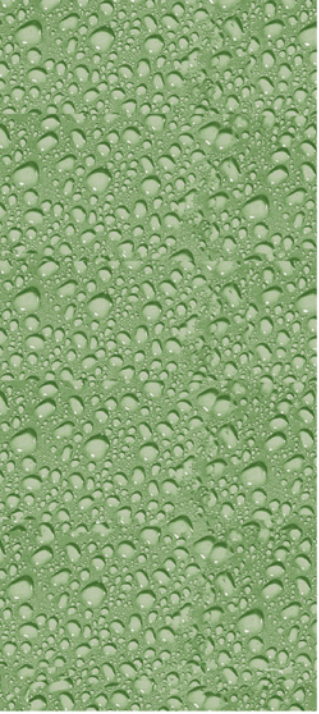


Cody's Data Cleaning Techniques Using SAS®

Second Edition

Ron Cody

SAS® Press Series

sas

Cody's Data Cleaning Techniques Using SAS®

Second Edition

Ron Cody

The correct bibliographic citation for this manual is as follows: Cody, Ron. 2008. *Cody's Data Cleaning Techniques Using SAS®, Second Edition.* Cary, NC: SAS Institute Inc.

Cody's Data Cleaning Techniques Using SAS®, Second Edition

ISBN 978-1-59994-659-7

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

1st printing, April 2008

SAS® Publishing 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 Publishing Web site at **support.sas.com/publishing** or call 1-800-727-3228.

Table of Contents

List of Programs ix
Preface xv
Acknowledgments xvii

1 Checking Values of Character Variables

Introduction 1
Using PROC FREQ to List Values 1
Description of the Raw Data File PATIENTS.TXT 2
Using a DATA Step to Check for Invalid Values 7
Describing the VERIFY, TRIM, MISSING, and NOTDIGIT Functions 9
Using PROC PRINT with a WHERE Statement to List Invalid Values 13
Using Formats to Check for Invalid Values 15
Using Informats to Remove Invalid Values 18

2 Checking Values of Numeric Variables

Introduction 23
Using PROC MEANS, PROC TABULATE, and PROC UNIVARIATE to Look for Outliers 24
Using an ODS SELECT Statement to List Extreme Values 34
Using PROC UNIVARIATE Options to List More Extreme Observations 35
Using PROC UNIVARIATE to Look for Highest and Lowest Values by Percentage 37
Using PROC RANK to Look for Highest and Lowest Values by Percentage 43
Presenting a Program to List the Highest and Lowest Ten Values 47
Presenting a Macro to List the Highest and Lowest "n" Values 50
Using PROC PRINT with a WHERE Statement to List Invalid Data Values 52
Using a DATA Step to Check for Out-of-Range Values 54
Identifying Invalid Values versus Missing Values 55

Listing Invalid (Character) Values in the Error Report 57
Creating a Macro for Range Checking 60
Checking Ranges for Several Variables 62
Using Formats to Check for Invalid Values 66
Using Informats to Filter Invalid Values 68
Checking a Range Using an Algorithm Based on Standard Deviation 71
Detecting Outliers Based on a Trimmed Mean and Standard Deviation 73
Presenting a Macro Based on Trimmed Statistics 76
Using the TRIM Option of PROC UNIVARIATE and ODS to Compute Trimmed Statistics 80
Checking a Range Based on the Interquartile Range 86

3 Checking for Missing Values

Introduction 91
Inspecting the SAS Log 91
Using PROC MEANS and PROC FREQ to Count Missing Values 93
Using DATA Step Approaches to Identify and Count Missing Values 96
Searching for a Specific Numeric Value 100
Creating a Macro to Search for Specific Numeric Values 102

4 Working with Dates

Introduction 105
Checking Ranges for Dates (Using a DATA Step) 106
Checking Ranges for Dates (Using PROC PRINT) 107
Checking for Invalid Dates 108
Working with Dates in Nonstandard Form 111
Creating a SAS Date When the Day of the Month Is Missing 113
Suspending Error Checking for Known Invalid Dates 114

5 Looking for Duplicates and "n" Observations per Subject

Introduction 117
Eliminating Duplicates by Using PROC SORT 117
Detecting Duplicates by Using DATA Step Approaches 123
Using PROC FREQ to Detect Duplicate ID's 126
Selecting Patients with Duplicate Observations by Using a Macro List and SQL 129
Identifying Subjects with "n" Observations Each (DATA Step Approach) 130
Identifying Subjects with "n" Observations Each (Using PROC FREQ) 132

6 Working with Multiple Files

Introduction 135
Checking for an ID in Each of Two Files 135
Checking for an ID in Each of "n" Files 138
A Macro for ID Checking 140
More Complicated Multi-File Rules 143
Checking That the Dates Are in the Proper Order 147

7 Double Entry and Verification (PROC COMPARE)

Introduction 149
Conducting a Simple Comparison of Two Data Sets 150
Using PROC COMPARE with Two Data Sets That Have an Unequal Number of Observations 159
Comparing Two Data Sets When Some Variables Are Not in Both Data Sets 161

8 Some PROC SQL Solutions to Data Cleaning

Introduction 165
A Quick Review of PROC SQL 166
Checking for Invalid Character Values 166
Checking for Outliers 168

Checking a Range Using an Algorithm Based on the Standard Deviation 169
Checking for Missing Values 170
Range Checking for Dates 172
Checking for Duplicates 173
Identifying Subjects with "n" Observations Each 174
Checking for an ID in Each of Two Files 174
More Complicated Multi-File Rules 176

9 Correcting Errors

Introduction 181
Hardcoding Corrections 181
Describing Named Input 182
Reviewing the UPDATE Statement 184

10 Creating Integrity Constraints and Audit Trails

Introducing SAS Integrity Constraints 187
Demonstrating General Integrity Constraints 188
Deleting an Integrity Constraint Using PROC DATASETS 193
Creating an Audit Trail Data Set 193
Demonstrating an Integrity Constraint Involving More than One Variable 200
Demonstrating a Referential Constraint 202
Attempting to Delete a Primary Key When a Foreign Key Still Exists 205
Attempting to Add a Name to the Child Data Set 207
Demonstrating the Cascade Feature of a Referential Constraint 208
Demonstrating the SET NULL Feature of a Referential Constraint 210
Demonstrating How to Delete a Referential Constraint 211

11 DataFlux and dfPower Studio

Introduction 213
Examples 215

Appendix Listing of Raw Data Files and SAS Programs

Programs and Raw Data Files Used in This Book 217
Description of the Raw Data File PATIENTS.TXT 217
Layout for the Data File PATIENTS.TXT 218
Listing of Raw Data File PATIENTS.TXT 218
Program to Create the SAS Data Set PATIENTS 219
Listing of Raw Data File PATIENTS2.TXT 220
Program to Create the SAS Data Set PATIENTS2 221
Program to Create the SAS Data Set AE (Adverse Events) 221
Program to Create the SAS Data Set LAB_TEST 222
Listings of the Data Cleaning Macros Used in This Book 222

Index 239

List of Programs

1 Checking Values of Character Variables

Program 1-1	Writing a Program to Create the Data Set PATIENTS	3
Program 1-2	Using PROC FREQ to List All the Unique Values for Character Variables	4
Program 1-3	Using the Keyword _CHARACTER_ in the TABLES Statement	6
Program 1-4	Using a DATA _NULL_ Step to Detect Invalid Character Data	7
Program 1-5	Using PROC PRINT to List Invalid Character Values	13
Program 1-6	Using PROC PRINT to List Invalid Character Data for Several Variables	14
Program 1-7	Using a User-Defined Format and PROC FREQ to List Invalid Data Values	15
Program 1-8	Using a User-Defined Format and a DATA Step to List Invalid Data Values	17
Program 1-9	Using a User-Defined Informat to Set Invalid Data Values to Missing	19
Program 1-10	Using a User-Defined Informat with the INPUT Function	21

2 Checking Values of Numeric Variables

Program 2-1	Using PROC MEANS to Detect Invalid and Missing Values	24
Program 2-2	Using PROC TABULATE to Display Descriptive Data	25
Program 2-3	Using PROC UNIVARIATE to Look for Outliers	26
Program 2-4	Using an ODS SELECT Statement to Print Only Extreme Observations	34

Program 2-5	Using the NEXTROBS= Option to Print the 10 Highest and Lowest Observations	35
Program 2-6	Using the NEXTRVALS= Option to Print the 10 Highest and Lowest Values	36
Program 2-7	Using PROC UNIVARIATE to Print the Top and Bottom "n" Percent of Data Values	38
Program 2-8	Creating a Macro to List the Highest and Lowest "n" Percent of the Data Using PROC UNIVARIATE	40
Program 2-9	Creating a Macro to List the Highest and Lowest "n" Percent of the Data Using PROC RANK	45
Program 2-10	Creating a Program to List the Highest and Lowest 10 Values	47
Program 2-11	Presenting a Macro to List the Highest and Lowest "n" Values	50
Program 2-12	Using a WHERE Statement with PROC PRINT to List Out-of-Range Data	53
Program 2-13	Using a DATA _NULL_ Step to List Out-of-Range Data Values	54
Program 2-14	Presenting a Program to Detect Invalid (Character) Data Values, Using _ERROR_	56
Program 2-15	Including Invalid Values in Your Error Report	58
Program 2-16	Writing a Macro to List Out-of-Range Data Values	61
Program 2-17	Writing a Program to Summarize Data Errors on Several Variables	62
Program 2-18	Detecting Out-of-Range Values Using User-Defined Formats	67
Program 2-19	Using User-Defined Informats to Filter Invalid Values	69
Program 2-20	Detecting Outliers Based on the Standard Deviation	71
Program 2-21	Computing Trimmed Statistics	73
Program 2-22	Detecting Outliers Based on Trimmed Statistics	75
Program 2-23	Creating a Macro to Detect Outliers Based on Trimmed Statistics	77
Program 2-24	Using the TRIM= Option of PROC UNIVARIATE	80
Program 2-25	Using ODS to Capture Trimmed Statistics from PROC UNIVARIATE	81
Program 2-26	Presenting a Macro to List Outliers of Several Variables Based on Trimmed Statistics (Using PROC UNIVARIATE)	83
Program 2-27	Detecting Outliers Based on the Interquartile Range	87

3 Checking for Missing Values

Program 3-1	Counting Missing and Non-missing Values for Numeric and Character Variables	94
Program 3-2	Writing a Simple DATA Step to List Missing Data Values and an ID Variable	96
Program 3-3	Attempting to Locate a Missing or Invalid Patient ID by Listing the Two Previous ID's	97
Program 3-4	Using PROC PRINT to List Data for Missing or Invalid Patient ID's	98
Program 3-5	Listing and Counting Missing Values for Selected Variables	99
Program 3-6	Identifying All Numeric Variables Equal to a Fixed Value (Such as 999)	101
Program 3-7	Creating a Macro to Search for Specific Numeric Values	102

4 Working with Dates

Program 4-1	Checking That a Date Is within a Specified Interval (DATA Step Approach)	106
Program 4-2	Checking That a Date Is within a Specified Interval (Using PROC PRINT and a WHERE Statement)	107
Program 4-3	Reading Dates with the MMDDYY10. Informat	108
Program 4-4	Listing Missing and Invalid Dates by Reading the Date Twice, Once with a Date Informat and the Second as Character Data	109
Program 4-5	Listing Missing and Invalid Dates by Reading the Date as a Character Variable and Converting to a SAS Date with the INPUT Function	110
Program 4-6	Removing the Missing Values from the Invalid Date Listing	111
Program 4-7	Demonstrating the MDY Function to Read Dates in Nonstandard Form	112
Program 4-8	Creating a SAS Date When the Day of the Month Is Missing	113
Program 4-9	Substituting the 15th of the Month When the Date of the Month Is Missing	114

Program 4-10 Suspending Error Checking for Known Invalid Dates by Using the ?? Informat Modifier 115
Program 4-11 Demonstrating the ?? Informat Modifier with the INPUT Function 115

5 Looking for Duplicates and "n" Observations per Subject

Program 5-1 Demonstrating the NODUPKEY Option of PROC SORT 118
Program 5-2 Demonstrating the NODUPRECS Option of PROC SORT 120
Program 5-3 Demonstrating a Problem with the NODUPRECS (NODUP) Option 121
Program 5-4 Removing Duplicate Records Using PROC SQL 123
Program 5-5 Identifying Duplicate ID's 123
Program 5-6 Creating the SAS Data Set PATIENTS2 (a Data Set Containing Multiple Visits for Each Patient) 125
Program 5-7 Identifying Patient ID's with Duplicate Visit Dates 126
Program 5-8 Using PROC FREQ and an Output Data Set to Identify Duplicate ID's 127
Program 5-9 Producing a List of Duplicate Patient Numbers by Using PROC FREQ 128
Program 5-10 Using PROC SQL to Create a List of Duplicates 129
Program 5-11 Using a DATA Step to List All ID's for Patients Who Do Not Have Exactly Two Observations 131
Program 5-12 Using PROC FREQ to List All ID's for Patients Who Do Not Have Exactly Two Observations 132

6 Working with Multiple Files

Program 6-1 Creating Two Test Data Sets for Chapter 6 Examples 136
Program 6-2 Identifying ID's Not in Each of Two Data Sets 136
Program 6-3 Creating a Third Data Set for Testing Purposes 138
Program 6-4 Checking for an ID in Each of Three Data Sets (Long Way) 139
Program 6-5 Presenting a Macro to Check for ID's Across Multiple Data Sets 141
Program 6-6 Creating Data Set AE (Adverse Events) 143
Program 6-7 Creating Data Set LAB_TEST 144

Program 6-8 Verifying That Patients with an Adverse Event of "X" in Data Set AE Have an Entry in Data Set LAB_TEST 146
Program 6-9 Adding the Condition That the Lab Test Must Follow the Adverse Event 147

7 Double Entry and Verification (PROC COMPARE)

Program 7-1 Creating Data Sets ONE and TWO from Two Raw Data Files 151
Program 7-2 Running PROC COMPARE 152
Program 7-3 Demonstrating the TRANSPOSE Option of PROC COMPARE 156
Program 7-4 Using PROC COMPARE to Compare Two Data Records 157
Program 7-5 Running PROC COMPARE on Two Data Sets of Different Length 160
Program 7-6 Creating Two Test Data Sets, DEMOG and OLDDEMOG 161
Program 7-7 Comparing Two Data Sets That Contain Different Variables 162
Program 7-8 Adding a VAR Statement to PROC COMPARE 163

8 Some PROC SQL Solutions to Data Cleaning

Program 8-1 Demonstrating a Simple SQL Query 166
Program 8-2 Using PROC SQL to Look for Invalid Character Values 167
Program 8-3 Using SQL to Check for Out-of-Range Numeric Values 168
Program 8-4 Using SQL to Check for Out-of-Range Values Based on the Standard Deviation 169
Program 8-5 Using SQL to List Missing Values 170
Program 8-6 Using SQL to Perform Range Checks on Dates 172
Program 8-7 Using SQL to List Duplicate Patient Numbers 173
Program 8-8 Using SQL to List Patients Who Do Not Have Two Visits 174
Program 8-9 Creating Two Data Sets for Testing Purposes 175
Program 8-10 Using SQL to Look for ID's That Are Not in Each of Two Files 175
Program 8-11 Using SQL to Demonstrate More Complicated Multi-File Rules 176
Program 8-12 Example of LEFT, RIGHT, and FULL Joins 177

9 Correcting Errors

Program 9-1	Hardcoding Corrections Using a DATA Step	181
Program 9-2	Describing Named Input	182
Program 9-3	Using Named Input to Make Corrections	183
Program 9-4	Demonstrating How UPDATE Works	184

10 Creating Integrity Constraints and Audit Trails

Program 10-1	Creating Data Set HEALTH to Demonstrate Integrity Constraints	189
Program 10-2	Creating Integrity Constraints Using PROC DATASETS	190
Program 10-3	Creating Data Set NEW Containing Valid and Invalid Data	192
Program 10-4	Attempting to Append Data Set NEW to the HEALTH Data Set	192
Program 10-5	Deleting an Integrity Constraint Using PROC DATASETS	193
Program 10-6	Adding User Messages to the Integrity Constraints	194
Program 10-7	Creating an Audit Trail Data Set	195
Program 10-8	Using PROC PRINT to List the Contents of the Audit Trail Data Set	196
Program 10-9	Reporting the Integrity Constraint Violations Using the Audit Trail Data Set	197
Program 10-10	Correcting Errors Based on the Observations in the Audit Trail Data Set	199
Program 10-11	Demonstrating an Integrity Constraint Involving More than One Variable	200
Program 10-12	Added the Survey Data	201
Program 10-13	Creating Two Data Sets and a Referential Constraint	203
Program 10-14	Attempting to Delete a Primary Key When a Foreign Key Still Exists	205
Program 10-15	Attempting to Add a Name to the Child Data Set	207
Program 10-16	Demonstrate the CASCADE Feature of a Referential Integrity Constraint	209
Program 10-17	Demonstrating the SET NULL Feature of a Referential Constraint	210
Program 10-18	Demonstrating How to Delete a Referential Constraint	212

Preface to the Second Edition

Although this book is titled *Cody's Data Cleaning Techniques Using SAS*, I hope that it is more than that. It is my hope that not only will you discover ways to detect data errors, but you will also be exposed to some DATA step programming techniques and SAS procedures that might be new to you.

I have been teaching a two-day data cleaning workshop for SAS, based on the first edition of this book, for several years. I have thoroughly enjoyed traveling to interesting places and meeting other SAS programmers who have a need to find and fix errors in their data. This experience has also helped me identify techniques that other SAS users will find useful.

There have been some significant changes in SAS since the first edition was published—specifically, SAS®9. SAS®9 includes many new functions that make the task of finding and correcting data errors much easier. In addition, SAS®9 allows you to create integrity constraints and audit trails. Integrity constraints are rules about your data that are stored in the data descriptor portion of a SAS data set. These rules prevent data that violates any of these constraints to be rejected when you try to add it to an existing data set. In addition, SAS can create an audit trail data set that shows which new observations were added and which observations were rejected, along with the reason for their rejection.

So, besides a new chapter on integrity constraints and audit trails, I have added several macros that might make your data cleaning tasks easier. I also corrected or removed several programs that the compulsive programmer in me could not allow to remain.

Finally, a short description of a SAS product called DataFlux® was added. DataFlux is a comprehensive collection of programs, with an interactive front-end, that perform many advanced data cleaning techniques such as address standardization and fuzzy matching.

I hope you enjoy this new edition.

Ron Cody
Winter 2008

Preface to the First Edition

What is data cleaning? In this book, we define data cleaning to include:

- Making sure that the raw data values were accurately entered into a computer readable file.
- Checking that character variables contain only valid values.
- Checking that numeric values are within predetermined ranges.
- Checking if there are missing values for variables where complete data is necessary.
- Checking for and eliminating duplicate data entries.
- Checking for uniqueness of certain values, such as patient IDs.
- Checking for invalid date values.
- Checking that an ID number is present in each of "n" files.
- Verifying that more complex multi-file rules have been followed.

This book provides many programming examples to accomplish the tasks listed above. In many cases, a given problem is solved in several ways. For example, numeric outliers are detected in a DATA step by using formats and informats, by using SAS procedures, and SQL queries. Throughout the book, there are useful macros that you may want to add to your collection of data cleaning tools. However, even if you are not experienced with SAS macros, most of the macros that are presented are first shown in non-macro form, so you should still be able to understand the programming concepts.

But, there is another purpose for this book. It provides instruction on intermediate and advanced SAS programming techniques. One of the reasons for providing multiple solutions to data cleaning problems is to demonstrate specific features of SAS programming. For those cases, the tools that are developed can be the jumping-off point for more complex programs.

Many applications that require accurate data entry use customized, and sometimes very expensive, data entry and verification programs. A chapter on PROC COMPARE shows how SAS can be used in a double-entry data verification process.

I have enjoyed writing this book. Writing any book is a learning experience and this book is no exception. I hope that most of the egregious errors have been eliminated. If any remain, I take full responsibility for them. Every program in the text has been run against sample data. However, as experience will tell, no program is foolproof.

Acknowledgments

This is a very special acknowledgment since my good friend and editor, Judy Whatley has retired from SAS Institute. As a matter of fact, the first edition of this book (written in 1999) was the first book she and I worked on together. Since then Judy has edited three more of my books. Judy, you are the best!

Now I have a new editor, John West. I have known John for some time, enjoying our talks at various SAS conferences. John has the job of seeing through the last phases of this book. I expect that John and I will be working on more books in the future—what else would I do with my "spare" time? Thank you, John, for all your patience.

There was a "cast of thousands" (well, perhaps a small exaggeration) involved in the review and production of this book and I would like to thank them all. To start, there were reviewers who worked for SAS who read either the entire book or sections where they had particular expertise. They are: Paul Grant, Janice Bloom, Lynn Mackay, Marjorie Lampton, Kathryn McLawhorn, Russ Tyndall, Kim Wilson, Amber Elam, and Pat Herbert.

In addition to these internal reviewers, I called on "the usual suspects," my friends who were willing to spend time to carefully read every word and program. For this second edition, they are: Mike Zdeb, Joanne Dipietro, and Sylvia Brown. While all three of these folks did a great job, I want to acknowledge that Mike Zdeb went above and beyond, pointing out techniques and tips (many of which were unknown to me) that, I think, made this a much better book.

The production of a book also includes lots of other people who provide such support as copy editing, cover design, and marketing. I wish to thank all of these people as well for their hard work: Mary Beth Steinbach, managing editor; Joel Byrd, copyeditor; Candy Farrell, technical publishing specialist; Jennifer Dilley, technical publishing specialist; Patrice Cherry, cover designer; Liz Villani, marketing specialist; and Shelly Goodin, marketing specialist.

Ron Cody
Winter 2008

1 Checking Values of Character Variables

Introduction 1
Using PROC FREQ to List Values 1
Description of the Raw Data File PATIENTS.TXT 2
Using a DATA Step to Check for Invalid Values 7
Describing the VERIFY, TRIM, MISSING, and NOTDIGIT Functions 9
Using PROC PRINT with a WHERE Statement to List Invalid Values 13
Using Formats to Check for Invalid Values 15
Using Informats to Remove Invalid Values 18

Introduction

There are some basic operations that need to be routinely performed when dealing with character data values. You may have a character variable that can take on only certain allowable values, such as 'M' and 'F' for gender. You may also have a character variable that can take on numerous values but the values must fit a certain pattern, such as a single letter followed by two or three digits. This chapter shows you several ways that you can use SAS software to perform validity checks on character variables.

Using PROC FREQ to List Values

This section demonstrates how to use PROC FREQ to check for invalid values of a character variable. In order to test the programs you develop, use the raw data file PATIENTS.TXT, listed in the Appendix. You can use this data file and, in later sections, a SAS data set created from this raw data file for many of the examples in this text.

You can download all the programs and data files used in this book from the SAS Web site: http://support.sas.com/publishing. Click the link for SAS Press Companion Sites and select *Cody's Data Cleaning Techniques Using SAS, Second Edition.* Finally, click the link for Example Code and Data and you can download a text file containing all of the programs, macros, and text files used in this book.

Description of the Raw Data File PATIENTS.TXT

The raw data file PATIENTS.TXT contains both character and numeric variables from a typical clinical trial. A number of data errors were included in the file so that you can test the data cleaning programs that are developed in this text. Programs, data files, SAS data sets, and macros used in this book are stored in the folder C:\BOOKS\CLEAN. For example, the file PATIENTS.TXT is located in a folder (directory) called C:\BOOKS\CLEAN. You will need to modify the INFILE and LIBNAME statements to fit your own operating environment.

Here is the layout for the data file PATIENTS.TXT.

Variable Name	Description	Starting Column	Length	Variable Type	Valid Values
Patno	Patient Number	1	3	Character	Numerals only
Gender	Gender	4	1	Character	'M' or 'F'
Visit	Visit Date	5	10	MMDDYY10.	Any valid date
HR	Heart Rate	15	3	Numeric	Between 40 and 100
SBP	Systolic Blood Pressure	18	3	Numeric	Between 80 and 200
DBP	Diastolic Blood Pressure	21	3	Numeric	Between 60 and 120
Dx	Diagnosis Code	24	3	Character	1 to 3 digit numeral
AE	Adverse Event	27	1	Character	'0' or '1'

There are several character variables that should have a limited number of valid values. For this exercise, you expect values of Gender to be 'F' or 'M', values of Dx the numerals 1 through 999, and values of AE (adverse events) to be '0' or '1'. A very simple approach to identifying invalid character values in this file is to use PROC FREQ to list all the unique values of these variables. Of course, once invalid values are identified using this technique, other means will have to be employed to locate specific records (or patient numbers) containing the invalid values.

Use the program PATIENTS.SAS (shown next) to create the SAS data set PATIENTS from the raw data file PATIENTS.TXT (which can be downloaded from the SAS Web site or found listed in the Appendix). This program is followed with the appropriate PROC FREQ statements to list the unique values (and their frequencies) for the variables Gender, Dx, and AE.

Program 1-1 Writing a Program to Create the Data Set PATIENTS

```
*-------------------------------------------------------*
|PROGRAM NAME: PATIENTS.SAS in C:\BOOKS\CLEAN           |
|PURPOSE: To create a SAS data set called PATIENTS      |
*-------------------------------------------------------*;
libname clean "c:\books\clean";

data clean.patients;
   infile "c:\books\clean\patients.txt" truncover /* take care of problems
                                                     with short records */;

   input @1  Patno    $3.
         @4  Gender   $1.
         @5  Visit    mmddyy10.
         @15 Hr       3.
         @18 SBP      3.
         @21 DBP      3.
         @24 Dx       $3.
         @27 AE       $1.;

   LABEL Patno   = "Patient Number"
         Gender  = "Gender"
         Visit   = "Visit Date"
         HR      = "Heart Rate"
         SBP     = "Systolic Blood Pressure"
         DBP     = "Diastolic Blood Pressure"
         Dx      = "Diagnosis Code"
         AE      = "Adverse Event?";
    format visit mmddyy10.;
run;
```

The DATA step is straightforward. Notice the TRUNCOVER option in the INFILE statement. This will seem foreign to most mainframe users. If you do not use this option and you have short records, SAS will, by default, go to the next record to read data. The TRUNCOVER option prevents this from happening. The TRUNCOVER option is also useful when you are using list input (delimited data values). In this case, if you have more variables on the INPUT statement than there are in a single record on the data file, SAS will supply a missing value for all the remaining variables. One final note about INFILE options: If you have long record lengths (greater than 256 on PCs and UNIX platforms) you need to use the LRECL= option to change the default logical record length.

Next, you want to use PROC FREQ to list all the unique values for your character variables. To simplify the output from PROC FREQ, use the NOCUM (no cumulative statistics) and NOPERCENT (no percentages) TABLES options because you only want frequency counts for each of the unique character values. (Note: Sometimes the percent and cumulative statistics can be useful—the choice is yours.) The PROC statements are shown in Program 1-2.

Program 1-2 Using PROC FREQ to List All the Unique Values for Character Variables

```
title "Frequency Counts for Selected Character Variables";
proc freq data=clean.patients;
   tables Gender Dx AE / nocum nopercent;
run;
```

Here is the output from running Program 1-2.

```
Frequency Counts for Selected Character Variables

The FREQ Procedure

      Gender

Gender    Frequency
____________________
2                  1
F                 12
M                 14
X                  1
f                  2

Frequency Missing = 1


Diagnosis Code

Dx    Frequency
________________
1             7
2             2
3             3
4             3
5             3
6             1
7             2
X             2

Frequency Missing = 8
```

(*continued*)

```
Adverse Event?

AE    Frequency
_______________
0            19
1            10
A             1

Frequency Missing = 1
```

Let's focus in on the frequency listing for the variable Gender. If valid values for Gender are 'F', 'M', and missing, this output would point out several data errors. The values '2' and 'X' both occur once. Depending on the situation, the lowercase value 'f' may or may not be considered an error. If lowercase values were entered into the file by mistake, but the value (aside from the case) was correct, you could change all lowercase values to uppercase with the UPCASE function. More on that later. The invalid Dx code of 'X' and the adverse event of 'A' are also easily identified. At this point, it is necessary to run additional programs to identify the location of these errors. Running PROC FREQ is still a useful first step in identifying errors of these types, and it is also useful as a last step, after the data have been cleaned, to ensure that all the errors have been identified and corrected.

For those users who like shortcuts, here is another way to have PROC FREQ select the same set of variables in the example above, without having to list them all.

Program 1-3 Using the Keyword _CHARACTER_ in the TABLES Statement

```
title "Frequency Counts for Selected Character Variables";
proc freq data=clean.patients(drop=Patno);
   tables _character_ / nocum nopercent;
run;
```

The keyword _CHARACTER_ in this example is equivalent to naming all the character variables in the CLEAN.PATIENTS data set. Since you don't want the variable Patno included in this list, you use the DROP= data set option to remove it from the list.

Using a DATA Step to Check for Invalid Values

Your next task is to use a DATA step to identify invalid data values and to determine where they occur in the raw data file (by listing the patient number).

This time, DATA step processing is used to identify invalid character values for selected variables. As before, you will check Gender, Dx, and AE. Several different methods are used to identify these values.

First, you can write a simple DATA step that reports invalid data values by using PUT statements in a DATA _NULL_ step. Here is the program.

Program 1-4 Using a DATA _NULL_ Step to Detect Invalid Character Data

```
title "Listing of invalid patient numbers and data values";
data _null_;
   set clean.patients;
   file print; ***send output to the output window;
   ***check Gender;
   if Gender not in ('F' 'M' ' ') then put Patno= Gender=;
   ***check Dx;
   if verify(trim(Dx),'0123456789') and not missing(Dx)
      then put Patno= Dx=;
   /*******************************************
   SAS 9 alternative:
   if notdigit(trim(Dx)) and not missing(Dx)
      then put Patno= Dx=;
   *******************************************/

   ***check AE;
   if AE not in ('0' '1' ' ') then put Patno= AE=;
run;
```

Before discussing the output, let's spend a moment looking over the program. First, notice the use of the DATA _NULL_ statement. Because the only purpose of this program is to identify invalid data values and print them out, there is no need to create a SAS data set. The reserved data set name _NULL_ tells SAS not to create a data set. This is a major efficiency technique. In this program, you avoid using all the resources to create a data set when one isn't needed.

The FILE PRINT statement causes the results of any subsequent PUT statements to be sent to the Output window (or output device). Without this statement, the results of the PUT statements would be sent to the SAS Log. Gender and AE are checked by using the IN operator. The statement

```
if X in ('A' 'B' 'C') then . . .;
```

is equivalent to

```
if X = 'A' or X = 'B' or X = 'C' then . . .;
```

That is, if X is equal to any of the values in the list following the IN operator, the expression is evaluated as true. You want an error message printed when the value of Gender is not one of the acceptable values ('F', 'M', or missing). Therefore, place a NOT in front of the whole expression, triggering the error report for invalid values of Gender or AE. You can separate the values in the list by spaces or commas. An equivalent statement to the one above is:

```
if X in ('A','B','C') then . . .;
```

There are several alternative ways that the gender checking statement can be written. The method above uses the IN operator.

A straightforward alternative to the IN operator is

```
if not (Gender eq 'F' or Gender eq 'M' or Gender = ' ') then
put Patno= Gender=;
```

Another possibility is

```
if Gender ne 'F' and Gender ne 'M' and Gender ne ' ' then
put Patno= Gender=;
```

While all of these statements checking for Gender and AE produce the same result, the IN operator is probably the easiest to write, especially if there are a large number of possible values to check. Always be sure to consider whether you want to identify missing values as invalid or not. In the statements above, you are allowing missing values as valid codes. If you want to flag missing values as errors, do not include a missing value in the list of valid codes.

If you want to allow lowercase M's and F's as valid values, you can add the single line

```
Gender = upcase(Gender);
```

immediately before the line that checks for invalid gender codes. As you can probably guess, the UPCASE function changes all lowercase letters to uppercase letters.

If you know from the start that you may have both upper- and lowercase values in your raw data file, you could use the $UPCASE informat to convert all lowercase values to uppercase. For example, to read all Gender values in uppercase, you could use:

```
@4 Gender $upcase1.
```

to replace the line that reads Gender values in Program 1-1.

A statement similar to the gender checking statement is used to test the adverse events.

There are so many valid values for Dx (any numeral from 1 to 999) that the approach you used for Gender and AE would be inefficient (and wear you out typing) if you used it to check for invalid Dx codes. The VERIFY function is one of the many possible ways you can check to see if there is a value other than the numerals 0 to 9 as a Dx value. The next section describes the VERIFY function along with several other functions.

Describing the VERIFY, TRIM, MISSING, and NOTDIGIT Functions

The verify function takes the form:

```
verify(character_variable,verify_string)
```

where *verify_string* is a character value (either the name of a character variable or a series of values placed in single or double quotes). The VERIFY function returns the first position in the *character_variable* that contains a character that is not in the *verify_string*. If the *character_variable* does not contain any invalid values, the VERIFY function returns a 0. To make this clearer, let's look at some examples of the VERIFY function.

Suppose you have a variable called ID that is stored in five bytes and is supposed to contain only the letters X, Y, Z, and digits 0 through 5. For example, valid values for ID would be X1234 or 34Z5X. You could use the VERIFY function to see if the variable ID contained any characters other than X, Y, Z and the digits 0 through 5 like this:

```
Position = verify(ID,'XYZ012345');
```

Suppose you had an ID value of X12B44. The value of Position in the line above would be 4, the position of the first invalid character in ID (the letter B). If no invalid characters are found, the VERIFY function returns a 0. Therefore, you can write an expression like the following to list invalid values of ID:

```
if verify(ID,'XYZ012345') then put "Invalid value of ID:" ID;
```

This may look strange to you. You might prefer the statement:

```
if verify(ID,'XYZ012345') gt 0 then put "Invalid value of ID:" ID;
```

However, these two statements are equivalent. Any numerical value in SAS other than 0 or missing is considered TRUE. You usually think of true and false values as 1 or 0—and that is what SAS returns to you when it evaluates an expression. However, it is often convenient to use values other than 1 to represent TRUE. When SAS evaluates the VERIFY function in either of the two statements above, it returns a 4 (the position of the first invalid character in the ID). Since 4 is neither 0 or missing, SAS interprets it as TRUE and the PUT statement is executed.

There is one more potential problem when using the VERIFY function. Suppose you had an ID equal to 'X123'. What would the expression

```
verify(ID,'XYZ012345')
```

return? You might think the answer is 0 since you only see valid characters in the ID (X, 1, 2, and 3). However, the expression above returns a 5! Why? Because that is the position of the first trailing blank. Since ID is stored in 5 bytes, any ID with fewer than 5 characters will contain trailing blanks—and blanks, even though they are sometimes hard to see, are still considered characters to be tested by the VERIFY function.

To avoid problems with trailing blanks, you can use the TRIM function to remove any trailing blanks before the VERIFY function operates. Therefore, the expression

```
verify(trim(ID),'XYZ012345')
```

will return a 0 for all valid values of ID, even if they are shorter than 5 characters.

There is one more problem to solve. That is, the expression above will return a 1 for a missing value of ID. (Think of character missing values as blanks). The MISSING function is a useful way to test for missing values. It returns a value of TRUE if its argument contains a missing value and a value of FALSE otherwise. And, this function can take character or numeric arguments! The MISSING function has become one of this author's favorites. It makes your SAS programs much more readable. For example, take the line in Program 1-4 that uses the MISSING function:

```
if verify(trim(Dx),'0123456789') and not missing(Dx)
   then put Patno= Dx=;
```

Without the MISSING function, this line would read:

```
if verify(trim(Dx),'0123456789') and Dx ne ' '
   then put Patno= Dx=;
```

If you start using the MISSING function in your SAS programs, you will begin to see statements like the one above as clumsy or even ugly.

You are now ready to understand the VERIFY function that checked for invalid Dx codes. The verify string contained the characters (numerals) 0 through 9. Thus, if the Dx code contains any character other than 0 through 9, it returns the position of this offending character, which would

have to be a 1, 2, or 3 (Dx is three bytes in length), and the error message would be printed. Output from Program 1-4 is shown below:

```
Listing of invalid patient numbers and data values
Patno=002 Dx=X
Patno=003 gender=X
Patno=004 AE=A
Patno=010 gender=f
Patno=013 gender=2
Patno=002 Dx=X
Patno=023 gender=f
```

Note that patient 002 appears twice in this output. This occurs because there is a duplicate observation for patient 002 (in addition to several other purposely included errors), so that the data set can be used for examples later in this book, such as the detection of duplicate ID's and duplicate observations.

If you have SAS 9 or higher, you can use the NOTDIGIT function.

```
notdigit(character_value)
```

is equivalent to

```
verify(character_value,'0123456789')
```

That is, the NOTDIGIT function returns the first position in *character_value* that is not a digit. The NOTDIGIT function treats trailing blanks the same way that the VERIFY function does, so if you have character strings of varying lengths, you may want to use the TRIM function to remove trailing blanks.

Using the NOTDIGIT function, you could replace the VERIFY function in Program 1-4 like this:

```
if notdigit(trim(Dx)) and not missing(Dx)
   then put Patno= Dx=;
```

Suppose you want to check for valid patient numbers (Patno) in a similar manner. However, you want to flag missing values as errors (every patient must have a valid ID). The following statement:

```
if notdigit(trim(Patno)) then put "Invalid ID for PATNO=" Patno;
```

will work in the same way as your check for invalid Dx codes except that missing values will now be listed as errors.

Using PROC PRINT with a WHERE Statement to List Invalid Values

There are several alternative ways to identify the ID's containing invalid data. As with most of the topics in this book, you will see several ways of accomplishing the same task. Why? One reason is that some techniques are better suited to an application. Another reason is to teach some additional SAS programming techniques. Finally, under different circumstances, some techniques may be more efficient than others.

One very easy alternative way to list the subjects with invalid data is to use PROC PRINT followed by a WHERE statement. Just as you used an IF statement in a DATA step in the previous section, you can use a WHERE statement in a similar manner with PROC PRINT and avoid having to write a DATA step altogether. For example, to list the ID's with invalid GENDER values, you could write a program like the one shown in Program 1-5.

Program 1-5 Using PROC PRINT to List Invalid Character Values

```
title "Listing of invalid gender values";
proc print data=clean.patients;
   where Gender not in ('M' 'F' ' ');
   id Patno;
   var Gender;
run;
```

It's easy to forget that WHERE statements can be used within SAS procedures. SAS programmers who have been at it for a long time (like the author) often write a short DATA step first and use PUT statements or create a temporary SAS data set and follow it with a PROC PRINT. The program above is both shorter and more efficient than a DATA step followed by a PROC PRINT. However, the WHERE statement does require that all variables already exist in the data set being processed. DATA _NULL_ steps, however, tend to be fairly efficient and are a reasonable alternative as well as the more flexible approach.

The output from Program 1-5 follows.

```
Listing of invalid gender values

Patno    gender

 003       X
 010       f
 013       2
 023       f
```

This program can be extended to list invalid values for all the character variables. You simply add the other invalid conditions to the WHERE statement as shown in Program 1-6.

Program 1-6 Using PROC PRINT to List Invalid Character Data for Several Variables

```
title "Listing of invalid character values";
proc print data=clean.patients;
   where Gender not in ('M' 'F' ' ')           or
         notdigit(trim(Dx)) and not missing(Dx) or
         AE not in ('0' '1' ' ');
   id Patno;
   var Gender Dx AE;
run;
```

The resulting output is shown next.

```
Listing of invalid character values

Patno    Gender    Dx    AE

 002       F       X     0
 003       X       3     1
 004       F       5     A
 010       f       1     0
 013       2       1
 002       F       X     0
 023       f             0
```

Notice that this output is not as informative as the one produced by the DATA _NULL_ step in Program 1-4. It lists all the patient numbers, genders, Dx codes, and adverse events even when only one of the variables has an error (patient 002, for example). So, there is a trade-off—the simpler program produces slightly less desirable output. We could get philosophical and extend this concept to life in general, but that's for some other book.

Using Formats to Check for Invalid Values

Another way to check for invalid values of a character variable from raw data is to use user-defined formats. There are several possibilities here. One, you can create a format that leaves all valid character values as is and formats all invalid values to a single error code. Let's start out with a program that simply assigns formats to the character variables and uses PROC FREQ to list the number of valid and invalid codes. Following that, you will extend the program by using a DATA step to identify which ID's have invalid values. Program 1-7 uses formats to convert all invalid data values to a single value.

Program 1-7 Using a User-Defined Format and PROC FREQ to List Invalid Data Values

```
proc format;
   value $gender 'F','M' = 'Valid'
                 ' '     = 'Missing'
                 other   = 'Miscoded';
```

```
   value $ae '0','1' = 'Valid'
             ' '     = 'Missing'
              other  = 'Miscoded';
run;

title "Using formats to identify invalid values";
proc freq data=clean.patients;
   format Gender $gender.
          AE     $ae.;
   tables Gender AE/ nocum nopercent missing;
run;
```

For the variables GENDER and AE, which have specific valid values, you list each of the valid values in the range to the left of the equal sign in the VALUE statement. Format each of these values with the value 'Valid'.

You may choose to combine the missing value with the valid values if that is appropriate, or you may want to keep track of missing values separately as was done here. Finally, any value other than the valid values or a missing value will be formatted as 'Miscoded'. All that is left is to run PROC FREQ to count the number of 'Valid', 'Missing', and 'Miscoded' values. The TABLES option MISSING causes the missing values to be listed in the body of the PROC FREQ output. (Important note: When you use the MISSING TABLES option with PROC FREQ and you are outputting percentages, the percentages are computed by dividing the number of a particular value by the total number of observations, missing or non-missing.) Here is the output from PROC FREQ.

```
Using formats to identify invalid values

The FREQ Procedure

       Gender

Gender      Frequency
_____________________
Missing             1
Miscoded            4
Valid              26


   Adverse Event?

AE          Frequency
_____________________
Missing             1
Valid              29
Miscoded            1
```

This output isn't particularly useful. It doesn't tell you which observations (patient numbers) contain missing or invalid values. Let's modify the program by adding a DATA step, so that ID's with invalid character values are listed.

Program 1-8 Using a User-Defined Format and a DATA Step to List Invalid Data Values

```
proc format;
   value $gender 'F','M' = 'Valid'
                 ' '     = 'Missing'
                 other   = 'Miscoded';

   value $ae '0','1' = 'Valid'
             ' '     = 'Missing'
              other  = 'Miscoded';
run;

title "Listing of invalid patient numbers and data values";
data _null_;
   set clean.patients(keep=Patno Gender AE);
   file print; ***Send output to the output window;
```

```
   if put(Gender,$gender.) = 'Miscoded' then put Patno= Gender=;
   if put(AE,$ae.) = 'Miscoded' then put Patno= AE=;
run;
```

The "heart" of this program is the PUT function. To review, the PUT function is similar to the INPUT function. It takes the following form:

```
character_variable = put(variable, format)
```

where *character_variable* is a character variable that contains the value of the variable listed as the first argument to the function, formatted by the *format* listed as the second argument to the function. The result of a PUT function is always a character variable, and the function is frequently used to perform numeric-to-character conversions. In Program 1-8, the first argument of the PUT function is a character variable you want to test and the second argument is the corresponding character format. The result of the PUT function for any invalid data values would be the value 'Miscoded'.

Here is the output from Program 1-8.

```
Listing of invalid patient numbers and data values
Patno=003 gender=X
Patno=004 AE=A
Patno=010 gender=f
Patno=013 gender=2
Patno=023 gender=f
```

Using Informats to Remove Invalid Values

PROC FORMAT is also used to create informats. Remember that formats are used to control how variables look in output or how they are classified by such procedures as PROC FREQ. Informats modify the value of variables as they are read from the raw data, or they can be used with an INPUT function to create new variables in the DATA step. User-defined informats are created in much the same way as user-defined formats. Instead of a VALUE statement that creates formats, an INVALUE statement is used to create informats. The only difference between the two is that informat names can only be 31 characters in length. (Note: For those curious readers, the reason is that informats and formats are both stored in the same catalog and an "@" is placed before informats to distinguish them from formats.) The following is a program that changes invalid values for GENDER and AE to missing values by using a user-defined informat.

Program 1-9 Using a User-Defined Informat to Set Invalid Data Values to Missing

```
*-----------------------------------------------------------------*
| Purpose: To create a SAS data set called PATIENTS2              |
|          and set any invalid values for Gender and AE to        |
|          missing, using a user-defined informat                 |
*-----------------------------------------------------------------*;
libname clean "c:\books\clean";

proc format;
   invalue $gen    'F','M' = _same_
                   other   = ' ';
   invalue $ae    '0','1' = _same_
                   other  = ' ';
run;

data clean.patients_filtered;
   infile "c:\books\clean\patients.txt" pad;
   input @1  Patno    $3.
         @4  Gender   $gen1.
         @27 AE       $ae1.;

   label Patno   = "Patient Number"
         Gender  = "Gender"
         AE      = "adverse event?";
run;

title "Listing of data set PATIENTS_FILTERED";
proc print data=clean.patients_filtered;
   var Patno Gender AE;
run;
```

Notice the INVALUE statements in the PROC FORMAT above. The keyword _SAME_ is a SAS reserved value that does what its name implies—it leaves any of the values listed in the range specification unchanged. The keyword OTHER in the subsequent line refers to any values not matching one of the previous ranges. Notice also that the informats in the INPUT statement use the user-defined informat name followed by the number of columns to be read, the same method that is used with predefined SAS informats.

Output from the PROC PRINT is shown next.

Listing of data set PATIENTS_FILTERED

Obs	Patno	Gender	AE
1	001	M	0
2	002	F	0
3	003		1
4	004	F	
5	XX5	M	0
6	006		1
7	007	M	0
8		M	0
9	008	F	0
10	009	M	1
11	010		0
12	011	M	1
13	012	M	0
14	013		
15	014	M	1
16	002	F	0
17	003	M	0
18	015	F	1
19	017	F	0
20	019	M	0
21	123	M	0
22	321	F	1
23	020	F	0
24	022	M	1
25	023		0
26	024	F	0
27	025	M	1
28	027	F	0
29	028	F	0
30	029	M	1
31	006	F	0

Notice that invalid values for GENDER and AE are now missing values, including the two lowercase 'f's (patient numbers 010 and 023).

Let's add one more feature to this program. By using the keyword UPCASE in the informat specification, you can automatically convert the values being read to uppercase before the ranges are checked. Here are the PROC FORMAT statements, rewritten to use this option.

```
proc format;
   invalue $gen (upcase)  'F' = 'F'
                          'M' = 'M'
                        other = ' ';
   invalue $ae '0','1' = _same_
               other  = ' ';
run;
```

The UPCASE option is placed in parenthesis following the informat name. Notice some other changes as well. You cannot use the keyword _SAME_ anymore because the value is changed to uppercase for comparison purposes, but the _SAME_ specification would leave the original lowercase value unchanged. By specifying each value individually, the lowercase 'f' (the only lowercase GENDER value) would match the range 'F' and be assigned the value of an uppercase 'F'.

The output of this data set is identical to the output for Program 1-9 except the value of GENDER for patients 010 and 023 is an uppercase 'F'.

If you want to preserve the original value of the variable, you can use a user-defined informat with an INPUT function instead of an INPUT statement. You can use this method to check a raw data file or a SAS data set. Program 1-10 reads the SAS data set CLEAN.PATIENTS and uses user-defined informats to detect errors.

Program 1-10 Using a User-Defined Informat with the INPUT Function

```
proc format;
   invalue $gender 'F','M' = _same_
                   other  = 'Error';
   invalue $ae     '0','1' = _same_
                   other   = 'Error';
run;

data _null_;
   file print;
   set clean.patients;
```

```
   if input (Gender,$gender.) = 'Error' then
      put @1 "Error for Gender for patient:" Patno" value is " Gender;
   if input (AE,$ae.) = 'Error' then
      put @1 "Error for AE for patient:" Patno" value is " AE;
run;
```

The advantage of this program over Program 1-9 is that the original values of the variables are not lost.

Output from Program 1-10 is shown below:

```
Listing of invalid character values
Error for Gender for patient:003  value is X
Error for AE for patient:004  value is A
Error for Gender for patient:006  value is
Error for Gender for patient:010  value is f
Error for Gender for patient:013  value is 2
Error for AE for patient:013  value is
Error for Gender for patient:023  value is f
```

2 Checking Values of Numeric Variables

Introduction 23
Using PROC MEANS, PROC TABULATE, and PROC UNIVARIATE to Look for Outliers 24
Using an ODS SELECT Statement to List Extreme Values 34
Using PROC UNIVARIATE Options to List More Extreme Observations 35
Using PROC UNIVARIATE to Look for Highest and Lowest Values by Percentage 37
Using PROC RANK to Look for Highest and Lowest Values by Percentage 43
Presenting a Program to List the Highest and Lowest Ten Values 47
Presenting a Macro to List the Highest and Lowest "n" Values 50
Using PROC PRINT with a WHERE Statement to List Invalid Data Values 52
Using a DATA Step to Check for Out-of-Range Values 54
Identifying Invalid Values versus Missing Values 55
Listing Invalid (Character) Values in the Error Report 57
Creating a Macro for Range Checking 60
Checking Ranges for Several Variables 62
Using Formats to Check for Invalid Values 66
Using Informats to Filter Invalid Values 68
Checking a Range Using an Algorithm Based on Standard Deviation 71
Detecting Outliers Based on a Trimmed Mean and Standard Deviation 73
Presenting a Macro Based on Trimmed Statistics 76
Using the TRIM Option of PROC UNIVARIATE and ODS to Compute Trimmed Statistics 80
Checking a Range Based on the Interquartile Range 86

Introduction

The techniques for checking invalid numeric data are quite different from the techniques for checking invalid character data that you saw in the last chapter. Although there are usually many different values a numeric variable can take on, there are several techniques that you can use to help identify data errors.

One of the most basic techniques is to examine the distribution of each variable, possibly using a graphical approach (histogram, stem-and-leaf plot, etc.). Another approach is to look at some of the highest and lowest values of each variable. For certain variables, it is possible to determine upper and lower limits, and any value outside that range should be examined. Finally, there are a variety of techniques that look at the distribution of data values and determine possible errors. You may see that most of the data values fall within a certain range and values beyond this range can be examined. This chapter develops programs based on these ideas.

Using PROC MEANS, PROC TABULATE, and PROC UNIVARIATE to Look for Outliers

One of the simplest ways to check for invalid numeric values is to run either PROC MEANS or PROC UNIVARIATE. By default, PROC MEANS lists the minimum and maximum values, along with the n, mean, and standard deviation. PROC UNIVARIATE is somewhat more useful in detecting invalid values, because it provides you with a listing of the five highest and five lowest values (this number can be changed with options), along with graphical output (stem-and-leaf plots and box plots). Let's first look at how you can use PROC MEANS for very simple checking of numeric variables. The program below checks the three numeric variables, heart rate (HR), systolic blood pressure (SBP), and diastolic blood pressure (DBP), in the PATIENTS data set.

Program 2-1 Using PROC MEANS to Detect Invalid and Missing Values

```
libname clean "c:\books\clean";

title "Checking numeric variables in the patients data set";
proc means data=clean.patients n nmiss min max maxdec=3;
   var HR SBP DBP;
run;
```

This program used the PROC MEANS options N, NMISS, MIN, MAX, and MAXDEC=3. The N and NMISS options report the number of non-missing and missing observations for each variable, respectively. The MIN and MAX options list the smallest and largest non-missing values for each variable. The MAXDEC=3 option is used so that the minimum and maximum values will be printed to three decimal places. Because HR, SBP, and DBP are supposed to be integers, you might have thought to set the MAXDEC option to 0. However, you might want to catch any data errors where a decimal point was entered by mistake.

Here is the output from Program 2-1.

```
Checking numeric variables in the patients data set

The MEANS Procedure

                                          N
Variable  Label                        N  Miss       Minimum       Maximum
------------------------------------------------------------------------
Hr        Heart Rate                  28     3        10.000       900.000
SBP       Systolic Blood Pressure     27     4        20.000       400.000
DBP       Diastolic Blood Pressure    28     3         8.000       200.000
```

This output is of limited use. It does show the number of non-missing and missing observations along with the highest and lowest values. Inspection of the minimum and maximum values for all three variables shows that there are probably some data errors in the PATIENTS data set. If you want a slightly prettier output, you can use PROC TABULATE to accomplish the same task. For an excellent reference on PROC TABULATE, let me suggest a book written by Lauren E. Haworth, called *PROC TABULATE by Example*, published by SAS Institute Inc., Cary, NC, as part of the SAS Press Series.

Here is the equivalent PROC TABULATE program, followed by the output. (Assume that the libref CLEAN has been previously defined in this program and in any future programs where it is not included in the program.)

Program 2-2 Using PROC TABULATE to Display Descriptive Data

```
title "Statistics for numeric variables";
proc tabulate data=clean.patients format=7.3;
   var HR SBP DBP;
   tables HR SBP DBP,
          n*f=7.0 nmiss*f=7.0 mean min max / rtspace=18;
   keylabel n     = 'Number'
            nmiss = 'Missing'
            mean  = 'Mean'
            min   = 'Lowest'
            max   = 'Highest';
run;
```

The option FORMAT=7.3 tells the procedure to use this format (a field width of 7 with 3 places to the right of the decimal point) for all the output, unless otherwise specified. The analysis variables HR, SBP, and DBP are listed in a VAR statement. Let's place these variables on the row dimension and the statistics along the column dimension. The TABLE option RTSPACE=18 allows for 18 spaces for all row labels, including the spaces for the lines forming the table. In addition, the format 7.0 is to be used for N and NMISS in the table. Finally, the KEYLABEL statement replaces the keywords for the selected statistics with more meaningful labels. Below is the output from PROC TABULATE.

Statistics for numeric variables

	Number	Missing	Mean	Lowest	Highest
Heart Rate	28	3	107.393	10.000	900.000
Systolic Blood Pressure	27	4	144.519	20.000	400.000
Diastolic Blood Pressure	28	3	88.071	8.000	200.000

A more useful procedure might be PROC UNIVARIATE. Running this procedure for your numeric variables yields much more information.

Program 2-3 Using PROC UNIVARIATE to Look for Outliers

```
title "Using PROC UNIVARIATE to Look for Outliers";
proc univariate data=clean.patients plot;
   id Patno;
   var HR SBP DBP;
run;
```

The procedure option PLOT provides you with several graphical displays of the data; a stem-and-leaf plot, a box plot, and a normal probability plot. Output from this procedure is shown next. (Note: To save some space, only the output for the variables HR and DBP is shown.)

```
Using PROC UNIVARIATE to Look for Outliers

The UNIVARIATE Procedure
Variable:  Hr  (Heart Rate)

                          Moments

N                          28    Sum Weights                28
Mean                107.392857    Sum Observations          3007
Std Deviation       161.086436    Variance            25948.8399
Skewness            4.73965876    Kurtosis            23.7861582
Uncorrected SS         1023549    Corrected SS        700618.679
Coeff Variation     149.997347    Std Error Mean       30.442475


              Basic Statistical Measures

    Location                    Variability

Mean     107.3929     Std Deviation          161.08644
Median    74.0000     Variance                   25949
Mode      68.0000     Range                  890.00000
                      Interquartile Range     27.00000


           Tests for Location: Mu0=0

Test           -Statistic-    -----p Value------

Student's t    t  3.527731    Pr > |t|    0.0015
Sign           M        14    Pr >= |M|   <.0001
Signed Rank    S       203    Pr >= |S|   <.0001
```

(*continued*)

```
Quantiles (Definition 5)

Quantile      Estimate

100% Max           900
99%                900
95%                210
90%                208
75% Q3              87
50% Median          74
25% Q1              60
10%                 22
5%                  22
1%                  10
0% Min              10

Using PROC UNIVARIATE to Look for Outliers

The UNIVARIATE Procedure
Variable:  Hr  (Heart Rate)

                Extreme Observations

--------Lowest--------        --------Highest-------

Value   Patno      Obs        Value   Patno      Obs

   10   020         23           90                8
   22   023         25          101   004          4
   22   014         15          208   017         19
   48   022         24          210   008          9
   58   019         20          900   321         22
```

(*continued*)

```
              Missing Values

                      -----Percent Of-----
Missing                             Missing
  Value       Count     All Obs         Obs

      .           3        9.68      100.00

   Stem Leaf                          #  Boxplot
      9 0                             1     *
      8
      7
      6
      5
      4
      3
      2 11                            2     *
      1 0                             1     +
      0 12256666777777888889999      24  +--0--+
        ----+----+----+----+----
    Multiply Stem.Leaf by 10**+2


Using PROC UNIVARIATE to Look for Outliers

The UNIVARIATE Procedure
Variable:  Hr  (Heart Rate)

                      Normal Probability Plot
     950+                                          *
        |
        |
     650+
        |                                                 +
        |                                            ++++++
     350+                                      +++++++
        |                                  ++++++ *  *
        |                             ++++++     *
      50+     *   *    **  *+**+***** * **  *
         +----+----+----+----+----+----+----+----+----+----+
             -2        -1         0        +1        +2
```

(*continued*)

```
Using PROC UNIVARIATE to Look for Outliers

The UNIVARIATE Procedure
Variable:  DBP  (Diastolic Blood Pressure)

                          Moments

N                          28    Sum Weights                28
Mean                88.0714286    Sum Observations         2466
Std Deviation       37.2915724    Variance            1390.66138
Skewness            1.06190956    Kurtosis            3.67139184
Uncorrected SS          254732    Corrected SS        37547.8571
Coeff Variation      42.342418    Std Error Mean      7.04744476

              Basic Statistical Measures

    Location                    Variability

Mean     88.07143     Std Deviation           37.29157
Median   81.00000     Variance                    1391
Mode     78.00000     Range                  192.00000
                      Interquartile Range     26.00000

NOTE: The mode displayed is the smallest of 2 modes with a count of 3.

           Tests for Location: Mu0=0

Test           -Statistic-    -----p Value------

Student's t    t  12.49693    Pr > |t|    <.0001
Sign           M        14    Pr >= |M|   <.0001
Signed Rank    S       203    Pr >= |S|   <.0001

Quantiles (Definition 5)

Quantile      Estimate

100% Max           200
99%                200
```

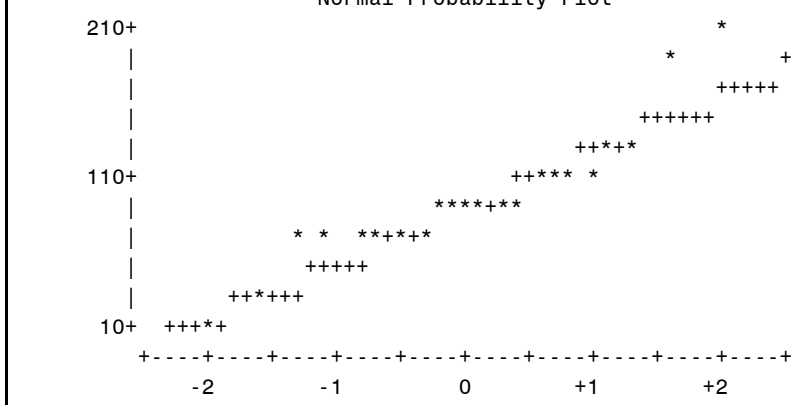
(*continued*)

```
75% Q3             100
50% Median          81
25% Q1              74
10%                 64
5%                  20
1%                   8
0% Min               8

Using PROC UNIVARIATE to Look for Outliers

The UNIVARIATE Procedure
Variable:  DBP  (Diastolic Blood Pressure)

               Extreme Observations

--------Lowest--------        --------Highest-------

Value   Patno      Obs        Value   Patno      Obs

    8   020         23          106   027         28
   20   011         12          120   004          4
   64   013         14          120   010         11
   68   025         27          180   009         10
   68   006          6          200   321         22


              Missing Values

                       -----Percent Of-----
Missing                            Missing
  Value       Count     All Obs         Obs

      .           3        9.68      100.00

   Stem Leaf                     #  Boxplot
     20 0                        1     *
     18 0                        1     *
     16
     14
     12 00                       2     |
     10 0026                     4  +-----+
      8 000244800                9  *--+--*
```

(*continued*)

```
      6 488044888               9  +-----+
      4
      2 0                       1     0
      0 8                       1     0
        ----+----+----+----+
```

```
    Multiply Stem.Leaf by 10**+1
Using PROC UNIVARIATE to Look for Outliers

The UNIVARIATE Procedure
Variable:  DBP  (Diastolic Blood Pressure)
```



You certainly get lots of information from PROC UNIVARIATE, perhaps too much information. Starting off, you see some descriptive univariate statistics (hence the procedure name) for each of the variables listed in the VAR statement. Most of these statistics are not very useful in the data checking operation. The number of non-missing observations (N), the number of observations not equal to zero (Num ^= 0), and the number of observations greater than zero (Num > 0) are probably the only items that are of interest to you at this time.

One of the most important sections of the PROC UNIVARIATE output, for data checking purposes, is the section labeled "Extreme Observations." Here you see the five lowest and five highest values for each of your variables. For example, for the variable HR (Heart Rate), there are three possible data errors under the column label "Lowest" (10, 22, and 22) and three possible data errors under the column label "Highest" (208, 210, and 900). Obviously, having knowledge of reasonable values for each of your variables is essential if this information is to be of any use. Next to the listing of the highest and lowest values is the patient number (Patno) value. This

column is listed because of the ID statement included in PROC UNIVARIATE. A column listing the observation number (Obs) is also printed and is rarely (if ever) useful.

The section that follows contains a stem-and-leaf plot and a box plot. These two data visualizations come from an area of statistics known as exploratory data analysis (EDA). (For an excellent reference see *Exploratory Data Analysis*, by John Tukey, Reading, Massachusetts: Addison-Wesley.) Let's focus on the plots for the variable DBP (diastolic blood pressure). The stem-and-leaf plot can be thought of as a sideways histogram. For this variable, the diastolic blood pressures are grouped in 20-point intervals. For example, the stem labeled "8" represents the values from 80 to 99. Instead of simply placing X's or some other symbol to represent the bar in this sideways histogram, the next digit in the value is used instead. Thus, you see that there were three values of 80, one 82, two 84's, one 88, and two 90's. You can ignore these values and just think of the stem-and-leaf plot as a histogram, or you might be interested in the additional information that the leaf values give you. We should also point out that if any interval contains more than 49 values, PROC UNIVARIATE will automatically switch to a horizontal bar chart instead of the stem-and-leaf plot. Finally, you may want to use a HISTOGRAM statement to produce a traditional histogram instead of the stem-and-leaf plot and box plot.

A quick examination of this plot shows that there were some abnormally low and high diastolic blood pressure values. This useful information complements the "Extreme Observations" information. The "Extreme Observations" only lists the five highest and five lowest values; the stem-and-leaf plot shows all the values in your data set.

To the right of the stem-and-leaf plot is a box plot. This plot shows the mean (+ sign), the median (the dashed line between the two asterisks), and the 25^{th} and 75^{th} percentiles (the bottom and top of the box, respectively). The distance between the top and bottom of the box is called the interquartile range and can also be found earlier in the output labeled as "Q1-Q3." Extending from both the top and bottom of the box are whiskers and outliers. The whiskers represent data values within one-and-a-half interquartile ranges above or below the box. (Note: The EDA people call the top and bottom of the box the hinges.) Any data values more than one-and-a-half but less than three interquartile ranges above or below the box (hinges) are represented by 0's. Two data values for DBP (8 and 20) fit this description in the box plot on page 32. Finally, any data values more than three interquartile ranges above or below the top and bottom hinges are represented by asterisks. For your DBP variable, two data points, 180 and 200, fit this description.

For those readers who are big fans of box plots, PROC BOXPLOT can be used to produce much nicer looking box plots using graphics characters instead of the somewhat crude box plots produced by PROC UNIVARIATE. Please refer to the SAS online documentation at support.sas.com for more details on this procedure.

The final graph, called a normal probability plot, is of interest to statisticians and helps determine deviations from a theoretical distribution called a normal or Gaussian distribution. The information displayed in the normal probability plot may not be useful for your data cleaning task because you are looking for data errors and are not particularly interested if the data are normally distributed or not.

Using an ODS SELECT Statement to List Extreme Values

You can use an ODS (output delivery system) statement before PROC UNIVARIATE to limit the output to only the section on extreme values. You use an ODS SELECT statement to do this. The following program prints only the extreme values for the three variables in question.

Program 2-4 Using an ODS SELECT Statement to Print Only Extreme Observations

```
ods select extremeobs;

title "Using proc univariate to look for outliers";
proc univariate data=clean.patients;
   id Patno;
   var HR SBP DBP;
run;
```

In this example, the name of the output object is extremeobs. If you want to obtain the names of the output objects from any SAS procedure, you add the statement:

```
ods trace on;
```

before the procedure and

```
ods trace off;
```

after the procedure. This results in a list of all the output objects printed to the SAS Log. You can then match up the names with the SAS output. You can modify the trace on statement to read:

```
ods trace on / listing;
```

and the names of the output objects will be printed to the output device, along with the normal output. This makes it easier to match up the output object names with the various portions of the output.

The section of output showing the "Extreme Observations" for the variable heart rate (HR) follows:

```
Using proc univariate to look for outliers

The UNIVARIATE Procedure
Variable:  Hr  (Heart Rate)

                Extreme Observations

--------Lowest--------        --------Highest-------

Value   Patno      Obs        Value   Patno      Obs

   10   020         23           90                8
   22   023         25          101   004          4
   22   014         15          208   017         19
   48   022         24          210   008          9
   58   019         20          900   321         22
```

Using PROC UNIVARIATE Options to List More Extreme Observations

There are two PROC UNIVARIATE options, NEXTROBS= and NEXTRVALS=, that allow you to have the procedure list more (or possibly fewer) extreme observations than the default number (five highest and lowest). The option NEXTROBS=n allows you to specify how many extreme observations you would like to see in the output. For example, the statements below will print the 10 highest and lowest values for the variable DBP:

Program 2-5 Using the NEXTROBS= Option to Print the 10 Highest and Lowest Observations

```
ODS select extremeobs;
title "The 10 highest and lowest observations for DBP";
proc univariate data=clean.patients nextrobs=10;
   id Patno;
   var DBP;
run;
```

Running this program results in the listing below:

```
The 10 highest and lowest observations for DBP

The UNIVARIATE Procedure
Variable:  DBP  (Diastolic Blood Pressure)

                Extreme Observations

--------Lowest--------        --------Highest-------

Value   Patno      Obs        Value   Patno      Obs

    8   020         23           90   014         15
   20   011         12           90   028         29
   64   013         14          100   003          3
   68   025         27          100                8
   68   006          6          102   007          7
   70   019         20          106   027         28
   74   003         17          120   004          4
   74   012         13          120   010         11
   78   023         25          180   009         10
   78   002         16          200   321         22
```

A similar option, NEXTRVALS=n lists the n highest and lowest unique values. To see how this works, compared to the NEXTROBS option, look at the result of changing the option NEXTROBS to NEXTRVALS in Program 2-5.

Program 2-6 Using the NEXTRVALS= Option to Print the 10 Highest and Lowest Values

```
ODS select extremevalues;

title "The 10 highest and lowest values for DBP";
proc univariate data=clean.patients nextrvals=10;
   id Patno;
   var DBP;
run;
```

Here is the output:

```
The 10 highest and lowest values for DBP

The UNIVARIATE Procedure
Variable:  DBP  (Diastolic Blood Pressure)

                  Extreme Values

---------Lowest--------   --------Highest--------

 Order   Value     Freq    Order   Value     Freq

     1       8        1        9      82        1
     2      20        1       10      84        2
     3      64        1       11      88        1
     4      68        2       12      90        2
     5      70        1       13     100        2
     6      74        2       14     102        1
     7      78        3       15     106        1
     8      80        3       16     120        2
     9      82        1       17     180        1
    10      84        2       18     200        1
```

Notice the difference. The NEXTRVALS options lists the 10 highest and lowest unique values in the data set, while NEXTROBS lists the 10 highest and lowest values (even if some values appear more than once). An additional column (Freq) in the listing above indicates the number of times each of the unique values occur.

Using PROC UNIVARIATE to Look for Highest and Lowest Values by Percentage

In the previous section, you used PROC UNIVARIATE with options NEXTROBS or NEXTRVALS, along with an ODS SELECT statement to list the lowest and highest values for a variable in your data set.

Instead of looking at the "n" highest and lowest values, you may want to see the top and bottom "n" percent of the data values. There are two quite different approaches to this problem. One uses PROC UNIVARIATE to output the cutoff values for the desired percentiles; the other uses PROC RANK to divide the data set into groups. Here is the PROC UNIVARIATE approach.

Program 2-7 Using PROC UNIVARIATE to Print the Top and Bottom "n" Percent of Data Values

```
libname clean "c:\books\clean";
proc univariate data=clean.patients noprint;
   var HR;
   id Patno;
   output out=tmp pctlpts=10 90 pctlpre = L_;  ❶
run;

data hilo;
   set clean.patients(keep=Patno HR);  ❷
   ***Bring in upper and lower cutoffs for variable;
   if _n_ = 1 then set tmp;  ❸
   if HR le L_10 and not missing(HR) then do;
      Range = 'Low ';
      output;
   end;
   else if HR ge L_90 then do;
      Range = 'High';
      output;
   end;
run;

proc sort data=hilo;
   by HR;  ❹
run;
title "Top and Bottom 10% for Variable HR";
proc print data=hilo;
   id Patno;
   var Range HR;
run;
```

PROC UNIVARIATE can be used to create an output data set containing information that is normally printed out by the procedure. Because you only want the output data set and not the listing from the procedure, use the NOPRINT procedure option. As you did before, you are supplying PROC UNIVARIATE with an ID statement so that the ID variable (Patno in this case) will be included in the output data set. Line ❶ defines the name of the output data set and specifies the information you want it to include. The keyword OUT= names your data set (TMP) and PCTLPTS= instructs the program to create two variables; one to hold the value of HR at the 10^{th} percentile and the other for the 90^{th} percentile. In order for this procedure to create the

variable names for these two variables, the keyword PCTLPRE= (percentile prefix) is used. PROC UNIVARIATE combines this prefix with the percentile values to create variable names. In this program, since you set the prefix to L_, two variables, L_10 and L_90, will hold the cutoff values at the 10^{th} and 90^{th} percentile, respectively.

Data set TMP contains only one observation and three variables: Patno (because of the ID statement), L_10, and L_90. The value of L_10 is 22 (10% of the HR values are below this value) and the value of L_90 is 208 (90% of the HR values are below this value).

You want to add the two values of L_10 and L_90 to every observation in the original PATIENTS data set. The SET statement in line ❷ brings in an observation from the PATIENTS data set, keeping only the variables Patno and HR. Line ❸ is executed only on the first iteration of this DATA step (when _N_ is equal to 1). Because all variables brought in with a SET statement are automatically retained, the values for L_10 and L_90 are added to every observation in the data set HILO.

Finally, for each observation coming in from the PATIENTS data set, the value of HR is compared to the lower and upper cutoff points defined by L_10 and L_90. If the value of HR is at or below the value of L_10 (and not missing), Range is set to the value 'Low' and the observation is added to the data set HILO. Likewise, if the value of HR is at or above the value of L_90, Range is set to 'High' and the observation is added to the data set HILO. Before you print out the contents of the data set HILO, you sort it first ❹ so that the values are in order from low to high. The final listing from this program is shown next.

```
Top and Bottom 10% for Variable HR

Patno    Range     HR

 020     Low       10
 014     Low       22
 023     Low       22
 017     High     208
 008     High     210
 321     High     900
```

The next program turns Program 2-7 into a macro.

Program 2-8 Creating a Macro to List the Highest and Lowest "n" Percent of the Data Using PROC UNIVARIATE

```
*-------------------------------------------------------------*
| Program Name: HILOWPER.SAS  in c:\books\clean               |
| Purpose: To list the n percent highest and lowest values for  |
|          a selected variable.                               |
| Arguments: Dsn    - Data set name                           |
|            Var     - Numeric variable to test               |
|            Percent - Upper and Lower percentile cutoff      |
|            Idvar   - ID variable to print in the report     |
| Example: %hilowper(Dsn=clean.patients,                      |
|                    Var=SBP,                                 |
|                    Perecent=10,                             |
|                    Idvar=Patno)                             |
*-------------------------------------------------------------*;

%macro hilowper(Dsn=,     /* Data set name                    */
                Var=,     /* Variable to test                 */
                Percent=, /* Upper and lower percentile cutoff */
                Idvar=    /* ID variable                      */);

   ***Compute upper percentile cutoff;
   %let Up_per = %eval(100 - &Percent);

   proc univariate data=&Dsn noprint;
      var &Var;
      id &Idvar;
      output out=tmp pctlpts=&Percent &Up_per pctlpre = L_;
   run;

   data hilo;
      set &Dsn(keep=&Idvar &Var);
      if _n_ = 1 then set tmp;
      if &Var le L_&percent and not missing(&Var) then do;
         range = 'Low';
         output;
      end;
      else if &Var ge L_&Up_per then do;
         range = 'High';
         output;
      end;
```

```
   run;

   proc sort data=hilo;
      by &Var;
   run;

   title "Low and High Values for Variables";
   proc print data=hilo;
      id &Idvar;
      var Range &Var;
   run;

   proc datasets library=work nolist;
     delete tmp hilo;
   run;
   quit;

%mend hilowper;
```

First, a brief explanation. A macro program is a piece of SAS code where parts of the code are substituted with variable information by the macro processor before the code is processed in the usual way by the SAS compiler. The macro in Program 2-8 is named HILOWPER, and it begins with a %MACRO statement and ends with a %MEND (macro end) statement. The first line of the macro contains the macro name, followed by a list of arguments. This style of listing the arguments is called Named Parameters. You list the name of the parameter, followed by an equal sign and, if you wish, a default value. An alternative way to list calling arguments when defining a macro is with Positional Parameters (where it is the order that is important).

When the macro is called, the macro processor replaces each of these arguments with the values you specify. Then, in the macro program, every macro variable (that is, every variable name preceded by an ampersand (&)) is replaced by the assigned value. For example, if you want to use this macro to look for the top and bottom 10 percent of the values in your data, in the PATIENTS data set, you would call the macro like this:

```
%hilowper(Dsn=clean.patients,
          Var=SBP,
          Perecent=10,
          Idvar=Patno)
```

The macro processor will substitute these calling arguments for the &variables in the macro program. For example, the SET statement will become:

```
set clean.patients(keep=Patno SBP);
```

&Dsn was replaced by CLEAN.PATIENTS, &Idvar was replaced by Patno, and &Var was replaced by SBP. To be sure this concept is clear (and to help you understand how the macro processor works), you can call the macro with the MPRINT option turned on. This option lists the macro-generated code in the SAS Log. Here is part of the SAS Log containing the macro-generated statements when the macro is called with the above arguments:

```
456  %hilowper(Dsn=clean.patients,
457            Var=SBP,
458            Percent=10,
459            Idvar=Patno)
MPRINT(HILOWPER):   ***Compute upper percentile cutoff;
MPRINT(HILOWPER):   proc univariate data=clean.patients noprint;
MPRINT(HILOWPER):   var SBP;
MPRINT(HILOWPER):   id Patno;
MPRINT(HILOWPER):   output out=tmp pctlpts=10 90 pctlpre = L_;
MPRINT(HILOWPER):   run;

MPRINT(HILOWPER):   data hilo;
MPRINT(HILOWPER):   set clean.patients(keep=Patno SBP);
MPRINT(HILOWPER):   if _n_ = 1 then set tmp;
MPRINT(HILOWPER):   if SBP le L_10 and not missing(SBP) then do;
MPRINT(HILOWPER):   range = 'Low ';
MPRINT(HILOWPER):   output;
MPRINT(HILOWPER):   end;
MPRINT(HILOWPER):   else if SBP ge L_90 then do;
MPRINT(HILOWPER):   range = 'High';
MPRINT(HILOWPER):   output;
MPRINT(HILOWPER):   end;
MPRINT(HILOWPER):   run;
```

Looking at this section of code, you can see how the macro variables have been replaced by their respective calling arguments. By the way, the missing semicolon at the end of the line where the macro is called is not a mistake—you don't need or want it. The reason is that this macro code contains SAS statements that already end in semicolons. Some macros may only generate a portion of a SAS statement, and a semicolon in the middle of a SAS statement will cause an error.

Let's go through this program step by step. The macro variable Percent holds the value of the lower cutoff. You compute the upper percentile cutoff (Up_Per) by subtracting the lower percentile cutoff from 100. (Note: The %EVAL function is needed here to perform the integer arithmetic. If the value of Percent was 10, the value of &Up_Per, without the %EVAL function, would be the text string "100 - 10" instead of 90.)

The output statement is identical to the one in the non-macro version of this program with the percentile values replaced by the corresponding macro variables. The DATA step is likewise similar with macro variables replacing their respective values.

PROC DATASETS is used to delete the temporary data set TMP. This is not really necessary, but it is preferable not to leave temporary data sets behind when you execute a macro. The NOLIST option prevents printing of the directory in the SAS Log.

To demonstrate this macro, the three lines below call the macro to list the highest and lowest 10% of the values for heart rate (HR), systolic blood pressure (SBP), and diastolic blood pressure (DBP) in the data set PATIENTS.

```
%hilowper(Dsn=clean.patients, Var=HR, Percent=10, Idvar=Patno)
%hilowper(Dsn=clean.patients, Var=SBP, Percent=10, Idvar=Patno)
%hilowper(Dsn=clean.patients, Var=DBP, Percent=10, Idvar=Patno)
```

Using PROC RANK to Look for Highest and Lowest Values by Percentage

There is an alternative way to list the highest and lowest "n" percent of the data values, that is, by using PROC RANK. First of all, what are ranks? If you had a variable X with values of 7, 3, 2, and 8, the corresponding ranks would be 3, 2, 1, and 4. That is, the smallest X is rank 1, the next smallest X is rank 2, and so forth.

PROC RANK takes an input data set and produces a new, output data set containing all the variables in the input data set, plus some new variables representing the ranks of variables in your input data set. If you supply a list of variables on a VAR statement and a list of variable names on a RANKS statement, PROC RANK will compute the ranks of every variable on the VAR statement and output the rank values to each of the variable names you list on the RANKS statement. For example, if your input data set is called (INPUT) and it contains the variables Subj, X, Y, and Z, you could produce a new data set (NEW) containing the rank values of X, Y, and Z like this:

```
proc rank data=input out=new;
   var X Y Z;
   ranks RX RY RZ;
run;
```

Beware! If you do not supply a RANKS statement, PROC RANK will replace your original data values with their ranks. It is usually a good idea to use VAR and RANKS statements with this procedure.

PROC RANK has a very useful option (GROUPS=) that allows you to group your data values. When you use this option, PROC RANK will group your data values into the number of groups you request. For example, if you set GROUPS=4, the variable listed on the RANKS statement will now have values of 0, 1, 2, and 3, with those observations in groups 0 being in the bottom quartile and observations in group 3 being in the top quartile.

If you want to see the top and bottom 10 percent of your data values, you have two choices. One way is to set GROUPS=10 (remember the group numbers will go from 0 to 9) and you can select the observations with group values of 0 and 9. Another choice is to set GROUPS=100 (generating groups numbers from 0 to 99). You can then select groups 0 to 9 (lowest 10 percent) and 90 to 99 (highest 10 percent). The program that follows uses the latter method.

We will only present a macro version using this method. It should be quite easy to write a non-macro version based on this program, if you wish.

Program 2-9 Creating a Macro to List the Highest and Lowest "n" Percent of the Data Using PROC RANK

```
*-------------------------------------------------------------*
| Macro Name: top_bottom_nPercent                             |
| Purpose: To list the upper and lower n% of values           |
| Arguments: Dsn     - Data set name (one- or two-level       |
|            Var     - Variable to test                       |
|            Percent - Upper and lower n%                     |
|            Idvar   - ID variable                            |
| Example: %top_bottom_nPercent(Dsn=clean.patients,           |
|                               Var=SBP,                      |
|                               Percent=10,                   |
|                               Idvar=Patno)                  |
*-------------------------------------------------------------*;

%macro top_bottom_nPercent
   (Dsn=,
    Var=,
    Percent=,
    Idvar=);
   %let Bottom = %eval(&Percent - 1);  ❶
   %let Top = %eval(100 - &Percent);  ❷

   proc format;
      value rnk 0 - &Bottom = 'Low'
                &Top - 99    = 'High';
   run;

   proc rank data=&Dsn(keep=&Var &Idvar)
             out=new(where=(&Var is not missing))
             groups=100;  ❸
      var &Var;
      ranks Range;
   run;

   ***Sort and keep top and bottom n%;
   proc sort data=new(where=(Range le &Bottom or
                             Range ge &Top));  ❹
      by  &Var;
   run;
```

```
   ***Produce the report;
   proc print data=new;
   title "Upper and Lower &Percent.% Values for %upcase(&Var)";  ❺
      id &Idvar;
      var Range &Var;
      format Range rnk.;
   run;

   proc datasets library=work nolist;  ❻
      delete new;
   run;
   quit;

%mend top_bottom_nPercent;
```

The program first computes two macro variables Bottom ❶ and Top ❷. Remember that since macro values are text, you need to use the %EVAL function to perform arithmetic on these values. For example, if you set the macro variable Percent equal to 10, Bottom will be 9 and Top will be 90.

The key to the whole program is PROC RANK ❸, which uses the GROUPS= option to divide the data set into 100 groups. The lowest "n" percent of your data values will be in groups 0 to &Bottom and the top "n" percent will be in groups &Top to 99. The WHERE data set option on data set NEW removes the missing values from this data set. The group values will be stored in the variable called Range (listed on the RANKS statement).

The sort ❹ accomplishes two things: 1) It subsets the data set with the WHERE data set option, keeping only the top and bottom groups, and 2) it puts the data values in order from the smallest to the largest. All that is left to do is to print the report ❺ and delete the temporary data set ❻. The format RNK. replaces the bottom and top values of Range to the values 'Low' and 'High' respectively.

Issue the following statement to see a list of the top and bottom 10% of your values for SBP (systolic blood pressure):

```
%top_bottom_nPercent(Dsn=clean.patients,
                     Var=SBP,
                     Percent=10,
                     Idvar=Patno)
```

This produces the following output:

```
Upper and Lower 10% Values for SBP

Patno    Range    SBP

 020     Low       20
 023     Low       34
 011     High     300
 321     High     400
```

Presenting a Program to List the Highest and Lowest Ten Values

As you saw earlier, PROC UNIVARIATE, with the option NEXTROBS= and an ODS SELECT statement, can quickly and conveniently list the "n" highest and lowest values of a variable. We present a program and macro to accomplish this same goal. Why? First, you can obtain the list for each variable by entering a single macro call. Second, the program and macro use several interesting programming features. Let's start with the program:

Program 2-10 Creating a Program to List the Highest and Lowest 10 Values

```
proc sort data=clean.patients(keep=Patno HR
          where=(HR is not missing)) out=tmp;
   by HR;
run;
data _null_;
   set tmp nobs=Num_obs;
   call symputx('Num',Num_obs);
   stop;
run;

%let High = %eval(&Num - 9);

title "Ten Highest and Ten Lowest Values for HR";
data _null_;
   set tmp(obs=10)         /* lowest values */
       tmp(firstobs=&High) /* highest values */;
   file print;
   if _n_ le 10 then do;
      if _n_ = 1 then put / "Ten Lowest Values" ;
      put "Patno = " Patno @15 "Value = " HR;
   end;
```

```
   else if _n_ ge 11 then do;
      if _n_ = 11 then put / "Ten Highest Values" ;
      put "Patno = " Patno @15 "Value = " HR;
   end;
run;
```

The program starts by sorting the data set by HR. The KEEP= data set option brings in the two variables Patno and HR and the WHERE= option removes the missing values. The output data set, TMP, holds the sorted data.

In the DATA _NULL_ step, you use the SET option NOBS=Num_obs to place the number of observations in data set TMP into the variable Num_obs. You want this value in a macro variable so it will be available in the next DATA step, so you use CALL SYMPUTX to place the value of the variable Num_obs into a macro variable called NUM. (Note: If you use CALL SYMPUT instead of CALL SYMPUTX, the program executes in an identical manner, but a note is written to the SAS Log about conversion of numeric values to character values.)

You can see that listing the lowest 10 non-missing values involves listing the first 10 observations in sorted data set TMP. How about the 10 highest values? Since you now know how many observations are in data set TMP, you can subtract 9 to determine the observation at which to start listing the 10 highest values. (You subtract 9 since you are going to list that value as well as the next 9 values, giving you a total of 10 values.)

The last DATA _NULL_ step uses two SET statements on the same data set (TMP). (Credit for this clever approach goes to Mike Zdeb, who suggested this method.) The first SET statement uses the data set option OBS=10. This option says to stop when you reach observation number 10, and therefore lists the first (lowest) 10 values in the data set. The second SET statement uses the data set option FIRSTOBS=&High. This says to start at the observation stored in the macro variable &High. This will list the last (highest) 10 values in the data set.

OK, you are almost finished. A test for observation 1 and observation &High is made so that the program can print the two messages (Ten Lowest Values and Ten Highest Values). Below is the output you obtain when this program is run:

```
Ten Highest and Ten Lowest Values for HR

Ten Lowest Values
Patno = 020   Value = 10
Patno = 014   Value = 22
Patno = 023   Value = 22
Patno = 022   Value = 48
Patno = 003   Value = 58
Patno = 019   Value = 58
Patno = 012   Value = 60
Patno = 123   Value = 60
Patno = 028   Value = 66
Patno = 003   Value = 68

Ten Highest Values
Patno = 002   Value = 84
Patno = 002   Value = 84
Patno = 009   Value = 86
Patno = 001   Value = 88
Patno = 007   Value = 88
Patno =       Value = 90
Patno = 004   Value = 101
Patno = 017   Value = 208
Patno = 008   Value = 210
Patno = 321   Value = 900
```

Presenting a Macro to List the Highest and Lowest "n" Values

This section uses the logic of Program 2-10 to produce a macro that lists the "n" highest and lowest (non-missing) values in a data set. First the macro, then the explanation:

Program 2-11 Presenting a Macro to List the Highest and Lowest "n" Values

```
*-------------------------------------------------------------*
| Macro Name: highlow                                         |
| Purpose: To list the "n" highest and lowest values          |
| Arguments: Dsn     - Data set name (one- or two-level       |
|            Var     - Variable to list                       |
|            Idvar   - ID variable                            |
|            n       - Number of values to list               |
| Example: %highlow(Dsn=clean.patients,                       |
|                   Var=SBP,                                  |
|                   Idvar=Patno,                              |
|                   n=7)                                      |
*-------------------------------------------------------------*;

%macro highlow(Dsn=,     /* Data set name          */
               Var=,     /* Variable to list       */
               Idvar=,   /* ID Variable            */
               n=        /* Number of high and low
                            values to list         */);
   proc sort data=&Dsn(keep=&Idvar &Var
             where=(&Var is not missing)) out=tmp;
      by &Var;
   run;
   data _null_;
      set tmp nobs=Num_obs;
      call symput('Num',Num_obs);
      stop;
   run;

   %let High = %eval(&Num - &n + 1);
```

```
   title "&n Highest and Lowest Values for &Var";
   data _null_;
      set tmp(obs=&n)         /* lowest values */
          tmp(firstobs=&High) /* highest values */;
      file print;
      if _n_ le &n then do;
         if _n_ = 1 then put / "&n Lowest Values" ;
         put "&Idvar = " &Idvar @15 "Value = " &Var;
      end;
      else if _n_ ge %eval(&n + 1) then do;
         if _n_ = %eval(&n + 1) then put / "&n Highest Values" ;
         put "&Idvar = " &Idvar @15 "Value = " &Var;
      end;
   run;
   proc datasets library=work nolist;
      delete tmp;
   run;
   quit;
%mend highlow;
```

Since the program has already been described, understanding the macro version is straightforward. All that was necessary to turn the program into a macro was to make the values of the data set name, the variables to test, the ID variable name, and the number of observations to list into macro calling arguments. You may want to place this macro into your macro library and use it as an alternative to PROC UNIVARIATE to list the "n" highest and lowest values of a numeric variable in a data set.

If you call the macro like this:

```
%highlow(Dsn=clean.patients, Var=SBP, Idvar=Patno, n=7)
```

you will generate the following output:

```
7 Highest and Lowest Values for HR

7 Lowest Values
Patno = 020   Value = 10
Patno = 014   Value = 22
Patno = 023   Value = 22
Patno = 022   Value = 48
Patno = 003   Value = 58
Patno = 019   Value = 58
Patno = 012   Value = 60

7 Highest Values
Patno = 001   Value = 88
Patno = 007   Value = 88
Patno =       Value = 90
Patno = 004   Value = 101
Patno = 017   Value = 208
Patno = 008   Value = 210
Patno = 321   Value = 900
```

Using PROC PRINT with a WHERE Statement to List Invalid Data Values

This section examines ways to detect possible data errors where you can determine reasonable ranges for each variable. This works quite well for variables such as heart rates and blood pressures, but may not be feasible for other types of variables, such as financial values that may take on a very large range of possible values.

One simple way to check each numeric variable for invalid values, where you can determine reasonable values, is to use PROC PRINT, followed by the appropriate WHERE statement.

Suppose you want to check all the data for any patient having a heart rate outside the range of 40 to 100, a systolic blood pressure outside the range of 80 to 200, and a diastolic blood pressure outside the range of 60 to 120. For this example, missing values are not treated as invalid. The PROC PRINT step in Program 2-12 reports all patients with out-of-range values for heart rate, systolic blood pressure, or diastolic blood pressure.

Program 2-12 Using a WHERE Statement with PROC PRINT to List Out-of-Range Data

```
title "Out-of-range values for numeric variables";
proc print data=clean.patients;
   where (HR not between 40 and 100 and HR is not missing)      or
         (SBP not between 80 and 200 and SBP is not missing)    or
         (DBP not between 60 and 120 and DBP is not missing);
   id Patno;
   var HR SBP DBP;
run;
```

You don't need the parentheses in the WHERE statements because the AND operator is evaluated before the OR operator. However, because this author can never seem to remember the order of operation of Boolean operators, the parentheses were included for clarity. Extra parentheses do no harm.

The resulting output is shown next.

```
Out-of-range values for numeric variables

Patno     HR    SBP    DBP

 004     101    200    120
 008     210      .      .
 009      86    240    180
 010       .     40    120
 011      68    300     20
 014      22    130     90
 017     208      .     84
 321     900    400    200
 020      10     20      8
 023      22     34     78
```

A disadvantage of this listing is that an observation is printed if one or more of the variables is outside the specified range. To obtain a more precise listing that shows only the data values outside the normal range, you can use a DATA step as described in the next section.

Using a DATA Step to Check for Out-of-Range Values

A simple DATA _NULL_ step can also be used to produce a report on out-of-range values. The approach here is the same as the one described in Chapter 1.

Program 2-13 Using a DATA _NULL_ Step to List Out-of-Range Data Values

```
title "Listing of patient numbers and invalid data values";
data _null_;
   file print; ***send output to the output window;
   set clean.patients(keep=Patno HR SBP DBP);
   ***Check HR;
   if (HR lt 40 and not missing(HR)) or HR gt 100 then
      put Patno= HR=;
   ***Check SBP;
   if (SBP lt 80 and not missing(SBP)) or SBP gt 200 then
      put Patno= SBP=;
   ***Check DBP;
   if (DBP lt 60 and not missing(HR)) or DBP gt 120 then
      put Patno= DBP=;
run;
```

Here is the output from Program 2-13.

```
Listing of patient numbers and invalid data values
Patno=004 HR=101
Patno=008 HR=210
Patno=008 DBP=.
Patno=009 SBP=240
Patno=009 DBP=180
Patno=010 SBP=40
Patno=011 SBP=300
Patno=011 DBP=20
Patno=014 HR=22
Patno=017 HR=208
Patno=123 DBP=.
Patno=321 HR=900
Patno=321 SBP=400
Patno=321 DBP=200
Patno=020 HR=10
Patno=020 SBP=20
Patno=020 DBP=8
Patno=023 HR=22
Patno=023 SBP=34
```

Notice that a statement such as "`if HR lt 40`" includes missing values because missing values are interpreted by SAS programs as the smallest possible value. Therefore, the following statement

```
if HR lt 40 or HR gt 100 then put Patno= HR=;
```

will produce a listing that includes missing heart rates as well as out-of-range values (which may be what you want).

Identifying Invalid Values versus Missing Values

This error report is fine as far as it goes, but there is a slight problem: Patient number 027 had a value of 'NA' for heart rate in the raw data file, but this value did not appear in this listing. Why not?

When the SAS data set PATIENTS was created, the variables HR, SBP, and DBP were all read with a 3. informat. When the INPUT statement attempted to read the value 'NA', SAS generated an error message and listed the offending line of data in the SAS Log (you did read the Log, didn't you?). It also generated a missing value for the value of HR for patient 027. Therein lies the problem—once you have a SAS data set, there is no way to tell if the original data value was truly missing or if the missing value was the result of invalid data.

Because missing values are not treated as errors in Program 2-13, no error listing is produced for patient number 027. If you would like to include invalid character values (such as NA) as errors, you have to be reading from the original raw data file. One way to test for invalid data values is to use the internal _ERROR_ variable to check if such a value was processed by the INPUT statement. _ERROR_ is a built-in SAS variable that is part of the PDV (Program Data Vector). It is reset to zero at each iteration of the DATA step and it is set to one if SAS encounters an error in the processing the INPUT statement.

The program below tests the _ERROR_ variable and prints out an error message if there was an invalid value for HR, SBP, or DBP. Later you will see a way to identify exactly which variables contained invalid (character) values.

Program 2-14 Presenting a Program to Detect Invalid (Character) Data Values, Using _ERROR_

```
title "Listing of patient numbers and invalid data values";
data _null_;
   infile "c:\books\clean\patients.txt" pad;
   file print; ***send output to the output window;
   ***Note: we will only input those variables of interest;
   input @1  Patno    $3.
         @15 HR        3.
         @18 SBP       3.
         @21 DBP       3.;
   ***Check HR;
   if (HR lt 40 and not missing(HR)) or HR gt 100 then
      put Patno= HR=;
   ***Check SBP;
   if (SBP lt 80 and not missing(SBP)) or SBP gt 200 then
      put Patno= SBP=;
   ***Check DBP;
   if (DBP lt 60 and not missing(HR)) or DBP gt 120 then
      put Patno= DBP=;
   if _error_ eq 1 then
      put Patno= "had one or more invalid character values" /
          HR= SBP= DBP=;
run;
```

The program cannot tell which variable has caused an error, so it prints a general message indicating that there was an error reading one or more values for a patient. It is certainly possible to distinguish between invalid character values in numeric fields from true missing values. One possible approach is to use an enhanced numeric informat. Another is to read all of the numeric

variables as character data, test the values, and then convert to numeric for range checking. In a later section we demonstrate how a user-defined enhanced numeric informat can be used for this purpose. Here is the output from Program 2-14:

```
Listing of patient numbers and invalid data values
Patno=004 HR=101
Patno=008 HR=210
Patno=009 SBP=240
Patno=009 DBP=180
Patno=010 SBP=40
Patno=011 SBP=300
Patno=011 DBP=20
Patno=014 HR=22
Patno=017 HR=208
Patno=321 HR=900
Patno=321 SBP=400
Patno=321 DBP=200
Patno=020 HR=10
Patno=020 SBP=20
Patno=020 DBP=8
Patno=023 HR=22
Patno=023 SBP=34
Patno=027 had one or more invalid character values
HR=. SBP=166 DBP=106
```

Looking at this output, you could make a reasonable guess that HR was the variable that contained the invalid value.

Listing Invalid (Character) Values in the Error Report

Although the program above identified an invalid value of HR for patient 027, it did not show the actual value that caused the error. Of course the SAS Log would list the error, but either you may miss it, or there may be more errors than the default number printed by SAS (20), so listing them in a report may be useful.

The next program reads the values of HR, SBP, and DBP as character data, tests for invalid values, and then performs a character-to-numeric conversion and tests the resulting numeric value, using the same method as Program 2-13:

Program 2-15 Including Invalid Values in Your Error Report

```
title "Listing of Invalid and Out-of-Range Values";
data _null_;
   file print; ***Send output to the output window;
   infile "c:\books\clean\patients.txt" pad;
   ***Note: we will only input those variables of interest;
   input @1  Patno    $3.
         @15 C_HR     $3.
         @18 C_SBP    $3.
         @21 C_DBP    $3.;
   ***Check HR;
   if not missing(C_HR) then do;
      if notdigit(trim(C_HR))then put Patno=
      "has an invalid value for HR of " C_HR;
      else do;
         HR = input(C_HR,8.);
         if HR lt 40 or HR gt 100 then put Patno= HR=;
      end;
   end;
   ***Check SBP;
   if not missing (C_SBP) then do;
      if notdigit(trim(C_SBP)) then put Patno=
      "has an invalid value of SBP of " C_SBP;
      else do;
         SBP = input(C_SBP,8.);
         if SBP lt 80 or SBP gt 200 then put Patno= SBP=;
      end;
   end;
```

```
   ***Check DBP;
   if not missing(C_DBP) then do;
      if notdigit(trim(C_DBP)) then put Patno=
      "has an invalid value for DBP of " C_DBP;
      else do;
         DBP = input(C_DBP,8.);
         if DBP lt 60 or DBP gt 120 then put Patno= DBP=;
      end;
   end;
   drop C_: ;
run;
```

In order to print invalid (character) values for your numeric variables, you need to do quite a bit of extra work. If you expect these types of errors will be rare, you may want to rely on the SAS Log to identify them.

Notice that this program reads the values for HR, SBP, and DBP as character. The test for each variable begins with a check to see if the value is missing. If not, a check to see if there are any non-digit characters in the string is performed. The NOTDIGIT function returns the position of the first character in a character value that is not a digit. If the character value contains only digits, the NOTDIGIT function returns a zero. Trailing blanks in a character string are treated as non-digits, so a TRIM function is used to remove trailing blanks.

If only digits are found, an INPUT function performs the character-to-numeric conversion. Finally, once the conversion is performed, the test for out-of-range values is accomplished. There is one last feature of this program that may be unfamiliar to you. That is, the use of a variable list '`C_:`' in the DROP statement. Anywhere that SAS allows you to supply a variable list, you can use a list in the form: *var*`:`. This is interpreted by SAS as all variable names that begin with '*var*'. So, the DROP statement above will drop all variables that begin with '`C_`'.

Below is the output from running Program 2-15:

```
Listing of Invalid and Out-of-Range Values
Patno=004 HR=101
Patno=008 HR=210
Patno=009 SBP=240
Patno=009 DBP=180
Patno=010 SBP=40
Patno=011 SBP=300
Patno=011 DBP=20
Patno=014 HR=22
Patno=017 HR=208
Patno=321 HR=900
Patno=321 SBP=400
Patno=321 DBP=200
Patno=020 HR=10
Patno=020 SBP=20
Patno=020 DBP=8
Patno=023 HR=22
Patno=023 SBP=34
Patno=027 has an invalid value for HR of NA
```

Notice that the invalid character value is listed for patient 027.

Creating a Macro for Range Checking

Because range checking is such a common data cleaning task, it makes some sense to automate the procedure somewhat. Looking at Program 2-13, you see that the range checking lines all look similar except for the variable names and the low and high cutoff values. Even if you are not a macro expert, the following program should not be too difficult to understand. (For an excellent review of macro programming, I recommend two books in the SAS Press Series: *SAS Macro Programming Made Easy*, *Second Edition*, by Michele M. Burlew, and *Carpenter's Complete Guide to the SAS Macro Language*, *Second Edition*, by Art Carpenter, both published by SAS Institute Inc., Cary, NC.)

Program 2-16 Writing a Macro to List Out-of-Range Data Values

```
*-------------------------------------------------------------*
| Program Name: RANGE.SAS  in C:\books\clean                  |
| Purpose: Macro that takes lower and upper limits for a      |
|          numeric variable and an ID variable to print out   |
|          an exception report to the Output window.          |
| Arguments: Dsn    - Data set name                           |
|            Var    - Numeric variable to test                |
|            Low    - Lowest valid value                      |
|            High   - Highest valid value                     |
|            Idvar  - ID variable to print in the exception   |
|                     report                                  |
| Example: %range(Dsn=CLEAN.PATIENTS,                         |
|                 Var=HR,                                     |
|                 Low=40,                                     |
|                 High=100,                                   |
|                 Idvar=Patno)                                |
*-------------------------------------------------------------*;

%macro range(Dsn=    /* Data set name              */,
             Var=    /* Variable you want to check  */,
             Low=    /* Low value                  */,
             High=   /* High value                 */,
             Idvar=  /* ID variable                */);

title "Listing of Out of range Data Values";
   data _null_;
      set &Dsn(keep=&Idvar &Var);
      file print;
      if (&Var lt &Low and not missing(&Var)) or &Var gt &High then
         put "&Idvar:" &Idvar  @18 "Variable:&VAR"
                              @38 "Value:" &Var
                              @50 "out-of-range";
   run;

%mend range;
```

You can see that converting Program 2-13 to a macro is quite straightforward. You just need to replace the pieces of the program that may change (the data set name, the variable to list, the low and high values, and the ID variable) and replace them with macro variables.

The results from running the macro with the above calling arguments are listed next:

```
Listing of Out of range Data Values
Patno:004        Variable:HR         Value:101   out-of-range
Patno:008        Variable:HR         Value:210   out-of-range
Patno:014        Variable:HR         Value:22    out-of-range
Patno:017        Variable:HR         Value:208   out-of-range
Patno:321        Variable:HR         Value:900   out-of-range
Patno:020        Variable:HR         Value:10    out-of-range
Patno:023        Variable:HR         Value:22    out-of-range
```

While this saves on programming time, it is not as efficient as a program that checks all the numeric variables in one DATA step. However, sometimes it is reasonable to sacrifice computer time for human time. In the next section we present a macro that will allow you to test for invalid ranges for as many variables as you wish.

Checking Ranges for Several Variables

Most of the programs and macros you have seen so far test out-of-range values for individual variables. In this section, the range checking macro developed in Program 2-16 is expanded to do two things: One, make the macro more flexible so that it can either treat missing values as valid or invalid; and two, allow the macro to be called multiple times with different numeric variables and produce one consolidated report when finished. The macro is listed first, followed by a step-by-step explanation.

Program 2-17 Writing a Program to Summarize Data Errors on Several Variables

```
*-------------------------------------------------------------*
| PROGRAM NAME: ERRORS.SAS  in c:\books\clean                 |
| PURPOSE: Accumulates errors for numeric variables in a SAS  |
|          data set for later reporting.                      |
|          This macro can be called several times with a      |
|          different variable each time. The resulting errors |
|          are accumulated in a temporary SAS data set called |
|          errors.                                            |
| ARGUMENTS: Dsn=    - SAS data set name (assigned with a %LET) |
|            Idvar=  - Id variable (assigned with a %LET)     |
|                                                             |
|            Var     = The variable name to test              |
|            Low     = Lowest valid value                     |
|            High    = Highest valid value                    |
```

```
|            Missing = IGNORE (default) Ignore missing values   |
|                      ERROR Missing values flagged as errors   |
|                                                               |
| EXAMPLE: %let Dsn = clean.patients;                           |
|          %let Idvar = Patno;                                  |
|                                                               |
|          %errors(Var=HR, Low=40, High=100, Missing=error)     |
|          %errors(Var=SBP, Low=80, High=200, Missing=ignore)   |
|          %errors(Var=DBP, Low=60, High=120)                   |
|                                                               |
|          Test the numeric variables HR, SBP, and DBP in data  |
|          set clean.patients for data outside the ranges       |
|          40 to 100, 80 to 200, and 60 to 120 respectively.    |
|          The ID variable is Patno and missing values are to   |
|          be flagged as invalid for HR but not for SBP or DBP. |
*---------------------------------------------------------------*;
%macro errors(Var=,    /* Variable to test     */
              Low=,    /* Low value            */
              High=,   /* High value           */
              Missing=ignore
                       /* How to treat missing values         */
                       /* Ignore is the default.  To flag     */
                       /* missing values as errors set        */
                       /* Missing=error                       */);
data tmp;
   set &dsn(keep=&Idvar &Var);
   length Reason $ 10 Variable $ 32;
   Variable = "&Var";
   Value = &Var;
   if &Var lt &Low and not missing(&Var) then do;
      Reason='Low';
      output;
   end;
   %if %upcase(&Missing) ne IGNORE %then %do;
   else if missing(&Var) then do;
      Reason='Missing';
      output;
   end;
   %end;
```

```
   else if &Var gt &High then do;
        Reason='High';
      output;
      end;
      drop &Var;
   run;
   proc append base=errors data=tmp;
   run;

%mend errors;

***Error Reporting Macro - to be run after ERRORS has been called
   as many times as desired for each numeric variable to be tested;

%macro report;
   proc sort data=errors;
      by &Idvar;
   run;

proc print data=errors;
      title "Error Report for Data Set &Dsn";
      id &Idvar;
      var Variable Value Reason;
   run;

   proc datasets library=work nolist;
      delete errors;
      delete tmp;
   run;
   quit;

%mend report;
```

To avoid having to enter the data set name and the ID variable each time this macro is called, the two macro variables &Dsn and &Idvar are assigned with %LET statements. Calling arguments to the macro are the name of the numeric variable to be tested, the lower and upper valid values for this variable, and an indicator to determine if missing values are to be listed in the error report or not. To keep the macro somewhat efficient, only the variable in question and the ID variable are added to the TMP data set because of the KEEP= data set option. The variables REASON and VARIABLE hold values for why the observation was selected and the name of the variable being

tested. Because the name of the numeric variable to be tested changes each time the macro is called, a variable called VALUE is assigned the value of the numeric variable.

The range is first checked with missing values being ignored. If the value of the macro variable &Missing is not equal to 'IGNORE' then an additional check is made to see if the value is missing. Finally, each error found is added to the temporary data set ERRORS by using PROC APPEND. This is the most efficient method of adding observations to an existing SAS data set. Each time the ERRORS macro is called, all the invalid observations will be added to the ERRORS data set.

The second macro, REPORT, is a macro that should be called once after the ERRORS macro has been called for each of the desired numeric variable range checks. The REPORT macro is simple. It sorts the ERRORS data set by the ID variable, so that all errors for a particular ID will be grouped together. Finally, as you have done in the past, use PROC DATASETS to clean up the WORK data sets that were created.

To demonstrate how these two macros work, the ERRORS macro is called three times, for the variables heart rate (HR), systolic blood pressure (SBP), and diastolic blood pressure (DBP), respectively. For the HR variable, you want missing values to appear in the error report; for the other two variables, you do not want missing values listed as errors. Here is the calling sequence:

```
***Calling the ERRORS macro;

***Set two macro variables;
%let dsn=clean.patients;
%let Idvar = Patno;

%errors(Var=HR, Low=40, High=100, Missing=error)
%errors(Var=SBP, Low=80, High=200, Missing=ignore)
%errors(Var=DBP, Low=60, High=120)

***Generate the report;
%report
```

And finally, the report that is produced:

```
Error Report for Data Set clean.patients

Patno    Variable    Value    Reason

 004        HR         101     High
 008        HR         210     High
 009        SBP        240     High
 009        DBP        180     High
 010        HR           .     Missing
 010        SBP         40     Low
 011        SBP        300     High
 011        DBP         20     Low
 014        HR          22     Low
 017        HR         208     High
 020        HR          10     Low
 020        SBP         20     Low
 020        DBP          8     Low
 023        HR          22     Low
 023        SBP         34     Low
 027        HR           .     Missing
 029        HR           .     Missing
 321        HR         900     High
 321        SBP        400     High
 321        DBP        200     High
```

The clear advantage of this technique is that it provides a report that lists all the out-of-range or missing value errors for each patient, all in one place.

Using Formats to Check for Invalid Values

Just as you did with character values in Chapter 1, you can use user-defined formats to check for out-of-range data values. Program 2-18 uses formats to find invalid data values, based on the same ranges used in Program 2-13 in this chapter.

Program 2-18 Detecting Out-of-Range Values Using User-Defined Formats

```
proc format;
   value hr_ck  40-100, . = 'OK';
   value sbp_ck 80-200, . = 'OK';
   value dbp_ck 60-120, . = 'OK';
run;

title "Listing of patient numbers and invalid data values";
data _null_;
   set clean.patients(keep=Patno HR SBP DBP);
   file print; ***send output to the output window;
   if put(HR,hr_ck.)   ne 'OK' then put Patno= HR=;
   if put(SBP,sbp_ck.) ne 'OK' then put Patno= SBP=;
   if put(DBP,dbp_ck.) ne 'OK' then put Patno= DBP=;
run;
```

This is a fairly simple and efficient program. The user-defined formats HR_CK., SBP_CK., and DBP_CK. all assign the formatted value 'OK' for any data value in the acceptable range. In the DATA step, the result of the PUT function is the value of the first argument (the variable to be tested) formatted by the format specified as the second argument of the function. For example, any value of heart rate between 40 and 100 (or missing) falls into the format range 'OK'. A value of 22 for heart rate does not fall within the range of 40 to 100 or missing and the formatted value 'OK' is not assigned. In that case, the PUT function for heart rate does not return the value 'OK' and the IF statement condition is true. The appropriate PUT statement is then executed and the invalid value is printed to the print file.

Output from this program is shown next:

```
Listing of patient numbers and invalid data values
Patno=004 HR=101
Patno=008 HR=210
Patno=009 SBP=240
Patno=009 DBP=180
Patno=010 SBP=40
Patno=011 SBP=300
Patno=011 DBP=20
Patno=014 HR=22
Patno=017 HR=208
Patno=321 HR=900
Patno=321 SBP=400
Patno=321 DBP=200
Patno=020 HR=10
Patno=020 SBP=20
Patno=020 DBP=8
Patno=023 HR=22
Patno=023 SBP=34
```

Using Informats to Filter Invalid Values

If you want a quick and easy way to test for numeric values in a certain range and simply want to set any other value to missing, you can use a user-defined informat.

Program 2-19 Using User-Defined Informats to Filter Invalid Values

```
proc format;
   invalue hr_ck  40-100 = _same_
                  other  = .;
   invalue sbp_ck 80-200 = _same_
                  other  = .;
   invalue dbp_ck 60-120 = _same_
                  other  = .;
run;

title "Using User-Defined Informats to Filter Invalid Values";
data valid_numerics;
   infile "c:\books\clean\patients.txt" pad;
   file print; ***send output to the output window;
   ***Note: we will only input those variables of interest;
   input @1  Patno    $3.
         @15 HR        hr_ck3.
         @18 SBP       sbp_ck3.
         @21 DBP       dbp_ck3.;
run;
```

PROC FORMAT is used to create three informats (note the use of INVALUE statements instead of the usual VALUE statements). For the informat HR_CK, any numeric value in the range 40 to 100 is unchanged (the keyword _same_ accomplishes this). Any other value is set to missing.

Running Program 2-19 results in the following output:

```
Using User-Defined Informats to Filter Invalid Values

Patno    HR    SBP    DBP

 001     88    140     80
 002     84    120     78
 003     68    190    100
 004      .    200    120
 XX5     68    120     80
 006     72    102     68
 007     88    148    102
         90    190    100
 008      .      .      .
 009     86      .      .
 010      .      .    120
 011     68      .      .
 012     60    122     74
 013     74    108     64
 014      .    130     90
 002     84    120     78
 003     58    112     74
 015     82    148     88
 017      .      .     84
 019     58    118     70
 123     60      .      .
 321      .      .      .
 020      .      .      .
 022     48    114     82
 023      .      .     78
 024     76    120     80
 025     74    102     68
 027      .    166    106
 028     66    150     90
 029      .      .      .
 006     82    148     84
```

Checking a Range Using an Algorithm Based on Standard Deviation

The remaining programs and macros in this chapter use the distributions of data values to determine possible errors. For example, you could decide to flag all values more than two standard deviations from the mean. However, if you had some severe data errors, the standard deviation could be so badly inflated that obviously incorrect data values might lie within two standard deviations. A possible workaround for this would be to compute the standard deviation after removing some of the highest and lowest values. For example, you could compute a standard deviation of the middle 80% of your data and use this to decide on outliers. Another popular alternative is to use an algorithm based on the interquartile range (the difference between the 25^{th} percentile and the 75^{th} percentile).

Let's first see how you could identify data values more than two standard deviations from the mean. You can use PROC MEANS to compute the mean and standard deviation followed by a short DATA step to select the outliers, as shown in Program 2-20.

Program 2-20 Detecting Outliers Based on the Standard Deviation

```
libname clean "c:\books\clean";
***Output means and standard deviations to a data set;
proc means data=clean.patients noprint;
   var HR;
   output out=means(drop=_type_ _freq_)
          mean=M_HR
          std=S_HR;
run;

title "Outliers for HR Based on 2 Standard Deviations";
data _null_;
   file print;
   set clean.patients(keep=Patno HR);
   ***bring in the means and standard deviations;
   if _n_ = 1 then set means;
   if HR lt M_HR – 2*S_HR and not missing(HR) or
      HR gt M_HR + 2*S_HR then put Patno= HR=;
run;
```

The PROC MEANS step computes the mean and standard deviation for heart rate (HR). To compare each of the raw data values against the mean plus or minus 2 standard deviations, you need to add the values for the mean and standard deviation to each observation in the PATIENTS data set.

You use the same trick you used earlier, that is, you execute a SET statement only once, when _N_ is equal to one. Because all the variables brought into the program data vector (PDV) with a SET statement are retained, the variables M_HR and S_HR will be in the PDV and will be added to each observation in the PATIENTS data set. The IF statement then checks for all values of HR that are more than 2 standard deviations from the mean (omitting missing values). The results of running this program on the PATIENTS data follow:

```
Outliers for HR Based on 2 Standard Deviations
Patno=321 HR=900
```

Well that didn't work very well! To see exactly why, take a look at the MEANS data set:

```
Listing of Data Set MEANS

  M_HR       S_HR

107.393    161.086
```

The mean heart rate is about 100 and the standard deviation is about 160. Either these folks are drinking a lot of coffee or something else is going on. If you have memorized all the data values in the PATIENTS data set, you will remember that there was one heart rate equal to 900. That extreme value inflated the mean to over 100 and also inflated the standard deviation to over 160. A good rule to remember is that extreme values can have a really devastating effect on the standard deviation. The mean minus 2 standard deviations is a negative number (there were no negative heart rates) and the mean plus 2 standard deviations was over 400. That is the reason that only one heart rate (900) was detected by this program.

One way to fix this problem is to compute trimmed statistics. This is done by first removing some values from the top and bottom of the data set, as we demonstrate in the next section.

Detecting Outliers Based on a Trimmed Mean and Standard Deviation

A quick and easy way to compute trimmed statistics and output them to a SAS data set is to first run PROC RANK with the GROUPS= option to divide the data set into "n" groups. For example, if you want to trim 10% from the top and bottom of your data set, you would need to set GROUPS equal to 10 and remove all the observations for HR in the top and bottom group. Below is a program to trim 20% off the top and bottom of the heart rate values and compute the mean and standard deviation:

Program 2-21 Computing Trimmed Statistics

```
proc rank data=clean.patients(keep=Patno HR) out=tmp groups=5;
   var HR;
   ranks R_HR;
run;

proc means data=tmp noprint;
   where R_HR not in (0,4);
   *Trimming the top and bottom 20%;
   var HR;
   output out=means(drop=_type_ _freq_)
          mean=M_HR
          std=S_HR;
run;
```

To see exactly what is happening here, first take a look at the first few observations in the output data set created by PROC RANK (TMP):

```
First 20 Observations in Data Set TMP

Obs    Patno     HR    R_HR

  1     001      88       3
  2     002      84       3
  3     003      68       1
  4     004     101       4
  5     XX5      68       1
  6     006      72       2
  7     007      88       3
  8              90       4
  9     008     210       4
 10     009      86       3
 11     010       .       .
 12     011      68       1
 13     012      60       1
 14     013      74       2
 15     014      22       0
 16     002      84       3
 17     003      58       0
 18     015      82       3
 19     017     208       4
 20     019      58       0
```

The values of the variable R_HR vary from 0 to 4. Observations with R_HR equal to 0 represent heart rates in the bottom 20%; observations with R_HR equal to 4 represent heart rates in the top 20%. By running PROC MEANS with a WHERE statement, you can compute the mean and standard deviation of HR with the top and bottom 20% of the values removed. The resulting MEANS data set is shown next:

```
Listing of Data Set MEANS

  M_HR       S_HR

75.2941    9.37926
```

This represents quite a change from the values computed from the original data. You can now go ahead and use these values in Program 2-20, with one minor change.

When you compute a standard deviation from trimmed data, you obtain a smaller value than if you use all the data values (since the trimmed data has less variation). If you trimmed 20% of the data values from the top and bottom of the data and you had a variable that was normally distributed, your estimate of the standard deviation based on the trimmed data would be too small by a factor of 2.12. If you want to base your decision to reject values beyond two standard deviations, you probably want to adjust the standard deviation you obtained from the trimmed data by that factor. This factor was incorporated in the program below as a macro variable called MULT (for multiplier).

This value will change depending on how much trimming is done. If you only trim a few percent from the top and bottom of the data values, MULT will be close to 1; if you trim a lot (say 25% from the top and bottom), this factor will be larger. The table below shows several trimming values along with the appropriate MULT factors:

Trim Value (from the top and bottom)	Multiplicative Factor
5%	1.24
10%	1.49
20%	2.12
25%	2.59

The entire program (including computation of the trimmed statistics) is shown in Program 2-22:

Program 2-22 Detecting Outliers Based on Trimmed Statistics

```
proc rank data=clean.patients(keep=Patno HR) out=tmp groups=5;
   var HR;
   ranks R_HR;
run;

proc means data=tmp noprint;
   where R_HR not in (0,4);  ***the middle 60%;
   var HR;
   output out=means(drop=_type_ _freq_)
          mean=M_HR
          std=S_HR;
run;
```

```
%let N_sd = 2;
%let Mult = 2.12;

title "Outliers Based on Trimmed Statistics";
data _null_;
   file print;

   set clean.patients;
   if _n_ = 1 then set means;
   if HR lt M_HR – &N_sd*S_HR*&Mult and not missing(HR) or
      HR gt M_HR + &N_sd*S_HR*&Mult then put Patno= HR=;
run;
```

Here is the output from Program 2-22.

```
Outliers Based on Trimmed Statistics
Patno=008 HR=210
Patno=014 HR=22
Patno=017 HR=208
Patno=321 HR=900
Patno=020 HR=10
Patno=023 HR=22
```

Notice that the method based on a non-trimmed standard deviation reported only one HR as an outlier (Patno=321, HR=900) while the method based on a trimmed mean identified six values.

Presenting a Macro Based on Trimmed Statistics

Since you may want to use trimmed statistics to detect outliers for some of your numeric variables and you may want different trim values, we present a macro that will allow you to test any numeric variable for outliers, based on your choice of the number of standard deviations and the percent of trimming from the top and bottom of the data values. Since you may want to run this for several variables, the data set name and the name of the ID variable are first assigned with %LET statements. You can then call the macro, once for each variable you want to test. The macro is based on Program 2-22 and you should be able to follow the logic of the macro if you understood the previous program. After the macro listing, we will demonstrate how to use it for several variables with different values of trimming. First, here is the macro:

Program 2-23 Creating a Macro to Detect Outliers Based on Trimmed Statistics

```
%macro trimmed
   (/* the data set name (DSN) and ID variable (IDVAR)
       need to be assigned with %let statements
       prior to calling this macro */
       Var=,     /* Variable to test for outliers */
       N_sd=2,   /* Number of standard deviations */
       Trim=10   /* Percent top and bottom trim   */
                 /* Valid values of Trim are      */
                 /* 5, 10, 20, and 25             */);

   /*************************************************************

   Example:
   %let dsn=clean.patients;
   %let idvar=Patno;

   %trimmed(Var=HR,
            N_sd=2,
            Trim=20)

   *************************************************************/
   title "Outliers for &Var based on &N_sd Standard Deviations";
   title2 "Trimming &Trim% from the Top and Bottom of the Values";

   %if &Trim eq 5 or
       &Trim eq 10 or
       &Trim eq 20 or
       &Trim eq 25 %then %do;

   %let NGroups = %eval(100/&Trim);
   %if &Trim = 5 %then %let Mult = 1.24;
   %else %if &trim = 10 %then %let Mult = 1.49;
   %else %if &trim = 20 %then %let Mult = 2.12;
   %else %if &trim = 25 %then %let Mult = 2.59;

   proc rank data=&dsn(keep=&Idvar &Var)
             out=tmp groups=&NGroups;
      var &var;
      ranks rank;
   run;
```

```
   proc means data=tmp noprint;
      where rank not in (0,%eval(&Ngroups - 1));
      var &Var;
      output out=means(drop=_type_ _freq_)
             mean=Mean
             std=Sd;
   run;

   data _null_;
      file print;
      set &dsn;
      if _n_ = 1 then set means;
      if &Var lt Mean - &N_sd*&Mult*Sd and
         not missing(&Var) or
         &Var gt Mean + &N_sd*&Mult*Sd
         then put &Idvar= &Var=;
   run;

   proc datasets library=work;
      delete means;
   run;
   quit;
   %end;

   %else %do;
   data _null_;
      file print;
      put "You entered a value of &trim for the Trim Value."/
          "It must be 5, 10, 20, or 25";
   run;
   %end;

%mend trimmed;
```

To demonstrate this macro, if you want to list outliers for HR based on 2 standard deviations with a 10% (from the top and bottom) trim, call the macro like this:

```
%let Dsn=clean.patients;
%let Idvar=Patno;
%trimmed(Var=HR, N_sd=2, Trim=10)
```

resulting in the following:

```
Outliers for HR based on 2 Standard Deviations
Trimming 10% from the Top and Bottom of the Values
Patno=008 HR=210
Patno=017 HR=208
Patno=321 HR=900
```

Calling the macro again with a 25% trim like this:

```
%trimmed(Var=HR, N_sd=2, Trim=25)
```

results in the following:

```
Outliers for HR based on 2 Standard Deviations
Trimming 25% from the Top and Bottom of the Values
Patno=008 HR=210
Patno=014 HR=22
Patno=017 HR=208
Patno=321 HR=900
Patno=020 HR=10
Patno=023 HR=22
```

Finally, to list outliers for SBP and DBP, using a cutoff of 2 standard deviations and a 20% trim, you call the macro like this:

```
%trimmed(Var=SBP, N_sd=2, Trim=20)
%trimmed(Var=DBP, N_sd=2, Trim=20)
```

yielding the following:

```
Outliers for SBP based on 2 Standard Deviations
Trimming 20% from the Top and Bottom of the Values
Patno=009 SBP=240
Patno=010 SBP=40
Patno=011 SBP=300
Patno=321 SBP=400
Patno=020 SBP=20
Patno=023 SBP=34

Outliers for DBP based on 2 Standard Deviations
Trimming 20% from the Top and Bottom of the Values
Patno=009 DBP=180
Patno=011 DBP=20
Patno=321 DBP=200
Patno=020 DBP=8
```

Using the TRIM Option of PROC UNIVARIATE and ODS to Compute Trimmed Statistics

Using the TRIM= PROC UNIVARIATE option, you can compute and print trimmed statistics or, by using the Output Delivery System (ODS), you can write these statistics to a SAS data set. If the value of TRIM is an integer (from 0 to half the number of non-missing observations), this number of values are trimmed from the top and bottom of the data. If the TRIM value is a proportion between 0 and .5, this proportion of values will be trimmed from the top and bottom of the data. As an example, the following program will print statistics for Heart Rate (HR) with 5 values trimmed off both ends of the distribution:

Program 2-24 Using the TRIM= Option of PROC UNIVARIATE

```
title "Trimmed statistics for HR with TRIM=5";
proc Univariate data=clean.patients trim=5;
   var HR;
run;
```

Part of the output from Program 2-24 showing the trimmed statistics is shown next:

```
                          Trimmed Means

Percent    Number               Std Error
Trimmed   Trimmed     Trimmed     Trimmed     95% Confidence
in Tail   in Tail        Mean        Mean          Limits             DF

  17.86         5    74.33333    3.593489   66.75173  81.91493       17

          Trimmed Means

Percent
Trimmed      t for H0:
in Tail       Mu0=0.00   Pr > |t|

  17.86       20.68556     <.0001
```

The next program shows how to use an ODS OUTPUT statement to capture the trimmed values to a SAS data set.

Program 2-25 Using ODS to Capture Trimmed Statistics from PROC UNIVARIATE

```
ods output TrimmedMeans=Trimmed5(keep=VarName Mean Stdmean DF);
ods listing close;

proc univariate data=clean.patients trim=5;
  var HR SBP DBP;
run;

ods output close;
ods listing;
```

The first ODS statement requests that the trimmed statistics produced by PROC UNIVARIATE be sent to a data set that you have named TRIMMED5. (Please see the explanation of how to obtain the names of output objects following Program 2-4.) The other ODS statement closes the listing file since you do not want to print the values—only to place them in a SAS data set. Finally, the last two ODS statements close the OUTPUT destination (the SAS data set TRIMMED5) and reopen the listing file. Do not use the NOPRINT option of PROC UNIVARIATE to turn off the listing file. If you do so, the ODS OUTPUT statement will not be able to output anything to your SAS data set.

The listing below shows the contents of data set TRIMMED5:

```
Listing of data set TRIMMED5

       Var
Obs    Name        Mean      StdMean        DF

 1     HR         74.3333    3.593489       17
 2     SBP       133.7647    9.790173       16
 3     DBP        84.0000    3.645085       17
```

Using the ideas developed in Program 2-25, you can create a general-purpose macro that will report outliers, based on trimmed statistics. Although this macro, presented as Program 2-26, is quite long, the logic is straightforward. The first step is to use PROC UNIVARIATE with the TRIM= option, along with an ODS OUTPUT statement, to create a data set like the one listed above. This data set contains the standard error of the mean rather than the standard deviation (the value we need). To compute the standard deviation from the standard error, you multiply by the square root of the number of observations. Luckily the data output data set also contains a variable called DF (degrees of freedom) which is equal to n – 1. You can, therefore, compute the standard deviation by multiplying the standard error by the square root of DF + 1.

A slight complication in using the data set produced by the ODS OUTPUT statement involves combining this data set with your original data. To do this, you need to restructure your original data set in what is sometimes referred to as "normal form." For example, if your original data set contained variables HR, SBP, and DBP (all in one observation) you would need to restructure it to contain three observations for each original observation, with variables VarName and Value. This allows you to merge your original data with the trimmed statistics generated by PROC UNIVARIATE. We will list several observations from each of the data sets following the presentation of the macro so you can see exactly how this works.

The final steps involve putting the list of outliers back in ID order and presenting it in a report. Here is the macro:

Program 2-26 Presenting a Macro to List Outliers of Several Variables Based on Trimmed Statistics (Using PROC UNIVARIATE)

```
%macro auto_outliers(
   Dsn=,      /* Data set name                          */
   ID=,       /* Name of ID variable                    */
   Var_list=, /* List of variables to check             */
              /* separate names with spaces             */
   Trim=.1,   /* Integer 0 to n = number to trim        */
              /* from each tail; if between 0 and .5, */
              /* proportion to trim in each tail        */
   N_sd=2     /* Number of standard deviations          */);
   ods listing close;
   ods output TrimmedMeans=trimmed(keep=VarName Mean Stdmean DF);
   proc univariate data=&Dsn trim=&Trim;
     var &Var_list;
   run;
   ods output close;

   data restructure;
      set &Dsn;
      length Varname $ 32;
      array vars[*] &Var_list;
      do i = 1 to dim(vars);
         Varname = vname(vars[i]);
         Value = vars[i];
         output;
      end;
      keep &ID Varname Value;
   run;

   proc sort data=trimmed;
      by Varname;
   run;

   proc sort data=restructure;
      by Varname;
   run;

   data outliers;
      merge restructure trimmed;
      by Varname;
```

```
      Std = StdMean*sqrt(DF + 1);
      if Value lt Mean - &N_sd*Std and not missing(Value)
         then do;
            Reason = 'Low  ';
            output;
         end;
      else if Value gt Mean + &N_sd*Std
         then do;
         Reason = 'High';
         output;
      end;
   run;

   proc sort data=outliers;
      by &ID;
   run;

   ods listing;
   title "Outliers based on trimmed Statistics";
   proc print data=outliers;
      id &ID;
      var Varname Value Reason;
   run;

   proc datasets nolist library=work;
      delete trimmed;
      delete restructure;
      *Note: work data set outliers not deleted;
   run;
   quit;
%mend auto_outliers;
```

If you call the macro like this:

```
%auto_outliers(Dsn=clean.patients,
               ID=Patno,
               Var_list=HR SBP DBP,
               Trim=.2,
               N_sd=2)
```

you will produce the following report:

```
Outliers based on trimmed Statistics

Patno    Varname    Value    Reason

 004       DBP       120      High
 004       SBP       200      High
 008       HR        210      High
 009       DBP       180      High
 009       SBP       240      High
 010       DBP       120      High
 010       SBP        40      Low
 011       DBP        20      Low
 011       SBP       300      High
 014       HR         22      Low
 017       HR        208      High
 020       DBP         8      Low
 020       HR         10      Low
 020       SBP        20      Low
 023       HR         22      Low
 023       SBP        34      Low
 321       DBP       200      High
 321       HR        900      High
 321       SBP       400      High
```

As promised earlier, if you look at the first few observations from each of the temporary data sets created by this macro, you will see more clearly how it works. Below are the selected observations from data sets TRIMMED and RESTRUCTURE:

```
Listing of Data Set TRIMMED

Var
Name        Mean     StdMean        DF

DBP      83.7500    3.570381        15
HR       74.5000    3.867447        15
SBP     131.7333    7.871275        14
```

```
Listing of Data Set RESTRUCTURE (partial listing)

Patno    Varname    Value

 001       DBP        80
 002       DBP        78
 003       DBP       100
     .
     .
     .
 001       HR         88
 002       HR         84
 003       HR         68
     .
     .
     .
 001       SBP       140
 002       SBP       120
 003       SBP       190
     .
     .
     .
```

Now you can see that if you merge these two data sets by Varname, you will have a value, the mean, and the standard deviation all in a single observation so you can test the values of your variables against the trimmed statistics.

Checking a Range Based on the Interquartile Range

Yet another way to look for outliers is a method devised by advocates of exploratory data analysis (EDA). This is a robust method, much like the previous method based on a trimmed mean. It uses the interquartile range (the distance from the 25^{th} percentile to the 75^{th} percentile) and defines an outlier as a multiple of the interquartile range above or below the upper or lower hinge, respectively. For those not familiar with EDA terminology, the lower hinge is the value corresponding to the 25^{th} percentile (the value below which 25% of the data values lie). The upper hinge is the value corresponding to the 75% percentile. For example, you may want to examine any data values more than two interquartile ranges above the upper hinge or below the lower hinge. This is an attractive method because it is independent of the distribution of the data values.

An easy way to determine the interquartile range and the upper and lower hinges is to use PROC MEANS to output these quantities. Presented next is a macro which is similar to the one in Program 2-23, but this one uses the number of interquartile ranges instead of an estimate of the standard deviation.

Program 2-27 Detecting Outliers Based on the Interquartile Range

```
%macro interquartile
   (/* the data set name (Dsn) and ID variable (Idvar)
       need to be assigned with %let statements
       prior to calling this macro */
       var=,     /* Variable to test for outliers  */
       n_iqr=2   /* Number of interquartile ranges */);

/*************************************************************
This macro will list outliers based on the interquartile range.

Example: To list all values beyond 1.5 interquartile ranges
   from a data set called clean.patients for a variable
   called hr, use the following:

   %let Dsn=clean.patients;
   %let Idvar=Patno;

   %interquartile(var=HR,
                  n_iqr=1.5)
*************************************************************/

   title "Outliers Based on &N_iqr Interquartile Ranges";

   proc means data=&dsn noprint;  ❶
      var &var;
      output out=tmp
             q1=Lower
             q3=Upper
             qrange=Iqr;
   run;
```

```
   data _null_;  ❷
      set &dsn(keep=&Idvar &Var);
      file print;
      if _n_ = 1 then set tmp;
      if &Var le Lower - &N_iqr*Iqr and not missing(&Var) or
         &Var ge Upper + &N_iqr*Iqr then
         put &Idvar= &Var=;
   run;

   proc datasets library=work;
      delete tmp;
   run;
   quit;
%mend interquartile;
```

Use PROC MEANS to output the values of the 25th and 75th percentile to a data set ❶. In the DATA _NULL_ step that follows, any values more than "n" interquartile ranges (the macro variable N_IQR) below the lower hinge or above the upper hinge are flagged as errors and reported ❷.

To demonstrate this macro, the calling sequence below checks for outliers more than 1.5 interquartile ranges above or below the upper or lower hinge, respectively. The calling statement is (assume that the macro values for &Dsn and &Idvar have already been assigned):

```
%interquartile(Var=HR, N_iqr=1.5)
```

with the resulting output shown next.

```
Outliers Based on 1.5 Interquartile Ranges
Patno=008 HR=210
Patno=017 HR=208
Patno=321 HR=900
Patno=020 HR=10
```

Summary

In this chapter you saw three basic tools to help you look for possible invalid numeric values. The first, most basic tool was to list the highest and lowest values (based on a number of observations or a percentage). The second tool was based on the concept that for some variables, it is possible to detect possible data errors by looking for values above or below known cutoff values. The third tool involved various ways to detect outliers based on the data values themselves. Three macros, TRIMMED, AUTO_OUTLIERS, and INTERQUARTILE were presented that should make this process fast and easy.

3 Checking for Missing Values

Introduction 91
Inspecting the SAS Log 91
Using PROC MEANS and PROC FREQ to Count Missing Values 93
Using DATA Step Approaches to Identify and Count Missing Values 96
Searching for a Specific Numeric Value 100
Creating a Macro to Search for Specific Numeric Values 102

Introduction

Many data sets contain missing values. There are several ways in which missing values can enter a SAS data set. First of all, the raw data value may be missing, either intentionally or accidentally. Next, an invalid raw value can cause a missing SAS value to be created. For example, reading a character value with a numeric informat will generate a missing value. Invalid dates are another common cause of missing values generated by SAS. Finally, many operations, such as assignment statements, can create missing values. This chapter investigates ways to detect and count missing values for numeric and character variables.

Inspecting the SAS Log

It is vitally important to carefully inspect the SAS Log, especially when creating a SAS data set for the first time. A log filled with messages about invalid data values is a clue that something may be wrong, either with the data or the program. If you know that a numeric field contains invalid character values, you may choose to read those data values with a character informat and to perform a character-to-numeric conversion (using the INPUT function) yourself. This will keep the SAS Log cleaner and make it easier to spot unexpected errors. Let's look at portions of the SAS Log that were generated when the PATIENTS data set was created.

```
NOTE: Libref LEARN was successfully assigned as follows:
      Engine:        V9
      Physical Name: c:\books\learning
NOTE: Libref CLEAN was successfully assigned as follows:
      Engine:        V9
      Physical Name: c:\books\clean

NOTE: AUTOEXEC processing completed.
```

```
1    *-------------------------------------------------------*
2    |PROGRAM NAME: PATIENTS.SAS in C:\BOOKS\CLEAN           |
3    |PURPOSE: To create a SAS data set called PATIENTS      |
4    *-------------------------------------------------------*;
5    libname clean "C:\BOOKS\CLEAN";
NOTE: Libref CLEAN was successfully assigned as follows:
      Engine:        V9
      Physical Name: C:\BOOKS\CLEAN
6
7    data clean.patients;
8       infile "c:\books\clean\patients.txt"
9              lrecl=30 pad; /* Pad short records
10                           with blanks  */
11
12      input @1  Patno    $3. @4  gender   $1.
13            @5  Visit    mmddyy10.
14            @15 HR       3.
15            @18 SBP      3.
16            @21 DBP      3.
17            @24 Dx       $3.
18            @27 AE       $1.;
19
20      LABEL Patno   = "Patient Number"
21            Gender  = "Gender"
22            Visit   = "Visit Date"
23            HR      = "Heart Rate"
24            SBP     = "Systolic Blood Pressure"
25            DBP     = "Diastolic Blood Pressure"
26            Dx      = "Diagnosis Code"
27            AE      = "Adverse Event?";
28       format visit mmddyy10.;
29   run;

NOTE: The infile "c:\books\clean\patients.txt" is:
      File Name=c:\books\clean\patients.txt,
      RECFM=V,LRECL=30

NOTE: Invalid data for Visit in line 7 5-14.
RULE:     ----+----1----+----2----+----3----+----4----+----5----+----6--
7         007M08/32/1998 88148102   0
Patno=007 gender=M Visit=. HR=88 SBP=148 DBP=102 Dx=  AE=0 _ERROR_=1
_N_=7
NOTE: Invalid data for Visit in line 12 5-14.
12        011M13/13/1998 68300 20  41
Patno=011 gender=M Visit=. HR=68 SBP=300 DBP=20 Dx=4 AE=1 _ERROR_=1
_N_=12
```

```
NOTE: Invalid data for Visit in line 21 5-14.
21        123M15/12/1999 60        10
Patno=123 gender=M Visit=. HR=60 SBP=. DBP=. Dx=1 AE=0 _ERROR_=1 _N_=21
NOTE: Invalid data for Visit in line 23 5-14.
23        020F99/99/9999 10 20  8   0
Patno=020 gender=F Visit=. HR=10 SBP=20 DBP=8 Dx=  AE=0 _ERROR_=1 _N_=23
NOTE: Invalid data for Visit in line 28 5-14.
NOTE: Invalid data for HR in line 28 15-17.
28        027FNOTAVAIL  NA 166106  70
Patno=027 gender=F Visit=. HR=. SBP=166 DBP=106 Dx=7 AE=0 _ERROR_=1
_N_=28
NOTE: 31 records were read from the infile
      "c:\books\clean\patients.txt".
      The minimum record length was 26.
      The maximum record length was 27.
NOTE: The data set CLEAN.PATIENTS has 31 observations and 8 variables.
```

The first invalid data message is generated by an invalid date (08/32/1998). This will be discussed in more detail in Chapter 4, "Working with Dates". For now, realize that a numeric missing value (remember that dates are stored as numeric values) will be generated as a result of this invalid date. Several more invalid date messages follow. A missing value for heart rate (HR) was generated for patient number 027 because of the character value 'NA' (not available or not applicable) that was entered. Before going any further, the invalid dates need to be checked, and a decision needs to be made concerning the 'NA' value for heart rate.

Using PROC MEANS and PROC FREQ to Count Missing Values

There are several procedures that will count missing values for you. It may be normal to have missing values for certain variables in your data set. There may also be variables for which no missing values are permitted (such as a patient ID). An easy way to count missing values for numeric variables is by using PROC MEANS; for character variables, PROC FREQ will provide this information. Program 3-1 is a simple program that can be used to check the number of numeric and character missing values in the PATIENTS data set.

Program 3-1 Counting Missing and Non-missing Values for Numeric and Character Variables

```
libname clean "c:\books\clean";

title "Missing value check for the patients data set";
proc means data=clean.patients n nmiss;
run;

proc format;
   value $misscnt ' '   = 'Missing'
                  other = 'Nonmissing';
run;

proc freq data=clean.patients;
   tables _character_ / nocum missing;
   format _character_ $misscnt.;
run;
```

The check for numeric missing values is straightforward. By using the N and NMISS options with PROC MEANS, you get a count of the non-missing and missing values for all your numeric variables (the default if no VAR statement is included). You could also choose to use a VAR statement to list only the variables of interest.

Counting missing values for character variables takes an extra step. First, you do not simply want to create one-way frequencies for all the character variables. Some variables, such as patient ID (Patno) can, conceivably, have thousands of values. By creating a character format that has only two value ranges, one for missing and the other for everything else, you can have PROC FREQ count missing and non-missing values for you.

Notice also that it is necessary to use the SAS keyword _CHARACTER_ in the TABLES statement (or to provide a list of character variables). PROC FREQ can produce frequency tables for numeric as well as character variables. Finally, the TABLES option MISSING includes the missing values in the body of the frequency listing. (Note: If you use the MISSING option with PROC FREQ and you request percentages, the percentage calculation uses all the values, missing or non-missing, in the denominator rather than just the number of non-missing values.) Examination of the listing from these two procedures is a good first step in your investigation of missing values. The output from Program 3-1 is shown next.

Missing value check for the patients data set

The MEANS Procedure

Variable	Label	N	N Miss
Visit	Visit Date	24	7
HR	Heart Rate	28	3
SBP	Systolic Blood Pressure	27	4
DBP	Diastolic Blood Pressure	28	3

Missing value check for the patients data set

The FREQ Procedure

Patient Number

Patno	Frequency	Percent
Missing	1	3.23
Nonmissing	30	96.77

Gender

gender	Frequency	Percent
Missing	1	3.23
Nonmissing	30	96.77

Diagnosis Code

Dx	Frequency	Percent
Missing	8	25.81
Nonmissing	23	74.19

(*continued*)

```
          Adverse Event?

AE            Frequency     Percent
_____________________________________
Missing               1        3.23
Nonmissing           30       96.77
```

Using DATA Step Approaches to Identify and Count Missing Values

Counting missing values is not usually enough. If you have variables for which missing values are not allowed, you need to locate the observations so that the original data values can be checked and the errors corrected. A simple DATA step with a PUT statement is one approach. Program 3-2 checks for any missing visit dates, heart rates (HR), or adverse events (AE).

Program 3-2 Writing a Simple DATA Step to List Missing Data Values and an ID Variable

```
title "Listing of missing values";
data _null_;
   file print; ***send output to the output window;
   set clean.patients(keep=Patno Visit HR AE);
   if missing(Visit) then
      put "Missing or invalid visit date for ID " Patno;
   if missing(HR) then put "Missing or invalid HR for ID " Patno;
   if missing(AE) then put "Missing value for ID " Patno;
run;
```

Notice how convenient it is to use the MISSING function to test for both numeric or character missing values. Using the MISSING function also makes the program easier to read (at least for this author). Output from running Program 3-2 is shown next.

```
Listing of missing values
Missing or invalid visit date for ID 007
Missing or invalid HR for ID 010
Missing or invalid visit date for ID 011
Missing value for ID 013
Missing or invalid visit date for ID 015
Missing or invalid visit date for ID 123
Missing or invalid visit date for ID 321
Missing or invalid visit date for ID 020
Missing or invalid visit date for ID 027
Missing or invalid HR for ID 027
Missing or invalid HR for ID 029
```

What do you do about missing patient numbers? Obviously, you can't list which patient number is missing because you don't have that information. One possibility is to report the patient number or numbers preceding the missing number (in the original order of data entry). If you sort the data set first, all the missing values will "float" to the top and you will not have a clue as to which patients they belong to. Here is a program that prints out the two previous patient ID's when a missing ID is found.

Program 3-3 Attempting to Locate a Missing or Invalid Patient ID by Listing the Two Previous ID's

```
title "Listing of missing patient numbers";
data _null_;
   set clean.patients;
   ***Be sure to run this on the unsorted data set;
   file print;
   Prev_id = lag(Patno);
   Prev2_id = lag2(Patno);
   if missing(Patno) then put "Missing patient ID. Two previous ID's are:"
      Prev2_id "and " Prev_id / @5 "Missing record is number " _n_;
   else if notdigit(trim(Patno)) then
      put "Invalid patient ID:" patno +(-1)". Two previous ID's are:"
      Prev2_id "and " Prev_id / @5 "Missing record is number " _n_;
run;
```

Although there are several solutions to listing the patient numbers from the preceding observations, the LAG function serves the purpose here. The LAG function returns the value of its argument the last time the function executed. If you execute this function for every iteration of the DATA step, it returns the value of Patno from the previous observation. The LAG2 function, when used in the same manner, returns the value of Patno from the observation before that. Remember to execute the LAG and LAG2 functions for every observation.

When Program 3-3 is run and a missing patient number is encountered, the two lagged variables will be the ID's from the previous two observations. The assumption in this program is that there are not more than three missing patient numbers in a row. If that is a possibility, you could list more than two previous patient ID's or include patient ID's following the missing one as well. Notice that we added the observation number to the output by printing the value of the internal SAS variable _N_. This provides one additional clue in finding the missing patient number.

The last part of the program uses the NOTDIGIT function to test for any invalid, non-missing values for the patient number. NOTDIGIT returns the first character in a character value that is not a digit. The TRIM function ensures that any trailing blanks (which are treated as non-digits) are removed before the test is made.

Here is the output from Program 3-3:

```
Listing of missing patient numbers
Invalid patient ID:XX5. Two previous ID's are:003 and 004
    Missing record is number 5
Missing patient ID. Two previous ID's are:006 and 007
    Missing record is number 8
```

Another approach is to list the values of all the variables for any missing or invalid patient ID. This may give a clue to the identity of the missing ID. Using PROC PRINT with a WHERE statement makes this an easy task, as demonstrated by the SAS code in Program 3-4.

Program 3-4 Using PROC PRINT to List Data for Missing or Invalid Patient ID's

```
title "Data listing for patients with missing or invalid ID's";
proc print data=clean.patients;
   where missing(Patno) or notdigit(trim(Patno));
run;
```

Here is the corresponding output:

```
Data listing for patients with missing or invalid ID's

Obs    Patno    gender         Visit    HR    SBP    DBP    Dx    AE

  5     XX5       M       05/07/1998    68    120     80    1     0
  8               M       11/11/1998    90    190    100          0
```

Before leaving this section on DATA step detection of missing values, let's modify Program 3-2, which listed missing dates, heart rates, and adverse events, to count the number of each missing variable as well.

Program 3-5 Listing and Counting Missing Values for Selected Variables

```
title "Listing of missing values";
data _null_;
   set clean.patients(keep=Patno Visit HR AE) end=last;
   file print; ***Send output to the output window;
   if missing(Visit) then do;
      put "Missing or invalid visit date for ID " Patno;
      N_visit + 1;
   end;
   if missing(HR) then do;
      put "Missing or invalid HR for ID " Patno;
      N_HR + 1;
   end;
   if missing(AE) then do;
      put "Missing AE for ID " Patno;
      N_AE + 1;
   end;

   if last then
           put // "Summary of missing values" /
           25*'-' /
           "Number of missing dates = " N_visit /
           "Number of missing HR's = " N_HR /
           "Number of missing adverse events = " N_AE;
run;
```

Each time a missing value is located, the respective missing counter is incremented by 1. Because you only want to see the totals once after all the data lines have been read, use the END= option in the SET statement to create the logical variable LAST. LAST will be true when the last observation is being read from the PATIENTS data set. So, in addition to the earlier listing, you have the additional lines of output shown next.

```
Listing of missing values
Missing or invalid visit date for ID 007
Missing or invalid HR for ID 010
Missing or invalid visit date for ID 011
Missing AE for ID 013
Missing or invalid visit date for ID 015
Missing or invalid visit date for ID 123
Missing or invalid visit date for ID 321
Missing or invalid visit date for ID 020
Missing or invalid visit date for ID 027
Missing or invalid HR for ID 027
Missing or invalid HR for ID 029


Summary of missing values
-------------------------
Number of missing dates = 7
Number of missing HR's = 3
Number of missing adverse events = 1
```

Searching for a Specific Numeric Value

Specific values such as 999 or 9999 are sometimes used to denote missing values. For example, numeric values that are left blank in Dbase files are stored as zeros. If 0 is a valid value for some of your variables, this can lead to problems. So, values such as 999 or 9999 are sometimes used instead of zeros or blanks. Program 3-6 searches a SAS data set for all numeric variables set to a specific value and produces a report which shows the variable name and the observation where the specific value was found.

The "trick" in this program is the relatively unknown function called VNAME (see *SAS Functions by Example* by this author for more details). This function returns the variable name of an array element. The first program (Program 3-6) searches a SAS data set for a specific value. The program is then generalized by using the data set name and the specific value as calling arguments in a macro. Here is the first program.

Program 3-6 Identifying All Numeric Variables Equal to a Fixed Value (Such as 999)

```
***Create test data set;
data test;
   input X Y A $ X1-X3 Z $;
datalines;
1 2 X 3 4 5 Y
2 999 Y 999 1 999 J
999 999 R 999 999 999 X
1 2 3 4 5 6 7
;

***Program to detect the specified values;
data _null_;
   set test;
   file print;
   array nums[*] _numeric_;
   length Varname $ 32;
   do __i = 1 to dim(nums);
      if nums[__i] = 999 then do;
         Varname = vname(nums[__i]);
         put "Value of 999 found for variable " Varname
             "in observation " _n_;
      end;
   end;
   drop __i;
run;
```

Key to this program is the use of _NUMERIC_ in the ARRAY statement. Because this ARRAY statement follows the SET statement, the array NUMS will contain all the numeric variables in the data set TEST. The next step is to examine each of the elements in the NUMS array, determine if a value of 999 is found, and then determine the variable name associated with that array element. The DO loop uses the index variable __i in the hopes that there will not be any variables in the data set to be tested with that name.

Now for the "trick." As you search for values of 999 for each of the numeric variables, you can use the VNAME function to return the variable name that corresponds to the array element. In this program, the variable name is stored in the variable Varname. All that is left to do is write out the variable names and observation numbers. Output from Program 3-6 is shown below:

```
Listing of missing values
Value of 999 found for variable Y in observation 2
Value of 999 found for variable X1 in observation 2
Value of 999 found for variable X3 in observation 2
Value of 999 found for variable X in observation 3
Value of 999 found for variable Y in observation 3
Value of 999 found for variable X1 in observation 3
Value of 999 found for variable X2 in observation 3
Value of 999 found for variable X3 in observation 3
```

Creating a Macro to Search for Specific Numeric Values

Before we leave this chapter, let's modify the program above so that it only produces a summary report on variables with specific missing values. In addition, we will turn the program into a macro so that it will be more generally useful. (If you are uncomfortable with SAS macros, you can simply take the body of the code out of the macro and replace all the variable names or values that start with ampersands (&) with variables names and numbers.) Here is the macro:

Program 3-7 Creating a Macro to Search for Specific Numeric Values

```
*-------------------------------------------------------------------*
| Macro name: find_value.sas  in c:\books\clean                     |
| purpose: Identifies any specified value for all numeric vars      |
| Calling arguments: dsn=   sas data set name                       |
|                    value=   numeric value to search for           |
| example:  to find variable values of 9999 in data set test, use   |
|                                                                   |
|           %find_value(dsn=test, value=9999)                       |
*-------------------------------------------------------------------*;
%macro find_value(dsn=,      /* The data set name                 */
                  value=999  /* Value to look for, default is 999 */ );
   title "Variables with &value as missing values";
   data temp;
      set &dsn;
      file print;
      length Varname $ 32;
      array nums[*] _numeric_;
```

```
      do __i = 1 to dim(nums);
         if nums[__i] = &value then do;
         Varname = vname(nums[__i]);
         output;
         end;
      end;
      keep Varname;
   run;
   proc freq data=temp;
      tables Varname / out=summary(keep=Varname Count)
                       nocum;
   run;
   proc datasets library=work;
      delete temp;
   run;
   quit;
%mend find_value;
```

There are several points in this macro that need explanation: First, the two calling arguments in this macro are the data set name and the value that you want to search for. The program creates an array of all numeric variables in the data set to be tested. A DO loop then tests to see if any of the numeric variables contain the specified value (such as 999). If so, the VNAME function determines the variable name corresponding to the array element and an observation is output to data set TEMP. This data set contains the single variable Varname. To help make this clear, here is a listing of data set TEMP when the macro is run on data set TEST looking for a value of 999:

```
Listing of data set TEMP

Obs    Varname

 1       Y
 2       X1
 3       X3
 4       X
 5       Y
 6       X1
 7       X2
 8       X3
```

You can now use PROC FREQ to list each of the variable names for which the specific value is found, along with the frequency. The final section of the macro deletes the temporary data set TEMP. Here is the output from calling the FIND_VALUE macro as follows:

```
%find_value(dsn=test, value=999)
```

```
Variables with 999 as missing values

The FREQ Procedure

Varname    Frequency     Percent
_____________________________________
X                  1       12.50
X1                 2       25.00
X2                 1       12.50
X3                 2       25.00
Y                  2       25.00
```

4 Working with Dates

Introduction 105
Checking Ranges for Dates (Using a DATA Step) 106
Checking Ranges for Dates (Using PROC PRINT) 107
Checking for Invalid Dates 108
Working with Dates in Nonstandard Form 111
Creating a SAS Date When the Day of the Month Is Missing 113
Suspending Error Checking for Known Invalid Dates 114

Introduction

SAS dates seem mysterious to many people, but by understanding how SAS dates are stored, you will see that they are really quite simple. SAS dates are stored as numeric variables and represent the number of days from a fixed point in time, January 1, 1960. The confusion develops in the many ways that SAS software can read and write dates. Typically, dates are read as MM/DD/YYYY or some similar form. There are informats to read almost any conceivable date notation. Regardless of how a date is read, the informat performs the conversion to a SAS date, and it is stored just like any other numeric value. If you print out a date value without a SAS date format, it will appear as a number (the number of days from January 1, 1960) rather than a date in one of the standard forms. When date information is not in a standard form, you can read the month, day, and year information as separate variables and use the MDY (month-day-year) function to create a SAS date. Let's look at some ways to perform data cleaning and validation with dates.

Checking Ranges for Dates (Using a DATA Step)

Suppose you want to determine if the visit dates in the PATIENTS data set are between June 1, 1998 and October 15, 1999. You can use a DATA step approach, much the same way you did when checking that numeric variables were within a specified range. The only difference is that the range boundaries have to be SAS dates. Let's see how this works. Program 4-1 checks for dates in the specified range and ignores missing values.

Program 4-1 Checking That a Date Is within a Specified Interval (DATA Step Approach)

```
libname clean "c:\books\clean";

title "Dates before June 1, 1998 or after October 15, 1999";
data _null_;
   file print;
   set clean.patients(keep=Visit Patno);
   if Visit lt '01jun1998'd and not missing(Visit) or
      Visit gt '15oct1999'd then put Patno= Visit= mmddyy10.;
run;
```

The key to this program is the use of the date constants in the IF statement. If you want SAS to turn a date into a SAS date (the number of days from 1/1/1960), the dates must be written in this fashion. Date constants are written as a one- or two-digit day, a three-character month name, and a two- or four-digit year, placed in single or double quotes, and followed by a lowercase or uppercase 'D'. You also need to add a date format in the PUT statement so that the date will be printed in a standard date format. Output from Program 4-1 is shown next.

```
Dates before June 1, 1998 or after October 15, 1999
Patno=XX5 Visit=05/07/1998
Patno=010 Visit=10/19/1999
Patno=003 Visit=11/12/1999
Patno=028 Visit=03/28/1998
Patno=029 Visit=05/15/1998
```

Checking Ranges for Dates (Using PROC PRINT)

You can accomplish the same objective by using a PROC PRINT statement in combination with a WHERE statement. Besides being easier to code, this approach allows you to use the keyword BETWEEN with the WHERE statement, making the logic somewhat simpler to follow. The SAS code is shown in Program 4-2.

Program 4-2 Checking That a Date Is within a Specified Interval (Using PROC PRINT and a WHERE Statement)

```
title "Dates before June 1, 1998 or after October 15, 1999";
proc print data=clean.patients;
   where Visit not between '01jun1998'd and '15oct1999'd and
   not missing(Visit);
   id Patno;
   var Visit;
   format Visit date9.;
run;
```

Output from this procedure contains the identical information as the previous DATA step approach. For variety, we chose the DATE9. date format to replace the mmddyy10. format that was associated with the Visit variable in the DATA step.

```
Dates before June 1, 1998 or after October 15, 1999

Patno        Visit

 XX5     07MAY1998
 010     19OCT1999
 003     12NOV1999
 028     28MAR1998
 029     15MAY1998
```

Checking for Invalid Dates

Some of the dates in the PATIENTS data set are missing and some are invalid dates, which were converted to missing values during the input process. If you want to distinguish between the two, you must work from the raw data, not the SAS data set. If you attempt to read an invalid date with a SAS date informat, an error message will appear in the SAS Log. This is one clue that you have errors in your date values. Program 4-3 reads the raw data file PATIENTS.TXT. The resulting SAS Log follows.

Program 4-3 Reading Dates with the MMDDYY10. Informat

```
data dates;
   infile "c:\cleaning\patients.txt" truncover;
   input @5 Visit mmddyy10.;
   format Visit mmddyy10.;
run;
```

The SAS Log that results from running this program is shown next.

```
52   data dates;
53      infile "c:\books\clean\patients.txt" truncover lrecl=20;
54      input @5 Visit mmddyy10.;
55      format Visit mmddyy10.;
56   run;

NOTE: The infile "c:\books\clean\patients.txt" is:
      File Name=c:\books\clean\patients.txt,
      RECFM=V,LRECL=20

NOTE: Invalid data for Visit in line 7 5-14.
RULE:     ----+----1----+----2----+----3----+----4----+----5----+----6--
7         007M08/32/1998 88148
Visit=. _ERROR_=1 _N_=7
NOTE: Invalid data for Visit in line 12 5-14.
12        011M13/13/1998 68300
Visit=. _ERROR_=1 _N_=12
NOTE: Invalid data for Visit in line 21 5-14.
21        123M15/12/1999 60
Visit=. _ERROR_=1 _N_=21
NOTE: Invalid data for Visit in line 23 5-14.
23        020F99/99/9999 10 20
Visit=. _ERROR_=1 _N_=23
NOTE: Invalid data for Visit in line 28 5-14.
```

```
28        027FNOTAVAIL  NA 166
Visit=. _ERROR_=1 _N_=28
NOTE: 31 records were read from the infile
      "c:\books\clean\patients.txt".
      The minimum record length was 20.
      The maximum record length was 20.
```

There are several reasons why these dates caused error reports in the SAS Log. In some cases, the month was greater than 12, possibly caused by reading a date that was actually in day-month-year form rather than month-day-year form. Other dates such as 99/99/9999 were an attempt to indicate that no date information was available. Obviously, the data value of NOTAVAIL (not available) caused an error. Remember that once the errors exceed a default number, they will no longer be reported in the SAS Log. This number can be adjusted by setting the system option ERRORS=. If you have no missing date values in your data, any missing date value must have been generated by an invalid date. You can use this idea to list missing and invalid dates. The plan is to read the date value twice; once with a SAS date informat, the other as a character value. This way, you can see the original date value that caused the error. Program 4-4 demonstrates this.

Program 4-4 Listing Missing and Invalid Dates by Reading the Date Twice, Once with a Date Informat and the Second as Character Data

```
title "Listing of missing and invalid dates";
data _null_;
   file print;
   infile "c:\books\clean\patients.txt" truncover;
   input @1 Patno $3.
         @5 Visit mmddyy10.
         @5 V_date $char10.;
   format Visit mmddyy10.;
   if missing(Visit) then put Patno= V_date=;
run;
```

Here you read the date twice, first with the SAS date informat MMDDYY10. and then with the character informat $CHAR10. This way, even though the SAS System substitutes a missing value for VISIT, the variable V_DATE will contain the actual characters that were entered in the date field.

This is a good place to discuss the difference between the $n. and $CHARn informats: The more common $n informat will left-justify a character value. That is, if there are leading blanks in the columns being read, the resulting character value in your data set will start with the first non-blank character. The $CHARn informat reads the specified columns as is, and leaves leading blanks, if there are any, in the character value. We chose to use the $CHAR informat in the program above so that you could see exactly what was contained in the 10 columns holding the date.

An alternative is to read the original date only once as a character value, and use the INPUT function to create the SAS date. Remember that the INPUT function "reads" the value of the first argument, using the informat listed as the second argument. The alternative program is shown in Program 4-5.

Program 4-5 Listing Missing and Invalid Dates by Reading the Date as a Character Variable and Converting to a SAS Date with the INPUT Function

```
title "Listing of missing and invalid dates";
data _null_;
   file print;
   infile "c:\books\clean\patients.txt" truncover;
   input @1 Patno $3.
         @5 V_date $char10.;
   Visit = input(V_date,mmddyy10.);
   format Visit mmddyy10.;
   if missing(Visit) then put Patno= V_date=;
run;
```

Running either Program 4-4 or Program 4-5 results in the following output.

```
Listing of missing and invalid dates
Patno=007 V_date=08/32/1998
Patno=011 V_date=13/13/1998
Patno=015 V_date=
Patno=123 V_date=15/12/1999
Patno=321 V_date=
Patno=020 V_date=99/99/9999
Patno=027 V_date=NOTAVAIL
```

If you want to ignore real missing values, you only need to make a slight change as shown in Program 4-6.

Program 4-6 Removing the Missing Values from the Invalid Date Listing

```
title "Listing only invalid dates";
data _null_;
   file print;
   infile "c:\books\clean\patients.txt" truncover;
   input @1 Patno $3.
         @5 V_date $char10.;

   Visit = input(V_date,mmddyy10.);
   format Visit mmddyy10.;
   if missing(Visit) and not missing(V_date) then put Patno= V_date=;
run;
```

If the numeric date value is missing and the character value holding the date is not missing, that indicates that you have located a non-missing, invalid value in the raw data and the dates will be printed. Output from Program 4-6 is shown below:

```
Listing only invalid dates
Patno=007 V_date=08/32/1998
Patno=011 V_date=13/13/1998
Patno=123 V_date=15/12/1999
Patno=020 V_date=99/99/9999
Patno=027 V_date=NOTAVAIL
```

Working with Dates in Nonstandard Form

Although SAS software can read dates in almost every conceivable form, there may be times when you have date information for which there is no SAS informat or you are given a SAS data set where, instead of a variable representing a date, you have separate variables for month, day, and year. In either case, there is an easy solution—use the MDY (month day year) function.

As an example suppose you have a month value (a number from 1 to 12) in columns 6–7, a day of the month value (a number from 1 to 31) in columns 13–14, and a four-digit year value in columns 20–23. Enter the three variable names for the month, day, and year as arguments to this function, and it will return a SAS date. Program 4-7 demonstrates how this works.

Program 4-7 Demonstrating the MDY Function to Read Dates in Nonstandard Form

```
data nonstandard;
   input Patno $ 1-3 Month 6-7 Day 13-14 Year 20-23;
   Date = mdy(Month,Day,Year);
   format date mmddyy10.;
datalines;
001  05     23     1998
006  11     01     1998
123  14     03     1998
137  10            1946
;
title "Listing of data set NONSTANDARD";
proc print data=nonstandard;
   id Patno;
run;
```

Notice that an invalid MONTH value (observation three) and a missing DAY value (observation four) were included intentionally. The listing of the data set NONSTANDARD follows.

```
Listing of data set NONSTANDARD

Patno    Month    Day    Year          Date

 001        5      23    1998    05/23/1998
 006       11       1    1998    11/01/1998
 123       14       3    1998             .
 137       10       .    1946             .
```

In the two cases where a date could not be computed, a missing value was generated. Inspection of the SAS Log also shows that the MDY function had an invalid value and a missing value.

Creating a SAS Date When the Day of the Month Is Missing

Some of your date values may be missing the day of the month, but you would still like to create a SAS date by using either the 1st or the 15th of the month as the day. There are two possibilities here. One method is to use the MONYY informat that reads dates in the form of a three-character month name and a two- or four-digit year. If your dates are in this form, SAS will create a SAS date using the first of the month as the day value. The other method of creating a SAS date from only month and year values is to use the MDY function, substituting a value such as 15 for the day argument. An example is shown in Program 4-8.

Program 4-8 Creating a SAS Date When the Day of the Month Is Missing

```
data no_day;
   input @1 Date1 monyy7. @8 Month 2. @10 Year 4.;
   Date2 = mdy(Month,15,Year);
   format Date1 Date2 mmddyy10.;
datalines;
JAN98  011998
OCT1998101998
;
title "Listing of data set NO_DAY";
proc print data=NO_DAY;
run;
```

DATE1 is a SAS date created by the MONYY SAS informat; DATE2 is created by the MDY function, using the 15th of the month as the missing day value. Output from PROC PRINT is shown next.

```
Listing of data set NO_DAY

Obs         Date1    Month    Year         Date2

 1     01/01/1998       1     1998    01/15/1998
 2     10/01/1998      10     1998    10/15/1998
```

Let's extend this idea a bit further. Suppose most of your dates have month, day, and year values, but for any date where the only piece missing is the day of the month, you want to substitute the 15th of the month. Program 4-9 will accomplish this goal.

Program 4-9 Substituting the 15th of the Month When the Date of the Month Is Missing

```
data miss_day;
   input @1  Patno  $3.
         @4  Month   2.
         @6  Day     2.
         @8  Year    4.;
   if not missing(Day) then Date = mdy(Month,Day,Year);
   else Date = mdy(Month,15,Year);
   format Date mmddyy10.;
datalines;
00110211998
00205  1998
00344  1998
;
title "Listing of data set MISS_DAY";
proc print data=miss_day;
run;
```

If the day value is not missing, the MDY function uses all three values of month, day, and year to compute a SAS date. If the day value is missing, the 15th of the month is used. As before, if there is an invalid date (such as for patient 003), a missing date value is generated. Here are the three observations created by this program.

```
Listing of data set MISS_DAY

Obs    Patno    Month    Day    Year          Date

 1      001       10      21    1998    10/21/1998
 2      002        5       .    1998    05/15/1998
 3      003       44       .    1998             .
```

Suspending Error Checking for Known Invalid Dates

As you saw earlier, invalid date values can fill your SAS Log with lots of errors. There are times when you know that invalid date values were used to represent missing dates or other specific values. If you would like to prevent the automatic listing of date errors in the SAS Log, you can use the double question mark (??) modifier in your INPUT statement or with the INPUT

function. This modifier prevents the NOTES and data listings to be printed in the SAS Log and also keeps the SAS internal variable _ERROR_ at 0.

Program 4-10 uses the ?? modifier in the INPUT statement to prevent error messages from printing in the SAS Log.

Program 4-10 Suspending Error Checking for Known Invalid Dates by Using the ?? Informat Modifier

```
data dates;
   infile "c:\books\clean\patients.txt" truncover;
   input @5 visit ?? mmddyy10.;
   format visit mmddyy10.;
run;
```

When this program is run, there will be no error messages in the SAS Log caused by invalid dates. **Turn off SAS error checking only when you plan to detect errors in other ways, or when you already know all about your invalid dates**.

Program 4-11 shows an example of using the ?? informat modifier with the INPUT function. The following program is identical to Program 4-5, with the addition of the ?? modifier to keep the SAS Log free of error messages.

Program 4-11 Demonstrating the ?? Informat Modifier with the INPUT Function

```
title "Listing of missing and invalid dates";
data _null_;
   file print;
   infile "c:\books\clean\patients.txt" truncover;
   input @1 Patno $3.
         @5 V_date $char10.;
   Visit = input(V_date,?? mmddyy10.);
   format Visit mmddyy10.;
   if missing(Visit) then put Patno= V_date=;
run;
```

Remember that you can use the ?? modifier before the informat argument of the INPUT function as well as the more traditional use with the INPUT statement. If you have a lot of known date errors, the overriding of the error messages will also improve program efficiency. (Remember that this also sets the value of _ERROR_ to 0.)

5 Looking for Duplicates and “n” Observations per Subject

Introduction 117
Eliminating Duplicates by Using PROC SORT 117
Detecting Duplicates by Using DATA Step Approaches 123
Using PROC FREQ to Detect Duplicate ID's 126
Selecting Patients with Duplicate Observations by Using a Macro List and SQL 129
Identifying Subjects with "n" Observations Each (DATA Step Approach) 130
Identifying Subjects with "n" Observations Each (Using PROC FREQ) 132

Introduction

Besides checking for invalid data values in a data set, it may be necessary to check for either duplicate ID's or duplicate observations. Duplicate observations are easy to fix; just eliminate the duplicates (although you may want to find out how the duplicates got there). Duplicate ID's with different data values present another problem. One possible cause of this is that the same ID was used for more than one person. Another possibility is that different data values were entered more than once for the same person. A third possibility is that multiple records per ID are expected. There are several ways to detect and eliminate unwanted duplicates in a SAS data set. This chapter explores some of them. (For more detailed information about working with data sets with multiple observations per subject, please take a look at *Longitudinal Data and SAS: A Programmer's Guide*, which was written by this author and published by SAS Institute Inc., Cary, NC.)

Eliminating Duplicates by Using PROC SORT

Suppose you have a data set where each patient is supposed to be represented by a single observation. To demonstrate what happens when you have multiple observations with the same ID, some duplicates in the PATIENTS data set were included on purpose. Observations with duplicate ID numbers are shown next.

Listing of Duplicated ID's in the Patients Data Set

Patno	gender	Visit	HR	SBP	DBP	Dx	AE
002	F	11/13/1998	84	120	78	X	0
002	F	11/13/1998	84	120	78	X	0
003	X	10/21/1998	68	190	100	3	1
003	M	11/12/1999	58	112	74		0
006		06/15/1999	72	102	68	6	1
006	F	07/07/1999	82	148	84	1	0

Notice that patient number 002 is a true duplicate observation. For patient numbers 003 and 006, the duplicate ID's contain different values.

Two very useful options of PROC SORT are NODUPKEY and NODUPRECS (also called NODUP). The NODUPKEY option automatically eliminates multiple observations where the BY variables have the same value. For example, to automatically eliminate multiple patient ID's (Patno) in the PATIENTS data set (which you probably would not want to do; this is for illustration only), you could use PROC SORT with the NODUPKEY option as shown in Program 5-1.

Program 5-1 Demonstrating the NODUPKEY Option of PROC SORT

```
proc sort data=clean.patients out=single nodupkey;
   by Patno;
run;

title "Data Set SINGLE - Duplicated ID's Removed from PATIENTS";
proc print data=single;
   id Patno;
run;
```

Notice that two options, OUT= and NODUPKEY, are used here. The OUT= option is used to create the new data set SINGLE, leaving the original data set PATIENTS unchanged. Shown next is a listing of the SINGLE data set.

Data Set SINGLE - Duplicated ID's Removed from PATIENTS

Patno	gender	Visit	HR	SBP	DBP	Dx	AE
	M	11/11/1998	90	190	100		0
001	M	11/11/1998	88	140	80	1	0
002	F	11/13/1998	84	120	78	X	0
003	X	10/21/1998	68	190	100	3	1
004	F	01/01/1999	101	200	120	5	A
006		06/15/1999	72	102	68	6	1
007	M	.	88	148	102		0
008	F	08/08/1998	210	.	.	7	0
009	M	09/25/1999	86	240	180	4	1
010	f	10/19/1999	.	40	120	1	0
011	M	.	68	300	20	4	1
012	M	10/12/1998	60	122	74		0
013	2	08/23/1999	74	108	64	1	
014	M	02/02/1999	22	130	90		1
015	F	.	82	148	88	3	1
017	F	04/05/1999	208	.	84	2	0
019	M	06/07/1999	58	118	70		0
020	F	.	10	20	8		0
022	M	10/10/1999	48	114	82	2	1
023	f	12/31/1998	22	34	78		0
024	F	11/09/1998	76	120	80	1	0
025	M	01/01/1999	74	102	68	5	1
027	F	.	.	166	106	7	0
028	F	03/28/1998	66	150	90	3	0
029	M	05/15/1998	.	.	.	4	1
123	M	.	60	.	.	1	0
321	F	.	900	400	200	5	1
XX5	M	05/07/1998	68	120	80	1	0

The NODUPKEY option eliminated the second observation for each of the three duplicate ID's. The only indication that duplicates were removed is in the NOTE in the SAS Log, which is shown next.

```
54   proc sort data=clean.patients out=single nodupkey;
55      by Patno;
56   run;

NOTE: There were 31 observations read from the data set CLEAN.PATIENTS.
NOTE: 3 observations with duplicate key values were deleted.
NOTE: The data set WORK.SINGLE has 28 observations and 8 variables.
NOTE: PROCEDURE SORT used (Total process time):
      real time           0.04 seconds
      cpu time            0.02 seconds
```

This method of looking for duplicate ID's is really only useful if the SAS Log shows that no duplicates were removed. If the SAS Log shows duplicate key values were deleted and duplicates were not expected, you need to see which ID's had duplicate data and the nature of the data.

If you use the NODUPKEY option with more than one BY variable, only those observations with identical values on each of the BY variables will be deleted. For example, if you sort by patient number (Patno) and visit date (VISIT), only the duplicate for patient number 002 will be deleted when you use the NODUPKEY option, because the two observations for patient number 002 are the only ones with the same patient number and visit date.

The option NODUPRECS also deletes duplicates, but only for two observations where all the variables have identical values. Program 5-2 demonstrates this option.

Program 5-2 Demonstrating the NODUPRECS Option of PROC SORT

```
proc sort data=clean.patients out=single noduprecs;
   by Patno;
run;
```

Listing the data set SINGLE, which is created by this procedure, shows that only the second observation for patient number 002 was deleted (see the listing below):

Listing of data set SINGLE (partial listing)

Patno	gender	Visit	HR	SBP	DBP	Dx	AE
	M	11/11/1998	90	190	100		0
001	M	11/11/1998	88	140	80	1	0
002	F	11/13/1998	84	120	78	X	0
003	X	10/21/1998	68	190	100	3	1
003	M	11/12/1999	58	112	74		0
004	F	01/01/1999	101	200	120	5	A
006		06/15/1999	72	102	68	6	1
006	F	07/07/1999	82	148	84	1	0
007	M	.	88	148	102		0
008	F	08/08/1998	210	.	.	7	0

Even though the NODUPRECS option worked as expected in this example, there is a slight problem with this option: That is, it doesn't always work the way you want or expect. An example of this strange behavior is shown next, followed by an explanation and a way to avoid this problem.

Program 5-3 Demonstrating a Problem with the NODUPRECS (NODUP) Option

```
data multiple;
   input Patno $ x y;
datalines;
001 1 2
006 1 2
009 1 2
001 3 4
001 1 2
009 1 2
001 1 2
;
proc sort data=multiple out=single noduprecs;
   by Patno;
run;
```

Here is data set SINGLE:

```
Listing of data set SINGLE

Patno    x    y

 001     1    2
 001     3    4
 001     1    2
 006     1    2
 009     1    2
```

Data set SINGLE still contains duplicate records (shaded in gray). To understand what is happening, look at data set SINGLE before the duplicates are removed: Here it is:

```
Listing of data set SINGLE (without NODUPRECS option)

Obs    Patno    x    y

 1      001     1    2
 2      001     3    4
 3      001     1    2
 4      001     1    2
 5      006     1    2
 6      009     1    2
 7      009     1    2
```

When you sort by Patno, only the last two observations (numbers 3 and 4) are **successive** duplicates, and the NODUPRECS option only removes successive duplicates. (Note: You may want to check the SAS online documentation at support.sas.com or another reference from time to time, since there is a possibility that the behavior of this option may change in future releases.) So, to be sure that you remove all duplicates with the NODUPRECS option, you may want to include several BY variables so that you can feel confident that any duplicate records are successive. You could use _ALL_ for your list of BY variables, guaranteeing that all duplicate observations were successive. However, with a large number of variables and observations, this might require extensive computer resources. Using several BY variables should usually be enough. One way to guarantee that all duplicates are removed is to use PROC SQL with the DISTINCT keyword like this:

Program 5-4 Removing Duplicate Records Using PROC SQL

```
proc sql;
   create table single as
   select distinct *
   from multiple;
quit;
```

Data set SINGLE above, will have all duplicate records removed. Note that this method uses comparable computer resources to using PROC SORT with BY _ALL_.

My thanks to Mike Zdeb, who first brought the peculiar behavior of the NODUPRECS option to my attention.

Detecting Duplicates by Using DATA Step Approaches

Let's explore the ways that will allow you to detect duplicate ID's and duplicate observations in a data set. One very good way to approach this problem is to use the temporary SAS variables FIRST. and LAST. To see how this works, look at Program 5-5, which prints out all observations that have duplicate patient numbers.

Program 5-5 Identifying Duplicate ID's

```
proc sort data=clean.patients out=tmp;
   by Patno;
run;

data dup;
   set tmp;
   by Patno;
   if first.Patno and last.Patno then delete;
run;

title "Listing of duplicates from data set CLEAN.PATIENTS";
proc print data=dup;
   id Patno;
run;
```

It's first necessary to sort the data set by the ID variable. In the above program, the original data set was left intact and a new data set (TMP) was created for the sorted observations. After you have a sorted data set, a short DATA step will remove patients that have a single observation, leaving a data set of duplicates. The key here is the BY statement following the SET statement. When a SET statement is followed by a BY statement, the temporary SAS variables FIRST.*by_variable_name* and LAST.*by _variable_name* are created. In this example, there is only one BY variable (Patno) so the two temporary SAS variables first.Patno and last.Patno are created. If an observation is the first one in a BY group (in this case, the first occurrence of a patient number), first.Patno will be true (equal to one). If an observation is the last one in a BY group, last.Patno will be true. Obviously, if first.Patno and last.Patno are both true, there is only one observation for that patient number.

Therefore, the data set DUP contains only observations where there is more than one observation for each patient number. Output from Program 5-5 is shown next.

```
Listing of duplicates from data set CLEAN.PATIENTS

Patno    gender         Visit    HR    SBP    DBP    Dx    AE

 002       F       11/13/1998    84    120     78    X     0
 002       F       11/13/1998    84    120     78    X     0
 003       X       10/21/1998    68    190    100    3     1
 003       M       11/12/1999    58    112     74          0
 006               06/15/1999    72    102     68    6     1
 006       F       07/07/1999    82    148     84    1     0
```

Next, you want to consider the case where there are two or more observations for each patient, and each observation is supposed to have a different visit date (Visit). The data set PATIENTS2 was created to demonstrate this situation. For this data set, two observations with the same patient ID and visit date would constitute an error. Refer to the Appendix for a listing of the raw data file PATIENTS2.TXT from which the PATIENTS2 data set was created.

If you would like to create the PATIENTS2 data set for test purposes, run the short DATA step shown in Program 5-6.

Program 5-6 Creating the SAS Data Set PATIENTS2 (a Data Set Containing Multiple Visits for Each Patient)

```
data clean.patients2;
   infile "c:\books\clean\patients2.txt" truncover;
   input @1  Patno  $3.
         @4  Visit  mmddyy10.
         @14 HR      3.
         @17 SBP     3.
         @20 DBP     3.;
   format Visit mmddyy10.;
run;
```

A listing of the resulting data set follows.

```
Listing of data set CLEAN.PATIENTS2

Patno         Visit     HR    SBP    DBP

 001     06/12/1998     80    130     80
 001     06/15/1998     78    128     78
 002     01/01/1999     48    102     66
 002     01/10/1999     70    112     82
 002     02/09/1999     74    118     78
 003     10/21/1998     68    120     70
 004     03/12/1998     70    102     66
 004     03/13/1998     70    106     68
 005     04/14/1998     72    118     74
 005     04/14/1998     74    120     80
 006     11/11/1998    100    180    110
 007     09/01/1998     68    138    100
 007     10/01/1998     68    140     98
```

Notice that there are from one to three observations for each patient. Also, notice that patient 005 has two observations with the same Visit date and different data values. To detect this situation, use the variables FIRST. and LAST., except with two BY variables instead of one, as shown in Program 5-7.

Program 5-7 Identifying Patient ID's with Duplicate Visit Dates

```
proc sort data=clean.patients2 out=tmp;
   by Patno Visit;
run;

data dup;
   set tmp;
   by Patno Visit;
   if first.Visit and last.Visit then delete;
run;

title "Listing of Duplicates from Data Set CLEAN.PATIENTS2";
proc print data=dup;
   id Patno;
run;
```

You sort as before, only this time, execute a two-level sort, by Patno and Visit. This places the observations in the data set TMP in patient number and visit date order. The SAS temporary variable first.Visit will be true for the first unique patient number and visit date combination. Only the two identical visit dates for patient 005 will be selected for the data set DUP as demonstrated in the listing shown next.

```
Listing of Duplicates from Data Set CLEAN.PATIENTS2

Patno         Visit    HR    SBP    DBP

 005     04/14/1998    72    118     74
 005     04/14/1998    74    120     80
```

Using PROC FREQ to Detect Duplicate ID's

Another way to find duplicates uses PROC FREQ to count the number of observations for each value of the patient ID variable (Patno). Use the patient ID variable and the OUT= option in the TABLES statement to create a SAS data set that contains the value of Patno and the frequency count (PROC FREQ uses the variable name Count to hold the frequency information). After you have this information, you can use it to select the original duplicate observations from your data set. To demonstrate how this works, Program 5-8 identifies duplicate patient numbers from the PATIENTS data set.

Program 5-8 Using PROC FREQ and an Output Data Set to Identify Duplicate ID's

```
proc freq data=clean.patients noprint;
   tables Patno / out=dup_no(keep=Patno Count
                             where=(Count gt 1));
run;

proc sort data=clean.patients out=tmp;
   by Patno;
run;

proc sort data=dup_no;
   by Patno;
run;

data dup;
   merge tmp dup_no(in=Yes_dup drop=Count);
   by Patno;
   if Yes_dup;
run;

title "Listing of data set dup";
proc print data=dup;
run;
```

PROC FREQ uses the NOPRINT option because you only want the output data set, not the actual PROC FREQ listing. The OUT= option in the TABLES statement creates a SAS data set called DUP_NO, which contains the variables Patno and Count. The WHERE data set option restricts this data set to those observations where Count is greater than one (the duplicates).

Next, sort both the original data set PATIENTS (to the temporary data set TMP) and the DUP_NO data set by Patno. The final DATA step merges the two data sets. The key to the entire program is the IN= option in this MERGE statement. The DUP_NO data set only contains patient numbers where the value of Count is greater than one. The logical variable Yes_dup, created by this IN= data set option, is true whenever the DUP_NO data set is making a contribution to the observation being formed. Thus, only the duplicates will be placed in the DUP data set, as shown in the next listing.

```
Listing of data set dup

Obs    Patno    Gender         Visit     HR    SBP    DBP    Dx    AE

 1      002       F       11/13/1998     84    120     78    X     0
 2      002       F       11/13/1998     84    120     78    X     0
 3      003       X       10/21/1998     68    190    100    3     1
 4      003       M       11/12/1999     58    112     74          0
 5      006               06/15/1999     72    102     68    6     1
 6      006       F       07/07/1999     82    148     84    1     0
```

If all you need to do is identify the patient ID's with more than one observation, you can avoid the MERGE step because the output data set from PROC FREQ contains the variable Patno as well as the frequency (Count). So, the much simpler program is shown in Program 5-9.

Program 5-9 Producing a List of Duplicate Patient Numbers by Using PROC FREQ

```
proc freq data=clean.patients noprint;
   tables Patno / out=dup_no(keep=Patno Count
                             where=(Count gt 1));
run;

title "Patients with duplicate observations";
proc print data=dup_no noobs;
run;
```

This program is considerably more efficient than the program requiring sorts and merging. The output produced by Program 5-9 is shown next.

```
Patients with duplicate observations

Patno    COUNT

 002       2
 003       2
```

Selecting Patients with Duplicate Observations by Using a Macro List and SQL

Another quick, easy, and efficient way to select observations with duplicate ID's is to create a macro variable that contains all the patient ID's in the duplicate data set (DUP_NO). Using a short SQL step, you can create a list of patient numbers, separated by spaces or commas (both will work) and placed in quotes, that can subsequently be used as the argument of an IN operator. Program 5-10 demonstrates this.

Program 5-10 Using PROC SQL to Create a List of Duplicates

```
proc sql noprint;
   select quote(Patno)
      into :Dup_list separated by " "
      from dup_no;
quit;

title "Duplicates selected using SQL and a macro variable";
proc print data=clean.patients;
   where Patno in (&Dup_list);
run;
```

The SELECT clause uses the QUOTE function, which places the patient numbers in quotes. INTO :Dup_list assigns the list of patient numbers to a macro variable (Dup_list) and separates each of the quoted values with a space.

To help understand what this PROC SQL program is doing, the macro variable &Dup_list will contain:

```
"002" "003" "006"
```

Finally, following a PROC PRINT statement, you supply a WHERE statement that selects only those patient numbers that are contained in the list of duplicate patient numbers. Once the macro processor resolves the macro variable &Dup_list, the WHERE statement will read:

```
where Patno in ("002" "003" "006");
```

Notice that this method does not require a sort or a DATA step. Inspection of the output from Program 5-10 shows that the observations are in the original order of the observations in the PATIENTS data set.

```
Duplicates selected using SQL and a macro variable

Obs    Patno    gender         Visit     HR    SBP    DBP    Dx    AE

  2     002       F       11/13/1998    84    120     78    X     0
  3     003       X       10/21/1998    68    190    100    3     1
  6     006               06/15/1999    72    102     68    6     1
 16     002       F       11/13/1998    84    120     78    X     0
 17     003       M       11/12/1999    58    112     74          0
 31     006       F       07/07/1999    82    148     84    1     0
```

To see an SQL solution, please refer to Chapter 8.

Identifying Subjects with "n" Observations Each (DATA Step Approach)

Besides identifying duplicates, you may need to verify that there are "n" observations per subject in a raw data file or in a SAS data set. For example, if each patient in a clinical trial was seen twice, you might want to verify that there are two observations for each patient ID in the file or data set. You can accomplish this task by using a DATA step approach or by using PROC FREQ, the same two methods used earlier to detect duplicates. First, let's look at the DATA step approach.

The key to this approach is the use of the variables FIRST. and LAST., which are created when you use a SET statement followed by a BY statement. To test the programs, let's use the data set PATIENTS2, which was created previously.

Inspection of the PATIENTS2 listing on page 125 shows that patient 002 has three observations, patients 001, 004, 005, and 007 have two observations, and patients 003 and 006 have only one observation.

Program 5-11 lists all the patient ID's which do not have exactly two observations each.

Program 5-11 Using a DATA Step to List All ID's for Patients Who Do Not Have Exactly Two Observations

```
proc sort data=clean.patients2(keep=Patno) out=tmp;
   by Patno;
run;

title "Patient ID's for patients with other than two observations";
data _null_;
   file print;
   set tmp;
   by Patno; ❶
   if first.Patno then n = 0;  ❷
   n + 1;  ❸
   if last.Patno and n ne 2 then put
      "Patient number " Patno "has " n "observation(s).";  ❹
run;
```

The first step is to sort the data set by patient ID (Patno), because you will use a BY Patno statement in the DATA_NULL_ step ❶. In this example, you create a new data set to hold the sorted observations (since it only contains the one variable Patno). Notice that you only need to keep the patient ID variable in the temporary data set (TMP) because that provides sufficient information to count the number of observations per patient. If each observation contains a large number of variables, this will save processing time.

Use a DATA _NULL_ step to do the counting and output the invalid patient ID's. When you are processing the first observation for any patient, the temporary variable first.Patno will be true and n will be set to 0 ❷. The counter (n) is incremented by 1 ❸. Statement ❸ is a SUM statement (notice there is no equal sign) which causes n to be automatically retained. Finally, when you reach the last observation for any patient ID ❹, output an error statement if your counter is not equal to 2. The next listing shows that this program works properly.

```
Patient ID's for patients with other than two observations
Patient number 002 has 3 observation(s).
Patient number 003 has 1 observation(s).
Patient number 006 has 1 observation(s).
```

What if a patient with two observations really only had one visit, but the single observation was duplicated by mistake? You should probably run one of the programs to detect duplicates before running this program.

Identifying Subjects with "n" Observations Each (Using PROC FREQ)

You can use PROC FREQ to count the number of observations per subject, just as you did to detect duplicates. Use the variable Count to determine the number of observations for each value of Patno, as shown in the next program.

Program 5-12 Using PROC FREQ to List All ID's for Patients Who Do Not Have Exactly Two Observations

```
proc freq data=clean.patients2 noprint;
   tables Patno / out=dup_no(keep=Patno Count
                             where=(Count ne 2));
run;

title "Patient ID's for patients with other than two observations";
proc print data=dup_no noobs;
run;
```

The output data set from PROC FREQ (DUP_NO) contains the variables Patno and the frequency (Count), and there is one observation for each patient who did not have exactly two visits. All that is left to do is to print out the observations in data set DUP_NO. Output from Program 5-12 is shown next.

```
Patient ID's for patients with other than two observations

Patno    COUNT

 002       3
 003       1
 006       1
```

It is usually easier to let a PROC do the work, as in this example, rather than doing all the work yourself with a DATA step. But, then again, some of us really enjoy the DATA step.

6 Working with Multiple Files

Introduction 135
Checking for an ID in Each of Two Files 135
Checking for an ID in Each of "n" Files 138
A Macro for ID Checking 140
More Complicated Multi-File Rules 143
Checking That the Dates Are in the Proper Order 147

Introduction

This chapter addresses data validation techniques where multiple files or data sets are involved. You will see how to verify that a given ID exists in multiple files and how to verify some more complicated multi-file rules. Because it is impossible to anticipate all the possible multi-file rules you may need, much of this chapter is devoted to programming tips and techniques rather than actual examples that you can use directly.

Checking for an ID in Each of Two Files

One requirement of a large project may be that a particular ID value exists in each of several SAS data sets. Let's start out by demonstrating how you can easily check that an ID is in each of two files. This will be generalized later to include an arbitrary number of files.

The technique demonstrated in this section is to merge the two files in question, using the ID variable as a BY variable. The key to the program is the IN= data set option that sets a logical variable to true or false, depending on whether or not the data set provides values to the current observation being created. An example will make this clearer. In Program 6-1 are the SAS statements to create two SAS data sets for testing purposes.

Program 6-1 Creating Two Test Data Sets for Chapter 6 Examples

```
data one;
   input Patno x y;
datalines;
1 69 79
2 56 .
3 66 99
5 98 87
12 13 14
;
data two;
   input Patno z;
datalines;
1 56
3 67
4 88
5 98
13 99
;
```

Notice that ID's 2 and 12 are in data set ONE but not in data set TWO; ID's 4 and 13 are in data set TWO but not in data set ONE. Program 6-2 gives detailed information on the unmatched ID's.

Program 6-2 Identifying ID's Not in Each of Two Data Sets

```
proc sort data=one;
   by Patno;
run;

proc sort data=two;
   by Patno;
run;

title "Listing of missing ID's";
data _null_;
   file print;
   merge one(in=Inone)
         two(in=Intwo)  end=Last;  ❶
   by Patno;  ❷

   if not Inone then do;  ❸
```

```
      put "ID " Patno "is not in data set one";
      n + 1;
   end;

   else if not Intwo then do;  ❹
      put "ID " Patno "is not in data set two";
      n + 1;
   end;

   if Last and n eq 0 then
      put "All ID's match in both files";  ❺
run;
```

Before you can merge the two data sets, they must first be sorted by the BY variable (unless the two data sets are indexed). The MERGE statement ❶ is the key to this program. Each of the data set names is followed by the data set option IN=*logical_variable*. In addition, the END= *variable_name* MERGE option creates a logical variable that is set to 1 (true) when the last observation from all data sets has been processed. Using a MERGE statement would be useless in this application without the BY statement ❷.

Since it is almost always an error to leave out a BY statement following a MERGE, you should consider using the SAS system option MERGENOBY= to check for this possibility. Valid values for this option are NOWARN (the default), WARN, and ERROR. If you use WARN, SAS will perform the merge but will add a warning message in the SAS Log. If you use ERROR, the DATA step will stop and an error message will be printed to the SAS Log. We strongly recommend that you set this option to ERROR, like this:

```
options mergenoby = error;
```

If, for some reason, you actually want to perform a MERGE without a BY statement, you can set this option to NOWARN before the DATA step and back to ERROR following the DATA step.

One final note: Although this program runs correctly even if there are multiple observations with the same BY variable in both data sets, you would be wise to check for unexpected duplicates as described in the previous chapter or to use the NODUPKEY option of PROC SORT on one of the data sets.

Let's "play computer" to see how this program works. Both data sets contain an observation for Patno = 1. Therefore, the two logical variables Inone and Intwo are both true, neither of the IF statements ❸ or ❹ is true, and a message is not printed to the output file. The next value of Patno is 2, from data set ONE. Because this value is not in data set TWO, the values of Inone and Intwo

are true and false, respectively. Therefore, statement ❹ is true, and the appropriate message is printed to the output file. When you reach Patno = 4, which exists in data set TWO but not in data set ONE, statement ❸ is true, and its associated message is printed out. Note that any time a value of Patno is missing from one of the files, the variable n is incremented. If you reach the last observation in the files being merged, the logical variable Last is true. If there are no ID errors, n will still be 0 and statement ❺ will be true. When you run Program 6-2, the following output is obtained.

```
Listing of missing ID's
ID 2 is not in data set two
ID 4 is not in data set one
ID 12 is not in data set two
ID 13 is not in data set one
```

All the ID errors are correctly displayed. If you want to extend this program to more than two data sets, the program could become long and tedious. A macro approach can be used to accomplish the ID checking task with an arbitrary number of data sets. The next section demonstrates such a program.

Checking for an ID in Each of "n" Files

Data set THREE is added to the mix to demonstrate how to approach this problem when there are more than two data sets. First, run Program 6-3 to create the new data set THREE.

Program 6-3 Creating a Third Data Set for Testing Purposes

```
data three;
   input Patno Gender $;
datalines;
1 M
2 F
3 M
5 F
6 M
12 M
13 M
;
```

Before developing a macro, let's look at Program 6-4, which is a rather simple but slightly tedious program to accomplish the ID checks.

Program 6-4 Checking for an ID in Each of Three Data Sets (Long Way)

```
proc sort data=one(keep=Patno) out=tmp1;
   by Patno;
run;

proc sort data=two(keep=Patno) out=tmp2;
   by Patno;
run;

proc sort data=three(keep=Patno) out=tmp3;
   by Patno;
run;

title "Listing of missing ID's and data set names";
data _null_;
   file print;
   merge tmp1(in=In1)
         tmp2(in=In2)
         tmp3(in=In3)  end=Last;
   by Patno;

   if not In1 then do;
      put "ID " Patno "missing from data set one";
      n + 1;
   end;

   if not In2 then do;
      put "ID " Patno "missing from data set two";
      n + 1;
   end;
```

```
   if not In3 then do;
      put "ID " Patno "missing from data set three";
      n + 1;
   end;

   if Last and n eq 0 then
      put "All id's match in all files";

run;
```

Program 6-4 can be extended to accommodate any number of data sets. The output is shown next.

```
Listing of missing ID's and data set names
ID 2 missing from data set two
ID 4 missing from data set one
ID 4 missing from data set three
ID 6 missing from data set one
ID 6 missing from data set two
ID 12 missing from data set two
ID 13 missing from data set one
```

A pattern begins to emerge. Notice the sorts and IF statements follow a pattern that can be automated by writing a macro. The ID checking macro is developed in the next section.

A Macro for ID Checking

If you need to check that a given ID is present in each of "n" files and "n" is large, the DATA step approach above becomes too tedious. Using the same logic as Program 6-4, you can create a macro that will check for ID's across as many files as you wish.

Program 6-5 Presenting a Macro to Check for ID's Across Multiple Data Sets

```
*-------------------------------------------------------------------*
| Program Name: check_id.sas  in c:\books\clean                     |
| Purpose: Macro which checks if an ID exists in each of n files    |
| Arguments: The name of the ID variable, followed by as many       |
|            data sets names as desired, separated by BLANKS        |
| Example: %check_id(ID = Patno,                                    |
|                    Dsn_list=one two three)                        |
| Date: Sept 17, 2007                                               |
*-------------------------------------------------------------------*;
%macro check_id(ID=,       /* ID variable              */
                Dsn_list=  /* List of data set names,  */
                           /* separated by spaces      */);
   %do i = 1 %to 99;
     /* break up list into data set names */
      %let Dsn = %scan(&Dsn_list,&i);
      %if &Dsn ne %then %do; /* If non null data set name       */
         %let n = &i;        /* When you leave the loop, n will */
                             /* be the number of data sets      */
         proc sort data=&Dsn(keep=&ID) out=tmp&i;
            by &ID;
         run;
      %end;
   %end;

   title  "Report of data sets with missing ID's";
   title2 "-----------------------------------";
   data _null_;
      file print;
      merge

      %do i = 1 %to &n;
         tmp&i(in=In&i)
      %end;
```

```
      end=Last;
      by &ID;

      if Last and nn eq 0 then do;
         put "All ID's Match in All Files";
         stop;
      end;

      %do i = 1 %to &n;
         %let Dsn = %scan(&Dsn_list,&i);
         if not In&i then do;
            put "ID " &ID "missing from data set &dsn";
            nn + 1;
         end;
      %end;

      run;

%mend check_id;
```

The two arguments used in this macro are the name of the ID variable and a list of data set names (separated by blanks). The first step is to break up the list of data set names into individual names. This is accomplished with the %SCAN function, which is similar to the non-macro SCAN function. It breaks strings into "words" where words are any string of characters separated by one of the default delimiters. Since a blank is one of these delimiters (and all of the other default delimiters are not valid as part of a data set name), this function will extract the individual data set names and stop when it runs out of data set names. For each of the data sets you obtain with the %SCAN function, you use PROC SORT to sort the data set by the ID variable and create a set of temporary data sets (tmp1, tmp2, etc.). When you run out of data set names, the %SCAN function returns a missing value and the program continues to the DATA _NULL_ step. The remainder of the program is similar to the previous program where a MERGE statement is constructed, using the multiple data set names followed by the IN= data set option.

To invoke this macro to check the three data sets (ONE, TWO, and THREE) with the ID variable Patno, you would write:

```
%check_id(Patno,one two three)
```

The resulting output is shown below:

```
Report of data sets with missing ID's
-------------------------------------
ID 2 missing from data set two
ID 4 missing from data set one
ID 4 missing from data set three
ID 6 missing from data set one
ID 6 missing from data set two
ID 12 missing from data set two
ID 13 missing from data set one
```

More Complicated Multi-File Rules

Every data collection project will have its own set of rules. The programs in this section are intended to demonstrate techniques rather than be exact models of data validation rules for multiple files.

In this first example, you want to be sure an observation has been added to the laboratory test data set (LAB_TEST) if there was an adverse event of 'X' entered for any patient ID in the adverse event data set (AE). The AE and the LAB_TEST data sets can be created by running the two programs below:

Program 6-6 Creating Data Set AE (Adverse Events)

```
libname clean "c:\books\clean";

data clean.ae;
   input @1  Patno   $3.
         @4  Date_ae mmddyy10.
         @14 A_event $1.;
   label Patno   = 'Patient ID'
         Date_ae = 'Date of AE'
         A_event = 'Adverse event';
   format Date_ae mmddyy10.;
datalines;
00111/21/1998W
00112/13/1998Y
00311/18/1998X
00409/18/1998O
00409/19/1998P
```

```
01110/10/1998X
01309/25/1998W
00912/25/1998X
02210/01/1998W
02502/09/1999X
;
```

Program 6-7 Creating Data Set LAB_TEST

```
libname clean "c:\books\clean";

data clean.lab_test;
   input @1  Patno    $3.
         @4  Lab_date date9.
         @13 WBC      5.
         @18 RBC      4.;
   label  Patno    = 'Patient ID'
          Lab_date = 'Date of lab test'
          WBC      = 'White blood cell count'
          RBC      = 'Red blood cell count';
   format Lab_date mmddyy10.;
datalines;
00115NOV1998 90005.45
00319NOV1998 95005.44
00721OCT1998 82005.23
00422DEC1998110005.55
02501JAN1999 82345.02
02210OCT1998 80005.00
;
```

Here are the listings of the AE and LAB_TEST data sets.

```
Listing of Data Set AE

Patno       Date_ae    A_event

 001     11/21/1998       W
 001     12/13/1998       Y
 003     11/18/1998       X
 004     09/18/1998       O
 004     09/19/1998       P
 011     10/10/1998       X
 013     09/25/1998       W
 009     12/25/1998       X
 022     10/01/1998       W
 025     02/09/1999       X
```

Note that each patient ID (Patno) may have more than one adverse event.

```
Listing of Data Set LAB_TEST

Patno      Lab_date     WBC      RBC

 001     11/15/1998     9000    5.45
 003     11/19/1998     9500    5.44
 007     10/21/1998     8200    5.23
 004     12/22/1998    11000    5.55
 025     01/01/1999     8234    5.02
 022     10/10/1998     8000    5.00
```

Note that each patient has only one observation in this data set.

According to our rule, any patient with an adverse event of 'X' should have one observation in the LAB_TEST data set. Patients 003, 011, 009, and 025 all had an adverse event of 'X'; however, only patients 003 and 025 had an observation in the LAB_TEST data set (although the date of the lab test for patient 025 is earlier than the date of the AE, let's ignore this for now). One approach to locating these two omissions is to merge the AE and LAB_TEST data sets by patient ID, selecting only those patients with an AE of 'X'. Then, using the IN= data set option on the merge, you can locate any missing observations. This is shown in Program 6-8. An explanation follows.

Program 6-8 Verifying That Patients with an Adverse Event of "X" in Data Set AE Have an Entry in Data Set LAB_TEST

```
libname clean "c:\books\clean";

proc sort data=clean.ae(where=(A_event = 'X')) out=ae_x;
   by Patno;
run;

proc sort data=clean.lab_test(keep=Patno Lab_date) out=lab;
   by Patno;
run;

data missing;
   merge ae_x
         lab(in=In_lab);
   by Patno;
   if not In_lab;
run;

title "Patients with AE of X who are missing a lab test entry";
proc print data=missing label;
   id Patno;
   var Date_ae A_event;
run;
```

Each of the two data sets is first sorted by patient ID (Patno). In addition, by using a WHERE= data set option, only those observations in the adverse events data set with event 'X' are selected. The key to the program is the IN= data set option to create the temporary logical variable In_lab. Because there should be an observation in LAB_TEST for every patient with an adverse event of 'X', anytime the logical variable In_lab is false, you have located a patient with a missing laboratory test. The output from Program 6-8 is shown next.

```
Patients with AE of X who are missing a lab test entry

Patient                  Adverse
  ID       Date of AE     event

  009      12/25/1998       X
  011      10/10/1998       X
```

Checking That the Dates Are in the Proper Order

It was mentioned earlier that the date of the laboratory test for patient 025 was prior to the date of the adverse event. You should modify your program to detect this. Remembering that dates are just numbers (the number of days from January 1, 1960), you can easily compare the two dates in the last DATA step, as shown in Program 6-8.

Program 6-9 Adding the Condition That the Lab Test Must Follow the Adverse Event

```
title  "Patients with AE of X Who Are Missing Lab Test Entry";
title2 "or the Date of the Lab Test Is Earlier Than the AE Date";
title3 "-----------------------------------------------------";

data _null_;
   file print;
   merge ae_x(in=In_ae)
         lab(in=In_lab);

   by Patno;
   if not In_lab then put
      "No lab test for patient " Patno "with adverse event X";
   else if In_ae and missing(Lab_date) then put
      "Date of lab test is missing for patient "
      Patno /
      "Date of AE is " Date_ae /;
   else if In_ae and Lab_date lt Date_ae then put
      "Date of lab test is earlier than date of AE for patient "
      Patno /
      "  date of AE is " Date_ae " date of lab test is " Lab_date /;
run;
```

One IF statement checks if the laboratory date is missing, the other IF statement tests if the laboratory date is prior to (less than) the date of the adverse event, and an appropriate message is printed. Output from this program is shown next.

```
Patients with AE of X Who Are Missing Lab Test Entry
or the Date of the Lab Test Is Earlier Than the AE Date
-------------------------------------------------------
No lab test for patient 009 with adverse event X
No lab test for patient 011 with adverse event X
Date of lab test is earlier than date of AE for patient 025
  date of AE is 02/09/1999  date of lab test is 01/01/1999
```

Programs to verify multi-file rules can become very complicated. However, many of the techniques discussed in this chapter should prove useful.

7 Double Entry and Verification (PROC COMPARE)

Introduction 149
Conducting a Simple Comparison of Two Data Sets 150
Using PROC COMPARE with Two Data Sets That Have an Unequal Number of Observations 159
Comparing Two Data Sets When Some Variables Are Not in Both Data Sets 161

Introduction

Many critical data applications require that you have the data entered twice and then compare the resulting files for discrepancies. This is usually referred to as double entry and verification. In the "old days," when I was first learning to use computers, most data entry was done using a keypunch (although my boys will tell you that, in my day, it was done with a hammer and chisel on stone tablets). The most common method of double entry and verification was done on a special machine called a verifier. The original cards were placed in the input hopper and a keypunch operator (preferably not the one who entered the data originally) re-keyed the information from the data entry form. If the information being typed matched the information already punched on the card, the card was accepted and a punch was placed, usually in column 81 of the card. If the information did not match, a check could be made to see whether the error was on the original card or in the re-keying of the information.

Today, there are several programs that accomplish the same goal by having all the data entered twice and then comparing the resulting data files. Some of these programs are quite sophisticated and also quite expensive. SAS software has a very flexible procedure called PROC COMPARE, which can be used to compare the contents of two SAS data sets. You can refer to the *Base SAS Procedures Guide*, published by SAS Institute Inc., Cary, NC, in hard-copy format or to the SAS online documentation at support.sas.com for more information. This chapter presents some simple examples using PROC COMPARE.

Conducting a Simple Comparison of Two Data Sets

The simplest application of PROC COMPARE is presented first, determining if the contents of two SAS data sets are identical. Suppose you have two people enter data from some coding forms and the two data sets are called FILE_1.TXT and FILE_2.TXT. A listing of the two files is shown next.

```
FILE_1.TXT

001M10211946130 80
002F12201950110 70
003M09141956140 90
004F10101960180100
007m10321940184110

FILE_2.TXT

001M1021194613080
002F12201950110 70
003M09141956144 90
004F10101960180100
007M10231940184110
```

Here is the file format.

Variable	Description	Starting Column	Length	Type
PATNO	Patient Number	1	3	Numeric
GENDER	Gender	4	1	Character
DOB	Date of Birth	5	8	mmddyyyy
SBP	Systolic Blood Pressure	13	3	Numeric
DBP	Diastolic Blood Pressure	16	3	Numeric

The data, without mistakes, should have been:

```
Correct Data Representation

001M10211946130 80
002F12201950110 70
003M09141956140 90
004F10101960180100
007M10231940184110
```

A visual inspection of the two original files shows the following discrepancies.

- For patient 001, there is a space missing before the 80 at the end of the line in FILE_2.TXT.
- For patient 003, SBP is 144 instead of 140 in FILE_2.TXT.
- For patient 007, the gender is entered in lowercase and the digits are interchanged in the day field of the date in FILE_1.TXT.

Let's see how to use PROC COMPARE to detect these differences. You have some choices here. One way to proceed is to create two SAS data sets, as shown in Program 7-1.

Program 7-1 Creating Data Sets ONE and TWO from Two Raw Data Files

```
data one;
   infile "c:\books\clean\file_1.txt" truncover;
   input @1  Patno  3.
         @4  Gender $1.
         @5  DOB    mmddyy8.
         @13 SBP    3.
         @16 DBP    3.;
   format DOB mmddyy10.;
run;
```

```
data two;
   infile "c:\books\clean\file_2.txt" truncover;
   input @1  Patno  3.
         @4  Gender $1.
         @5  DOB    mmddyy8.
         @13 SBP    3.
         @16 DBP    3.;
   format dob mmddyy10.;
run;
```

Then run PROC COMPARE as shown in Program 7-2.

Program 7-2 Running PROC COMPARE

```
title "Using PROC COMPARE to compare two data sets";
proc compare base=one compare=two;
   id Patno;
run;
```

The procedure options BASE= and COMPARE= identify the two data sets to be compared. In this example, data set ONE was arbitrarily chosen as the base data set. (The option DATA= may be used in place of BASE= because they are equivalent.)

The variable listed on the ID statement must be a unique identifier or a combination of variables that create a unique identifier.

Here is the output from PROC COMPARE.

```
Using PROC COMPARE to compare two data sets

The COMPARE Procedure
Comparison of WORK.ONE with WORK.TWO
(Method=EXACT)

Data Set Summary

Dataset            Created          Modified  NVar    NObs

WORK.ONE  21SEP07:14:56:03  21SEP07:14:56:03     5       5
WORK.TWO  21SEP07:14:56:03  21SEP07:14:56:03     5       5


Variables Summary

Number of Variables in Common: 5.
Number of ID Variables: 1.


Observation Summary

Observation      Base  Compare  ID

First Obs           1        1  Patno=1
First Unequal       3        3  Patno=3
Last  Unequal       5        5  Patno=7
Last  Obs           5        5  Patno=7

Number of Observations in Common: 5.
Total Number of Observations Read from WORK.ONE: 5.
Total Number of Observations Read from WORK.TWO: 5.

Number of Observations with Some Compared Variables Unequal: 2.
Number of Observations with All Compared Variables Equal: 3.

Values Comparison Summary

Number of Variables Compared with All Observations Equal: 1.
Number of Variables Compared with Some Observations Unequal: 3.
Number of Variables with Missing Value Differences: 1.
Total Number of Values which Compare Unequal: 3.
Maximum Difference: 4.
```

(*continued*)

```
Using PROC COMPARE to compare two data sets

The COMPARE Procedure
Comparison of WORK.ONE with WORK.TWO
(Method=EXACT)

Variables with Unequal Values

Variable  Type  Len  Ndif   MaxDif  MissDif

Gender    CHAR    1     1                0
DOB       NUM     8     1        0       1
SBP       NUM     8     1    4.000       0


Value Comparison Results for Variables

_____________________________________________________
          ||  Base Value           Compare Value
   Patno  ||  Gender                Gender
 _______  ||  _                     _
          ||
       7  ||  m                     M
_____________________________________________________


_____________________________________________________
          ||       Base    Compare
   Patno  ||        DOB         DOB      Diff.     % Diff
 _______  ||  _________  _________  _________  _________
          ||
       7  ||          .   10/23/40          .          .
_____________________________________________________


_____________________________________________________
          ||       Base    Compare
   Patno  ||        SBP         SBP      Diff.     % Diff
 _______  ||  _________  _________  _________  _________
          ||
       3  ||   140.0000   144.0000     4.0000     2.8571
_____________________________________________________
```

Notice that the left-adjusted value of 80 for patient 001 in FILE_1.TXT was not flagged as an error. Why? Because SAS correctly reads left-adjusted numeric values and the comparison is between the two SAS data sets, not the raw files themselves. Also, the incorrect date of 10/32/1940 (patient number 005 in FILE_1) was shown as a missing value in the output. If you inspect the SAS Log, you will see the incorrect date was flagged as an error. When invalid dates are encountered, SAS substitutes a missing value for the date. If you do not want this to happen, you can use a character informat instead of a date informat for data checking purposes.

You may find that the output from PROC COMPARE contains more information than you need. The procedure option BRIEF reduces the output considerably by producing only a short comparison summary and suppressing the lengthy output shown after Program 7-2. Here is the output from PROC COMPARE using the BRIEF option.

```
Using PROC COMPARE to compare two data sets

The COMPARE Procedure
Comparison of WORK.ONE with WORK.TWO
(Method=EXACT)

NOTE: Values of the following 3 variables compare unequal: Gender DOB
      SBP

Value Comparison Results for Variables


_____________________________________________________
          ||  Base Value           Compare Value
   Patno  ||  Gender                Gender
 _______  ||  _                     _
          ||
       7  ||  m                     M
_____________________________________________________


_____________________________________________________
          ||       Base    Compare
   Patno  ||        DOB         DOB      Diff.     % Diff
 _______  ||  _________  _________  _________  _________
          ||
       7  ||          .   10/23/40          .          .
_____________________________________________________
```

(*continued*)

Patno	Base SBP	Compare SBP	Diff.	% Diff
3	140.0000	144.0000	4.0000	2.8571

If you would rather have the differences in the two data sets listed by patient number, you can use the TRANSPOSE option (as shown below):

Program 7-3 Demonstrating the TRANSPOSE Option of PROC COMPARE

```
title "Demonstrating the TRANSPOSE Option";
proc compare base=one compare=two brief transpose;
   id Patno;
run;
```

Here is the output from Program 7-3:

```
Demonstrating the TRANSPOSE Option

The COMPARE Procedure
Comparison of WORK.ONE with WORK.TWO
(Method=EXACT)

Comparison Results for Observations

Patno=3:
Variable    Base Value       Compare         Diff.        % Diff
     SBP    140.000000    144.000000      4.000000      2.857143

Patno=7:
Variable    Base Value       Compare         Diff.        % Diff
     DOB             .    10/23/1940             .             .
  Gender             m             M

NOTE: Values of the following 3 variables compare unequal: Gender DOB
      SBP
```

If you want to simply simulate a data-entry, verify process, you can proceed in another way. You can read every variable using a character informat, as shown in Program 7-4. By doing this, you can compare any differences in the two raw data files.

Program 7-4 Using PROC COMPARE to Compare Two Data Records

```
data one;
   infile "c:\books\clean\file_1.txt"
      pad;
   input @1  Patno  $char3.
         @4  Gender $char1.
         @5  DOB    $char8.
         @13 SBP    $char3.
         @16 DBP    $char3.;
run;
data two;
   infile "c:\books\clean\file_2.txt"
      pad;
   input @1  Patno  $char3.
         @4  Gender $char1.
         @5  DOB    $char8.
         @13 SBP    $char3.
         @16 DBP    $char3.;
run;

title "Using PROC COMPARE to compare two data sets";
proc compare base=one compare=two brief;
   id Patno;
run;
```

Using the BRIEF option, you obtain the following output:

```
Using PROC COMPARE to compare two data sets

The COMPARE Procedure
Comparison of WORK.ONE with WORK.TWO
(Method=EXACT)

NOTE: Values of the following 4 variables compare unequal: gender dob
      sbp dbp

Value Comparison Results for Variables


_____________________________________________________________
        ||  Base Value           Compare Value
 patno  ||  gender                gender
 _____  ||  _                     _
        ||
 007    ||  m                     M
_____________________________________________________________


_____________________________________________________________
        ||  Base Value           Compare Value
 patno  ||  dob                   dob
 _____  ||  ________              ________
        ||
 007    ||  10321940              10231940
_____________________________________________________________


_____________________________________________________________
        ||  Base Value           Compare Value
 patno  ||  sbp                   sbp
 _____  ||  ___                   ___
        ||
 003    ||  140                   144
_____________________________________________________________


_____________________________________________________________
        ||  Base Value           Compare Value
 patno  ||  dbp                   dbp
 _____  ||  ___                   ___
        ||
 001    ||   80                   80
_____________________________________________________________
```

If you examine the output, you will notice the errors in Gender and SBP that you saw earlier. In addition, you can see the differences in the two dates (since it was read as character) and the DBP values. (Note: The differences show up only because the $CHAR informat was used. This informat maintains leading blanks while the $ informat will left-adjust character fields.)

Using PROC COMPARE with Two Data Sets That Have an Unequal Number of Observations

You can compare two data sets with unequal numbers of observations, providing you include an ID statement. To illustrate this, two new files, (FILE_1B.TXT and FILE_2B.TXT) were created. A new patient number (005) has been added to FILE_1.TXT to make FILE_1B.TXT, and patient number 004 has been omitted from FILE_2.TXT to make FILE_2B.TXT. Here are the listings of these two files.

```
FILE_1B.TXT

001M10211946130 80
002F12201950110 70
003M09141956140 90
004F10101960180100
005M01041930166 88
007m10321940184110

FILE_2B.TXT

001M1021194613080
002F12201950110 70
003M09141956144 90
007M10231940184110
```

The two SAS data sets (ONE_B and TWO_B) are created by running Program 7-1 again with the two new data files, and then running PROC COMPARE with the two options LISTBASE and LISTCOMP (Program 7-5). These two options tell PROC COMPARE to print information on the ID values that are not in both files, as seen below:

Program 7-5 Running PROC COMPARE on Two Data Sets of Different Length

```
title "Comparing Two Data Sets with Different ID Values";
proc compare base=one_b compare=two_b listbase listcompare;
   id Patno;
run;
```

Here is the output from Program 7-5 (partial listing).

```
Comparing Two Data Sets with Different ID Values (partial listing)

The COMPARE Procedure
Comparison of WORK.ONE_B with WORK.TWO_B

Variables Summary

Number of Variables in Common: 5.
Number of ID Variables: 1.

Comparison Results for Observations

Observation 4 in WORK.ONE_B not found in WORK.TWO_B: Patno=4.

Observation 5 in WORK.ONE_B not found in WORK.TWO_B: Patno=5.


Observation Summary

Observation      Base  Compare  ID

First Obs           1        1  Patno=1
First Unequal       3        3  Patno=3
Last  Unequal       6        4  Patno=7
Last  Obs           6        4  Patno=7
```

Comparing Two Data Sets When Some Variables Are Not in Both Data Sets

PROC COMPARE can also be used to compare selected variables between two data sets. Suppose you have one data set that contains demographic information on each patient in a clinical trial (DEMOG). In addition, you have another file from a previous study that contains some of the same patients and some of the same demographic information (OLDDEMOG). Program 7-6 creates these sample data sets.

Program 7-6 Creating Two Test Data Sets, DEMOG and OLDDEMOG

```
***Program to create data sets DEMOG and OLDDEMOG;
data demog;
   input  @1  Patno  3.
          @4  Gender $1.
          @5  DOB    mmddyy10.
          @15 Height 2.;
   format DOB mmddyy10.;
datalines;
001M10/21/194668
003F11/11/105062
004M04/05/193072
006F05/13/196863
;

data olddemog;
   input @1  Patno  3.
         @4  DOB    mmddyy8.
         @12 Gender $1.
         @13 Weight 3.;
   format DOB mmddyy10.;
datalines;
00110211946M155
00201011950F102
00404051930F101
00511111945M200
00605131966F133
;
```

You want to compare the date of birth (DOB) and gender (Gender) for each patient. PROC COMPARE will automatically compare all variables that are in common to both data sets, so you can proceed as shown in Program 7-7.

Program 7-7 Comparing Two Data Sets That Contain Different Variables

```
title "Comparing demographic information between two data sets";
proc compare base=olddemog compare=demog brief;
   id Patno;
run;
```

Here is the output from this procedure.

```
Comparing demographic information between two data sets

The COMPARE Procedure
Comparison of WORK.OLDDEMOG with WORK.DEMOG
(Method=EXACT)

NOTE: Data set WORK.OLDDEMOG contains 2 observations not in WORK.DEMOG.
NOTE: Data set WORK.DEMOG contains 1 observations not in WORK.OLDDEMOG.
NOTE: Values of the following 2 variables compare unequal: DOB Gender


Value Comparison Results for Variables

_____________________________________________________
          ||       Base    Compare
   Patno  ||        DOB        DOB      Diff.     % Diff
 _______  ||  _________  _________  _________  _________
          ||
       6  ||   05/13/66   05/13/68   731.0000    31.4544
_____________________________________________________


_____________________________________________________
          ||  Base Value           Compare Value
   Patno  ||  Gender                Gender
 _______  ||  _                     _
          ||
       4  ||  F                     M
_____________________________________________________
```

Suppose you only want to verify that the genders are correct between the two files. You can add a VAR statement to PROC COMPARE, which restricts the comparison to the variables listed in this statement. Program 7-8 demonstrates this.

Program 7-8 Adding a VAR Statement to PROC COMPARE

```
title "Comparing demographic information between two data sets";
proc compare base=olddemog compare=demog brief;
   id Patno;
   var Gender;
run;
```

Here is the output, which now only shows a gender comparison between the two files.

```
Comparing demographic information between two data sets

The COMPARE Procedure
Comparison of WORK.OLDDEMOG with WORK.DEMOG
(Method=EXACT)

NOTE: Data set WORK.OLDDEMOG contains 2 observations not in WORK.DEMOG.
NOTE: Data set WORK.DEMOG contains 1 observations not in WORK.OLDDEMOG.
NOTE: Values of the following 1 variables compare unequal: Gender


Value Comparison Results for Variables


_____________________________________________________
          ||  Base Value           Compare Value
   Patno  ||  Gender                Gender
 _______  ||  _                     _
          ||
       4  ||  F                     M
_____________________________________________________
```

You have seen a few examples of how PROC COMPARE can be used to perform a comparison between two raw data files or two SAS data sets. Before going out and spending a lot of money on custom-designed software, you may want to take a look at PROC COMPARE.

8 Some PROC SQL Solutions to Data Cleaning

Introduction 165
A Quick Review of PROC SQL 166
Checking for Invalid Character Values 166
Checking for Outliers 168
Checking a Range Using an Algorithm Based on the Standard Deviation 169
Checking for Missing Values 170
Range Checking for Dates 172
Checking for Duplicates 173
Identifying Subjects with "n" Observations Each 174
Checking for an ID in Each of Two Files 174
More Complicated Multi-File Rules 176

Introduction

It was a hard decision whether to group all the PROC SQL approaches together in one chapter or to include an SQL solution in each of the other chapters. I opted for the former. PROC SQL (Structured Query Language) is an alternative to the traditional DATA step and PROC approaches used in this book up to this point. Although PROC SQL is sometimes easier to program and more efficient, sometimes it is not. In this chapter, many of the data cleaning operations you performed earlier with DATA step and PROC solutions will be revisited. Be careful; some SAS programmers get carried away with PROC SQL and try to solve every problem with it. It is extremely powerful and useful, but it is not always the best solution to every problem. For a more complete discussion of PROC SQL, I recommend *PROC SQL: Beyond the Basics Using SAS*, by Kirk Lafler, *SAS SQL Procedure User's Guide, Version 8*, and *The Essential PROC SQL Handbook for SAS Users*, by Katherine Prairie, all published by SAS Institute Inc., Cary, NC.

A Quick Review of PROC SQL

PROC SQL can be used to list data to the output device (Output window), to create SAS data sets (also called tables in SQL terminology), to create SAS views, or to create macro variables. For many of your data cleaning operations, you will not be creating SAS data sets or views. By omitting the CREATE clause, the results of an SQL query will be sent to the Output window (unless the NOPRINT option is included). For example, if you have a data set called ONE and you want to list all observations where X is greater than 100, you would write a program like the one in Program 8-1.

Program 8-1 Demonstrating a Simple SQL Query

```
proc sql;
   select X
   from one
   where X gt 100;
quit;
```

The SELECT clause identifies which variables you want to select. This can be a single variable (as in this example), a list of variables (separated by commas, not spaces), or an asterisk (*), which means all the variables in the data set. The FROM clause identifies what data set to read. Finally, the WHERE clause selects only those observations for which the WHERE condition is true. This is a good time to mention that the SQL clauses have to be in a certain order: SELECT, FROM, WHERE, GROUP BY (having), ORDER BY. Thanks to Cynthia Zender (one of my reviewers), you can remember this order by the saying: ***S***an ***F***rancisco ***W***here the ***G***rateful Dead ***O***riginate.

One advantage of PROC SQL is that you can combine summary data with detail data in one step. Another is its ability to create a Cartesian product—a combining of every observation in one data set with every observation in another, a very difficult task using a DATA step. Many of these operations are demonstrated in the sections that follow.

Checking for Invalid Character Values

Let's start with checking for invalid character values. For these examples, let's use the SAS data set PATIENTS (see the Appendix for the program and data file), and look for invalid values for Gender, DX, and AE (ignoring missing values).

Program 8-2 Using PROC SQL to Look for Invalid Character Values

```
libname clean "c:\books\clean";
***Checking for invalid character data;
title "Checking for Invalid Character Data";
proc sql;
   select Patno,
          Gender,
          DX,
          AE
   from clean.patients
   where Gender not in ('M','F',' ')           or
         notdigit(trim(DX))and not missing(DX) or
         AE not in ('0','1',' ');
quit;
```

Because there is no CREATE clause, the observations meeting the WHERE condition will be printed to the Output window when the procedure is submitted. The variables listed in the SELECT clause are separated by commas in PROC SQL, not spaces as in a VAR statement in a DATA step. Notice that the SQL solution looks very much like the PROC PRINT solution used in Chapter 1.

An alternative to including a character missing value in the list of valid values (above) is to use the terms IS NULL or IS MISSING, part of the WHERE syntax. Thus, the WHERE clause above could also be written like this:

```
where Gender not in ('M','F') and Gender is not missing   or
      notdigit(trim(DX))and DX is not missing             or
      AE not in ('0','1') and AE is not missing;
```

Here is the output from running Program 8-2.

```
Checking for Invalid Character Data

Patient          Diagnosis  Adverse
Number   Gender  Code       Event?
_____________________________________
002      F       X          0
003      X       3          1
004      F       5          A
010      f       1          0
013      2       1
002      F       X          0
023      f                  0
```

Checking for Outliers

A similar program can be used to check for out-of-range numeric values. The SQL statements in Program 8-3 produce a report for heart rate, systolic blood pressure, and diastolic blood pressure readings outside specified ranges. In this example, missing values are not treated as errors.

Program 8-3 Using SQL to Check for Out-of-Range Numeric Values

```
title "Checking for out-of-range numeric values";
proc sql;
   select Patno,
          HR,
          SBP,
          DBP
   from clean.patients
   where HR  not between 40 and 100 and HR is not missing    or
         SBP not between 80 and 200 and SBP is not missing   or
         DBP not between 60 and 120 and DBP is not missing;
quit;
```

Running this procedure produces the output below. Notice that the column headings are variable labels rather than variable names.

```
Checking for out-of-range numeric values

                    Systolic  Diastolic
Patient     Heart     Blood      Blood
Number       Rate  Pressure   Pressure
──────────────────────────────────────
004           101       200        120
008           210         .          .
009            86       240        180
010             .        40        120
011            68       300         20
014            22       130         90
017           208         .         84
321           900       400        200
020            10        20          8
023            22        34         78
```

Checking a Range Using an Algorithm Based on the Standard Deviation

In Chapter 2 (Program 2-20), an algorithm to detect outliers based on standard deviation was described. Program 8-4 shows an SQL approach using the same algorithm and using the variable systolic blood pressure (SBP).

Program 8-4 Using SQL to Check for Out-of-Range Values Based on the Standard Deviation

```
title "Data values beyond two standard deviations";
proc sql;
   select Patno,
          SBP
   from clean.patients
   having SBP not between mean(SBP) - 2 * std(SBP) and
      mean(SBP) + 2 * std(SBP)                     and
      SBP is not missing;
quit;
```

This program uses two summary functions, MEAN and STD. When these functions are used, a HAVING clause is needed instead of the WHERE clause used earlier. In this example, all values more than two standard deviations away from the mean that are not missing are printed to the Output window.

Here is the output after running Program 8-4.

```
Data values beyond two standard deviations

         Systolic
Patient     Blood
Number   Pressure
_________________
011           300
321           400
```

This program misses some other possible data errors. Techniques developed in Chapter 2, such as using trimmed statistics or non-parametric approaches might produce more useful results.

Checking for Missing Values

It's particularly easy to use PROC SQL to check for missing values. The WHERE clause IS MISSING can be used for both character and numeric variables. The simple query shown in Program 8-5 checks the data set for all character and numeric missing values and prints out any observation that contains a missing value for one or more variables.

Program 8-5 Using SQL to List Missing Values

```
title "Observations with missing values";
proc sql;
   select *
   from clean.patients
   where Patno   is missing or
         Gender  is missing or
         Visit   is missing or
         HR      is missing or
         SBP     is missing or
         DBP     is missing or
```

```
        DX      is missing or
        AE      is missing;
quit;
```

The SELECT clause uses an asterisk (*) to indicate that all the variables in the data set are listed in the FROM clause.

Here is the output from Program 8-5.

```
Observations with missing values

                                          Systolic  Diastolic
Patient                          Heart      Blood      Blood  Diagnosis  Adverse
Number   Gender  Visit Date       Rate   Pressure   Pressure  Code       Event?
──────────────────────────────────────────────────────────────────────────────
006              06/15/1999        72        102         68  6          1
007      M                .        88        148        102             0
         M       11/11/1998        90        190        100             0
008      F       08/08/1998       210          .          .  7          0
010      f       10/19/1999         .         40        120  1          0
011      M                .        68        300         20  4          1
012      M       10/12/1998        60        122         74             0
013      2       08/23/1999        74        108         64  1
014      M       02/02/1999        22        130         90             1
003      M       11/12/1999        58        112         74             0
015      F                .        82        148         88  3          1
017      F       04/05/1999       208          .         84  2          0
019      M       06/07/1999        58        118         70             0
123      M                .        60          .          .  1          0
321      F                .       900        400        200  5          1
020      F                .        10         20          8             0
023      f       12/31/1998        22         34         78             0
027      F                .         .        166        106  7          0
029      M       05/15/1998         .          .          .  4          1
```

Range Checking for Dates

You can also use PROC SQL to check for dates that are out of range. Suppose you want a list of all patients in the PATIENTS data set that have nonmissing visit dates before June 1, 1998 or after October 15, 1999. You use the same programming statements used for checking out-of-range numeric quantities, except that you use a date constant in place of the actual number of days from January 1, 1960 (day 0 according to SAS). The SQL statements in Program 8-6 will do the trick.

Program 8-6 Using SQL to Perform Range Checks on Dates

```
title "Dates before June 1, 1998 or after October 15, 1999";
proc sql;
   select Patno,
          Visit
   from clean.patients
   where Visit not between '01jun1998'd and '15oct1999'd and
         Visit is not missing;
quit;
```

Here is the resulting output from Program 8-6.

```
Dates before June 1, 1998 or after October 15, 1999

Patient
Number   Visit Date
──────────────────
XX5      05/07/1998
010      10/19/1999
003      11/12/1999
028      03/28/1998
029      05/15/1998
```

Checking for Duplicates

In Chapter 5, you used PROC SORT with the NODUPRECS and NODUPKEY options to detect duplicates, as well as a DATA step approach using the FIRST. and LAST. temporary variables. Yes, you guessed it, there is an SQL answer also. If you have a GROUP BY clause in your PROC SQL and follow it with a COUNT function, you can count the frequency of each level of the GROUP BY variable. If you choose patient number (Patno) as the grouping variable, the COUNT function will tell you how many observations there are per patient. Remember to use a HAVING clause when you use summary functions such as COUNT. Look at the SQL code in Program 8-7.

Program 8-7 Using SQL to List Duplicate Patient Numbers

```
title "Duplicate Patient Numbers";
proc sql;
   select Patno,
          Visit
      from clean.patients
      group by Patno
      having count(Patno) gt 1;
quit;
```

In Program 8-7, you are telling PROC SQL to list any duplicate patient numbers. Note that multiple missing patient numbers will not appear in the listing because the COUNT function returns a frequency count only for nonmissing values. Here are the results:

```
Duplicate Patient Numbers

Patient
Number   Visit Date
──────────────────
002      11/13/1998
002      11/13/1998
003      10/21/1998
003      11/12/1999
006      06/15/1999
006      07/07/1999
```

Identifying Subjects with "n" Observations Each

Using the grouping capability of PROC SQL and the COUNT function, you can list all patients who do not have exactly "n" visits or observations in a data set, just as you did in Programs 5-11 and 5-12. Here is the program with an explanation following.

Program 8-8 Using SQL to List Patients Who Do Not Have Two Visits

```
title "Listing of patients who do not have two visits";
proc sql;
   select Patno,
          Visit
      from clean.patients2
      group by Patno
      having count(Patno) ne 2;
quit;
```

By first grouping the observations by patient number, you can then use the COUNT function, which returns the number of observations in a group. Here is the output from Program 8-8.

```
Listing of patients who do not have two visits

Patno       Visit
──────────────────
002    01/01/1999
002    02/09/1999
002    01/10/1999
003    10/21/1998
006    11/11/1998
```

Checking for an ID in Each of Two Files

Do you think PROC SQL can check if each patient number is in two files? Why else is there a section heading with that task listed? Of course you can! Now, on to the problem.

The equivalent of a DATA step merge is called a JOIN in SQL terms. Normally, a JOIN lists only those observations that have a matching value for the variables in each of the files. If you want all observations from both files, regardless if they have a corresponding observation in the other file, you perform a FULL JOIN (this is equivalent to a MERGE where no IN= variables are used). So, if you perform a FULL JOIN between two data sets and an ID value is not in both data sets, one

of the observations will have a missing value for the ID variable. Let's use the same data sets, ONE and TWO, that were used in Chapter 6. For convenience, the code to produce these data sets is shown in Program 8-9.

Program 8-9 Creating Two Data Sets for Testing Purposes

```
data one;
   input Patno X Y;
datalines;
1 69 79
2 56 .
3 66 99
5 98 87
12 13 14
;
data two;
   input Patno Z;
datalines;
1 56
3 67
4 88
5 98
13 99
;
```

Here is the PROC SQL solution for finding ID's that are missing from one of the files:

Program 8-10 Using SQL to Look for ID's That Are Not in Each of Two Files

```
title "Patient numbers not in both files";
proc sql;
   select One.patno as ID_one,
          Two.patno as ID_two
   from one full join two
   on One.patno eq Two.patno
   where One.patno is missing or Two.patno is missing;
quit;
```

Because the variable name Patno is used in both data sets, you can distinguish between them by adding either ONE. or TWO. before the variable name, depending on whether you are referring to the patient number from data set ONE or data set TWO. Also, to make it easier to keep track of

these two variables, an alias (created by using the AS clause) for each of these variable names was created (ID_one and ID_two).

The condition for the FULL JOIN is that the ID's match between the two data sets. This is specified in the ON clause. (Note: You can have a different variable name for the ID variable in each file without causing any problems. In the DATA step solution, using a MERGE statement, you need to rename one of the ID variables so that they are the same.) The WHERE condition will be true for any ID that is not in both data sets.

The output from this query contains all the observations where an ID value is not in both files, as shown next.

```
Patient numbers not in both files

  ID_one    ID_two
_________________
       2         .
       .         4
      12         .
       .        13
```

More Complicated Multi-File Rules

Let's start the discussion of more complicated multi-file rules by redoing the example from Chapter 6. To review, there are two files: AE, which recorded adverse events for patients in the study, and LAB_TEST, which contained the laboratory tests for people with various adverse events (see the Appendix for the programs to create these data sets). The goal is to list any patient who had an adverse event of 'X' anywhere in the adverse event data set who either did not have any entry in the laboratory data set or where the date of the lab test was before the date of the adverse event. The following SQL query will produce the same information as the DATA step solution shown in Chapter 6 in Program 6-8.

Program 8-11 Using SQL to Demonstrate More Complicated Multi-File Rules

```
title1 "Patients with an AE of X who did not have a";
title2 "labtest or where the date of the test is prior";
title3 "to the date of the visit";
proc sql;
   select AE.Patno as AE_Patno label="AE patient number",
          A_event,
```

```
          Date_ae,
          Lab_test.Patno as Labpatno label="Lab patient number",
          Lab_date
   from clean.ae left join clean.lab_test
   on AE.Patno=Lab_test.Patno
   where A_event = 'X'        and
         Lab_date lt Date_ae;
quit;
```

Because the variable Patno had the same label in both the AE and LAB_TEST data sets, a LABEL column modifier was used to re-label these variables so that they could be distinguished in the output listing. An alias (AE_Patno and Labpatno), as well as a label, was selected for each of these variables.

To help explain the difference between a LEFT JOIN, a RIGHT JOIN, and a FULL JOIN, let's execute all three with the data sets ONE and TWO, which were described in the previous section. In Program 8-12, the following SQL statements execute all three joins.

Program 8-12 Example of LEFT, RIGHT, and FULL Joins

```
proc sql;
   title "Left join";
   select one.Patno as One_id,
          two.patno as Two_id
   from one left join two
   on one.Patno eq two.Patno;

   title "Right join";
   select one.Patno as One_id,
          two.Patno as Two_id
   from one right join two
   on one.Patno eq two.Patno;

   title "Full join";
   select one.Patno as One_id,
          two.Patno as Two_id
   from one full join two
   on one.Patno eq two.Patno;
quit;
```

By inspecting the next three listings, it is very easy to see the difference among these three different JOIN operations.

```
Left join

  One_id    Two_id
_________________
       1         1
       2         .
       3         3
       5         5
      12         .

Right join

  One_id    Two_id
_________________
       1         1
       3         3
       .         4
       5         5
       .        13

Full join

  One_id    Two_id
_________________
       1         1
       2         .
       3         3
       .         4
       5         5
      12         .
       .        13
```

Now, back to our example. You want all patients in the AE data set with an adverse event equal to 'X', but only those observations in the LAB_TEST data set where the patient numbers were selected from the adverse event data set. Therefore, the LEFT JOIN operation was used.

Finally, if the lab date (Lab_date) is prior to the adverse event or if the lab date is missing, the statement `Lab_date lt Date_ae` will be true. Notice, in the output that follows, that the same three patients are listed with this query as were listed in the example in Chapter 6.

```
Patients with an AE of X who did not have a
labtest or where the date of the test is prior
to the date of the visit

AE                              Lab
patient  Adverse                patient     Date of
number   event    Date of AE  number     lab test
_____________________________________________
009      X        12/25/1998                    .
011      X        10/10/1998                    .
025      X        02/09/1999  025      01/01/1999
```

PROC SQL provides a very convenient way to conduct many of the data cleaning tasks described throughout this book. There are those of us who still feel more comfortable with DATA step and PROC approaches and others who feel that PROC SQL is the solution to all their problems. You may want to start using PROC SQL more often and become more comfortable with its syntax. Quite likely, you will wind up using a combination of traditional DATA step and PROC approaches along with PROC SQL for your data cleaning and other SAS programming needs.

9 Correcting Errors

Introduction 181
Hardcoding Corrections 181
Describing Named Input 182
Reviewing the UPDATE Statement 184

Introduction

Now that you have developed techniques for detecting data errors, it's time to think about how to correct the errors. The correcting process will vary, depending on your application. For example, you may have a large data warehouse where you can delete observations with errors (or possible errors) or replace possible data errors with missing values. For other applications, you may need to return to the original data forms, determine the correct values, and replace incorrect data values with corrected values.

Hardcoding Corrections

For small data sets where there are only a few corrections to be made, you may consider "hardcoding" your corrections. As an example, take a look at Program 9-1:

Program 9-1 Hardcoding Corrections Using a DATA Step

```
libname clean 'c:\books\clean';
data clean.patients_24Sept2007;
   set clean.patients;
   if Patno = '002' then DX = '3';
   else if Patno = '003' then do;
      Gender = 'F';
      SBP = 160;
   end;
```

```
   else if Patno = '004' then do;
      SBP = 188;
      DBP = 110;
      HR = 90;
   end;
***and so forth;
run;
```

This may seem somewhat inelegant (and it is), but if you need to make a few corrections, this method is reasonable. There are a couple of points you should notice about this program. First, you name the new data set to include a revision date, so that you can keep track of your data sets if you make more corrections in the future. Also, you will want to save this program with a name such as UPDATE_24Sept2007.sas so that you can recreate the corrected data set from the original data, should this need ever arise. Never overwrite your original data set. You may also want to insert a comment to document the source of your data change.

Describing Named Input

You are most likely familiar with three ways of reading raw data in a SAS DATA step: list input (used for delimited data), column input (where you specify the starting and ending column for each variable), and formatted input (where you use pointers and informats). There is one other method for reading raw data called "named input." This input method is somewhat like the middle pedal of a piano—it's always been there but nobody knows what it does or how to use it!

The reason we are introducing this input method will be made clear a little later in this chapter. The best way to explain named input is to show you an example. Take a look at Program 9-2:

Program 9-2 Describing Named Input

```
data named;
   length Char $ 3;
   input x=
         y=
         char=;
datalines;
x=3 y=4 char=abc
y=7
char=xyz z=9
;
```

This input method requires you to follow each variable name with an equal sign. In order to tell SAS that Char is a character variable, you use a LENGTH statement. In the raw data file, you need to enter the variable name, and equal sign, and the value. You can enter your data in any order and if you leave out variables, they will automatically be given missing values. Please take a look at the listing below:

```
Listing of data set NAMED

Char    x    y

abc     3    4
        .    7
xyz     .    .
```

This input method is a bit cumbersome, but you will see that it can be very useful for correcting your data. Let's enter some corrections to the PATIENTS data set (created in Program 1-1) using named input.

Program 9-3 Using Named Input to Make Corrections

```
data correct_24Sept2007;
   length Patno DX $ 3
          Gender Drug $ 1;
   input Patno=
         DX=
         Gender=
         Drug=
         HR=
         SBP=
         DBP=;
datalines;
Patno=002 DX=3
Patno=003 Gender=F SBP=160
Patno=004 SBP=188 DBP=110 HR=90
;
```

Here is a listing of data set CORRECT_24Sept2007:

Listing of CORRECT_24Sept2007

Patno	DX	Gender	Drug	HR	SBP	DBP
002	3			.	.	.
003		F		.	160	.
004				90	188	110

In this example, you can see the advantage of named input. Rather than having to place your data values in columns or having to enter periods for missing values, you can simply enter values for the variables you want to change. The question is, what do you do with this data set? Let's take a moment to review the UPDATE statement.

Reviewing the UPDATE Statement

If you MERGE two data sets that share variable names, the value from the right-most data set will replace the value from the data set to its left, even if the value in the right-most data set is a missing value. However, if you use an UPDATE statement instead of a MERGE statement, a missing value for a variable in the right-most data set will not replace the value in the data set to its left. The right-most data set is sometimes called the transaction data set, and it is used to update values in the master data set. To be sure this is clear, we will demonstrate how UPDATE works with two small data sets, INVENTORY and TRANSACTION. Take a look:

Program 9-4 Demonstrating How UPDATE Works

```
data inventory;
   length PartNo $ 3;
   input PartNo $ Quantity Price;
datalines;
133 200 10.99
198 105 30.00
933 45 59.95
;
data transaction;
   length PartNo $ 3;
   input Partno=
         Quantity=
         Price=;
```

```
datalines;
PartNo=133 Quantity=195
PartNo=933 Quantity=40 Price=69.95
;
proc sort data=inventory;
   by Partno;
run;
proc sort data=transaction;
   by PartNo;
run;
data inventory_24Sept2007;
   update inventory transaction;
   by Partno;
run;
```

To help visualize how UPDATE is working, here is a listing of the INVENTORY and TRANSACTION data sets (after the sort):

```
   Listing of data set INVENTORY

   Part
    No     Quantity    Price

   133        200      10.99
   198        105      30.00
   933         45      59.95

   Listing of Data Set TRANSACTION

   Part
    No     Quantity    Price

   133        195        .
   933         40      69.95
```

The TRANSACTION data set only contains values for two of the three part numbers. You have an updated quantity for part number 133, and for part number 933, you have a new Quantity and Price. Below is a listing of the updated data set:

Listing of data set INVENTORY_24Sept2007

Obs	Part No	Quantity	Price
1	133	195	10.99
2	198	105	30.00
3	933	40	69.95

Whatever method you decide to use when correcting your data, make sure that you are methodical. You should be able to recreate the final, corrected data set from the raw data at any time.

You may also decide to use FSEDIT to correct your errors (see the SAS online documentation for SAS 9 at support.sas.com for information on FSEDIT). Be sure that you have evoked its audit trail feature so that you can keep track of your corrections.

Once you have a clean data set, you probably want to keep it that way. As you will see in the next chapter, you can add integrity constraints to a data set that will not allow invalid data values to enter.

10 Creating Integrity Constraints and Audit Trails

Introducing SAS Integrity Constraints 187
Demonstrating General Integrity Constraints 188
Deleting an Integrity Constraint Using PROC DATASETS 193
Creating an Audit Trail Data Set 193
Demonstrating an Integrity Constraint Involving More than One Variable 200
Demonstrating a Referential Constraint 202
Attempting to Delete a Primary Key When a Foreign Key Still Exists 205
Attempting to Add a Name to the Child Data Set 207
Demonstrating the Cascade Feature of a Referential Constraint 208
Demonstrating the SET NULL Feature of a Referential Constraint 210
Demonstrating How to Delete a Referential Constraint 211

Introducing SAS Integrity Constraints

Starting with SAS 7, a feature called integrity constraints was implemented. Integrity constraints are rules that are stored within a SAS data set that can restrict data values accepted into the data set when new data is added with PROC APPEND, DATA step MODIFY, and SQL insert, delete, or update. These constraints are preserved when the data set is copied using PROC COPY, CPORT, or CIMPORT, or is sorted with PROC SORT. Procedures UPLOAD and DOWNLOAD also preserve constraints.

There are two types of integrity constraints. One type, called general integrity constraints, allows you to restrict data values that are added to a single SAS data set. You can specify specific values for a variable, a range of values, or a requirement that values of a variable be unique or non-missing. You can also create integrity constraints that reflect relationships among variables in a data set.

The other type, called referential integrity constraints, allows you to link data values among SAS data sets. For example, you may have a list of valid patient numbers in a demographic file. You could restrict any patient numbers added to another file to those that exist in the demographic file. You could also prevent the deletion of patient numbers in a demographic file unless all occurrences of those patient numbers were first removed from the related files. Finally, you could link the demographic file and one or more related files so that any change to a patient number in

the demographic file would automatically change all the associated patient number values in the related files.

Integrity constraints can keep a data set "pure" by rejecting any observations that violate one or more of the constraints. You can have SAS create an audit trail data set, providing you with information about observations that failed to meet one or more of the constraints. Using referential constraints, you can ensure consistency of variable values among multiple files.

Integrity constraints can be created with PROC SQL statements, PROC DATASETS, or SCL (SAS Component Language) programs. In this chapter, PROC DATASETS is used to create or delete integrity constraints. If you are already familiar with using SQL to manage integrity constraints, you may choose to use PROC SQL to create, delete, or manage your integrity constraints. Any SQL reference can give you details on the SQL approach. There is also some limited information on using PROC SQL to create and manage integrity constraints in the SAS online documentation at support.sas.com.

Demonstrating General Integrity Constraints

There are four types of general integrity constraints:

Check
: A user defined constraint on a single variable or a relationship among several variables. You can specify a list of values, a range of values, or a relationship among two or more variables.

Not Null
: Missing values are not allowed for this variable.

Unique
: Values for this variable must be unique.

Primary Key
: Values for this variable must be both unique and non-missing. This constraint is particularly useful for observation identifier variables.

To demonstrate how you could use integrity constraints to prevent bad data values being added to an existing SAS data set, let's first create a small data set called HEALTH by running the following program:

Program 10-1 Creating Data Set HEALTH to Demonstrate Integrity Constraints

```
data health;
   input Patno : $3. Gender : $1.  HR    SBP    DBP;
datalines;
001 M  88 140  80
002 F  84 120  78
003 M  58 112  74
004 F   . 200 120
007 M  88 148 102
015 F  82 148  88
;
```

A listing of this data set is shown next.

```
Listing of data set HEALTH
Patno    Gender    HR    SBP    DBP

 001       M       88    140     80
 002       F       84    120     78
 003       M       58    112     74
 004       F        .    200    120
 007       M       88    148    102
 015       F       82    148     88
```

The next program adds general integrity constraints to the HEALTH data set. The constraints are

- Gender must be 'F' or 'M'.
- HR (heart rate) must be between 40 and 100. Missing values are allowed.
- SBP (systolic blood pressure) must be between 80 and 200. Missing values are allowed.

- DBP (diastolic blood pressure) must be between 60 and 140. Missing values are not allowed.
- Patno (patient number) must be unique.

Here are the PROC DATASETS statements:

Program 10-2 Creating Integrity Constraints Using PROC DATASETS

```
proc datasets library=work nolist;
   modify health;
   ic create gender_chk = check
      (where=(gender in('F','M')));
   ic create hr_chk = check
      (where=( HR between 40 and 100 or HR is missing));
   ic create sbp_chk = check
      (where=(SBP between 80 and 200 or SBP is missing));
   ic create dbp_chk = check
      (where=(DBP between 60 and 140));
   ic create id_chk = unique(Patno);
run;
quit;
```

Note that if any of the observations in an existing data set violate any of the integrity constraints you want to create, that integrity constraint will not be created (in the example presented here, the existing data sets does not contain any IC violations). You have some choices: First, you can be sure that there are no integrity constraint violations in your data set before you run PROC DATASETS, or you can create a data set with all the variables but no observations (as shown in Program 10-11).

Running PROC CONTENTS will display the usual data set information as well as the integrity constraints. The PROC CONTENTS statements and the resulting output (edited) are shown next.

```
Alphabetic List of Variables and Attributes
#    Variable    Type    Len

5    DBP         Num       8
2    Gender      Char      1
3    HR          Num       8
1    Patno       Char      3
4    SBP         Num       8


               Alphabetic List of Integrity Constraints

  Integrity                    Where
# Constraint Type   Variables Clause

1 dbp_chk    Check             (DBP>=60 and DBP<=140)
2 gender_chk Check             Gender in ('F', 'M')
3 hr_chk     Check             (HR>=40 and HR<=100) or (HR is null)
4 id_chk     Unique Patno
5 sbp_chk    Check             (SBP>=80 and SBP<=200) or (SBP is null)


Alphabetic List of Indexes and Attributes
# of
              Unique    Owned    Unique
#    Index    Option    by IC    Values

1    Patno    YES       YES           6
```

Notice that each of the WHERE clauses that created the integrity constraints is listed in the output from PROC CONTENTS.

What happens when you try to append data that violates one or more of the integrity constraints? The short DATA step shown next creates a data set (NEW) containing four observations.

Program 10-3 Creating Data Set NEW Containing Valid and Invalid Data

```
data new;
   input Patno : $3. Gender : $1. HR SBP DBP;
datalines;
456 M 66 98 72
567 F 150 130 80
003 M 70 134 86
123 F 66 10 80
013 X 68 120 .
;
```

Here is a listing of data set NEW:

```
Listing of data set NEW
Patno    Gender     HR    SBP    DBP

 456       M        66     98     72
 567       F       150    130     80
 003       M        70    134     86
 123       F        66     10     80
 013       X        68    120      .
```

Data for patient 456 is valid and consistent with all the constraints; patient 567 has a heart rate (HR) outside the valid range; patient 003 is a duplicate patient number; patient 123 has a SBP outside the valid range; and patient 013 has an invalid Gender and a missing value for DBP, both of which violate constraints.

Let's see what happens when you attempt to append the observations in data set NEW to the original HEALTH data set:

Program 10-4 Attempting to Append Data Set NEW to the HEALTH Data Set

```
proc append base=health data=new;
run;
```

The SAS Log (shown next) shows that only one observation from data set NEW was appended to the HEALTH data set. In addition, you see that one observation failed the hr_chk constraint, one observation failed the sbp_chk constraint, one observation failed the gender_chk constraint, and one observation failed the id_chk constraint. Even though patient 013 failed two constraints (invalid Gender and a missing value for DBP), only one of the failed constraints was reported.

SAS reports only the first integrity constraint (in the order they were created) for each observation.

```
NOTE: Appending WORK.NEW to WORK.HEALTH.
WARNING: Add/Update failed for data set WORK.HEALTH because data
value(s) do not comply with integrity constraint hr_chk, 1 observations
rejected.
WARNING: Add/Update failed for data set WORK.HEALTH because data
value(s) do not comply with integrity constraint sbp_chk, 1
observations rejected.
WARNING: Add/Update failed for data set WORK.HEALTH because data
value(s) do not comply with integrity constraint gender_chk, 1
observations rejected.
WARNING: Add/Update failed for data set WORK.HEALTH because data
value(s) do not comply with integrity constraint id_chk, 1 observations
rejected.
NOTE: There were 5 observations read from the data set WORK.NEW.
NOTE: 1 observations added.
```

Deleting an Integrity Constraint Using PROC DATASETS

You can use PROC DATASETS to delete one or more integrity constraints. For example, if you want to remove the constraint on Gender from the HEALTH data set, you would proceed like this:

Program 10-5 Deleting an Integrity Constraint Using PROC DATASETS

```
proc datasets library=work nolist;
   modify health;
   ic delete gender_chk;
run;
quit;
```

Creating an Audit Trail Data Set

You may have a large data warehouse where you can simply throw out observations that fail one or more integrity constraints. If, however, you want to keep track of observations that failed one or more integrity constraints, you need to create an audit trail data set. An audit trail data set has the same name as your SAS data set with a type of "audit." To see the contents of an audit trail data set, say in a PROC PRINT or PROC REPORT, you need to use the TYPE=AUDIT data set option. We will demonstrate this in a moment. To make the audit trail more useful, you can add a user-defined message to each integrity constraint. Program 10-6 shows how to add your own

messages to the integrity constraints, making the audit trail data set more useful. (Note that the integrity constraints are the same ones defined in Program 10-2.)

Program 10-6 Adding User Messages to the Integrity Constraints

```
proc datasets library=work nolist;
   modify health;
   ic create gender_chk = check
      (where=(gender in('F','M')))
   message="Gender must be F or M"
   msgtype=user;

   ic create hr_chk = check
      (where=( HR between 40 and 100 or HR is missing))
   message="HR must be between 40 and 100 or missing"
   msgtype=user;

   ic create sbp_chk = check
      (where=(SBP between 80 and 200 or SBP is missing))
   message="SBP must be between 80 and 200 or missing"
   msgtype=user;

   ic create dbp_chk = check
      (where=(DBP between 60 and 140))
   message="DBP must be between 60 and 140"
   msgtype=user;

   ic create id_chk = unique(Patno)
   message="Patient number must be unique"
   msgtype=user;
run;
quit;
```

Notice that we have added our own message using the keyword MESSAGE= and also specified with MSGTYPE=USER that our message should be placed in the audit trail data set instead of the default system message.

Now it's time to create the audit trail data set. Program 10-7 below uses PROC DATASETS to do this:

Program 10-7 Creating an Audit Trail Data Set

```
proc datasets library=work nolist;
   audit health;
   initiate;
run;
quit;
```

Besides INITIATE, once you have created the audit trail data set, you can SUSPEND (suspend event logging but do not delete the audit trail data set), RESUME (resume event logging), and TERMINATE (terminate event logging and delete the audit trail data set).

Let's rerun Program 10-4 and see what happens. In this example, data set HEALTH was returned to its original condition with six observations and no errors.

```
211  proc append base=health data=new;
212  run;

NOTE: Appending WORK.NEW to WORK.HEALTH.
WARNING: HR must be between 40 and 100 or missing , 1 observations
rejected.
WARNING: SBP must be between 80 and 200 or missing , 1 observations
rejected.
WARNING: Gender must be F or M , 1 observations rejected.
WARNING: Patient number must be unique , 1 observations rejected.
NOTE: There were 5 observations read from the data set WORK.NEW.
NOTE: 1 observations added.
NOTE: The data set WORK.HEALTH has 7 observations and 5 variables.
```

Notice that the log now reports the user messages instead of just the name of the violated constraint.

Since the audit trail data set was initiated prior to running PROC APPEND, you can now look at this data set. Let's first print the entire audit trail data set using PROC PRINT, like this:

Program 10-8 Using PROC PRINT to List the Contents of the Audit Trail Data Set

```
title "Listing of audit trail data set";
proc print data=health(type=audit) noobs;
run;
```

Here is the listing:

```
Listing of audit trail data set
Patno Gender  HR SBP DBP        _ATDATETIME_ _ATOBSNO_ _ATRETURNCODE_

 456    M     66  98  72  28SEP2007:19:23:28      7                .
 567    F    150 130  80  28SEP2007:19:23:28      8                .
 003    M     70 134  86  28SEP2007:19:23:28      9                .
 123    F     66  10  80  28SEP2007:19:23:28     10                .
 013    X     68 120   .  28SEP2007:19:23:28     11                .
 003    M     70 134  86  28SEP2007:19:23:28      9      -2147348311
 013    X     68 120   .  28SEP2007:19:23:28     11      -2147348311
 123    F     66  10  80  28SEP2007:19:23:28     10      -2147348311
 567    F    150 130  80  28SEP2007:19:23:28      8      -2147348311

_ATUSERID_  _ATOPCODE_  _ATMESSAGE_
 Ron Cody       DA
 Ron Cody       DA
 Ron Cody       DA
 Ron Cody       DA
 Ron Cody       DA
 Ron Cody       EA      ERROR: Patient number must be unique
 Ron Cody       EA      ERROR: Gender must be F or M
 Ron Cody       EA      ERROR: SBP must be between 80 and 200 or missing
 Ron Cody       EA      ERROR: HR must be between 40 and 100 or missing
```

Remember that the audit trail data set has the same name as your SAS data set, so you must use the TYPE= data set option. Inspection of the listing shows the original six observations (with a value of the variable _ATOPCODE_ equal to DA (Data Added) and four additional observations listing errors. The _ATOPCODE_ variable has a value of EA (Error Add) for these observations, the _ATRETURNCODE_ value is non-missing, and the _ATMESSAGE_ variable contains the user message you created with the integrity constraints.

Other automatic variables that you may find useful in the audit trail data set are _ATDATETIME_, which reports the date and time a change was made, and _ATUSERID_, which reports who made the change.

To see only those observations that contained one or more integrity constraint violations, you can select observations where the value of _ATOPCODE_ is equal to 'EA' (error adding), 'ED' (error deleting), or 'EU' (error updating). Below is a list of all the _ATOPCODE_ values:

ATOPCODE Values	
Code	Modification
AL	Auditing is resumed
AS	Auditing is suspended
DA	Added data record image
DD	Deleted data record image
DR	Before-update record image
DW	After-update record image
EA	Observation add failed
ED	Observation delete failed
EU	Observation update failed

Armed with this information, you can use PROC REPORT (or PROC PRINT if you prefer) to create an error report, as follows:

Program 10-9 Reporting the Integrity Constraint Violations Using the Audit Trail Data Set

```
title "Integrity Constraint Violations";
proc report data=health(type=audit) nowd;
   where _ATOPCODE_ in ('EA' 'ED' 'EU');
   columns Patno Gender HR SBP DBP _ATMESSAGE_;
   define Patno / order "Patient Number" width=7;
   define Gender / display width=6;
   define HR / display "Heart Rate" width=5;
```

```
   define SBP / display width=3;
   define DBP / display width=3;
   define _atmessage_ / display "_IC Violation_"
                        width=30 flow;
run;
```

Here is the final report:

```
Integrity Constraint Violations

  Patient          Heart
  Number   Gender   Rate  SBP  DBP  _________IC Violation_________
  003      M          70  134   86  ERROR: Patient number must be
                                     unique
  013      X          68  120    .  ERROR: Gender must be F or M
  123      F          66   10   80  ERROR: SBP must be between 80
                                     and 200 or missing
  567      F         150  130   80  ERROR: HR must be between 40
                                     and 100 or missing
```

This is looking much better. The only (slight) problem is that if there is more than one integrity constraint violation in an observation, you only see the first one (based on the order they were created). For example, patient 013 had an invalid value for Gender (X) and DBP (missing) and the _ATMESSAGE_ listed the Gender violation. Therefore, you may correct the violation that is listed in the Audit Trail data set and still have the observation fail an additional integrity constraint.

If you create a data set consisting only of the observations containing integrity constraint errors (as in the report above), you can correct the reported errors and attempt to run PROC APPEND again. Then, if there are still uncorrected errors, you can continue to run PROC APPEND until all the observations that you want to add have been corrected.

As an example, we will correct each of the reported errors in the listing above as follows:

Program 10-10 Correcting Errors Based on the Observations in the Audit Trail Data Set

```
data correct_audit;
   set health(type=audit
              where=(_ATOPCODE_ in ('EA' 'ED' 'EU')));
   if Patno = '003' then Patno = '103';
   else if Patno = '013' then do;
      Gender = 'F';
      DBP = 88;
   end;
   else if Patno = '123' then SBP = 100;
   else if Patno = '567' then HR = 87;
   drop _AT: ;
run;

proc append base=health data=correct_audit;
run;
```

In Program 10-10, all of the errors have been corrected and the corrected file was appended to the original HEALTH file. The notation _AT: in the DROP statement was an instruction to drop all variables beginning with _AT. Looking at the SAS Log (below), you notice that no integrity constraints were violated and all four observations were added to the HEALTH data set.

```
106
107  proc append base=health data=correct_audit;
108  run;

NOTE: Appending WORK.CORRECT_AUDIT to WORK.HEALTH.
NOTE: There were 4 observations read from the data set WORK.CORRECT_AUDIT.
NOTE: 4 observations added.
NOTE: The data set WORK.HEALTH has 11 observations and 5 variables.
NOTE: PROCEDURE APPEND used (Total process time):
      real time           0.03 seconds
      cpu time            0.03 seconds
```

Demonstrating an Integrity Constraint Involving More than One Variable

Check integrity constraints can involve more than a single variable. Suppose you have a survey where you ask how much time a person spends in various activities, as a percentage. Each individual value cannot exceed 100%, and the sum of the values also cannot exceed 100%. The program below first creates a data set with all the required variables, but with zero observations (my thanks to Mike Zdeb, who suggested this method). Next, PROC DATASETS is used to create integrity constraints for each of the percentage variables as well as a constraint on the sum of these variables. The audit trail data set is created in this same procedure.

Program 10-11 Demonstrating an Integrity Constraint Involving More than One Variable

```
data survey;
   length ID $ 3;
   retain ID ' ' TimeTV TimeSleep TimeWork TimeOther .;
   stop;
run;

proc datasets library=work;
   modify survey;
   ic create ID_check = primary key(ID)
      message = "ID must be unique and nonmissing"
      msgtype = user;
   ic create TimeTV_max = check(where=(TimeTV le 100))
      message = "TimeTV must not be over 100"
      msgtype = user;
   ic create TimeSleep_max = check(where=(TimeSleep le 100))
      message = "TimeSleep must not be over 100"
      msgtype = user;
   ic create TimeWork_max = check(where=(TimeWork le 100))
      message = "TimeWork must not be over 100"
      msgtype = user;
   ic create TimeOther_max = check(where=(TimeOther le 100))
      message = "TimeOther must not be over 100"
      msgtype = user;
   ic create Time_total =
      check(where=(sum(TimeTV,TimeSleep,TimeWork,TimeOther) le 100))
      message = "Total percentage cannot exceed 100%"
      msgtype = user;
run;
```

```
   audit survey;
   initiate;
run;
quit;
```

Data set SURVEY has five variables and zero observations. The data set (with or without observations) must exist before you can add integrity constraints.

The last integrity constraint (Time_total) prevents any observations where the total percentage values exceed 100% from being added to the data set. When you create your data set of survey values and attempt to append it to data set SURVEY (which originally contains no observations), here is what happens:

Program 10-12 Added the Survey Data

```
data add;
   input ID $ TimeTV TimeSleep TimeWork TimeOther;
datalines;
001 10 40 40 10
002 20 50 40 5
003 10 . . .
004 0 40 60 0
005 120 10 10 10
;

proc append base=survey data=add;
run;

title "Integrity Constraint Violations";
proc report data=survey(type=audit) nowd;
   where _ATOPCODE_ in ('EA' 'ED' 'EU');
   columns ID TimeTV TimeSleep TimeWork TimeOther _ATMESSAGE_;
   define ID / order "ID Number" width=7;
   define TimeTV / display "Time spent watching TV" width=8;
   define TimeSleep / display "Time spent sleeping" width=8;
   define TimeWork / display "Time spent working" width=8;
   define TimeOther / display "Time spent in other activities"
                      width=10;
   define _atmessage_ / display "_Error Report_"
                        width=30 flow;
run;
```

The resulting report is shown next:

```
Integrity Constraint Violations

                Time                              Time
               spent      Time      Time    spent in
  ID       watching     spent     spent       other
  Number         TV  sleeping   working  activities  ______Error Report_______
  002            20        50        40           5  ERROR: Total percentage
                                                     cannot exceed 100%
  005           120        10        10          10  ERROR: TimeTV must not
                                                     be over 100
```

None of the individual percentages for ID 002 exceeds 100%, but since the total of these percentages exceeded 100%, the Time_total integrity constraint was violated and that observation was not added to the survey. Notice that the message for ID 005 indicated that the value for TimeTV exceeded 100%, even though this observation also violated the total percentage constraint as well. Remember that you only see the message corresponding to the first integrity constraint violation.

Demonstrating a Referential Constraint

In the beginning of this chapter, two classes of constraints were described: general and referential. General constraints relate to a single data set, while referential constraints restrict certain operations on one or more variables among two or more data sets. You choose one or more variables (listed as a primary key) from one data set (called the parent data set) and create relationships with the same variable or variables (called foreign keys) in one or more data sets (called child data sets). The purpose is to restrict modifications to either the primary or foreign keys.

If you update or delete an observation in the parent data set, you need to specify what action you wish to take. Possible actions are:

- RESTRICT prevents the values in the primary key variables from being updated or deleted in the parent data set if these same key values exist as a foreign key in a child data set. This is the default action if you do not specify an action.
- SET NULL allows you to update or delete primary key values when there are matching values in a foreign data set, but the values of the foreign keys are set to missing.

- CASCADE allows values of the primary key variables to be changed, with corresponding values of the foreign keys changed to the same values.

It is important that the variables that you define as primary and foreign key variables be the same type and length.

As an example, we will first create two data sets, MASTER_LIST and SALARY. Both data sets contain first name (FirstName) and last name (LastName). In this scenario you want to be sure that you cannot change or delete a name in the MASTER_LIST data set if that name appears in the SALARY data set. The program that follows creates the two data sets and sets up a referential constraint and two general constraints:

Program 10-13 Creating Two Data Sets and a Referential Constraint

```
data master_list;
   input FirstName : $12. LastName : $12. DOB : mmddyy10. Gender : $1.;
   format DOB mmddyy10.;
datalines;
Julie Chen 7/7/1988 F
Nicholas Schneider 4/15/1966 M
Joanne DiMonte 6/15/1983 F
Roger Clement 9/11/1988 M
;

data salary;
   input FirstName : $12. LastName : $12. Salary : comma10.;
datalines;
Julie Chen $54,123
Nicholas Schneider $56,877
Joanne DiMonte $67,800
Roger Clement $42,000
;

title "Listing of MASTER LIST";
proc print data=master_list;
run;
title "Listing of SALARY";
proc print data=salary;
run;
```

```
proc datasets library=work nolist;
   modify master_list;
   ic create prime_key = primary key (FirstName LastName);
   ic create gender_chk = check(where=(Gender in ('F','M')));
run;

   modify salary;
   ic create foreign_key = foreign key (FirstName LastName)
      references master_list
      on delete restrict on update restrict;
   ic create salary_chk = check(where=(Salary le 90000));
run;
quit;
```

Below is a listing of these two data sets:

```
Listing of MASTER LIST

       First
Obs    Name        LastName            DOB    Gender

 1     Julie       Chen          07/07/1988      F
 2     Nicholas    Schneider     04/15/1966      M
 3     Joanne      DiMonte       06/15/1983      F
 4     Roger       Clement       09/11/1988      M

Listing of SALARY

       First
Obs    Name        LastName        Salary

 1     Julie       Chen           $54,123
 2     Nicholas    Schneider      $56,877
 3     Joanne      DiMonte        $67,800
 4     Roger       Clement        $42,000
```

If you run PROC CONTENTS on data set MASTER_LIST, you will see a listing of both the general and referential constraints. Below, is a section of the PROC CONTENTS output showing this information:

```
                    Alphabetic List of Integrity Constraints

  Integrity                              Where                     On       On
# Constraint Type        Variables       Clause          Reference Delete   Update

1 gender_chk Check                       Gender in
                                         ('F', 'M')
2 prime_key  Primary Key FirstName
                         LastName
  foreign_   Referential                                 WORK.     Restrict Restrict
  key                                                    SALARY
```

You can see in this output that your primary key constraint (Prime_key) and foreign key constraint (Foreign_key) were created. (Please see the section in this chapter for an example of how to delete referential constraints).

Attempting to Delete a Primary Key When a Foreign Key Still Exists

The next program demonstrates what happens when you attempt to delete an observation from a parent data set when that key still exists in a child data set. Program 10-14, without any referential constraints, would delete "Joanne DiMonte" from the MASTER_LIST. However, because of the referential constraint, no observations are deleted.

Program 10-14 Attempting to Delete a Primary Key When a Foreign Key Still Exists

```
*Attempt to delete an observation in the master_list;
data master_list;
   modify master_list;
   if FirstName = 'Joanne' and LastName = 'DiMonte' then remove;
run;

title "Listing of MASTER_LIST";
proc print data=master_list;
run;
```

Because of the referential constraint, the data set is not altered. Below are a copy of the SAS Log and a listing of the MASTER_LIST data set:

```
210  *Attempt to delete an observation in the master_list;
211  data master_list;
212     modify master_list;
213     if FirstName = 'Joanne' and LastName = 'DiMonte' then remove;
214  run;

ERROR: Unable to delete observation.  One or more foreign keys exist, contain matching
       value(s), and are controlled by RESTRICT referential action.
NOTE: The SAS System stopped processing this step because of errors.
NOTE: There were 3 observations read from the data set WORK.MASTER_LIST.
NOTE: The data set WORK.MASTER_LIST has been updated.  There were 0 observations
      rewritten, 0 observations added and 0 observations deleted.
NOTE: There were 0 rejected updates, 0 rejected adds, and 1 rejected deletes.
```

```
Listing of MASTER_LIST

       First
Obs    Name        LastName            DOB    Gender

 1     Julie       Chen          07/07/1988      F
 2     Nicholas    Schneider     04/15/1966      M
 3     Joanne      DiMonte       06/15/1983      F
 4     Roger       Clement       09/11/1988      M
```

As you can see, no observations were deleted from the MASTER_LIST data set.

Attempting to Add a Name to the Child Data Set

This section demonstrates how the referential constraint prevents you from adding a foreign key value to the child data set when that key does not exist in the parent data set. To do this, we will attempt to add a new first and last name to the SALARY data set that does not already exist in the MASTER_LIST data set. Take a look at the following:

Program 10-15 Attempting to Add a Name to the Child Data Set

```
data add_name;
   input FirstName : $12. LastName : $12. Salary : comma10.;
   format Salary dollar9.;
datalines;
David York 77,777
;

proc append base=salary data=add_name;
run;
```

In this program, you are using PROC APPEND to add observations from ADD_NAME to the end of the SALARY data set. The data set named on the BASE= option names the original data set, the name on the DATA= option names the data set containing the observations you want to add. When you use PROC APPEND to add observations from one data set to the bottom of another data set, it does not read observations from the BASE= data set, a possibly huge efficiency gain. If you use a SET statement to perform this task, SAS will first read all observations from both data sets.

The name "David York" does not exist in the MASTER_LIST data set. Therefore, the referential constraint should prevent you from adding this name to the SALARY data set. Below is the SAS Log from running Program 10-15:

```
37   data add_name;
38      input FirstName : $12. LastName : $12. Salary : comma10.;
39      format Salary dollar9.;
40   datalines;

NOTE: The data set WORK.ADD_NAME has 1 observations and 3 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds


42   ;
43
44   proc append base=salary data=add_name;
45   run;

NOTE: Appending WORK.ADD_NAME to WORK.SALARY.
WARNING: Observation was not added/updated because a matching primary key value was
         not found for foreign key _FK0001_. (Occurred 1 times.)
NOTE: There were 1 observations read from the data set WORK.ADD_NAME.
NOTE: 0 observations added.
NOTE: The data set WORK.SALARY has 4 observations and 3 variables.
```

Demonstrating the Cascade Feature of a Referential Constraint

This example shows how updating primary key values will automatically update the corresponding foreign key values in all referencing foreign key data sets.

In the program that follows, realize that the two referential constraints were first deleted before being redefined. The program below creates the referential constraint that will change the value of FirstName and LastName in the SALARY (child) data set when the value is changed in MASTER_LIST (the parent data set):

Program 10-16 Demonstrate the CASCADE Feature of a Referential Integrity Constraint

```
proc datasets library=work nolist;
   modify master_list;
   ic create prime_key = primary key (FirstName LastName);
run;

   modify salary;
   ic create foreign_key = foreign key (FirstName LastName)
      references master_list
   on delete RESTRICT on update CASCADE;
run;
quit;

data master_list;
   modify master_list;
   if FirstName = 'Roger' and LastName = 'Clement' then
      LastName = 'Cody';
run;

title "master list";
proc print data=master_list;
run;
title "salary";
proc print data=salary;
run;
```

Because you have chosen CASCADE for the action to take place when you update a value of the primary key, the value of the two variables FirstName and LastName change in the SALARY data set. Here are listings of both data sets:

```
master list

       First
Obs    Name        LastName           DOB    Gender

 1     Julie       Chen         07/07/1988      F
 2     Nicholas    Schneider    04/15/1966      M
 3     Joanne      DiMonte      06/15/1983      F
 4     Roger       Cody         09/11/1988      M
```

```
salary

       First
Obs    Name        LastName        Salary

 1     Julie       Chen           $54,123
 2     Nicholas    Schneider      $56,877
 3     Joanne      DiMonte        $67,800
 4     Roger       Cody           $42,000
```

Demonstrating the SET NULL Feature of a Referential Constraint

You may want to delete a primary key in a parent data set and have all the foreign key references set to a missing value (leaving all the other data for that observation intact). To accomplish this, you use the clause ON DELETE SET NULL as demonstrated in Program 10-17. (Note: The two data sets MASTER_LIST and SALARY were returned to their original condition as shown following Program 10-13.)

Program 10-17 Demonstrating the SET NULL Feature of a Referential Constraint

```
proc datasets library=work nolist;
   modify master_list;
   ic create primary key (FirstName LastName);
run;

   modify salary;
   ic create foreign key (FirstName LastName) references master_list
   on delete SET NULL on update CASCADE;
run;
quit;

data master_list;
   modify master_list;
   if FirstName = 'Roger' and LastName = 'Clement' then
      remove;
run;
```

```
title "master list";
proc print data=master_list;
run;
title "salary";
proc print data=salary;
run;
```

When you run this program, the observation for Roger Clement is deleted and the values of FirstName and LastName are set to missing values (null) in the SALARY data set (as shown below).

```
Listing of MASTER LIST

       First
Obs    Name        LastName            DOB    Gender

 1     Julie       Chen          07/07/1988      F
 2     Nicholas    Schneider     04/15/1966      M
 3     Joanne      DiMonte       06/15/1983      F

Listing of SALARY

       First
Obs    Name        LastName        Salary

 1     Julie       Chen           $54,123
 2     Nicholas    Schneider      $56,877
 3     Joanne      DiMonte        $67,800
 4                                $42,000
```

Demonstrating How to Delete a Referential Constraint

If you want or need to delete a referential constraint, you can use PROC DATASETS (the same way you deleted general integrity constraints earlier). You also need to delete all foreign keys before you can delete a primary key. As an example, if you want to remove the referential constraints between the MASTER_LIST and SALARY data sets, you could proceed as follows:

Program 10-18 Demonstrating How to Delete a Referential Constraint

```
*delete prior referential integrity constraint;
*Note: Foreign key must be deleted before the primary key can be deleted;
proc datasets library=work nolist;
   modify salary;
   ic delete foreign_key;
run;

   modify master_list;
   ic delete prime_key;
run;
quit;
```

Notice that you remove the foreign key before the primary key, since you cannot delete a primary key constraint unless all of the foreign key constraints have been deleted first.

This chapter demonstrated a few ways that you can use integrity constraints to prevent data errors from being added to a SAS data set or to maintain the integrity of key values among several data sets.

11 DataFlux and dfPower Studio

Introduction 213
Examples 215

Introduction

A SAS product called DataFlux (and its graphical user interface, dfPower Studio) is a product designed to perform data cleaning functions.

This product allows you to collect, store, examine, and manage various data quality and integration logic and business rules. For example, you can:

- perform address standardization
- standardize company and product names
- perform "fuzzy" matches among files
- parse names so that matching can be accomplished
- identify which records belong to the same household
- perform redundant data identification
- eliminate duplicates using clustering algorithms to look for duplicates and near-duplicate records
- merge records
- perform data linking and joining
- create and edit regular expressions to verify consistent data formats
- create and modify phonetics library files for data matching
- create and modify look-up tables and parsing rules for complex parsing routines

Using the dfPower Studio integration tool, you can:

- store all business rules generated for each of the data quality objects
- allow reuse of these data quality jobs and objects across the various applications
- facilitate quick launches of various stored data management jobs with a few clicks of the mouse
- maintain all reports that are generated during data quality processing
- manage various configurations of the various data quality processes routinely implemented by the organization
- maintain information about batch jobs and the various schedules of these jobs

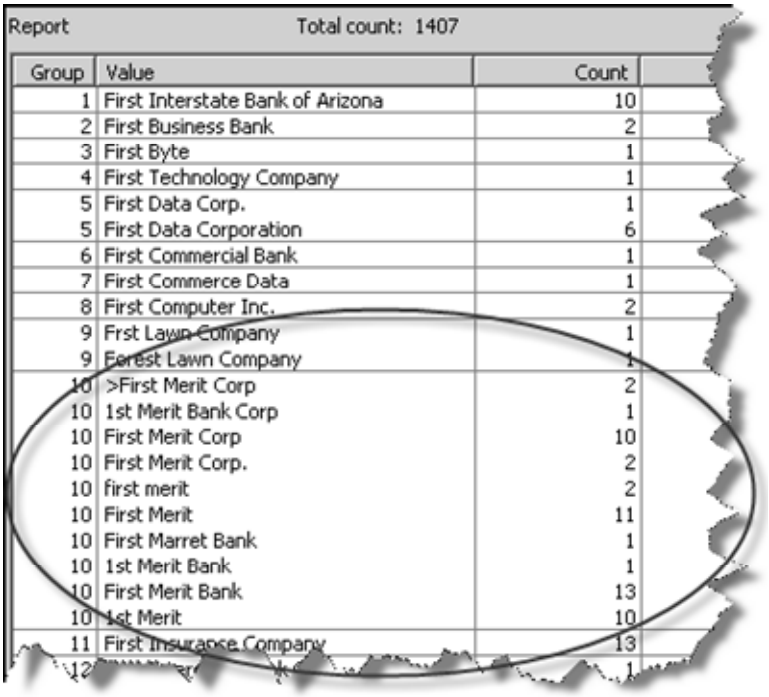
AIC data management (Analyze, Improve, Control) steps include:

- data profiling
- data quality
- data integration
- data enrichment
- data monitoring

Examples

Below are some sample screens, showing typical tasks performed by DataFlux and dfPower Studio:

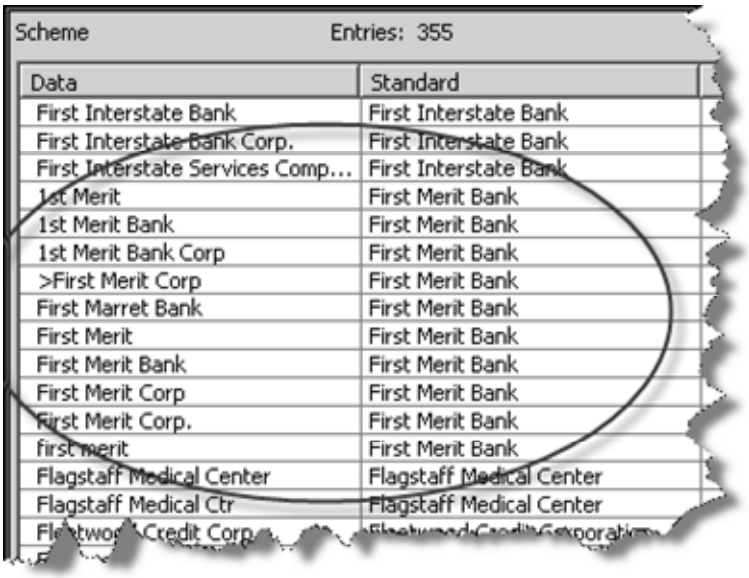
Performing a "fuzzy" sort causing differing names for the same company to be placed together:



Report Total count: 1407

Group	Value	Count
1	First Interstate Bank of Arizona	10
2	First Business Bank	2
3	First Byte	1
4	First Technology Company	1
5	First Data Corp.	1
5	First Data Corporation	6
6	First Commercial Bank	1
7	First Commerce Data	1
8	First Computer Inc.	2
9	Frst Lawn Company	1
9	Forest Lawn Company	1
10	>First Merit Corp	2
10	1st Merit Bank Corp	1
10	First Merit Corp	10
10	First Merit Corp.	2
10	first merit	2
10	First Merit	11
10	First Marret Bank	1
10	1st Merit Bank	1
10	First Merit Bank	13
10	1st Merit	10
11	First Insurance Company	13
12		1

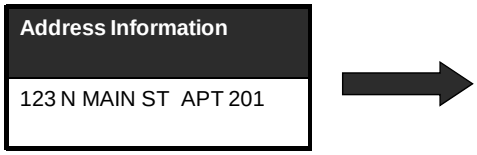
The resulting scheme shows how these names are standardized:



Scheme Entries: 355

Data	Standard
First Interstate Bank	First Interstate Bank
First Interstate Bank Corp.	First Interstate Bank
First Interstate Services Comp...	First Interstate Bank
1st Merit	First Merit Bank
1st Merit Bank	First Merit Bank
1st Merit Bank Corp	First Merit Bank
>First Merit Corp	First Merit Bank
First Marret Bank	First Merit Bank
First Merit	First Merit Bank
First Merit Bank	First Merit Bank
First Merit Corp	First Merit Bank
First Merit Corp.	First Merit Bank
first merit	First Merit Bank
Flagstaff Medical Center	Flagstaff Medical Center
Flagstaff Medical Ctr	Flagstaff Medical Center
Fle[illegible] Credit Corp	[illegible]

The following screen shows that DataFlux can parse items into their constituents:



Address Information
123 N MAIN ST APT 201

Parsing Information	
Street Number	123
Pre-direction	N
Street Name	MAIN
Street Type	ST
Post-direction	
Address Extension	APT
Address Extension Number	201

What is shown here is the tip of the iceberg. For more information on DataFlux and dfPower Studio, please contact SAS at (919) 677-8000 and request information or a demonstration of this powerful package.

Appendix Listing of Raw Data Files and SAS Programs

Programs and Raw Data Files Used in This Book 217
Description of the Raw Data File PATIENTS.TXT 217
Layout for the Data File PATIENTS.TXT 218
Listing of Raw Data File PATIENTS.TXT 218
Program to Create the SAS Data Set PATIENTS 219
Listing of Raw Data File PATIENTS2.TXT 220
Program to Create the SAS Data Set PATIENTS2 221
Program to Create the SAS Data Set AE (Adverse Events) 221
Program to Create the SAS Data Set LAB_TEST 222
Listings of the Data Cleaning Macros Used in This Book 222

Programs and Raw Data Files Used in This Book

You can download all the programs and data files used in this book from the SAS web site: http://support.sas.com/publishing. Click the link for SAS Press Companion Sites and select *Cody's Data Cleaning Techniques Using SAS, Second Edition*. Finally, click on the link for Example Code and Data and you can download a text file containing all of the programs, macros, and text files used in this book.

This Appendix also contains listings of the programs and data files to create the primary SAS data sets used in this book, as well as a listing of all the macros.

Description of the Raw Data File PATIENTS.TXT

The raw data file PATIENTS.TXT contains both character and numeric variables from a typical clinical trial. A number of data errors were included in the file so that you can test the data cleaning programs that are developed in this text. The programs in this book assume that the file PATIENTS.TXT is located in a directory (folder) called C:\BOOKS\CLEAN. This is the directory that is used throughout this book as the

location for data files, SAS data sets, SAS programs, and SAS macros. You should be able to modify the INFILE and LIBNAME statements to fit your own operating environment.

Layout for the Data File PATIENTS.TXT

Variable Name	Description	Starting Column	Length	Variable Type	Valid Values
PATNO	Patient Number	1	3	Character	Numerals only
GENDER	Gender	4	1	Character	'M' or 'F'
VISIT	Visit Date	5	10	MMDDYY10.	Any valid date
HR	Heart Rate	15	3	Numeric	Between 40 and 100
SBP	Systolic Blood Pressure	18	3	Numeric	Between 80 and 200
DBP	Diastolic Blood Pressure	21	3	Numeric	Between 60 and 120
DX	Diagnosis Code	24	3	Character	1 to 3 digit numeral
AE	Adverse Event	27	1	Character	'0' or '1'

Listing of Raw Data File PATIENTS.TXT

```
12345678901234567890123456 (ruler)
----------------------------------
001M11/11/1998 88140 80  10
002F11/13/1998 84120 78  X0
003X10/21/1998 68190100  31
004F01/01/1999101200120  5A
XX5M05/07/1998 68120 80  10
006 06/15/1999 72102 68  61
007M08/32/1998 88148102   0
   M11/11/1998 90190100   0
008F08/08/1998210        70
009M09/25/1999 86240180  41
010f10/19/1999    40120  10
```

```
011M13/13/1998 68300 20  41
012M10/12/98   60122 74   0
013208/23/1999 74108 64   1
014M02/02/1999 22130 90   1
002F11/13/1998 84120 78  X0
003M11/12/1999 58112 74   0
015F           82148 88  31
017F04/05/1999208    84  20
019M06/07/1999 58118 70   0
123M15/12/1999 60        10
321F          900400200  51
020F99/99/9999 10 20  8   0
022M10/10/1999 48114 82  21
023f12/31/1998 22 34 78   0
024F11/09/199876 120 80  10
025M01/01/1999 74102 68  51
027FNOTAVAIL  NA 166106  70
028F03/28/1998 66150 90  30
029M05/15/1998           41
006F07/07/1999 82148 84  10
```

Program to Create the SAS Data Set PATIENTS

```
*-----------------------------------------------------------*
|PROGRAM NAME: PATIENTS.SAS in C:\BOOKS\CLEAN               |
|PURPOSE: To create a SAS data set called PATIENTS          |
*-----------------------------------------------------------*;
libname clean "c:\books\clean";

data clean.patients;
   infile "c:\books\clean\patients.txt"
          lrecl=30 truncover; /* take care of problems
                                 with short records */

   input @1  Patno    $3. @4  gender   $1.
         @5  Visit    mmddyy10.
         @15 HR       3.
         @18 SBP      3.
         @21 DBP      3.
         @24 Dx       $3.
         @27 AE       $1.;
```

```
   LABEL Patno   = "Patient Number"
         Gender  = "Gender"
         Visit   = "Visit Date"
         HR      = "Heart Rate"
         SBP     = "Systolic Blood Pressure"
         DBP     = "Diastolic Blood Pressure"
         Dx      = "Diagnosis Code"
         AE      = "Adverse Event?";
    format visit mmddyy10.;
run;
```

Listing of Raw Data File PATIENTS2.TXT

```
Listing of the File PATIENTS2.TXT

         1         2
12345678901234567890123456 (ruler)
----------------------------------
00106/12/1998 80130 80
00106/15/1998 78128 78
00201/01/1999 48102 66
00201/10/1999 70112 82
00202/09/1999 74118 78
00310/21/1998 68120 70
00403/12/1998 70102 66
00403/13/1998 70106 68
00504/14/1998 72118 74
00504/14/1998 74120 80
00611/11/1998100180110
00709/01/1998 68138100
00710/01/1998 68140 98
```

Program to Create the SAS Data Set PATIENTS2

```
libname clean "c:\books\clean";

data clean.patients2;
   infile "c:\books\clean\patients2.txt" truncover;
   input @1  Patno  $3.
         @4  Visit  mmddyy10.
         @14 HR      3.
         @17 SBP     3.
         @20 DBP     3.;
   format Visit mmddyy10.;
run;
```

Program to Create the SAS Data Set AE (Adverse Events)

```
libname clean "c:\books\clean";

data clean.ae;
   input @1  Patno   $3.
         @4  Date_ae mmddyy10.
         @14 A_event $1.;
   label Patno   = 'Patient ID'
         Date_ae = 'Date of AE'
         A_event = 'Adverse event';
   format Date_ae mmddyy10.;
datalines;
00111/21/1998W
00112/13/1998Y
00311/18/1998X
00409/18/1998O
00409/19/1998P
01110/10/1998X
01309/25/1998W
00912/25/1998X
02210/01/1998W
02502/09/1999X
;
```

Program to Create the SAS Data Set LAB_TEST

```
libname clean "c:\books\clean";

data clean.lab_test;
   input @1  Patno    $3.
         @4  Lab_date date9.
         @13 WBC      5.
         @18 RBC      4.;
   label  Patno    = 'Patient ID'
          Lab_date = 'Date of lab test'
          WBC      = 'White blood cell count'
          RBC      = 'Red blood cell count';
   format Lab_date mmddyy10.;
datalines;
00115NOV1998 90005.45
00319NOV1998 95005.44
00721OCT1998 82005.23
00422DEC1998110005.55
02501JAN1999 82345.02
02210OCT1998 80005.00
;
```

Listing of the Data Cleaning Macros Used in This Book

Creating a Macro to List the Highest and Lowest "n" Percent of the Data Using PROC UNIVARIATE

```
*-------------------------------------------------------------*
| Program Name: HILOWPER.SAS  in c:\books\clean               |
| Purpose: To list the n percent highest and lowest values for  |
|          a selected variable.                               |
| Arguments: Dsn    - Data set name                           |
|            Var     - Numeric variable to test               |
|            Percent - Upper and Lower percentile cutoff      |
|            Idvar   - ID variable to print in the report     |
```

```
| Example: %hilowper(Dsn=clean.patients,                        |
|                    Var=SBP,                                   |
|                    Perecent=10,                               |
|                    Idvar=Patno)                               |
*---------------------------------------------------------------*;

%macro hilowper(Dsn=,     /* Data set name                     */
                Var=,     /* Variable to test                  */
                Percent=, /* Upper and lower percentile cutoff */
                Idvar=    /* ID variable                       */);

   ***Compute upper percentile cutoff;
   %let Up_per = %eval(100 - &Percent);

   proc univariate data=&Dsn noprint;
      var &Var;
      id &Idvar;
      output out=tmp pctlpts=&Percent &Up_per pctlpre = L_;
   run;

   data hilo;
      set &Dsn(keep=&Idvar &Var);
      if _n_ = 1 then set tmp;
      if &Var le L_&percent and not missing(&Var) then do;
         range = 'Low ';
         output;
      end;
      else if &Var ge L_&Up_per then do;
         range = 'High';
         output;
      end;
   run;

   proc sort data=hilo;
      by &Var;
   run;

   title "Low and High Values for Variables";
   proc print data=hilo;
      id &Idvar;
      var Range &Var;
   run;
```

```
   proc datasets library=work nolist;
     delete tmp hilo;
   run;
   quit;

%mend hilowper ;
```

Creating a Macro to List the Highest and Lowest "n" Percent of the Data Using PROC RANK

```
*-------------------------------------------------------------*
| Macro Name: top_bottom_nPercent                             |
| Purpose: To list the upper and lower n% of values           |
| Arguments: Dsn     - Data set name (one- or two-level       |
|            Var     - Variable to test                       |
|            Percent - Upper and lower n%                     |
|            Idvar   - ID variable                            |
| Example: %top_bottom_nPercent(Dsn=clean.patients,           |
|                               Var=SBP,                      |
|                               Percent=10,                   |
|                               Idvar=Patno)                  |
*-------------------------------------------------------------*;

%macro top_bottom_nPercent
   (Dsn=,
    Var=,
    Percent=,
    Idvar=);
   %let Bottom = %eval(&Percent - 1);
   %let Top = %eval(100 - &Percent);

   proc format;
      value rnk 0 - &Bottom = 'Low'
                &Top - 99    = 'High';
   run;

   proc rank data=&Dsn(keep=&Var &Idvar)
             out=new(where=(&Var is not missing))
             groups=100;
      var &Var;
      ranks Range;
   run;
```

```
   ***Sort and keep top and bottom n%;
   proc sort data=new(where=(Range le &Bottom or
                             Range ge &Top));
      by  &Var;
   run;

   ***Produce the report;
   proc print data=new;
   title "Upper and Lower &Percent.% Values for %upcase(&Var)";
      id &Idvar;
      var Range &Var;
      format Range rnk.;
   run;

   proc datasets library=work nolist;
      delete new;
   run;
   quit;

%mend top_bottom_nPercent;
```

Creating a Macro to List the Highest and Lowest "n" Values

```
*-------------------------------------------------------------*
| Macro Name: highlow                                         |
| Purpose: To list the "n" highest and lowest values          |
| Arguments: Dsn     - Data set name (one- or two-level       |
|            Var     - Variable to list                       |
|            Idvar   - ID variable                            |
|            n       - Number of values to list               |
| Example: %highlow(Dsn=clean.patients,                       |
|                   Var=SBP,                                  |
|                   Idvar=Patno,                              |
|                   n=7)                                      |
*-------------------------------------------------------------*;
```

```
%macro highlow(Dsn=,     /* Data set name          */
               Var=,     /* Variable to list       */
               Idvar=,   /* ID Variable            */
               n=        /* Number of high and low
                            values to list         */);
   proc sort data=&Dsn(keep=&Idvar &Var
             where=(&Var is not missing)) out=tmp;
      by &Var;
   run;
   data _null_;
      set tmp nobs=Num_obs;
      call symput('Num',Num_obs);
      stop;
   run;

   %let High = %eval(&Num - &n + 1);

   title "&n Highest and Lowest Values for &Var";
   data _null_;
      set tmp(obs=&n)         /* lowest values */
          tmp(firstobs=&High) /* highest values */;
      file print;
      if _n_ le &n then do;
         if _n_ = 1 then put / "&n Lowest Values" ;
         put "&Idvar = " &Idvar @15 "Value = " &Var;
      end;
      else if _n_ ge %eval(&n + 1) then do;
         if _n_ = %eval(&n + 1) then put / "&n Highest Values" ;
         put "&Idvar = " &Idvar @15 "Value = " &Var;
      end;
   run;
   proc datasets library=work nolist;
      delete tmp;
   run;
   quit;
%mend highlow;
```

Creating a Macro to List Out-of-Range Data Values

```
*-------------------------------------------------------------*
| Program Name: RANGE.SAS  in C:\books\clean                  |
| Purpose: Macro that takes lower and upper limits for a      |
|          numeric variable and an ID variable to print out   |
|          an exception report to the Output window.          |
| Arguments: Dsn    - Data set name                           |
|            Var    - Numeric variable to test                |
|            Low    - Lowest valid value                      |
|            High   - Highest valid value                     |
|            Idvar  - ID variable to print in the exception   |
|                     report                                  |
| Example: %range(Dsn=CLEAN.PATIENTS,                         |
|                 Var=HR,                                     |
|                 Low=40,                                     |
|                 High=100,                                   |
|                 Idvar=Patno)                                |
*-------------------------------------------------------------*;

%macro range(Dsn=    /* Data set name               */,
             Var=    /* Variable you want to check  */,
             Low=    /* Low value                   */,
             High=   /* High value                  */,
             Idvar=  /* ID variable                 */);

title "Listing of Out of range Data Values";
   data _null_;
      set &Dsn(keep=&Idvar &Var);
      file print;
      if (&Var lt &Low and not missing(&Var)) or &Var gt &High then
         put "&Idvar:" &Idvar  @18 "Variable:&VAR"
                               @38 "Value:" &Var
                               @50 "out-of-range";
   run;

%mend range;
```

Writing a Program to Summarize Data Errors on Several Variables

```
*-------------------------------------------------------------*
| PROGRAM NAME: ERRORS.SAS  in c:\books\clean                 |
| PURPOSE: Accumulates errors for numeric variables in a SAS  |
|          data set for later reporting.                      |
|          This macro can be called several times with a      |
|          different variable each time. The resulting errors |
|          are accumulated in a temporary SAS data set called |
|          errors.                                            |
| ARGUMENTS: Dsn=    - SAS data set name (assigned with a %LET) |
|            Idvar=  - Id variable (assigned with a %LET)     |
|                                                             |
|            Var     = The variable name to test              |
|            Low     = Lowest valid value                     |
|            High    = Highest valid value                    |
|            Missing = IGNORE (default) Ignore missing values |
|                      ERROR Missing values flagged as errors |
|                                                             |
| EXAMPLE: %let Dsn = clean.patients;                         |
|          %let Idvar = Patno;                                |
|                                                             |
|          %errors(Var=HR, Low=40, High=100, Missing=error)   |
|          %errors(Var=SBP, Low=80, High=200, Missing=ignore) |
|          %errors(Var=DBP, Low=60, High=120)                 |
|                                                             |
|          Test the numeric variables HR, SBP, and DBP in data |
|          set clean.patients for data outside the ranges     |
|          40 to 100, 80 to 200, and 60 to 120 respectively.  |
|          The ID variable is Patno and missing values are to |
|          be flagged as invalid for HR but not for SBP or DBP. |
*-------------------------------------------------------------*;
%macro errors(Var=,    /* Variable to test     */
              Low=,    /* Low value            */
              High=,   /* High value           */
              Missing=ignore
                       /* How to treat missing values         */
                       /* Ignore is the default.  To flag     */
                       /* missing values as errors set        */
                       /* Missing=error                       */);
```

```
data tmp;
   set &dsn(keep=&Idvar &Var);
   length Reason $ 10 Variable $ 32;
   Variable = "&Var";
   Value = &Var;
   if &Var lt &Low and not missing(&Var) then do;
      Reason='Low';
      output;
   end;
   %if %upcase(&Missing) ne IGNORE %then %do;
   else if missing(&Var) then do;
      Reason='Missing';
      output;
   end;
   %end;

   else if &Var gt &High then do;
        Reason='High';
      output;
      end;
      drop &Var;
   run;
   proc append base=errors data=tmp;
   run;

%mend errors;

***Error Reporting Macro - to be run after ERRORS has been called
   as many times as desired for each numeric variable to be tested;

%macro report;
   proc sort data=errors;
      by &Idvar;
   run;

proc print data=errors;
      title "Error Report for Data Set &Dsn";
      id &Idvar;
      var Variable Value Reason;
   run;
```

```
   proc datasets library=work nolist;
      delete errors;
      delete tmp;
   run;
   quit;

%mend report;
```

Creating a Macro to Detect Outliers Based on Trimmed Statistics

```
%macro trimmed
   (/* the data set name (DSN) and ID variable (IDVAR)
       need to be assigned with %let statements
       prior to calling this macro */
       Var=,     /* Variable to test for outliers */
       N_sd=2,   /* Number of standard deviations */
       Trim=10   /* Percent top and bottom trim   */
                 /* Valid values of Trim are      */
                 /* 5, 10, 20, and 25             */);

   /************************************************************

   Example:
   %let dsn=clean.patients;
   %let idvar=Patno;

   %trimmed(Var=HR,
            N_sd=2,
            Trim=20)

   ************************************************************/
   title "Outliers for &Var based on &N_sd Standard Deviations";
   title2 "Trimming &Trim% from the Top and Bottom of the Values";

   %if &Trim eq 5 or
       &Trim eq 10 or
       &Trim eq 20 or
       &Trim eq 25 %then %do;

   %let NGroups = %eval(100/&Trim);
   %if &Trim = 5 %then %let Mult = 1.24;
   %else %if &trim = 10 %then %let Mult = 1.49;
```

```
   %else %if &trim = 20 %then %let Mult = 2.12;
   %else %if &trim = 25 %then %let Mult = 2.59;

   proc rank data=&dsn(keep=&Idvar &Var)
             out=tmp groups=&NGroups;
      var &var;
      ranks rank;
   run;
   proc means data=tmp noprint;
      where rank not in (0,%eval(&Ngroups - 1));
      var &Var;
      output out=means(drop=_type_ _freq_)
             mean=Mean
             std=Sd;
   run;

   data _null_;
      file print;
      set &dsn;
      if _n_ = 1 then set means;
      if &Var lt Mean - &N_sd*&Mult*Sd and
         not missing(&Var) or
         &Var gt Mean + &N_sd*&Mult*Sd
         then put &Idvar= &Var=;
   run;

   proc datasets library=work;
      delete means;
   run;
   quit;
   %end;

   %else %do;
   data _null_;
      file print;
      put "You entered a value of &trim for the Trim Value."/
          "It must be 5, 10, 20, or 25";
   run;
   %end;

%mend trimmed;
```

Creating a Macro to List Outliers of Several Variables Based on Trimmed Statistics (Using PROC UNIVARIATE)

```
%macro auto_outliers(
   Dsn=,      /* Data set name                        */
   ID=,       /* Name of ID variable                  */
   Var_list=, /* List of variables to check           */
              /* separate names with spaces           */
   Trim=.1,   /* Integer 0 to n = number to trim      */
              /* from each tail; if between 0 and .5, */
              /* proportion to trim in each tail      */
   N_sd=2     /* Number of standard deviations        */);
   ods listing close;
   ods output TrimmedMeans=trimmed(keep=VarName Mean Stdmean DF);
   proc univariate data=&Dsn trim=&Trim;
      var &Var_list;
   run;
   ods output close;

   data restructure;
      set &Dsn;
      length Varname $ 32;
      array vars[*] &Var_list;
      do i = 1 to dim(vars);
         Varname = vname(vars[i]);
         Value = vars[i];
         output;
      end;
      keep &ID Varname Value;
   run;

   proc sort data=trimmed;
      by Varname;
   run;

   proc sort data=restructure;
      by Varname;
   run;
```

```
   data outliers;
      merge restructure trimmed;
      by Varname;
      Std = StdMean*sqrt(DF + 1);
      if Value lt Mean - &N_sd*Std and not missing(Value)
         then do;
            Reason = 'Low  ';
            output;
         end;
      else if Value gt Mean + &N_sd*Std
         then do;
         Reason = 'High';
         output;
      end;
   run;

   proc sort data=outliers;
      by &ID;
   run;

   ods listing;
   title "Outliers based on trimmed Statistics";
   proc print data=outliers;
      id &ID;
      var Varname Value Reason;
   run;

   proc datasets nolist library=work;
      delete trimmed;
      delete restructure;
      *Note: work data set outliers not deleted;
   run;
   quit;
%mend auto_outliers;
```

Detecting Outliers Based on the Interquartile Range

```
%macro interquartile
   (/* the data set name (Dsn) and ID variable (Idvar)
       need to be assigned with %let statements
       prior to calling this macro */
       var=,     /* Variable to test for outliers  */
       n_iqr=2   /* Number of interquartile ranges */);

/*************************************************************
This macro will list outliers based on the interquartile range.

Example: To list all values beyond 1.5 interquartile ranges
   from a data set called clean.patients for a variable
   called hr, use the following:

   %let Dsn=clean.patients;
   %let Idvar=Patno;

   %interquartile(var=HR,
                  n_iqr=1.5)
*************************************************************/

   title "Outliers Based on &N_iqr Interquartile Ranges";

   proc means data=&dsn noprint;
      var &var;
      output out=tmp
             q1=Lower
             q3=Upper
             qrange=Iqr;
   run;

   data _null_;
      set &dsn(keep=&Idvar &Var);
      file print;
      if _n_ = 1 then set tmp;
      if &Var le Lower - &N_iqr*Iqr and not missing(&Var) or
         &Var ge Upper + &N_iqr*Iqr then
         put &Idvar= &Var=;
   run;
```

```
   proc datasets library=work;
      delete tmp;
   run;
   quit;
%mend interquartile;
```

Creating a Macro to Search for Specific Numeric Values

```
*-------------------------------------------------------------------*
| Macro name: find_value.sas  in c:\books\clean                     |
| purpose: Identifies any specified value for all numeric vars      |
| Calling arguments: dsn=   sas data set name                       |
|                    value=   numeric value to search for           |
| example:  to find variable values of 9999 in data set test, use   |
|                                                                   |
|           %find_value(dsn=test, value=9999)                       |
*-------------------------------------------------------------------*;
%macro find_value(dsn=,      /* The data set name                 */
                  value=999  /* Value to look for, default is 999 */ );
   title "Variables with &value as missing values";
   data temp;
      set &dsn;
      file print;
      length Varname $ 32;
      array nums[*] _numeric_;
      do __i = 1 to dim(nums);
         if nums[__i] = &value then do;
         Varname = vname(nums[__i]);
         output;
         end;
      end;
      keep Varname;
   run;
   proc freq data=temp;
      tables Varname / out=summary(keep=Varname Count)
                       nocum;
   run;
   proc datasets library=work;
      delete temp;
   run;
   quit;
%mend find_value;
```

Creating a Macro to Check for ID's Across Multiple Data Sets

```
*-------------------------------------------------------------*
| Program Name: check_id.sas  in c:\books\clean               |
| Purpose: Macro which checks if an ID exists in each of n files |
| Arguments: The name of the ID variable, followed by as many  |
|            data sets names as desired, separated by BLANKS   |
| Example: %check_id(ID = Patno,                              |
|                    Dsn_list=one two three)                  |
| Date: Sept 17, 2007                                         |
*-------------------------------------------------------------*;
%macro check_id(ID=,       /* ID variable              */
                Dsn_list=  /* List of data set names,  */
                           /* separated by spaces      */);
   %do i = 1 %to 99;
     /* break up list into data set names */
      %let Dsn = %scan(&Dsn_list,&i);
      %if &Dsn ne %then %do; /* If non null data set name       */
         %let n = &i;        /* When you leave the loop, n will */
                             /* be the number of data sets      */
         proc sort data=&Dsn(keep=&ID) out=tmp&i;
            by &ID;
         run;
      %end;
   %end;

   title  "Report of data sets with missing ID's";
   title2 "-----------------------------------";
   data _null_;
      file print;
      merge

      %do i = 1 %to &n;
         tmp&i(in=In&i)
      %end;

      end=Last;
      by &ID;
```

```
      if Last and nn eq 0 then do;
         put "All ID's Match in All Files";
         stop;
      end;

      %do i = 1 %to &n;
         %let Dsn = %scan(&Dsn_list,&i);
         if not In&i then do;
            put "ID " &ID "missing from data set &dsn";
            nn + 1;
         end;
      %end;

      run;

%mend check_id;
```

Index

A

AIC data management 214
algorithm based on standard deviation, checking ranges with 71–72, 169–170
ALL keyword 122–123
ampersand (&) 41, 102
AND operator 53
APPEND procedure 187
  adding errors to data sets 65
  adding names to child data sets 207
  audit trail data 195, 198
  BASE= option 207
  DATA= option 207
ARRAY statement 101
asterisk (*) 171
ATDATETIME automatic variable 196
ATMESSAGE automatic variable 196, 198
ATOPCODE automatic variable 196–197
ATRETURNCODE automatic variable 196
ATUSERID automatic variable 196
AUDIT statement, DATASETS procedure 195
audit trails 193–200

B

bar charts 33
BASE= option
  APPEND procedure 207
  COMPARE procedure 152
BETWEEN keyword, WHERE statement (PRINT) 107
blanks, trailing 10, 12
Boolean operators 53
box plots 33
BOXPLOT procedure 33
BRIEF option, COMPARE procedure 155, 158
BY statement 124, 137
BY variables
  ALL keyword 122–123
  detecting duplicates 125
  ID variables as 135–138
  NODUPKEY option, SORT procedure 120

C

CALL SYMPUT routine 48
CALL SYMPUTX routine 48
Cartesian product 166
CASCADE feature 203, 208–210
case conversions 9
$CHAR informat 109–110, 159
CHARACTER keyword 6, 94
character variables
  checking for invalid dates 110
  checking for invalid values with DATA step 7–13
  checking for invalid values with formats 15–18
  checking for invalid values with SQL procedure 166–168
  checking for missing values 170
  checking values with IF statement 13
  counting missing values 93–96
  listing invalid values with WHERE statement 13–15
  listing values with FREQ procedure 1–6
  removing invalid values with informats 18–22
Check integrity constraint 188
child data sets 202, 207–208
CIMPORT procedure 187
COMPARE= option, COMPARE procedure 152

COMPARE procedure 149
BASE= option 152
BRIEF option 155, 158
COMPARE= option 152
comparing data sets with selected variables 161–163
comparing data sets with unequal observations 159–160
comparing two data sets 150–159
ID statement 152, 159–160
LISTBASE option 159–160
LISTCOMP option 159–160
TRANSPOSE option 156
VAR statement 163
CONTENTS procedure 190–191, 205
converting lowercase to uppercase 9
COPY procedure 187
corrections
See error handling
COUNT function 173–174
counting missing values 93–100
CPORT procedure 187
CREATE clause, SQL procedure 166

D

DATA _NULL_ step
checking for out-of-range values 54–55
checking range based on interquartile range 88
detecting invalid character data 7–9
identifying subjects with *n* observations 131
listing highest/lowest ten values 47–49
MERGE statement 137
WHERE statement comparison 14
DATA= option, APPEND procedure 207
data sets
adding errors to 65
adding general integrity constraints to 189–191
child 202, 207–208
comparing with selected variables 161–163
comparing with unequal observations 159–160
comparing two 150–159
creating 125, 143–144
creating audit trails 193–200
integrity constraints and 187–191, 202
parent 202
DATA step
checking for invalid values 7–13
checking for out-of-range values 54–55
checking ranges for dates 106
counting missing values 96–100
detecting duplicates 123–126
identifying missing values 96–100
identifying subjects with *n* observations 130–132
IF statement 13, 106
integrity constraints 187
listing invalid values 15, 17
reading data in 182
SQL procedure alternative 165
DataFlux 213–216
DATASETS procedure 43, 65
AUDIT statement 195
audit trail data sets 195
IC CREATE statement 190–191
integrity constraints 188, 190–191, 193, 200–202, 211–212
MESSAGE= option 194
MSGTYPE=USER option 194
NOLIST option 43
DATE9. format 107
dates
checking for invalid 108–111
checking order of 147–148
checking ranges 106–107, 172
creating when day of month is missing 113–114
printing 105
reading 105
storing 105
suspending error checking for unknown 114–116

working with nonstandard forms 111–112
dfPower Studio 213–216
DISTINCT option, SELECT clause (SQL) 122–123
DO loops 101, 103
double entry and verification
data sets with selected variables 161–163
data sets with unequal observations 159–160
defined 149
two data sets 150–159
DOWNLOAD procedure 187
DROP= data set option 6
DROP statement 59, 199
duplicate ID numbers
checking with SQL procedure 173
detecting 123–129
eliminating 117–123
duplicate observations
detecting 123–126
eliminating 117
identifying subjects with 130–133
selecting patients with 129–130

E

EDA (exploratory data analysis) 86
error handling
audit trail data and 199
describing named input 182–184
hardcoding corrections 181–182
suspending for unknown dates 114–116
UPDATE statement and 184–186
error reports
listing invalid values 57–60
reading invalid dates 108–109
ERROR variable 56, 115–116
errors, adding to data sets 65
ERRORS= system option 109
%EVAL function 43, 46
exploratory data analysis (EDA) 86
extreme observations, listing 34–37

F

files, multiple
See multiple files
filtering invalid values with informats 68–70
FIRST. temporary variable 123–125, 130
FIRSTOBS= data set option 48
foreign keys
adding names to child data sets 207–208
deleting primary keys 205–206
referential constraints and 202–203, 208–212
FORMAT= option, TABULATE procedure 25
FORMAT procedure 18
invalid values with informats 69
INVALUE statement 18–19, 21, 69
formats
checking for invalid values 15–18, 66–68
printing dates 105
FREQ procedure
checking invalid values 1–6
counting missing values 93–96
detecting duplicates 126–129
identifying subjects with *n* observations 132–133
listing character variable values 1–6
listing invalid values 15–16
listing variable names 104
MISSING option 94
TABLES statement 4, 6, 16, 94, 126–127
FROM clause, SQL procedure 166, 171
FSEDIT procedure 186
FULL JOIN operation 174–177
fuzzy sorts 215

G

Gaussian distribution 34
general integrity constraints
adding to data sets 189–191

general integrity constraints (*continued*)
defined 187, 202
types of 188
GROUP BY clause, SQL procedure 166, 173
GROUPS= option, RANK procedure 44, 46, 73

H

hardcoding corrections 181–182
HAVING clause, SQL procedure 169, 173
high values
finding by percentage 37–47
listing highest ten 35–37, 47–52
UNIVARIATE procedure 32, 35–43
HISTOGRAM statement, UNIVARIATE procedure 33
horizontal bar charts 33

I

IC CREATE statement, DATASETS procedure 190–191
ID checking
in each of *n* files 138–143
in multiple files 135–138, 174–176
macro for 140–143
ID numbers
See duplicate ID numbers
ID statement
COMPARE procedure 152, 159–160
UNIVARIATE procedure 38
ID variables
as BY variable 135–138
checking with SQL procedure 175
IF statement
checking character variable values 13
checking date order 148
checking ranges for dates 106
IN= data set option 127, 135–138, 142
IN operator 8
INFILE statement 4
informats 18
?? modifier 114–116
checking for invalid dates 108–111
filtering invalid values 68–70
in INPUT function 18, 21–22
in INPUT statement 19
in INVALUE statement (FORMAT) 18–19
reading dates 105
removing invalid values 18–22
INITIATE option, AUDIT statement (DATASETS) 195
INPUT function
?? informat modifier 114–116
checking for invalid dates 110
checking for missing values 91
checking values of numeric variables 59
informats in 18, 21–22
PUT function comparison 18
INPUT statement
?? informat modifier 114–116
ERROR variable 56
informats in 19
integrity constraints
See also general integrity constraints
See also referential integrity constraints
adding user messages 194–195
audit trail data sets and 193–200
Check 188
creating 190
data sets and 187–191, 202
defined 187–188
deleting 193
demonstrating 189–190, 202–205
involving multiple variables 200–202
Not Null 188
Primary Key 188
reporting violations 197–198
types of 187–188
Unique 188
interquartile range 33, 86–88
invalid dates, checking for 108–111
invalid values
checking with DATA step 7–13
checking with formats 15–18, 66–68

checking with FREQ procedure 1–6
checking with SQL procedure 166–168
filtering with informats 68–70
identifying missing values versus 55–57
listing in error report 57–60
listing with DATA step 15, 17
listing with FREQ procedure 15–16
listing with PRINT procedure 13–15, 52–54
listing with WHERE statement 13–15
looking for outliers 24–34
removing with informats 18–22
setting to missing 19
INVALUE statement, FORMAT procedure
filtering invalid values with informats 69
informats in 18–19
UPCASE keyword 21
IS MISSING keyword 167, 170
IS NULL keyword 167

J

JOIN operations 174–179

K

KEEP= data set option 48, 64
KEYLABEL statement, TABULATE procedure 26
keypunch machine, verifier 149

L

LAG function 98
LAG2 function 98
LAST. temporary variable 123–125, 130
LEFT JOIN operation 177–178
LENGTH statement 183
%LET statement 64, 76
LISTBASE option, COMPARE procedure 159–160
LISTCOMP option, COMPARE procedure 159–160
log
?? modifier 115
inspecting missing values 91–93
reading invalid dates 108–109
low values
finding by percentage 37–47
listing lowest ten 35–37, 47–52
UNIVARIATE procedure 32, 35–43
lowercase, converting to uppercase 9
LRECL= option, INFILE statement 4

M

%MACRO statement 41
macro variables 41, 102
macros
automating range checking 60–62
checking range based on interquartile range 86–88
checking ranges for several variables 62–66
defined 41
detecting outliers based on trimmed statistics 76–80
ID checking 140–143
listing highest/lowest percentage 40–41, 44–47
listing highest/lowest values 50–52
listing outliers of several variables 82–86
named parameters 41
searching for specific numeric 102–104
selecting patients with duplicate observations 129–130
semi-colons and 43
MAX option, MEANS procedure 24
MAXDEC= option, MEANS procedure 24
MDY function 105, 111–114
MEAN summary function 169
MEANS procedure
checking range based on interquartile range 86, 88
counting missing values 93–96
detecting outliers based on 24–25, 71–76
MAX option 24
MAXDEC= option 24
MIN option 24

MEANS procedure (*continued*)
N option 24, 26, 94
NMISS option 24, 94
VAR statement 94
WHERE statement 74
%MEND statement 41
MERGE statement 137, 176
MERGENOBY ERROR system option 137
MERGENOBY NOWARN system option 137
MERGENOBY system option 137
MERGENOBY WARN system option 137
MESSAGE= option, DATASETS procedure 194
messages, and integrity constraints 194–195
MIN option, MEANS procedure 24
MISSING function 11, 96
MISSING option
FREQ procedure 94
TABLES statement (FREQ) 16, 94
missing values
checking with INPUT function 91
checking with SQL procedure 170–171
counting 93–100
identifying invalid values versus 55–57
inspecting SAS log 91–93
named input method and 183
removing from listings 110–111
searching for specific numeric 100–104
setting invalid values to 19
MMDDYY10. format 107–109
MONYY informat 113
MPRINT system option 42
MSGTYPE=USER option, DATASETS procedure 194
multiple files
checking date order 147–148
checking IDs in 135–138, 174–176
checking IDs in each of "n" 138–143
complicated rules 143–147, 176–179

N

$n. informat 109
N option
KEYLABEL statement (TABULATE) 26
MEANS procedure 24, 26, 94
named input method 182–184
named parameters 41
names
adding to child data sets 207–208
obtaining for output objects 34
NEXTROBS= option, UNIVARIATE procedure 35–37
NEXTRVALS= option, UNIVARIATE procedure 35–37
NMISS option
KEYLABEL statement (TABULATE) 26
MEANS procedure 24, 94
NOBOS= option, SET statement 48
NOCUM option, TABLES statement (FREQ) 4
NODUPKEY option, SORT procedure 118–120, 137
NODUPRECS option, SORT procedure 118, 120–123
NOLIST option, DATASETS procedure 43
NOPERCENT option, TABLES statement (FREQ) 4
NOPRINT option, UNIVARIATE procedure 38, 81
normal distribution 34
normal probability plots 34
Not Null integrity constraint 188
NOT operator 8
NOTDIGIT function 12–13, 59
identifying missing values 98
NULL reserved data set name 7
NUMERIC keyword 101
numeric macros, searching for specific 102–104
numeric missing values, searching for specific 100–104
numeric variables
checking for missing values 170

checking for out-of-range values 54–55
checking ranges based on interquartile range 86–88
checking ranges with algorithm 71–72
checking values with INPUT function 59
computing trimmed statistics 80–86
counting missing values 93–96
creating range checking macro 60–62
detecting outliers based on standard deviation 73–76
detecting outliers based on trimmed statistics 73–80
filtering invalid values with informats 68–70
finding highest/lowest values by percentage 37–47
formats to check invalid values 66–68
identifying invalid versus missing values 55–57
listing extreme values 34–37
listing highest/lowest ten values 47–52
listing invalid values 52–54, 57–60
looking for outliers 24–34
range checking for multiple variables 62–66
searching for specific 100–102

O

observations
See also duplicate observations
comparing data sets with unequal observations 159–160
listing extreme observations 34–35
ODS (Output Delivery System) 80–86
ODS LISTING statement 81
ODS OUTPUT statement 81–82
ODS SELECT statement 34–35
operators 8, 53
OR operator 53
ORDER BY clause, SQL procedure 166
OTHER keyword 19
out-of-range values
checking for 54–55, 66–68
listing 52–54
OUT= option
OUTPUT statement (UNIVARIATE) 38
SORT procedure 118
TABLES statement (FREQ) 126–127
outliers
box plot example 33
checking with SQL procedure 168
detecting based on standard deviation 71–76
detecting based on trimmed mean 73–76
detecting based on trimmed statistics 76–80
listing outliers of several variables 82–86
looking for in numeric variables 24–34
Output Delivery System (ODS) 80–86
OUTPUT destination 81
output devices 8, 166
output objects, obtaining names 34
OUTPUT statement, UNIVARIATE procedure 38

P

parameters, named 41
parent data sets 202
patients, selecting with duplicate observations 129–130
PATIENTS.TXT raw data file 2–6
PCTLPRE= option, OUTPUT statement (UNIVARIATE) 38
PCTLPTS= option, OUTPUT statement (UNIVARIATE) 38
PDV (Program Data Vector) 56
percentage, finding values by 37–47
PLOT option, UNIVARIATE procedure 26
primary key
deleting when foreign key exists 205–206
referential constraints and 202, 208–212
Primary Key integrity constraint 188
PRINT procedure
checking ranges for dates 107

PRINT procedure (*continued*)
listing invalid values 13–15, 52–54
viewing audit trail data 193, 195–198
WHERE statement 13–15, 52–54, 98, 107, 130
printing dates 105
probability plots 34
Program Data Vector (PDV) 56
PUT function 18, 67
PUT statement
checking ranges for dates 106
formats checking for invalid values 67
identifying missing values 96
sending results to output device 8

Q

question mark (?) 114–116
QUOTE function 129

R

range checking
automating 60–62
based on interquartile range 86–88
checking for out-of-range values 54–55, 66–68
for dates 106–107, 172
for multiple variables 62–66
listing out-of-range values 52–54
with algorithm based on standard deviation 71–72, 169–170
RANK procedure
GROUPS= option 44, 46, 73
highest/lowest values by percentage 37, 43–47
RANKS statement 44, 46
VAR statement 44
RANKS statement, RANK procedure 44, 46
reading data, with DATA step 182
referential integrity constraints
adding names to child data sets 207–208
CASCADE feature 203, 208–210
defined 187–188, 202
deleting 211–212
deleting primary key when foreign key exists 205–206
demonstrating 202–205
primary key and 202, 208–212
RESTRICT feature 202
SET NULL feature 202, 210–211
REPORT procedure 193, 197–198
RESTRICT feature 202
RIGHT JOIN operation 177
RTSPACE= option, TABLE statement (TABULATE) 25

S

SAME keyword 19, 21
SAS Component Language (SCL) 188
SAS dates
See dates
SAS log
See log
SCAN function 142
%SCAN function 142
SCL (SAS Component Language) 188
SELECT clause, SQL procedure 166
asterisk (*) in 171
DISTINCT option 122–123
QUOTE function 129
semi-colon (;) 43
SET NULL feature 202, 210–211
SET statement
adding names to child data sets 207
detecting duplicates 124
example 39, 42
executing once 72
NOBS= option 48
SORT procedure
eliminating duplicates 117–123
NODUPKEY option 118–120, 137
NODUPRECS option 118, 120–123
OUT= option 118
%SCAN function and 142
sorts, fuzzy 215
SQL procedure 166
as DATA step alternative 165

checking for duplicates 173
checking for IDs in multiple files 174–176
checking for invalid character values 166–168
checking for missing values 170–171
checking for outliers 168
checking ranges based on standard deviation 169–170
CREATE clause 166
FROM clause 166, 171
GROUP BY clause 166, 173
HAVING clause 169, 173
identifying subjects with *n* observations 174
integrity constraints 187–188
JOIN operations 174–179
multi-file rules 176–179
ORDER BY clause 166
ordering clauses 166
removing duplicate records 122–123
SELECT clause 122–123, 129, 166, 171
selecting patients with duplicate observations 129–130
WHERE clause 166–167, 170, 176, 191
standard deviation
checking ranges 71–72, 169–170
computing from standard error 82
detecting outliers based on 71–76
standard error 82
STD summary function 169
stem-and-leaf plots 33
subjects, identifying with *n* observations 130–133
SUM statement 131
SUSPEND option, AUDIT statement (DATASETS) 195
SYMPUT CALL routine 48
SYMPUTX CALL routine 48

T

TABLE statement, TABULATE procedure 25
TABLES statement, FREQ procedure
CHARACTER keyword 6, 94
listing unique values 4
MISSING option 16, 94
NOCUM option 4
NOPERCENT option 4
OUT= option 126–127
TABULATE procedure
FORMAT= option 25
KEYLABEL statement 26
looking for outliers 25–26
TABLE statement 25
VAR statement 25
temporary variables 123–125, 130
TERMINATE option, AUDIT statement (DATASETS) 195
trailing blanks, removing 10, 12
TRANSPOSE option, COMPARE procedure 156
TRIM function 10, 12
identifying missing values 98
TRIM= option, UNIVARIATE procedure 80–82
trimmed statistics
computing 72–76, 80–86
detecting outliers based on 73–80
macro example 76–80
TRUNCOVER option, INFILE statement 4
TYPE= data set option 193, 196
TYPE=AUDIT data set option 193

U

Unique integrity constraint 188
unique values 94
UNIVARIATE procedure
highest/lowest values by percentage 32, 35–43
HISTOGRAM statement 33
ID statement 38
listing extreme values 35–37
looking for outliers 24, 26–33
NEXTROBS= option 35–37
NEXTRVALS= option 35–37

UNIVARIATE procedure (*continued*)
NOPRINT option 38, 81
ODS statement support 34
OUTPUT statement 38
PLOT option 26
TRIM= option 80–82
unknown dates, checking for 114–116
UPCASE function 5, 9
$UPCASE informat 9
UPCASE keyword 21
UPDATE statement 184–186
UPLOAD procedure 187
uppercase, converting lowercase to 9
user messages, and integrity constraints 194–195

V

VAR statement
COMPARE procedure 163
MEANS procedure 94
RANK procedure 44
TABULATE procedure 25
variables
See also character variables
automatic 196–197, 198
BY variables 120, 122–123, 125, 135–138
comparing data sets with selected variables 161–163
ERROR 56, 115–116
ID variables 135–138, 175
integrity constraints and multiple variables 200–202
listing variable names 104
macro variables 41, 102
range checking for multiple 62–66
temporary 123–125, 130
verifier keypunch machine 149
VERIFY function 9–13
VNAME function 100–103

W

WHERE clause, SQL procedure
checking for invalid character values 166–167
checking for missing values 170
integrity constraints and 191
multi-file rules 176
WHERE= data set option
checking values of numeric variables 46, 48
detecting duplicates 127
multiple files 146
WHERE= option, IC CREATE statement (DATASETS) 190–191
WHERE statement
listing invalid values 13–15
MEANS procedure 74
PRINT procedure 13–15, 52–54, 98, 107, 130
whiskers 33

Symbols

& (ampersand) 41, 102
* (asterisk) 171
?? informat modifier 114–116
; (semi-colon) 43

Books Available from SAS Press

Advanced Log-Linear Models Using SAS®
by **Daniel Zelterman**

Analysis of Clinical Trials Using SAS®: A Practical Guide
by **Alex Dmitrienko, Geert Molenberghs, Walter Offen,** *and* **Christy Chuang-Stein**

Analyzing Receiver Operating Characteristic Curves with SAS®
by **Mithat Gönen**

Annotate: Simply the Basics
by **Art Carpenter**

Applied Multivariate Statistics with SAS® Software, Second Edition
by **Ravindra Khattree**
and **Dayanand N. Naik**

Applied Statistics and the SAS® Programming Language, Fifth Edition
by **Ronald P. Cody**
and **Jeffrey K. Smith**

An Array of Challenges — Test Your SAS® Skills
by **Robert Virgile**

Basic Statistics Using SAS® Enterprise Guide®: A Primer
by **Geoff Der**
and **Brian S. Everitt**

Building Web Applications with SAS/IntrNet®: A Guide to the Application Dispatcher
by **Don Henderson**

Carpenter's Complete Guide to the SAS® Macro Language, Second Edition
by **Art Carpenter**

Carpenter's Complete Guide to the SAS® REPORT Procedure
by **Art Carpenter**

The Cartoon Guide to Statistics
by **Larry Gonick**
and **Woollcott Smith**

Categorical Data Analysis Using the SAS® System, Second Edition
by **Maura E. Stokes, Charles S. Davis,**
and **Gary G. Koch**

Cody's Data Cleaning Techniques Using SAS® Software
by **Ron Cody**

Common Statistical Methods for Clinical Research with SAS® Examples, Second Edition
by **Glenn A. Walker**

The Complete Guide to SAS® Indexes
by **Michael A. Raithel**

CRM Segmemtation and Clustering Using SAS® Enterprise Miner™
by **Randall S. Collica**

Data Management and Reporting Made Easy with SAS® Learning Edition 2.0
by **Sunil K. Gupta**

Data Preparation for Analytics Using SAS®
by **Gerhard Svolba**

Debugging SAS® Programs: A Handbook of Tools and Techniques
by **Michele M. Burlew**

Decision Trees for Business Intelligence and Data Mining: Using SAS® Enterprise Miner™
by **Barry de Ville**

Efficiency: Improving the Performance of Your SAS® Applications
by **Robert Virgile**

The Essential Guide to SAS® Dates and Times
by **Derek P. Morgan**

Fixed Effects Regression Methods for Longitudinal Data Using SAS®
by **Paul D. Allison**

Genetic Analysis of Complex Traits Using SAS®
by **Arnold M. Saxton**

The Global English Style Guide: Writing Clear, Translatable Documentation for a Global Market
by **John R. Kohl**

A Handbook of Statistical Analyses Using SAS®, Second Edition
by **B.S. Everitt**
and **G. Der**

Health Care Data and SAS®
by **Marge Scerbo, Craig Dickstein,**
and **Alan Wilson**

The How-To Book for SAS/GRAPH® Software
by **Thomas Miron**

In the Know... SAS® Tips and Techniques From Around the Globe, Second Edition
by **Phil Mason**

Instant ODS: Style Templates for the Output Delivery System
by **Bernadette Johnson**

Integrating Results through Meta-Analytic Review Using SAS® Software
by **Morgan C. Wang**
and **Brad J. Bushman**

Introduction to Data Mining Using SAS® Enterprise Miner™
by **Patricia B. Cerrito**

Introduction to Design of Experiments with JMP® Examples, Third Edition
by **Jacques Goupy**
and **Lee Creighton**

Learning SAS® by Example: A Programmer's Guide
by **Ron Cody**

The Little SAS® Book: A Primer
by **Lora D. Delwiche**
and **Susan J. Slaughter**

The Little SAS® Book: A Primer, Second Edition
by **Lora D. Delwiche**
and **Susan J. Slaughter**
(updated to include SAS 7 features)

The Little SAS® Book: A Primer, Third Edition
by **Lora D. Delwiche**
and **Susan J. Slaughter**
(updated to include SAS 9.1 features)

The Little SAS® Book for Enterprise Guide® 3.0
by **Susan J. Slaughter**
and **Lora D. Delwiche**

The Little SAS® Book for Enterprise Guide® 4.1
by **Susan J. Slaughter**
and **Lora D. Delwiche**

Logistic Regression Using the SAS® System: Theory and Application
by **Paul D. Allison**

Longitudinal Data and SAS®: A Programmer's Guide
by **Ron Cody**

Maps Made Easy Using SAS®
by **Mike Zdeb**

Measurement, Analysis, and Control Using JMP®: Quality Techniques for Manufacturing
by **Jack E. Reece**

Multiple Comparisons and Multiple Tests Using SAS® Text and Workbook Set
(books in this set also sold separately)
by **Peter H. Westfall, Randall D. Tobias, Dror Rom, Russell D. Wolfinger,** *and* **Yosef Hochberg**

Multiple-Plot Displays: Simplified with Macros
by **Perry Watts**

Multivariate Data Reduction and Discrimination with SAS® Software
by **Ravindra Khattree** *and* **Dayanand N. Naik**

Output Delivery System: The Basics
by **Lauren E. Haworth**

Painless Windows: A Handbook for SAS® Users, Third Edition
by **Jodie Gilmore**
(updated to include SAS 8 and SAS 9.1 features)

Pharmaceutical Statistics Using SAS®: A Practical Guide
Edited by **Alex Dmitrienko, Christy Chuang-Stein,** *and* **Ralph D'Agostino**

The Power of PROC FORMAT
by **Jonas V. Bilenas**

Predictive Modeling with SAS® Enterprise Miner™: Practical Solutions for Business Applications
by **Kattamuri S. Sarma**

PROC SQL: Beyond the Basics Using SAS®
by **Kirk Paul Lafler**

PROC TABULATE by Example
by **Lauren E. Haworth**

Professional SAS® Programmer's Pocket Reference, Fifth Edition
by **Rick Aster**

Professional SAS® Programming Shortcuts, Second Edition
by **Rick Aster**

Quick Results with SAS/GRAPH® Software
by **Arthur L. Carpenter** *and* **Charles E. Shipp**

Quick Results with the Output Delivery System
by **Sunil Gupta**

Reading External Data Files Using SAS®: Examples Handbook
by **Michele M. Burlew**

Regression and ANOVA: An Integrated Approach Using SAS® Software
by **Keith E. Muller** *and* **Bethel A. Fetterman**

SAS® For Dummies®
by **Stephen McDaniel** *and* **Chris Hemedinger**

SAS® for Forecasting Time Series, Second Edition
by **John C. Brocklebank** *and* **David A. Dickey**

SAS® for Linear Models, Fourth Edition
by **Ramon C. Littell, Walter W. Stroup,** *and* **Rudolf Freund**

SAS® for Mixed Models, Second Edition
by **Ramon C. Littell, George A. Milliken, Walter W. Stroup, Russell D. Wolfinger,** *and* **Oliver Schabenberger**

SAS® for Monte Carlo Studies: A Guide for Quantitative Researchers
by **Xitao Fan, Ákos Felsővályi, Stephen A. Sivo,** *and* **Sean C. Keenan**

SAS® Functions by Example
by **Ron Cody**

SAS® Graphics for Java: Examples Using SAS® AppDev Studio™ and the Output Delivery System
by **Wendy Bohnenkamp** *and* **Jackie Iverson**

SAS® Guide to Report Writing, Second Edition
by **Michele M. Burlew**

SAS® Macro Programming Made Easy, Second Edition
by **Michele M. Burlew**

SAS® Programming by Example
by **Ron Cody**
and **Ray Pass**

SAS® Programming for Enterprise Guide® Users
by **Neil Constable**

SAS® Programming in the Pharmaceutical Industry
by **Jack Shostak**

SAS® Survival Analysis Techniques for Medical Research, Second Edition
by **Alan B. Cantor**

SAS® System for Elementary Statistical Analysis, Second Edition
by **Sandra D. Schlotzhauer**
and **Ramon C. Littell**

SAS® System for Regression, Third Edition
by **Rudolf J. Freund**
and **Ramon C. Littell**

SAS® System for Statistical Graphics, First Edition
by **Michael Friendly**

The SAS® Workbook and *Solutions* Set
(books in this set also sold separately)
by **Ron Cody**

Saving Time and Money Using SAS®
by **Philip R. Holland**

Selecting Statistical Techniques for Social Science Data: A Guide for SAS® Users
by **Frank M. Andrews, Laura Klem, Patrick M. O'Malley, Willard L. Rodgers, Kathleen B. Welch,**
and **Terrence N. Davidson**

Statistics Using SAS® Enterprise Guide®
by **James B. Davis**

A Step-by-Step Approach to Using the SAS® System for Factor Analysis and Structural Equation Modeling
by **Larry Hatcher**

A Step-by-Step Approach to Using SAS® for Univariate and Multivariate Statistics, Second Edition
by **Norm O'Rourke, Larry Hatcher,**
and **Edward J. Stepanski**

Step-by-Step Basic Statistics Using SAS®: Student Guide and *Exercises*
(books in this set also sold separately)
by **Larry Hatcher**

Survival Analysis Using SAS®: A Practical Guide
by **Paul D. Allison**

Tuning SAS® Applications in the OS/390 and z/OS Environments, Second Edition
by **Michael A. Raithel**

Using SAS® in Financial Research
by **Ekkehart Boehmer, John Paul Broussard,**
and **Juha-Pekka Kallunki**

Validating Clinical Trial Data Reporting with SAS®
by **Carol I. Matthews**
and **Brian C. Shilling**

Visualizing Categorical Data
by **Michael Friendly**

Web Development with SAS® by Example, Second Edition
by **Frederick E. Pratter**

JMP® Books

Elementary Statistics Using JMP®
by **Sandra D. Schlotzhauer**

JMP® for Basic Univariate and Multivariate Statistics: A Step-by-Step Guide
by **Ann Lehman, Norm O'Rourke, Larry Hatcher,** *and* **Edward J. Stepanski**

JMP® Start Statistics: A Guide to Statistics and Data Analysis Using JMP®, Fourth Edition
by **John Sall, Lee Creighton,** *and* **Ann Lehman**

Regression Using JMP®
by **Rudolf J. Freund, Ramon C. Littell,** *and* **Lee Creighton**

support.sas.com/publishing