creating Paneled Graphs with PROC SGPANEL
12. 8.12 Specifying Image Properties and Saving Graphics Output
13. Chapter 9 Using Basic Statistical Procedures
  1. 9.1 Examining the Distribution of

## Data with PROC UNIVARIATE

2. 9.2 Creating Statistical Graphics with PROC UNIVARIATE
3. 9.3 Producing Statistics with PROC MEANS
4. 9.4 Testing Means with PROC TTEST
5. 9.5 Creating Statistical Graphics with PROC TTEST 258
6. 9.6 Testing Categorical Data with PROC FREQ
7. 9.7 Creating Statistical Graphics with PROC FREQ
8. 9.8 Examining Correlations with PROC CORR
9. 9.9 Creating Statistical Graphics with PROC CORR
10. 9.10 Using PROC REG for Simple Regression Analysis
11. 9.11 Creating Statistical Graphics with PROC REG
12. 9.12 Using PROC ANOVA for One-

## Way Analysis of Variance

### 13. 9.13 Reading the Output of PROC ANOVA

### 14. Chapter 10 Exporting Your Data

1. 10.1 Methods for Exporting Your Data
2. 10.2 Writing Delimited Files with the EXPORT Procedure
3. 10.3 Writing Delimited Files Using ODS
4. 10.4 Writing Microsoft Excel Files with the EXPORT Procedure
5. 10.5 Writing Microsoft Excel Files Using ODS
6. 10.6 Writing Raw Data Files with the DATA Step

### 15. Chapter 11 Debugging Your SAS Programs

1. 11.1 Writing SAS Programs That Work
2. 11.2 Fixing Programs That Don't

## Work

3. 11.3 Searching for the Missing Semicolon
4. 11.4 Note: INPUT Statement Reached Past the End of a Line
5. 11.5 Note: Lost Card
6. 11.6 Note: Invalid Data
7. 11.7 Note: Missing Values Were Generated
8. 11.8 Note: Numeric Values Have Been Converted to Character (or Vice Versa)
9. 11.9 DATA Step Produces Wrong Results but No Error Message
10. 11.10 Error: Invalid Option, Error: The Option Is Not Recognized, or Error: Statement IsNot Valid
11. 11.11 Note: Variable Is Uninitialized or Error: Variable Not Found
12. 11.12 SAS Truncates a Character Variable

13. 11.13 Saving Memory or Disk Space

16. Index