

Carpenter's Guide to Innovative SAS[®] Techniques



Art Carpenter

Contents

About This Book xvii

Acknowledgments xxv

About the Author xxvii

Part 1 Data Preparation 1

Chapter 1 Moving, Copying, Importing, and Exporting Data 3

1.1 LIBNAME Statement Engines 4

1.1.1 Using Data Access Engines to Read and Write Data 5

1.1.2 Using the Engine to View the Data 6

1.1.3 Options Associated with the Engine 6

1.1.4 Replacing EXCEL Sheets 7

1.1.5 Recovering the Names of EXCEL Sheets 8

1.2 PROC IMPORT and EXPORT 9

1.2.1 Using the Wizard to Build Sample Code 9

1.2.2 Control through the Use of Options 9

1.2.3 PROC IMPORT Data Source Statements 10

1.2.4 Importing and Exporting CSV Files 12

1.2.5 Preventing the Export of Blank Sheets 15

1.2.6 Working with Named Ranges 16

1.3 DATA Step INPUT Statement 17

1.3.1 Format Modifiers for Errors 18

1.3.2 Format Modifiers for the INPUT Statement 18

1.3.3 Controlling Delimited Input 20

1.3.4 Reading Variable-Length Records 24

1.4 Writing Delimited Files 28

1.4.1 Using the DATA Step with the DLM= Option 28

1.4.2 PROC EXPORT 29

1.4.3 Using the %DS2CSV Macro 30

1.4.4 Using ODS and the CSV Destination 31

1.4.5 Inserting the Separator Manually 31

1.5 SQL Pass-Through 32

1.5.1 Adding a Pass-Through to Your SQL Step 32

1.5.2 Pass-Through Efficiencies 33

1.6 Reading and Writing to XML 33

1.6.1 Using ODS 34

1.6.2 Using the XML Engine 34

Chapter 2 Working with Your Data 37

- 2.1 Data Set Options 38**
 - 2.1.1 REPLACE and REPEMPTY 40
 - 2.1.2 Password Protection 41
 - 2.1.3 KEEP, DROP, and RENAME Options 42
 - 2.1.4 Observation Control Using FIRSTOBS and OBS Data Set Options 43
- 2.2 Evaluating Expressions 45**
 - 2.2.1 Operator Hierarchy 45
 - 2.2.2 Using the Colon as a Comparison Modifier 46
 - 2.2.3 Logical and Comparison Operators in Assignment Statements 47
 - 2.2.4 Compound Inequalities 49
 - 2.2.5 The MIN and MAX Operators 50
 - 2.2.6 Numeric Expressions and Boolean Transformations 51
- 2.3 Data Validation and Exception Reporting 52**
 - 2.3.1 Date Validation 52
 - 2.3.2 Writing to an Error Data Set 55
 - 2.3.3 Controlling Exception Reporting with Macros 58
- 2.4 Normalizing - Transposing the Data 60**
 - 2.4.1 Using PROC TRANSPOSE 61
 - 2.4.2 Transposing in the DATA Step 63
- 2.5 Filling Sparse Data 65**
 - 2.5.1 Known Template of Rows 65
 - 2.5.2 Double Transpose 67
 - 2.5.3 Using COMPLETYPES with PROC MEANS or PROC SUMMARY 70
 - 2.5.4 Using CLASSDATA 70
 - 2.5.5 Using Preloaded Formats 72
 - 2.5.6 Using the SPARSE Option with PROC FREQ 73
- 2.6 Some General Concepts 73**
 - 2.6.1 Shorthand Variable Naming 73
 - 2.6.2 Understanding the ORDER= Option 77
 - 2.6.3 Quotes within Quotes within Quotes 79
 - 2.6.4 Setting the Length of Numeric Variables 81
- 2.7 WHERE Specifics 82**
 - 2.7.1 Operators Just for the WHERE 83
 - 2.7.2 Interaction with the BY Statement 86
- 2.8 Appending Data Sets 88**
 - 2.8.1 Appending Data Sets Using the DATA Step and SQL UNION 88
 - 2.8.2 Using the DATASETS Procedure's APPEND Statement 90

2.9 Finding and Eliminating Duplicates 90

- 2.9.1 Using PROC SORT 91
- 2.9.2 Using FIRST. and LAST. BY-Group Processing 92
- 2.9.3 Using PROC SQL 93
- 2.9.4 Using PROC FREQ 93
- 2.9.5 Using the Data Component Hash Object 94

2.10 Working with Missing Values 97

- 2.10.1 Special Missing Values 97
- 2.10.2 MISSING System Option 98
- 2.10.3 Using the CMISS, NMISS, and MISSING Functions 99
- 2.10.4 Using the CALL MISSING Routine 100
- 2.10.5 When Classification Variables are Missing 100
- 2.10.6 Missing Values and Macro Variables 101
- 2.10.7 Imputing Missing Values 101

Chapter 3 Just In the DATA Step 103**3.1 Working across Observations 105**

- 3.1.1 BY-Group Processing—Using FIRST. and LAST. Processing 105
- 3.1.2 Transposing to ARRAYs 107
- 3.1.3 Using the LAG Function 108
- 3.1.4 Look-Ahead Using a MERGE Statement 110
- 3.1.5 Look-Ahead Using a Double SET Statement 111
- 3.1.6 Look-Back Using a Double SET Statement 111
- 3.1.7 Building a FIFO Stack 113
- 3.1.8 A Bit on the SUM Statement 114

3.2 Calculating a Person's Age 114

- 3.2.1 Simple Formula 115
- 3.2.2 Using Functions 116
- 3.2.3 The Way Society Measures Age 117

3.3 Using DATA Step Component Objects 117

- 3.3.1 Declaring (Instantiating) the Object 119
- 3.3.2 Using Methods with an Object 119
- 3.3.3 Simple Sort Using the HASH Object 120
- 3.3.4 Stepping through a Hash Table 121
- 3.3.5 Breaking Up a Data Set into Multiple Data Sets 126
- 3.3.6 Hash Tables That Reference Hash Tables 128
- 3.3.7 Using a Hash Table to Update a Master Data Set 130

3.4 Doing More with the INTNX and INTCK Functions 132

- 3.4.1 Interval Multipliers 132
- 3.4.2 Shift Operators 133
- 3.4.3 Alignment Options 134
- 3.4.4 Automatic Dates 136

- 3.5 Variable Conversions 138**
 - 3.5.1 Using the PUT and INPUT Functions 138
 - 3.5.2 Decimal, Hexadecimal, and Binary Number Conversions 143
- 3.6 DATA Step Functions 143**
 - 3.6.1 The ANY and NOT Families of Functions 144
 - 3.6.2 Comparison Functions 145
 - 3.6.3 Concatenation Functions 147
 - 3.6.4 Finding Maximum and Minimum Values 147
 - 3.6.5 Variable Information Functions 148
 - 3.6.6 New Alternatives and Functions That Do More 154
 - 3.6.7 Functions That Put the Squeeze on Values 163
- 3.7 Joins and Merges 165**
 - 3.7.1 BY Variable Attribute Consistency 166
 - 3.7.2 Variables in Common That Are Not in the BY List 169
 - 3.7.3 Repeating BY Variables 170
 - 3.7.4 Merging without a Clear Key (Fuzzy Merge) 171
- 3.8 More on the SET Statement 172**
 - 3.8.1 Using the NOBS= and POINT= Options 172
 - 3.8.2 Using the INDSNAME= Option 174
 - 3.8.3 A Comment on the END= Option 175
 - 3.8.4 DATA Steps with Two SET Statements 175
- 3.9 Doing More with DO Loops 176**
 - 3.9.1 Using the DOW Loop 176
 - 3.9.2 Compound Loop Specifications 178
 - 3.9.3 Special Forms of Loop Specifications 178
- 3.10 More on Arrays 180**
 - 3.10.1 Array Syntax 180
 - 3.10.2 Temporary Arrays 181
 - 3.10.3 Functions Used with Arrays 182
 - 3.10.4 Implicit Arrays 183

Chapter 4 Sorting the Data 185

- 4.1 PROC SORT Options 186**
 - 4.1.1 The NODUPREC Option 186
 - 4.1.2 The DUPOUT= Option 187
 - 4.1.3 The TAGSORT Option 188
 - 4.1.4 Using the SORTSEQ Option 188
 - 4.1.5 The FORCE Option 190
 - 4.1.6 The EQUALS or NOEQUALS Options 190
- 4.2 Using Data Set Options with PROC SORT 190**
- 4.3 Taking Advantage of Known or Knowable Sort Order 191**

4.4 Metadata Sort Information 193**4.5 Using Threads 194****Chapter 5 Working with Data Sets 197****5.1 Automating the COMPARE Process 198****5.2 Reordering Variables on the PDV 200****5.3 Building and Maintaining Indexes 202**

5.3.1 Introduction to Indexing 203

5.3.2 Creating Simple Indexes 204

5.3.3 Creating Composite Indexes 206

5.3.4 Using the IDXWHERE and IDXNAME Options 206

5.3.5 Index Caveats and Considerations 207

5.4 Protecting Passwords 208

5.4.1 Using PROC PWENCODE 208

5.4.2 Protecting Database Passwords 209

5.5 Deleting Data Sets 211**5.6 Renaming Data Sets 211**

5.6.1 Using the RENAME Function 212

5.6.2 Using PROC DATASETS 212

Chapter 6 Table Lookup Techniques 213**6.1 A Series of IF Statements—The Logical Lookup 215****6.2 IF -THEN/ELSE Lookup Statements 215****6.3 DATA Step Merges and SQL Joins 216****6.4 Merge Using Double SET Statements 218****6.5 Using Formats 219****6.6 Using Indexes 221**

6.6.1 Using the BY Statement 222

6.6.2 Using the KEY= Option 222

6.7 Key Indexing (Direct Addressing)—Using Arrays to Form a Simple Hash 223

6.7.1 Building a List of Unique Values 223

6.7.2 Performing a Key Index Lookup 224

6.7.3 Using a Non-Numeric Index 226

6.8 Using the HASH Object 227

Part 2 Data Summary, Analysis, and Reporting 231

Chapter 7 MEANS and SUMMARY Procedures 233

- 7.1 Using Multiple CLASS Statements and CLASS Statement Options 234**
 - 7.1.1 MISSING and DESCENDING Options 236
 - 7.1.2 GROUPINTERNAL Option 237
 - 7.1.3 Order= Option 238
- 7.2 Letting SAS Name the Output Variables 238**
- 7.3 Statistic Specification on the OUTPUT Statement 240**
- 7.4 Identifying the Extremes 241**
 - 7.4.1 Using the MAXID and MINID Options 241
 - 7.4.2 Using the IDGROUP Option 243
 - 7.4.3 Using Percentiles to Create Subsets 245
- 7.5 Understanding the _TYPE_ Variable 246**
- 7.6 Using the CHARTYPE Option 248**
- 7.7 Controlling Summary Subsets Using the WAYS Statement 249**
- 7.8 Controlling Summary Subsets Using the TYPES Statement 250**
- 7.9 Controlling Subsets Using the CLASSDATA= and EXCLUSIVE Options 251**
- 7.10 Using the COMPLETETYPES Option 253**
- 7.11 Identifying Summary Subsets Using the LEVELS and WAYS Options 254**
- 7.12 CLASS Statement vs. BY Statement 255**

Chapter 8 Other Reporting and Analysis Procedures 257

- 8.1 Expanding PROC TABULATE 258**
 - 8.1.1 What You Need to Know to Get Started 258
 - 8.1.2 Calculating Percentages Using PROC TABULATE 262
 - 8.1.3 Using the STYLE= Option with PROC TABULATE 265
 - 8.1.4 Controlling Table Content with the CLASSDATA Option 267
 - 8.1.5 Ordering Classification Level Headings 269
- 8.2 Expanding PROC UNIVARIATE 270**
 - 8.2.1 Generating Presentation-Quality Plots 270
 - 8.2.2 Using the CLASS Statement 273
 - 8.2.3 Probability and Quantile Plots 275
 - 8.2.4 Using the OUTPUT Statement to Calculate Percentages 276
- 8.3 Doing More with PROC FREQ 277**
 - 8.3.1 OUTPUT Statement in PROC FREQ 277
 - 8.3.2 Using the NLEVELS Option 279

- 8.4 Using PROC REPORT to Better Advantage 280**
 - 8.4.1 PROC REPORT vs. PROC TABULATE 280
 - 8.4.2 Naming Report Items (Variables) in the Compute Block 280
 - 8.4.3 Understanding Compute Block Execution 281
 - 8.4.4 Using a Dummy Column to Consolidate Compute Blocks 283
 - 8.4.5 Consolidating Columns 284
 - 8.4.6 Using the STYLE= Option with LINES 285
 - 8.4.7 Setting Style Attributes with the CALL DEFINE Routine 287
 - 8.4.8 Dates within Dates 288
 - 8.4.9 Aligning Decimal Points 289
 - 8.4.10 Conditionally Executing the LINE Statement 290
- 8.5 Using PROC PRINT 291**
 - 8.5.1 Using the ID and BY Statements Together 291
 - 8.5.2 Using the STYLE= Option with PROC PRINT 292
 - 8.5.3 Using PROC PRINT to Generate a Table of Contents 295

Chapter 9 SAS/GRAPH Elements You Should Know—Even if You Don’t Use SAS/GRAPH 297

- 9.1 Using Title Options with ODS 298**
- 9.2 Setting and Clearing Graphics Options and Settings 300**
- 9.3 Using SAS/GRAPH Statements with Procedures That Are Not SAS/GRAPH Procedures 303**
 - 9.3.1 Changing Plot Symbols with the SYMBOL Statement 303
 - 9.3.2 Controlling Axes and Legends 306
- 9.4 Using ANNOTATE to Augment Graphs 309**

Chapter 10 Presentation Graphics—More than Just SAS/GRAPH 313

- 10.1 Generating Box Plots 314**
 - 10.1.1 Using PROC BOXPLOT 314
 - 10.1.2 Using PROC GPLOT and the SYMBOL Statement 315
 - 10.1.3 Using PROC SHEWHART 316
- 10.2 SAS/GRAPH Specialty Techniques and Procedures 317**
 - 10.2.1 Building Your Own Graphics Font 317
 - 10.2.2 Splitting a Text Line Using JUSTIFY= 319
 - 10.2.3 Using Windows Fonts 319
 - 10.2.4 Using PROC GKPI 320
- 10.3 PROC FREQ Graphics 323**

Chapter 11 Output Delivery System 325

11.1 Using the OUTPUT Destination 326

- 11.1.1 Determining Object Names 326
- 11.1.2 Creating a Data Set 327
- 11.1.3 Using the MATCH_ALL Option 330
- 11.1.4 Using the PERSIST= Option 330
- 11.1.5 Using MATCH_ALL= with the PERSIST= Option 331

11.2 Writing Reports to Excel 332

- 11.2.1 EXCELXP Tagset Documentation and Options 333
- 11.2.2 Generating Multisheet Workbooks 334
- 11.2.3 Checking Out the Styles 335

11.3 Inline Formatting Using Escape Character Sequences 337

- 11.3.1 Page X of Y 338
- 11.3.2 Superscripts, Subscripts, and a Dagger 340
- 11.3.3 Changing Attributes 341
- 11.3.4 Using Sequence Codes to Control Indentations, Spacing, and Line Breaks 342
- 11.3.5 Issuing Raw RTF Specific Commands 344

11.4 Creating Hyperlinks 345

- 11.4.1 Using Style Overrides to Create Links 345
- 11.4.2 Using the LINK= TITLE Statement Option 347
- 11.4.3 Linking Graphics Elements 348
- 11.4.4 Creating Internal Links 350

11.5 Traffic Lighting 352

- 11.5.1 User-Defined Format 352
- 11.5.2 PROC TABULATE 353
- 11.5.3 PROC REPORT 354
- 11.5.4 Traffic Lighting with PROC PRINT 355

11.6 The ODS LAYOUT Statement 356

11.7 A Few Other Useful ODS Tidbits 358

- 11.7.1 Using the ASIS Style Attribute 358
- 11.7.2 ODS RESULTS Statement 358

Part 3 Techniques, Tools, and Interfaces 361

Chapter 12 Taking Advantage of Formats 363

12.1 Using Preloaded Formats to Modify Report Contents 364

- 12.1.1 Using Preloaded Formats with PROC REPORT 365
- 12.1.2 Using Preloaded Formats with PROC TABULATE 367
- 12.1.3 Using Preloaded Formats with the MEANS and SUMMARY Procedures 369

12.2	Doing More with Picture Formats	370
12.2.1	Date Directives and the DATATYPE Option	371
12.2.2	Working with Fractional Values	373
12.2.3	Using the MULT and PREFIX Options	374
12.2.4	Display Granularity Based on Value Ranges – Limiting Significant Digits	376
12.3	Multilabel (MLF) Formats	377
12.3.1	A Simple MLF	377
12.3.2	Calculating Rolling Averages	378
12.4	Controlling Order Using the NOTSORTED Option	381
12.5	Extending the Use of Format Translations	382
12.5.1	Filtering Missing Values	382
12.5.2	Mapping Overlapping Ranges	383
12.5.3	Handling Text within Numeric Values	383
12.5.4	Using Perl Regular Expressions within Format Definitions	384
12.5.5	Passing Values to a Function as a Format Label	384
12.6	ANYDATE Informats	388
12.6.1	Reading in Mixed Dates	389
12.6.2	Converting Mixed DATETIME Values	389
12.7	Building Formats from Data Sets	390
12.8	Using the PVALUE Format	392
12.9	Format Libraries	393
12.9.1	Saving Formats Permanently	393
12.9.2	Searching for Formats	394
12.9.3	Concatenating Format Catalogs and Libraries	394
Chapter 13	Interfacing with the Macro Language	397
13.1	Avoiding Macro Variable Collisions—Make Your Macro Variables %Local	398
13.2	Using the SYMPUTX Routine	400
13.2.1	Compared to CALL SYMPUT	401
13.2.2	Using SYMPUTX to Save Values of Options	402
13.2.3	Using SYMPUTX to Build a List of Macro Variables	402
13.3	Generalized Programs—Variations on a Theme	403
13.3.1	Steps to the Generalization of a Program	403
13.3.2	Levels of Generalization and Levels of Macro Language Understanding	405
13.4	Utilizing Macro Libraries	406
13.4.1	Establishing an Autocall Library	406
13.4.2	Tracing Autocall Macro Locations	408
13.4.3	Using Stored Compiled Macro Libraries	408
13.4.4	Macro Library Search Order	409

13.5 Metadata-Driven Programs	409
13.5.1 Processing across Data Sets	409
13.5.2 Controlling Data Validations	410
13.6 Hard Coding—Just Don't Do It	415
13.7 Writing Macro Functions	417
13.8 Macro Information Sources	420
13.8.1 Using SASHELP and Dictionary tables	420
13.8.2 Retrieving System Options and Settings	422
13.8.3 Accessing the Metadata of a SAS Data Set	424
13.9 Macro Security and Protection	426
13.9.1 Hiding Macro Code	426
13.9.2 Executing a Specific Macro Version	427
13.10 Using the Macro Language IN Operator	430
13.10.1 What Can Go Wrong	430
13.10.2 Using the MINOPERATOR Option	431
13.10.3 Using the MINDELIMITER= Option	432
13.10.4 Compilation vs. Execution for these Options	432
13.11 Making Use of the MFILE System Option	433
13.12 A Bit on Macro Quoting	434

Chapter 14 Operating System Interface and Environmental Control 437

14.1 System Options	438
14.1.1 Initialization Options	438
14.1.2 Data Processing Options	441
14.1.3 Saving SAS System Options	444
14.2 Using an AUTOEXEC Program	446
14.3 Using the Configuration File	446
14.3.1 Changing the SASAUTOS Location	447
14.3.2 Controlling DM Initialization	449
14.4 In the Display Manager	449
14.4.1 Showing Column Names in ViewTable	450
14.4.2 Using the DM Statement	451
14.4.3 Enhanced Editor Options and Shortcuts	452
14.4.4 Macro Abbreviations for the Enhanced Editor	456
14.4.5 Adding Tools to the Application Tool Bar	461
14.4.6 Adding Tools to Pull-Down and Pop-up Menus	463
14.4.7 Adding Tools to the KEYS List	466
14.5 Using SAS to Write and Send E-mails	467

- 14.6 Recovering Physical Location Information 468**
 - 14.6.1 Using the PATHNAME Function 468
 - 14.6.2 SASHELP VIEWS and DICTIONARY Tables 468
 - 14.6.3 Determining the Executing Program Name and Path 469
 - 14.6.4 Retrieving the UNC (Universal Naming Convention) Path 470

Chapter 15 Miscellaneous Topics 473

- 15.1 A Few Miscellaneous Tips 474**
 - 15.1.1 Customizing Your NOTES, WARNINGS, and ERRORS 474
 - 15.1.2 Enhancing Titles and Footnotes with the #BYVAL and #BYVAR Options 475
 - 15.1.3 Executing OS Commands 477
- 15.2 Creating User-defined Functions Using PROC FCMP 479**
 - 15.2.1 Building Your Own Functions 479
 - 15.2.2 Storing and Accessing Your Functions 481
 - 15.2.3 Interaction with the Macro Language 482
 - 15.2.4 Viewing Function Definitions 483
 - 15.2.5 Removing Functions 484
- 15.3 Reading RTF as Data 485**
 - 15.3.1 RTF Diagram Completion 486
 - 15.3.2 Template Preparation 486
 - 15.3.3 RTF as Data 487

Appendix A Topical Index 489

Appendix B Usage Index 491

- Global Statements and Options 492**
 - Statements, Global 492
 - Macro Language 493
 - GOPTIONS, Graphics 493
 - Options, System 493
 - Options, Data Set 495
- Procedures: Steps, Statements, and Options 495**
 - Procedures 495
- DATA Step: Statements and Options 500**
 - Statements, DATA Step 500
 - Format Modifiers 501
 - Functions 501
 - Hash Object 504

Output Delivery System, ODS 504

- ODS Destinations and Tagsets 504
- ODS Attributes 505
- ODS Options 505
- ODS Statements 506

SAS Display Manager 506

- Display Manager Commands 506

References 507

User Publications 507

Generally Good Reading—Lots More to Learn 518

- SAS Documentation 518
- SAS Usage Notes 518
- Discussion Forums 518
- Newsletters, Corporate and Private Sites 519
- User Communities 519
- Publications 519
- Learning SAS 520

Index 521



Chapter 1

Moving, Copying, Importing, and Exporting Data

- 1.1 LIBNAME Statement Engines 4**
 - 1.1.1 Using Data Access Engines to Read and Write Data 5
 - 1.1.2 Using the Engine to View the Data 6
 - 1.1.3 Options Associated with the Engine 6
 - 1.1.4 Replacing EXCEL Sheets 7
 - 1.1.5 Recovering the Names of EXCEL Sheets 8
- 1.2 PROC IMPORT and EXPORT 9**
 - 1.2.1 Using the Wizard to Build Sample Code 9
 - 1.2.2 Control through the Use of Options 9
 - 1.2.3 PROC IMPORT Data Source Statements 10
 - 1.2.4 Importing and Exporting CSV Files 12
 - 1.2.5 Preventing the Export of Blank Sheets 15
 - 1.2.6 Working with Named Ranges 16
- 1.3 DATA Step INPUT Statement 17**
 - 1.3.1 Format Modifiers for Errors 18
 - 1.3.2 Format Modifiers for the INPUT Statement 18
 - 1.3.3 Controlling Delimited Input 20
 - 1.3.4 Reading Variable-Length Records 24
- 1.4 Writing Delimited Files 28**
 - 1.4.1 Using the DATA Step with the DLM= Option 28
 - 1.4.2 PROC EXPORT 29
 - 1.4.3 Using the %DS2CSV Macro 30
 - 1.4.4 Using ODS and the CSV Destination 31
 - 1.4.5 Inserting the Separator Manually 31

1.5 SQL Pass-Through 32

1.5.1 Adding a Pass-Through to Your SQL Step 32

1.5.2 Pass-Through Efficiencies 33

1.6 Reading and Writing to XML 33

1.6.1 Using ODS 34

1.6.2 Using the XML Engine 34

A great deal of the process of the preparation of the data is focused on the movement of data from one table to another. This transfer of data may be entirely within the control of SAS or it may be between disparate data storage systems. Although most of the emphasis in this book is on the use of SAS, not all data are either originally stored in SAS or even ultimately presented in SAS. This chapter discusses some of the aspects associated with moving data between tables as well as into and out of SAS.

When moving data into and out of SAS, Base SAS allows you only limited access to other database storage forms. The ability to directly access additional databases can be obtained by licensing one or more of the various SAS/ACCESS products. These products give you the ability to utilize the SAS/ACCESS engines described in Section 1.1 as well as an expanded list of databases that can be used with the IMPORT and EXPORT procedures (Section 1.2).

SEE ALSO

Andrews (2006) and Frey (2004) both present details of a variety of techniques that can be used to move data to and from EXCEL.

1.1 LIBNAME Statement Engines

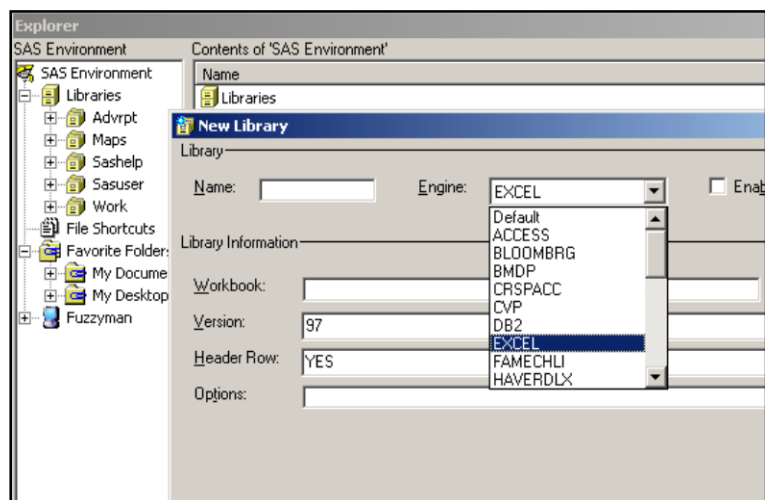
In SAS®9 a number of engines are available for the LIBNAME statement. These engines allow you to read and write data to and from sources other than SAS. These engines can reduce the need to use the IMPORT and EXPORT procedures.

The number of available engines depends on which products your company has licensed from SAS. One of the most popular is SAS/ACCESS® Interface to PC Files.

You can quickly determine which engines are available to you. An easy way to build this list is through the NEW LIBRARY window.

From the SAS Explorer right click on LIBRARIES and select NEW. Available engines appear in the ENGINE pull-down list.

Pulling down the engine list box on the 'New Library' dialog box shown to the right, indicates the engines,



including the EXCEL engine, among others, which are available to this user.

PROC SETINIT can also be used to determine which products have been licensed.

The examples in this section show various aspects of the EXCEL engine; however, most of what is demonstrated can be applied to other engines as well.

SEE ALSO

Choate and Martell (2006) discuss the EXCEL engine on the LIBNAME statement in more detail. Levin (2004) used engines to write to ORACLE tables.

1.1.1 Using Data Access Engines to Read and Write Data

In the following example, the EXCEL engine is used to create an EXCEL workbook, store a SAS data set as a sheet in that workbook, and then read the data back from the workbook into SAS.

```
libname toxls excel "&path\data\newwb.xls"; ❶

proc sort data=advrpt.demog
      out=toxls.demog; ❷
  by clinnum;
run;

data getdemog;
  set toxls.demog; ❸
run;

libname toxls clear; ❹
```

❶ The use of the EXCEL engine establishes the TOXLS *libref* so that it can be used to convert to and from the Microsoft Excel workbook NEWWB.XLS. If it does not already exist, the workbook will be created upon execution of the LIBNAME statement.

For many of the examples in this book, the macro variable &PATH is assumed to have been defined. It contains the upper portion of the path appropriate for the installation of the examples on your system. See the book's introduction and the AUTOEXEC.SAS in the root directory of the example code, which you may download from support.sas.com/authors.

❷ Data sets that are written to the TOXLS *libref* will be added to the workbook as named sheets. This OUT= option adds a sheet with the name of DEMOG to the NEWWB.XLS workbook.

❸ A sheet can be read from the workbook, and brought into the SAS world, simply by naming the sheet.

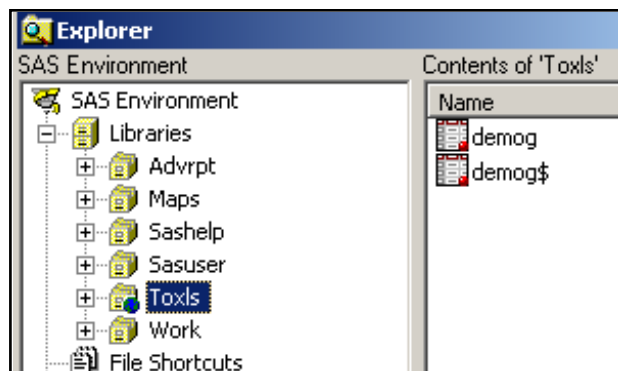
❹ As should be the case with any *libref*, when you no longer need the association, the *libref* should be cleared. This can be especially important when using data engines, since as long as the *libref* exists, access to the data by applications other than SAS is blocked. Until the *libref* is cleared, we are not able to view or work with any sheets in the workbook using Excel.

MORE INFORMATION

LIBNAME statement engines are also discussed in Sections 1.1.2 and 1.2.6. The XML engine is discussed in Section 1.6.2.

1.1.2 Using the Engine to View the Data

Once an access engine has been established by a *libref*, we are able to do almost all of the things that we typically do with SAS data sets that are held in a SAS library.



The SAS Explorer shows the contents of the workbook with each sheet appearing as a data table.

When viewing an EXCEL workbook through a SAS/ACCESS engine, each sheet appears as a data set. Indeed you can use the VIEWTABLE or View Columns tools against what are actually sheets. Notice in this image of the SAS

Explorer, that the DEMOG sheet shows up twice. Sheet names followed by a \$ are actually named ranges, which under EXCEL can actually be a portion of the entire sheet. Any given sheet can have more than one named range, so this becomes another way to filter or subset what information from a given sheet will be brought into SAS through the SAS/ACCESS engine.

1.1.3 Options Associated with the Engine

The SAS/ACCESS engine is acting like a translator between two methods of storing information, and sometimes we need to be able to control the interface. This can often be accomplished through the use of options that modify the translation process. Many of these same options appear in the PROC IMPORT/EXPORT steps as statements or options.

It is important to remember that not all databases store information in the same relationship as does SAS. SAS, for instance, is column based - an entire column (variable) will be either numeric or character. EXCEL, on the other hand, is cell based - a given cell can be considered numeric, while the cell above it in the same column stores text. When translating from EXCEL to SAS we can use options to establish guidelines for the resolution of ambiguous situations such as this.

Connection Options

For database systems that require user identification and passwords these can be supplied as options on the LIBNAME statement.

- USER User identification
- PASSWORD User password
- *others* Other connection options vary according to the database to which you are connecting

LIBNAME Statement Options

These options control how information that is passed through the interface is to be processed. Most of these options are database specific and are documented in the sections dealing with your database.

When working with EXCEL typical LIBNAME options might include:

- **HEADER** Determines if a header row exists or should be added to the table.
- **MIXED** Some columns contain both numeric and character information.
- **VER** Controls which type (version) of EXCEL is to be written.

Data Source Options

Some of the same options associated with PROC IMPORT (see Section 1.2.3) can also be used on the LIBNAME statement. These include:

- **GETNAMES** Incoming variable names are available in the *first row* of the incoming data.
- **SCANTEXT** A length is assigned to a character variable by scanning the incoming column and determining the maximum length.

1.1.4 Replacing EXCEL Sheets

While the EXCEL engine allows you to establish, view, and use a sheet in an Excel workbook as a SAS data set, you cannot update, delete or replace the sheet from within SAS. It is possible to replace the contents of a sheet, however, with the help of PROC DATASETS and the SCAN_TEXT=NO option on the LIBNAME statement. The following example shows how to replace the contents of an EXCEL sheet.

In the first DATA step the programmer has ‘accidentally’ used a WHERE clause ❶ that writes the

```
libname toxls excel "&path\data\newwb.xls";

data toxls.ClinicNames;
  set advrpt.clinicnames;
  where clinname>'X'; ❶
run;

* Running the DATA step a second time
* results in an error;
data toxls.ClinicNames; ❷
  set advrpt.clinicnames;
run;
```

incorrect data, in this case 0 observations, to the EXCEL sheet. Simply correcting and rerunning the DATA step ❷ will not work because the sheet already exists.

We could step out of SAS and use EXCEL to manually remove the bad sheet; however, we would rather do it from within SAS. First we must

```
libname toxls excel
               "&path\data\newwb.xls"
               scan_text=no ❸;
proc datasets library=toxls nolist;
  delete ClinicNames;
quit;
```

reestablish the *libref* using the SCAN_TEXT=NO option ❸. PROC DATASETS can then be used to *delete* the sheet. In actuality the sheet

has not truly been deleted, but merely cleared of all contents. Since the sheet is now truly empty and the SCAN_TEXT option is set to NO, we can now replace the empty sheet with the desired contents.

```
data toxls.ClinicNames; ❹
    set advrpt.clinicnames;
run;

libname toxls clear; ❺
```

The DATA step can now be rerun ❹, and the sheet contents will now be correct. When SAS has completed its work with the workbook, and before you can use the workbook using EXCEL you will need to clear the *libref*. This can be done using the CLEAR option on the LIBNAME statement ❺.

MORE INFORMATION

See Section 1.2 for more information on options and statements in PROC IMPORT and PROC EXPORT. In addition to PROC DATASETS, Section 5.4 discusses other techniques that can be used to delete tables. Section 14.4.5 also has an example of deleting data sets using PROC DATASETS.

SEE ALSO

Choate and Martell (2006) discuss this and numerous other techniques that can be used with EXCEL.

1.1.5 Recovering the Names of EXCEL Sheets

Especially when writing automated systems you may need to determine the names of workbook sheets. There are a couple of ways to do this.

If you know the *libref*(s) of interest, the automatic view SASHELP.VTABLE can be used in a

```
data sheetnames;
set sashelp.vtable;
where libname = 'TOXLS';
run;
```

DATA step to see the sheet names. This view contains one observation for every SAS data set in every SAS library in current use, and for the TOXLS *libref* the sheet names will be shown as data set names.

```
proc sql;
create table sheetnames as
select * from dictionary.members
where engine= 'EXCEL' ;
quit ;
```

When there are a number of active libraries, the process of building this table can be lengthy. As a general rule using the DICTIONARY.MEMBERS table in a PROC SQL step has a couple of advantages. It is usually quicker

than the SASHELP.VTABLE view, and it also has an ENGINE column which allows you to search without knowing the specific *libref*.

The KEEP statement or the preferred KEEP= data set option could have been used in these examples to reduce the number of variables (see Section 2.1.3).

MORE INFORMATION

SASHELP views and DICTIONARY tables are discussed further in Section 13.8.1.

SEE ALSO

A thread in the SAS Forums includes similar examples.

<http://communities.sas.com/thread/10348?tstart=0>

1.2 PROC IMPORT and EXPORT

Like the SAS/ACCESS engines discussed in Section 1.1, the IMPORT and EXPORT procedures are used to translate data into and out of SAS from a variety of data sources. The SAS/ACCESS product, which is usually licensed separately through SAS (but may be bundled with Base SAS), controls which databases you will be able to move data to and from. Even without SAS/ACCESS you can still use these two procedures to read and write text files such as comma separated variables (CSV), as well as files using the TAB and other delimiters to separate the variables.

1.2.1 Using the Wizard to Build Sample Code

The import/export wizard gives you a step-by-step guide to the process of importing or exporting data. The wizard is easy enough to use, but like all wizards does not lend itself to automated or batch processing. Fortunately the wizard is actually building a PROC IMPORT/EXPORT step in the background, and you can capture the completed code. For both the import and export process the last screen prompts you to ‘Create SAS Statements.’

```
PROC EXPORT DATA= WORK.A ❶
      OUTFILE= "C:\temp\junk.xls" ❷
      DBMS=EXCEL ❸
      REPLACE ❹;
      SHEET="junk"; ❺
RUN;
```

The following PROC EXPORT step was built using the EXPORT wizard. A simple inspection of the code indicates what needs to be changed for a future application of the EXPORT procedure. Usually this means that the wizard itself needs to be run infrequently.

- ❶ The DATA= option identifies the data set that is to be converted.
- ❷ In this case, since we are writing to EXCEL ❸ the OUTFILE= identifies the workbook.
- ❹ If the sheet already exists, it will be replaced.
- ❺ The sheet name can also be provided.

Converting the previous generic step to one that creates a CSV file is very straightforward.

```
PROC EXPORT DATA= sashelp.class
      OUTFILE= "&path\data\class.csv"
      DBMS=csv
      REPLACE;
RUN;
```

SEE ALSO

Raithel (2009) discusses the use of the EXPORT wizard to generate code in a sasCommunity.org tip.

1.2.2 Control through the Use of Options

There are only a few options that need to be specified. Of these most of the interesting ones are used when the data are being imported (clearly SAS already knows all about the data when it is being exported).

- **DBMS=** Identifies the incoming database structure (including .CSV and .TXT). Since database structures change with versions of the software, you should know the database version. Specific engines exist at the version level for some databases (especially Microsoft's EXCEL and ACCESS). The documentation discusses which engine is optimized for each software version.
- **REPLACE** Determines whether or not the destination target (data set, sheet, table) is replaced if it already exists.

1.2.3 PROC IMPORT Data Source Statements

These statements give you additional control over how the incoming data are to be read and interpreted. Availability of any given source statement depends on the type (DBMS=) of the incoming data.

- **DATAROW** First incoming row that contains data.
- **GETNAMES** The names of the incoming columns are available in the *first row* of the incoming data. Default column names when none are available on the incoming table are VAR1, VAR2, etc.
- **GUESSINGROWS** Number of rows SAS will scan before determining if an incoming column is numeric or character. This is especially important for mixed columns and early rows are all numeric. In earlier versions of SAS modifications to the SAS Registry were needed to change the number of rows used to determine the variable's type, which is fortunately no longer necessary.
- **RANGE and SHEET** For spreadsheets a specific sheet name, named range, or range within a sheet can be specified.
- **SCANTEXT and TEXTSIZE** PROC IMPORT assigns a length to a character variable by scanning the incoming column and determining the maximum.

When using GETNAMES to read column names from the source data, keep in mind that most databases use different naming conventions than SAS and may have column names that will cause problems when imported. By default illegal characters are replaced with an underscore (_) by PROC IMPORT. When you need the original column name, the system option VALIDVARNAME=ANY (see Section 14.1.2) allows a broader range of acceptable column names.

In the contrived data for the following example we have an EXCEL file containing a subject number and a response variable (SCALE). The import wizard can be used to generate a PROC IMPORT step that will read the XLS file (MAKESCALE.XLS) and create the data set WORK.SCALEDATA. This PROC IMPORT step creates two numeric variables.

	A	B
1	subject	scale
2	200	1
3	200	2
4	200	3
5	200	4
6	200	5
7	200	6
8	200	7
9	200	8
10	200	9

```
PROC IMPORT OUT= WORK.scaledata
            DATAFILE= "C:\Temp\makescale.xls"
                        DBMS=EXCEL REPLACE;

            RANGE="MAKESCALE";
            GETNAMES=YES; ❶
            MIXED=NO; ❷
            SCANTEXT=YES;
            USEDATE=YES;
            SCANTIME=YES;
            RUN;
```

Notice that the form of the supporting statements is different than form most procedures. They look more like options (option=value;) than like statements. The GETNAMES= statement ❶ is used to determine the variable names from the first column.

When importing data SAS must determine if a given column is to be numeric or character. A number of clues are utilized to make this determination. SAS will scan a number of rows for each column to try to determine if all the values are numeric. If a non-numeric value is found, the column will be read as a character variable; however, only some of the rows are scanned and consequently an incorrect determination is possible. ❷ The MIXED= statement is used to specify that the values in a given column are always of a single type (numeric or character). When set to YES, the IMPORT procedure will tend to create character variables in order to accommodate mixed types.

In this contrived example it turns out that starting with subject 271 the variable SCALE starts taking on non-numeric values. Using the previous PROC IMPORT step does not detect this change, and creates SCALE as a numeric variable. This, of course, means that data will be lost as SCALE will be missing for the observations starting from row 712.

706	270	5
707	270	6
708	270	7
709	270	8
710	270	9
711	270	10
712	271	a
713	271	b
714	271	c
715	271	d

GUESSINGROWS statements.

For PROC IMPORT to correctly read the information in SCALE it needs to be a character variable. We can encourage IMPORT to create a character variable by using the MIXED and

```
PROC IMPORT OUT= WORK.scaledata
            DATAFILE= "C:\Temp\makescale.xls"
                        DBMS=excel REPLACE;

            GETNAMES=YES;
            MIXED=YES; ❸
            RUN;
```

Changing the MIXED= value to YES ❸ is not necessarily sufficient to cause SCALE to be a character value; however, if the value of the DBMS option is changed from EXCEL to XLS ❹, the MIXED=YES statement ❸ is honored and SCALE is written as a character variable in the data set SCALEDATA.

```
PROC IMPORT OUT= WORK.scaledata
           DATAFILE= "C:\Temp\makescale.xls"
           DBMS=xls REPLACE; ❹
           GETNAMES=YES; ❸
           GUESSINGROWS=800; ❺
RUN;
```

When MIXED=YES is not practical the GUESSINGROWS= statement can sometimes be used to successfully determine the type for a variable.

GUESSINGROWS cannot be used when DBMS=EXCEL, however it can be used when DBMS=XLS. Since GUESSINGROWS ❺ changes the number of rows that are scanned prior to determining if the column should be numeric or character, its use can increase the time and resources required to read the data.

SEE ALSO

The SAS Forum thread <http://communities.sas.com/thread/12743?tstart=0> has a PROC IMPORT using Namerow= and Startrow= data source statements. The thread <http://communities.sas.com/thread/30405?tstart=0> discusses named ranges, and it and the thread <http://communities.sas.com/thread/12293?tstart=0> show the use of several data source statements.

1.2.4 Importing and Exporting CSV Files

Comma Separated Variable, CSV, files have been a standard file type for moving data between systems for many years. Fortunately we now have a number of superior tools available to us so that we do not need to resort to CSV files as often. Still they are commonly used and we need to understand how to work with them.

Both the IMPORT and EXPORT procedures can work with CSV files (this capability is a part of the Base SAS product and a SAS/ACCESS product is not required). Both do the conversion by first building a DATA step, which is then executed.

Building a DATA Step

When you use the import/export wizard to save the PROC step (see Section 1.2.1), the resulting DATA step is not saved. Fortunately you can still get to the generated DATA step by recalling the last submitted code.

1. Execute the IMPORT/EXPORT procedure.
2. While in the Display Manager, go to RUN→Recall Last Submit.

Once the code generated by the procedure is loaded into the editor, you can modify it for other purposes or simply learn from it. For the simple PROC EXPORT step in Section 1.2.1, the following code is generated:

```

/*****
*   PRODUCT:   SAS
*   VERSION:   9.1
*   CREATOR:   External File Interface
*   DATE:      11APR09
*   DESC:      Generated SAS Datasheet Code
*   TEMPLATE SOURCE: (None Specified.)
*****/

data _null_;
set SASHELP.CLASS                                end=EFIEOD;
%let _EFIERR_ = 0; /* set the ERROR detection macro variable */
%let _EFIREC_ = 0; /* clear export record count macro variable */
file 'C:\InnovativeTechniques\data\class.csv' delimiter=','
      DSD DROPOVER lrecl=32767;
  format Name $8. ;
  format Sex $1. ;
  format Age best12. ;
  format Height best12. ;
  format Weight best12. ;
if _n_ = 1 then /* write column names */
do;
  put
    'Name'
    ','
    'Sex'
    ','
    'Age'
    ','
    'Height'
    ','
    'Weight'
  ;
end;
do;
  EFIOUT + 1;
  put Name $ @;
  put Sex $ @;
  put Age @;
  put Height @;
  put Weight ;
  ;
end;
if _ERROR_ then call symputx('_EFIERR_',1); /*set ERROR detection
                                           macro variable*/
if EFIEOD then call symputx('_EFIREC_',EFIOUT);
run;

```

Headers are Not on Row 1

The ability to create column names based on information contained in the data is very beneficial. This is especially important when building a large SAS table from a CSV file with lots of columns. Unfortunately we do not always have a CSV file with the column headers in row 1. Since GETNAMES=YES assumes that the headers are in row 1 we cannot use GETNAMES=YES. Fortunately this is SAS, so there are alternatives.

The CSV file created in the PROC EXPORT step in Section 1.2.1 has been modified so that the column names are on row 3. The first few lines of the file are:


```

Class Data from SASHELP,,,,
Comma Separated rows; starting in row 3,,,,
Name,Sex,Age,Height,Weight
Alfred,M,14,69,112.5
Alice,F,13,56.5,84
Barbara,F,13,65.3,98
Carol,F,14,62.8,102.5
.... data not shown ....

```

The DATA step generated by PROC IMPORT (E1_2_3c_ImportWO.SAS), simplified somewhat for this example, looks something like:

```

data WORK.CLASSWO
infile "&path\Data\classwo.csv" delimiter = ',' ;
    MISSOVER DSD lrecl=32767 firstobs=4 ;
    informat VAR1 $8. ;
    informat VAR2 $1. ;
    informat VAR3 best32. ;
    informat VAR4 best32. ;
    informat VAR5 best32. ;
    format VAR1 $8. ;
    format VAR2 $1. ;
    format VAR3 best12. ;
    format VAR4 best12. ;
    format VAR5 best12. ;
input
            VAR1 $
            VAR2 $
            VAR3
            VAR4
            VAR5
;
run;

```

Clearly SAS has substituted VAR1, VAR2, and so on for the unknown variable names. If we knew the variable names, all we would have to do to fix the problem would be to rename the variables. The following macro reads the header row from the appropriate row in the CSV file, and uses that information to rename the columns in WORK.CLASSWO.

```

%macro rename(headrow=3, rawcsv=, dsn=);
%local lib ds i;
data _null_ ;
  infile "&path\Data\&rawcsv"
    scanover lrecl=32767 firstobs=&headrow;
  length temp $ 32767;
  input temp $;
  i=1;
  do while(scan(temp,i,',') ne ' ');
    call symputx('var' || left(put(i,4.)), scan(temp,i,','), '1');
    i+1;
  end;
  call symputx('varcnt', i-1, '1');
  stop;
run;

%* Determine the library and dataset name;
%if %scan(&dsn,2,.) = %then %do;
  %let lib=work;
  %let ds = %scan(&dsn,1,.);
%end;
%else %do;
  %let lib= %scan(&dsn,1,.);
  %let ds = %scan(&dsn,2,.);
%end;

proc datasets lib=&lib nolist;
  modify &ds;
  rename
  %do i = 1 %to &varcnt;
    var&i = &&var&i
  %end;
  ;
quit;
%mend rename;

%rename(headrow=3, rawcsv=classwo.csv, dsn=work.classwo)

```

SEE ALSO

McGuown (2005) also discusses the code generated by PROC IMPORT when reading a CSV file. King (2011) uses arrays and hash tables to read CSV files with unknown or varying variable lists. These flexible and efficient techniques could be adapted to the type of problem described in this section.

1.2.5 Preventing the Export of Blank Sheets

PROC EXPORT does not protect us from writing a blank sheet when our exclusion criteria excludes all possible rows from a given sheet ❶. In the following example we have inadvertently

```

proc export data=sashelp.class(where=(sex='q'❶))
  outfile='c:\temp\classmates.xls'
  dbms=excel2000
  replace;
  SHEET='sex: Q';
run;

```

asked to list all students with SEX='q'. There are none of course, and the resulting sheet is blank, except for the column headers.

We can prevent this from occurring by first identifying those levels of SEX that have one or more rows. There are a number of ways to generate a list of values of a variable; however, an SQL step is ideally suited to place those values into a macro variable for further processing.

The name of the data set that is to be exported, as well as the classification variable, are passed to the macro %MAKEXLS as named parameters.

```
%macro makexls(dsn=,class=);
%local valuelist listnum i value;
proc sql noprint;
select distinct &class ❷
  into :valuelist separated by ' ' ❸
  from &dsn;
%let listnum = %sqlobs;
quit;

%* One export for each sheet;
%do i = 1 %to &listnum; ❹
  %let value = %scan(&valuelist,&i,%str( )); ❺
  proc export data=&dsn(where=(&class="&value")) ❻
    outfile="c:\temp\&dsn..xls"
    dbms=excel2000
    replace;
    SHEET="&class:&value";
  run;
%end;
%mend makexls;
%makexls(dsn=sashelp.class,class=sex)
```

❷ An SQL step is used to build a list of distinct values of the classification variable.

❸ These values are saved in the macro variable &VALUELIST.

❹ A %DO loop is used to process across the individual values, which are extracted ❺ from the list using the %SCAN function.

❻ The PROC EXPORT step then creates a sheet for the selected value. ❼

SEE ALSO

A similar example which breaks a data set into separate sheets can be found in the article “Automatically_Separating_Data_into_Excel_Sheets” on sasCommunity.org.
http://www.sascommunity.org/wiki/Automatically_Separating_Data_into_Excel_Sheets

1.2.6 Working with Named Ranges

By default PROC IMPORT and the LIBNAME statement's EXCEL engine expect EXCEL data to be arranged in a certain way (column headers, if present, on row one column A; and data starting on row two). It is not unusual, however, for the data to be delivered as part of a report or as a subset of a larger table. One solution is to manually cut and paste the data onto a blank sheet so that it conforms to the default layout. It can often be much easier to create a named range.

	A	B	C	D	E	F	G
1	Data From SASHELP.CLASS						
2			Data Columns				
3	Variable Names	Name	Sex	Age	Height	Weight	
4		Alfred	M		14	69	112.5
5		Alice	F		13	56.5	84
6		Barbara	F		13	65.3	98
7		Carol	F		14	62.8	102.5
8		Henry	M		14	63.5	102.5
9		James	M		12	57.3	83
10		Jane	F		12	59.8	84.5
11		Janet	F		15	62.5	112.5
12		Jeffrey	M		13	62.5	84
13		John	M		12	59	99.5

The EXCEL spreadsheet shown here contains the SASHELP.CLASS data set (only part of which is shown here); however, titles and columns have been added. Using the defaults PROC IMPORT will not be able to successfully read this sheet.

To facilitate the use of this spreadsheet, a named range was created for the rectangle defined by C3-G22. This

range was given the name 'CLASSDATA'. This named range can now be used when reading the data from this sheet.

When reading a named range using the EXCEL engine on the LIBNAME statement, the named

```
libname seexls excel "&path\data\E1_2_6classmates.xls";
data class;
  set seexls.classdata; ❶
run;
libname seexls clear; ❷
```

range (CLASSDATA) is used just as you would the sheet name ❶.

❷ When using an

engine on the LIBNAME statement be sure to clear the *libref* so that you can use the spreadsheet outside of SAS.

When using PROC IMPORT to read a named range, the RANGE= statement ❸ is used to

```
proc import out=work.classdata
  datafile= "&path\data\E1_2_6classmates.xls"
  dbms=xls replace;
  getnames=yes;
  range='classdata'; ❸
run;
```

designate the named range of interest. Since the name of the named range is unique to the workbook, a sheet name is not required.

MORE INFORMATION

The EXCEL LIBNAME engine is introduced in Section 1.1.

1.3 DATA Step INPUT Statement

The INPUT statement is loaded with options that make it extremely flexible. Since there has been a great deal written about the basic INPUT statement, only a few of the options that seem to be under used have been collected here.

SEE ALSO

An overview about reading raw data with the INPUT statement can be found in the SAS documentation at <http://support.sas.com/publishing/pubcat/chaps/58369.pdf>. Schreier (2001) gives a short overview of the automatic _INFILE_ variable along with other information regarding the reading of raw data.

1.3.1 Format Modifiers for Errors

Inappropriate data within an input field can cause input errors that prevent the completion of the data set. As the data are read, a great many messages can also be generated and written to the LOG. The (?) and (??) format modifiers control error handling. Both the ? and the ?? suppress error messages in the LOG; however, the ?? also resets the automatic error variable (`_ERROR_`) to 0. This means that while both of these operators control what is written to the LOG only the ?? will necessarily prevent the step from terminating when the maximum error count is reached.

In the following step, the third data row contains an invalid value for AGE. AGE is assigned a missing value, and because of the ?? operator no 'invalid data' message is written to the LOG.

```
data base;
input age ?? name $;
datalines;
15   Fred
14   Sally
x    John
run;
```

MORE INFORMATION

The ?? modifier is used with the INPUT function in Sections 2.3.1 and 3.6.1.

SEE ALSO

The SAS Forum thread found at <http://communities.sas.com/message/48729> has an example that uses the ?? format modifier.

1.3.2 Format Modifiers for the INPUT Statement

Some of the most difficult input coding occurs when combining the use of informats with LIST style input. This style is generally required when columns are not equally spaced so informats can't be easily used, and the fields are delimited with blanks. LIST is also the least flexible input style. Informat modifiers include:

- & allows embedded blanks in character variables
- :
- ~ allows the use of quotation marks within data fields

Because of the inherent disadvantages of LIST input (space delimited fields), when it is possible, consider requesting a specific unique delimiter. Most recently generated files of this type utilize a non-blank delimiter, which allows you to take advantage of some of the options discussed in Section 1.3.3. Unfortunately many legacy files are space delimited, and we generally do not have the luxury of either requesting a specific delimiter or editing the existing file to replace the spaces with delimiters.

There are two problems in the data being read in the following code. The three potential INPUT statements (two of the three are commented) highlight how the ampersand and colon can be used to help read the data. Notice that DOB does not start in a consistent column and the second last name has an embedded blank.

```

title '1.3.2a List Input Modifiers';
data base;
length lname $15;
input fname $ dob mmddyy10. lname $ ; ❶
*input fname $ dob :mmddyy10. lname $ ; ❷
*input fname $ dob :mmddyy10. lname $ &; ❸
datalines;
Sam 12/15/1945 Johnson
Susan 10/10/1983 Mc Callister
run;

```

Using the first INPUT statement without informat modifiers ❶ shows, that for the second data line, both the date and the last name have been read incorrectly.

Obs	lname	fname	dob
1	Johnson	Sam	12/15/1945
2	83	Susan	10/10/ 2019

Assuming the second INPUT statement ❷ was commented and used, the colon modifier is placed in front of the date informat. The colon allows the format to essentially float to the appropriate starting point by using LIST input and then applying the informat once the value is found.

Obs	lname	fname	dob
1	Johnson	Sam	12/15/1945
2	Mc	Susan	10/10/1983

The birthdays are now being read correctly; however, Susan's last name is being split because the embedded blank is being interpreted as a field delimiter. The ampersand ❸ can be used to allow embedded spaces within a

field.

By placing an ampersand after the variable name (LNAME) ❸, the blank space becomes part of the variable rather than a delimiter. We are now reading both the date of birth and the last name correctly.

```
input fname $ dob :mmddyy10. lname $ &; ❸
```

Obs	lname	fname	dob
1	Johnson	Sam	12/15/1945
2	Mc Callister	Susan	10/10/1983

While the ampersand is also used as a macro language trigger, this will not be a problem

when it is used as an INPUT statement modifier as long as it is not immediately followed by text that could be interpreted as a macro variable name (letter or underscore). In this example the ampersand is followed by the semicolon so there will be no confusion with the macro language.

While the trailing ampersand can be helpful it can also introduce problems as well. If the data had been slightly more complex, even this solution might not have worked. The following data also contains a city name. Even though the city is not being read, the trailing & used with the last name

(LNAME) causes the city name to be confused with the last name.

```

title '1.3.2b List Input Modifiers';
data base;
length lname $15;
input fname $ dob :mmddyy10. lname $ &;
format dob mmddyy10.; ❹
datalines;
Sam 12/15/1945 Johnson   Seattle
Susan 10/10/1983 Mc Callister New York
; ❺
run;

```

Because of the trailing & and the length of LNAME (\$15) a portion of the city (New York) has been read into the LNAME for the second observation. On the first observation the last name is correct because more than one space separates Johnson and Seattle. Even with the trailing &, more than one space is still successfully seen as a field delimiter.

Obs	lname	fname	dob
1	Johnson	Sam	12/15/1945
2	Mc Callister Ne	Susan	10/10/1983

On the second observation the city would not have been confused with the last name had there been two or more spaces between the two fields.

❹ Placing the FORMAT statement within the DATA step causes the format to be associated with the variable DOB in subsequent steps. The INFORMAT statement is only used when reading the data.

❺ The DATALINES statement causes subsequent records to be read as data up to, but not including, the first line that contains a semicolon. In the previous examples the RUN statement doubles as the end of data marker. Many programmers use a separate semicolon to perform this task. Both styles are generally considered acceptable (as long as you are using the RUN statement to end your step).

With only a single space between the last name and the city, the trailing & alone is not sufficient to help the INPUT statement distinguish between these two fields. Additional variations of this example can be found in Section 1.3.3.

MORE INFORMATION

LIST input is a form of delimited input and as such these options also apply to the examples discussed in Section 1.3.3. When the date form is not consistent one of the *any date* informats may be helpful. See Section 12.6 for more information on the use of these specialized informats.

SEE ALSO

The SAS Forum thread <http://communities.sas.com/message/42690> discusses the use of list input modifiers.

1.3.3 Controlling Delimited Input

Technically LIST input is a form of delimited input, with the default delimiter being a space. This means that the modifiers shown in Section 1.3.2 apply to other forms of delimited input, including comma separated variable, CSV, files.

INFILE Statement Options

Options on the INFILE statement are used to control how the delimiters are to be interpreted.

- **DELIMITER** Specifies the character that delimits fields (other than the default - a space). This option is often abbreviated as DLM=.
- **DLMSTR** Specifies a single multiple character string as a delimiter.
- **DLMOPT** Specifies parsing options for the DLMSTR option.
- **DSD** Allows character fields that are surrounded by quotes (by setting the comma as the delimiter). Two successive delimiters are interpreted as individual delimiters, which allow missing values to be assigned appropriately. DSD also removes quotation marks from character values surrounded by quotes. If the comma is not the delimiter you will need to use the DLM= option along with the DSD option.

Some applications, such as Excel, build delimiter separated variable files with quotes surrounding the fields. This can be critical if a field's value can contain the field separator. For default list input, where a space is a delimiter, it can be very difficult to successfully read a field with an embedded blank (see Section 1.3.2 which discusses the use of trailing & to read embedded spaces). The DSD option alerts SAS to the *potential* of quoted character fields. The following example demonstrates simple comma-separated data.

```
data base;
length lname $15;
infile datalines ❶ dlm=','; ❷
*infile datalines dlm=',' dsd; ❸
input fname $ lname $ dob :mmddyy10.;
datalines;
'Sam','Johnson',12/15/1945
'Susan','Mc Callister',10/10/1983
run;
```

❶ Although the INFILE statement is often not needed when using the DATALINES, CARDS, or CARDS4 statements, it can be very useful when the options associated with the INFILE statement are needed. The *fileref* can be DATALINES or CARDS.

The DLM= option is used to specify the delimiter. In this example the field delimiter is specified as a comma ❷.

1.3.3a Delimited List Input Modifiers

Obs	lname	fname	dob
1	'Johnson'	'Sam'	12/15/1945
2	'Mc Callister'	'Susan'	10/10/1983

The fields containing character data have been quoted. Since we do not actually want the quote marks to be a part of the data fields, the DSD option ❸ alerts the parser to this possibility and the quotes themselves become a part of the field delimiting process.

```
infile datalines dlm=',' dsd; ❸
```

Using the DSD option results in data fields without the quotes.

1.3.3a Delimited List Input Modifiers

Obs	lname	fname	dob
1	Johnson	Sam	12/15/1945
2	Mc Callister	Susan	10/10/1983

On the INPUT Statement

The tilde (~) ❹ can be used to modify a format, much the same way as a colon (:); however, the two modifiers are not exactly the same.

```

title '1.3.3b Delimited List Input Modifiers';
title2 'Using the ~ Format Modifier';
data base;
length lname $15;
infile datalines dlm=',' dsd;
input fname $ lname $ birthloc $~❹15. dob :mmddyy10. ;
datalines;
'Sam','Johnson', 'Fresno, CA','12/15/1945'
'Susan','Mc Callister','Seattle, WA',10/10/1983
run;

```

The tilde format modifier correctly reads the BIRTHLOC field; however, it preserves the quote marks that surround the field. Like the colon, the tilde can either precede or follow the \$ for character variables. As an aside notice that for this example quote marks surround the numeric date value for the first row. The field is still processed correctly as a numeric SAS date value.

```

1.3.3b Delimited List Input Modifiers
Using the ~ Format Modifier

Obs      lname           fname      birthloc           dob
1        Johnson        Sam        'Fresno, CA'      12/15/1945
2        Mc Callister    Susan      'Seattle, WA'     10/10/1983

```

Replacing the tilde ❹ with a colon (:) would cause the BIRTHLOC value to be saved without the quote marks. If instead we supply a length for BIRTHLOC ❺, neither a format nor the tilde will be needed.

```

title '1.3.3c Delimited List Input Modifiers';
title2 'BIRTHLOC without a Format Modifier';
title3 'BIRTHLOC Length Specified';
data base;
length lname birthloc $15; ❺
infile datalines dlm=',' dsd;
input fname $ lname $ birthloc $ dob :mmddyy10. ;
datalines;
'Sam','Johnson', 'Fresno, CA',12/15/1945
'Susan','Mc Callister','Seattle, WA',10/10/1983
run;

```

```

1.3.3c Delimited List Input Modifiers
BIRTHLOC without a Format Modifier
BIRTHLOC Length Specified

Obs      lname           birthloc      fname      dob
1        Johnson        Fresno, CA    Sam        12/15/1945
2        Mc Callister    Seattle, WA   Susan      10/10/1983

```

Multiple Delimiters

It is possible to read delimited input streams that contain more than one delimiter. In the following small example two delimiters, a comma and a slash are both used to delimit the data values.

```
data imports;
infile cards dlm='/, ';
input id importcode $ value;
cards;
14,1,13
25/Q9,15
6,D/20
run;
```

Obs	id	importcode	value
1	14	1	13
2	25	Q9	15
3	6	D	20

```
data imports;
retain dlmvar '/, '; ❸
infile cards dlm=dlmvar;
input id importcode $ value;
cards;
14,1,13
25/Q9,15
6,D/20
run;
```

Notice that the DLM option causes either the comma or the slash to be used as field delimiters, but not the slash comma together as a single delimiter (see the DLMSTR option below to create a single multiple character delimiter).

❸ Because the INFILE statement is executed for each observation, the value assigned to the DLM option does not necessarily need to be a constant. It can also be a variable or can be changed using IF-THEN/ELSE logic. In the simplest form this variable could be assigned in a retain statement.

```
data imports;
infile cards;
input dlmvar $1. @;
infile cards dlm=dlmvar; ❹
input @2 id importcode $ value;
cards;
,14,1,13
/25/Q9/15
~6~D~20
run;
```

❹ This simple example demonstrates a delimiter that varies by observation. Here the first character of each line is the delimiter that is to be used in that line. The delimiter is read, stored, and then used on the INFILE statement. Here we are taking advantage of the executable nature of the INFILE statement.

Using DLMSTR

Unlike the DLM option, which designates one or more delimiters, the DLMSTR option declares a specific list of characters to use as a delimiter. Here the delimiter is the sequence of characters comma-comma-slash (,/,). Notice in the LISTING of the IMPORT data set, that extra commas and slashes are read as data.

```
data imports;
infile cards dlmstr=',/, ';
input id importcode $ value;
cards;
14,,/1/,/13
25,,/Q9,,/15
6,,/D,,/20
run;
```

1.3.3g Use a delimiter string

Obs	id	importcode	value
1	14	1/	13
2	25	Q9,	15
3	6	,D	20

SEE ALSO

The following SAS Forum thread discussed the use of the DLM and DLMSTR options <http://communities.sas.com/message/46192>. The use of the tilde when writing data was discussed on the following forum thread: <http://communities.sas.com/message/57848>. The INFILE and FILE statements are discussed in more detail by First (2008).

1.3.4 Reading Variable-Length Records

For most raw data files, including the small ones shown in most of the preceding examples, the number of characters on each row has not been consistent. Inconsistent record length can cause problems with lost data and incomplete fields. This is especially true when using the formatted style of input. Fortunately there are several approaches to reading this kind of data successfully.

The Problem Is

Consider the following data file containing a list of patients. Unless it has been built and defined as a fixed-length file, which is very unlikely on most operating systems including Windows, each record has a different length. The individual records physically stop after the last non-blank character. When we try to read the last name on the third row (Rachel's last name is unknown), we will be attempting to read past the end of the physical record and there will almost certainly be an error.

F	Linda	Maxwell
M	Ronald	Mercy
F	Rachel	
M	Mat	Most
M	David	Nabers
F	Terrie	Nolan
F	June	Olsen
M	Merv	Panda
M	Mathew	Perez
M	Robert	Pope
M	Arthur	Reilly
M	Adam	Robertson

The following code attempts to read the above data. However, we have a couple of problems.

```
filename patlist "&path\data\patientlist.txt";
data patients;
  infile patlist;
  input @2 sex $1.
        @8 fname $10.
        @18 lname $15.;
run;
title '1.3.4a Varying Length Records';
proc print data=patients;
run;
```

The LOG shows two notes; there is a LOST CARD and the INPUT statement reached past the end of the line.

NOTE: LOST CARD.

sex=M fname=Adam lname= _ERROR_=1 _N_=6

NOTE: 12 records were read from the infile PATLIST.

The minimum record length was 13.

The maximum record length was 26.

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

The resulting data set has a number of data problems. Even a quick inspection of the data shows that the data fields have become confused.

1.3.4a Varying Length Records

Obs	sex	fname	lname
1	F	Linda	M Ronald
2	F	M Mat	M David
3	F	Terrie	F June
4	M	Merv	M Mathew
5	M	Robert	M Arthur

Our INPUT statement requests SAS to read 15 spaces starting in column 18; however, there are never 15 columns available (the longest record is the last – Robertson – with a last name of 9 characters. To fill our request, it skips to column 1 of the next physical record to read the last name. When this happens the notes mentioned in the LOG are generated.

INFILE Statement Options (TRUNCOVER, MISSOVER)

Two INFILE statement options can be especially useful in controlling how SAS handles *short* records.

- **MISSOVER** Assigns missing values to variables beyond the end of the physical record. Partial variables are set to missing.
- **TRUNCOVER** Assigns missing values to variables beyond the end of the physical record. Partial variables are truncated, but not necessarily set to missing.
- **FLOWOVER** SAS finishes the logical record using the next physical record. This is the default.

```
title '1.3.4b Varying Length Records';
title2 'Using TRUNCOVER';
data patients(keep=sex fname lname);
  infile patlist truncover;
  input @2 sex $1.
        @8 fname $10.
        @18 lname $15.;
run;
```

The TRUNCOVER option is specified and as much information as possible is gathered from each record; however, SAS does not go to the next physical record to complete the observation.

1.3.4b Varying Length Records Using TRUNCOVER

Obs	sex	fname	lname
1	F	Linda	Maxwell
2	M	Ronald	Mercy
3	F	Rachel	
4	M	Mat	Most
5	M	David	Nabers
6	F	Terrie	Nolan
7	F	June	Olsen
8	M	Merv	Panda
9	M	Mathew	Perez
10	M	Robert	Pope
11	M	Arthur	Reilly
12	M	Adam	Robertson

Generally the TRUNCOVER option is easier to apply than the \$VARYING informat, and there is no penalty for including a TRUNCOVER option on the INFILE statement even when you think that you will not need it.

By including the TRUNCOVER option on the INFILE statement, we have now correctly read the data without skipping a record, while correctly assigning a missing value to Rachel's last name.

Using the \$VARYING Informat

The \$VARYING informat was created to be used with variable-length records. This informat allows us to determine the record length and then use that length for calculating how many columns to read. As a general rule, you should first attempt to use the more flexible and easier to apply TRUNCOVER option on the INFILE statement, before attempting to use the \$VARYING informat.

Unlike other informats \$VARYING utilizes a secondary value to determine how many bytes to read. Very often this value depends on the overall length of the record. The record length can be retrieved with the LENGTH= option ❶ and a portion of the overall record length is used to read the field with a varying width.

The classic use of the \$VARYING informat is shown in the following example, where the last field on the record has an inconsistent width from record to record. This is also the type of data

read for which the TRUNCOVER option was designed.

```
title2 'Using the $VARYING Informat';
data patients(keep=sex fname lname);
  infile patlist length=len ❶;
  input @; ❷
  namewidth = len-17; ❸
  input @2 sex $1.
        @8 fname $10.
        @18 lname $varying15. namewidth ❹;
run;
```

❶ The LENGTH= option on the INFILE statement specifies a temporary variable (LEN) which holds the length of the current record.

1.3.4c Varying Length Records Using the \$VARYING Informat

Obs	sex	fname	lname
1	F	Linda	Maxwell
2	M	Ronald	Mercy
3	F	M	Mat ❺
4	M	David	Nabers
5	F	Terrie	Nolan
6	F	June	Olsen
7	M	Merv	Panda
8	M	Mathew	Perez
9	M	Robert	Pope
10	M	Arthur	Reilly
11	M	Adam	Robertson

❷ An INPUT statement with just a trailing @ is used to load the record into the input buffer. Here the length is determined and loaded into the variable LEN. The trailing @ holds the record so that it can be read again.

❸ The width of the last name is calculated (total length less the number of characters to the left of the name). The variable NAMEWIDTH holds this value for use by the \$VARYING informat.

④ The width of the last name field for this particular record follows the \$VARYING15. informat. Here the width used with the \$VARYING informat is the widest possible value for LNAME and also establishes the variable's length.

Inspection of the resulting data shows that we are now reading the correct last name; however, we still have a data issue ⑤ for the third and fourth input lines. Since the third data line has *no* last name, the \$VARYING informat jumps to the next data record. The TRUNCOVER option on the INFILE statement discussed above addresses this issue successfully.

In fact for the third record the variable FNAME, which uses a \$10 informat, reaches beyond the end of the record and causes the data to be misread.

```
data patients(keep=sex fname lname namewidth ⑩);
  length sex $1 fname $10 lname $15; ⑥
  infile patlist length=len;
  input @;
  if len lt 8 then do; ⑦
    input @2 sex $;
  end;
  else if len le 17 then do; ⑧
    namewidth = len-7;
    input @2 sex $
          @8 fname $varying. namewidth;
  end;
  else do; ⑨
    namewidth = len-17;
    input @2 sex $
          @8 fname $
          @18 lname $varying. namewidth; ⑩
  end;
run;
```

⑥ Using a LENGTH statement to declare the variable lengths avoids the need to add a width to the informats.

⑦ Neither a first or last name is included. This code assumes that a gender (SEX) is always present.

⑧ The record is too short to have a last name, but must contain a first name of at least one letter.

⑨ The last name must have at least one letter.

⑩ The variable

NAMEWIDTH will contain the width of the rightmost variable. The value of this variable is generally of no interest, but it is kept here so that you can see its values change for each observation.

It is easy to see that the \$VARYING informat is more difficult to use than either the TRUNCOVER or the MISSOVER options. However, the \$VARYING informat can still be helpful. In the following simplified example suggested by John King there is no delimiter and yet the columns are not of constant width. To make things more interesting the variable with the inconsistent width is not on the end of the input string.

```
data datacodes;
  length dataname $15;
  input @1 width 2.
        dataname $varying. width
        datacode :2.;
  datalines;
5 Demog43
2 AE65
13lab_chemistry32
run;
```

The first field (WIDTH) contains the number of characters in the second field (DATANAME). This value is used with the \$VARYING informat to correctly read the data set name while not reading past the name and into the next field (DATACODE).

SEE ALSO

Cates (2001) discusses the differences between MISSOVER and TRUNCOVER. A good comparison of these options can also be found in the SAS documentation

<http://support.sas.com/documentation/cdl/en/basess/58133/HTML/default/viewer.htm#a002645812.htm>.

SAS Technical Support example #37763 uses the \$VARYING. informat to write a zero-length string in a REPORT example <http://support.sas.com/kb/37/763.html>.

1.4 Writing Delimited Files

Most modern database systems utilize metadata to make the data itself more useful. When transferring data to and from Excel, for instance, SAS can take advantage of this metadata. Flat files do not have the advantage of metadata and consequently more information must be transferred through the program itself. For this reason delimited data files should not be our first choice for transferring information from one database system to another. That said we do not always have that choice. We saw in Section 1.3 a number of techniques for reading delimited data.

Since SAS already knows all about a given SAS data set (it has access to the metadata), it is much more straightforward to write delimited files.

MORE INFORMATION

Much of the discussion on reading delimited data also applies when writing delimited data (see Section 1.3).

1.4.1 Using the DATA Step with the DLM= Option

When reading delimited data using the DATA step, the INFILE statement is used to specify a number of controlling options. Writing the delimited file is similar; however, the FILE statement is used. Many of the same options that appear on the INFILE statement can also be used on the FILE statement. These include:

- DLM=
- DLMSTR=
- DSD

While the DSD option by default implies a comma as the delimiter, there are differences between the uses of these two options. The DSD option will cause values which contain an embedded delimiter character to be double quoted. The DSD option also causes missing values to appear as two consecutive delimiters, while the DLM= alone writes the missing as either a period or a blank.

In the following example three columns from the ADVRPT.DEMOG data set are to be written to the comma separated variable (CSV) file. The FILE statement is used to specify the delimiter

```
filename outspot "&path\data\E1_4_1demog.csv";

data _null_;
  set advrpt.demog(keep=fname lname dob);
  file outspot dlm=',' ❶
          dsd; ❷
  if _n_=1 then put 'FName,LName,DOB'; ❸
  put fname lname dob mmddyy10.; ❹
run;
```

using the DLM= ❶ option. Just in case one of the fields contains the delimiter (a comma in this example), the Delimiter Sensitive Data

option, DSD ❷, is also included. Using the DSD option is a good general practice.

When you also want the first row to contain the column names, a conditional PUT ❸ statement can be used to write them. The data itself is also written using a PUT statement ❹.

MORE INFORMATION

The example in Section 1.4.4 shows how to insert the header row without explicitly naming the variables.

All the variables on the PDV can be written by using the statement PUT (_ALL_) (:); (see Section 1.4.5).

1.4.2 PROC EXPORT

Although a bit less flexible than the DATA step, the EXPORT procedure is probably easier to use for simple cases. However, it has some characteristics that make it ‘not so easy’ when the data are slightly less straightforward.

The EXPORT step shown here is intended to mimic the output file generated by the DATA step in Section 1.4.1; however, it is not successful and we need to understand why.

```
filename outspot "&path\data\E1_4_2demog.csv";

proc export data=advrpt.demog(keep=fname lname dob) ❶
  outfile=outspot ❷
  dbms=csv ❸ replace;
  delimiter=','; ❹
run;
```

❶ Three variables have been selected from ADVRPT.DEMOG and EXPORT is used to create a CSV file.

❷ The OUTFILE= option points to the *fileref* associated with the file to be created. Notice that the extension of the file’s name matches the selected database type ❸.

❸ The DBMS= option is used to declare the type for the generated file. In this case a CSV file. Other choices include TAB and DLM (and others if one of the SAS/ACCESS products has been licensed).

❹ The DELIMITER= option is used to designate the delimiter. It is not necessary in this example as the default delimiter for a CSV file is a comma. This option is most commonly used when DBMS is set to DLM and something other than a space, the default delimiter for DBMS=DLM, is desired as the delimiter.

A quick inspection of the file generated by the PROC EXPORT step shows that **all** the variables from the ADVRPT.DEMOG data set have been included in the file; however, only those variables in the KEEP= data set option have values. Data set options **1** cannot be used with the incoming data set when EXPORT creates delimited data. Either you will need to write all the variables or the appropriate variables need to be selected in a previous step (see Section 1.4.3). This behavior is an artifact of the way that PROC EXPORT writes the delimited file. PROC EXPORT writes a DATA step and builds the variable list from the metadata, ignoring the data set options. When the data are actually read into the constructed DATA step; however, the KEEP= data set option is applied, thus resulting in the missing values.

```
subject,clinnum,lname,fname,ssn,sex,dob,death,race,edu,wt,ht,symp,death2
,,Adams,Mary,,,12AUG51,,,,,,
,,Adamson,Joan,,,,,,
,,Alexander,Mark,,,15JAN30,,,,,
,,Antler,Peter,,,15JAN34,,,,,
,,Atwood,Teddy,,,14FEB50,,,,,
.... data not shown ....
```

1.4.3 Using the %DS2CSV Macro

The DS2CSV.SAS file is a macro that ships with Base SAS, and is accessed through the SAS autocall facility. Its original authorship predates many of the current capabilities discussed elsewhere in Section 1.4. The macro call is fairly straightforward; however, the macro code itself utilizes SCL functions and lists and is outside the scope of this book.

The macro is controlled through the use of a series of named or keyword parameters. Only a small

subset of this list of parameters is shown here.

```
data part;
  set advrpt.demog(keep=fname lname dob); 1
run;

%ds2csv(data=part, 2
         runmode=b, 3
         labels=n, 4
         csvfile=&path\data\E1_4_3demog.csv) 5
```

1 As was the case with PROC EXPORT in Section 1.4.2, if you need to eliminate observations or columns a separate step is required.

2 The data set to be processed is passed to the macro.

3 The macro can be executed on a server by using RUNMODE=Y.

4 By default the variable labels are used in the column header. Generally you will want the column names to be passed to the CSV file. This is done using the LABELS= parameter.

5 The CSVFILE= parameter is used to name the CSV file. This parameter does not accept a *fileref*.

SEE ALSO

A search of SAS documentation for the macro name, DS2CSV, will surface the documentation for this macro.

1.4.4 Using ODS and the CSV Destination

The Output Delivery System, ODS, and the CVS tagset can be used to generate CSV files. When you want to create a CSV file of the data, complete with column headers, the CSV destination can be used in conjunction with PROC PRINT.

```
ods csv file="&path\data\E1_4_4demog.csv" ❶
      options(doc='Help' ❷
              delimiter=";"); ❸
proc print data=advrpt.demog
      noobs; ❹
      var fname lname dob; ❺
run;
ods csv close; ❻
```

❶ The new delimited file is specified using the FILE= option.

❷ TAGSET options are specified in the OPTIONS list. A list of available options can

be seen using the DOC='HELP' option.

❸ The delimiter can be changed from a comma with the DELIMITER= option.

❹ The OBS column is removed using the NOOBS option.

❺ Select variables and variable order using the

```
"fname";"lname";"dob"
"Mary";"Adams";"12AUG51"
"Joan";"Adamson";"."
"Mark";"Alexander";"15JAN30"
"Peter";"Antler";"15JAN34"
"Teddy";"Atwood";"14FEB50"
.... data not shown ....
```

VAR statement in the PROC PRINT step.

❻ As always be sure to close the destination.

MORE INFORMATION

Chapter 11 discusses a number of aspects of the Output Delivery System.

SEE ALSO

There have been several SAS forum postings on the CSV destination.

<http://communities.sas.com/message/29026#29026>

<http://communities.sas.com/message/19459>

1.4.5 Inserting the Separator Manually

When using the DATA step to create the delimited file, the techniques shown in Section 1.4.1 will generally be sufficient. However you may occasionally require more control, or you may want to take control of the delimiter more directly.

One suggestion that has been seen in the literature uses the PUT statement to insert the delimiter.

```
data _null_;
  set advrpt.demog(keep=fname lname dob);
  file csv_a;
  if _n_=1 then put 'FName,LName,DOB';
  put (_all_)(' '); ❶
run;
```

Here the `_ALL_` variable list shortcut has been used to specify that all variables are to be written. This shortcut list requires a corresponding text, format, or other modifier for each of the variables. In this case we have specified a comma, e.g., `('')` ❶.

This approach will work to some extent, but it is not perfect in that a comma precedes each line of data.

The DSD option on the FILE statement ❷ implies a comma as the delimiter, although the DLM= option can be used to specify a different option (see Section 1.4.1). The `_ALL_` list abbreviation can still be used; however, a neutral modifier must also be selected. Either the colon (:) or the question mark (?) ❸, will serve the purpose.

```
data _null_;
  set advrpt.demog(keep=fname lname dob);
  file csv_b dsd; ❷
  if _n_=1 then put 'FName,LName,DOB';
  put (_all_) (?) ❸;
run;
```

Because the DSD option has been used, an approach such as this one will also work when one or more of the variables contain an embedded delimiter.

1.5 SQL Pass-Through

SQL pass through allows the user to literally pass instructions through a SAS SQL step to the server of another database. Passing code or SQL instructions out of the SQL step to the server can have a number of advantages, most notably significant efficiency gains.

1.5.1 Adding a Pass-Through to Your SQL Step

The pass-through requires three elements to be successful:

- A connection must be formed to the server/database. ❶
- Code must be passed to the server/database. ❷
- The connection must be closed. ❸

These three elements will be formulated as statements (❶ CONNECT and ❸ DISCONNECT) or as a clause within the FROM CONNECTION phrase ❷.

```
proc sql noprint;
  connect to odbc (dsn=clindat uid=Susie pwd=pigtails); ❶
  create table stuff as select * from connection to odbc (
    select * from q.org ❷
    for fetch only
  );
  disconnect from odbc; ❸
quit;
```

The connection that is established using the CONNECT statement ❶ and is then referred to in the FROM CONNECTION TO phrase.

Notice that the SQL code that is being passed to the database, not a SAS database, ❷ is within the parentheses. This code must be appropriate for the receiving database. In this case the pass through is to a DB2 table via an ODBC connection.

There are a number of types of connections and while ODBC connections, such as the one established in this example, are almost universally available in the Microsoft/Windows world, they are typically slower than SAS/ACCESS connections.

1.5.2 Pass-Through Efficiencies

When using PROC SQL to create and pass database-specific code to a database other than SAS, such as Oracle or DB2, it is important that you be careful with how you program the particular problem. Depending on how it is coded SQL can be very efficient or very inefficient, and this can be an even more important issue when you use pass-through techniques to create a data subset.

Passing information back from the server is usually slower than processing on the server. Design the pass-through to minimize the amount of returned information. Generally the primary database will be stored at a location with the maximum processing power. Take advantage of that power. At the very least minimizing the amount of information that has to be transferred back to you will help preserve your bandwidth.

In SQL, data sets are processed in memory. This means that large data set joins should be performed where available memory is maximized. When a join becomes memory bound subsetting the data before the join can be helpful. Know and understand your database and OS, some WHERE statements form clauses that are applied to the result of the join rather than to the incoming data set.

Even when you do not intend to write to the primary database that is being accessed using an SQL pass-through, extra process checking may be involved against that data table. These checks, which can be costly, can potentially be eliminated by designating the incoming data table as read-only. This can be accomplished in a number of ways. In DB2 using the clause `for fetch only` in the code that is being passed to the database eliminates write checks against the incoming table. In the DB2 pass-through example in Section 1.5.1 we only want to extract or fetch data. We speed up the process by letting the database know that we will not be writing any data – only fetching it.

MORE INFORMATION

An SQL step using pass-through code can be found in Section 5.4.2.

1.6 Reading and Writing to XML

Extensible Markup Language, XML, has a hierarchical structure while SAS data sets are record or observation based. Because XML is fast becoming a universal data exchange format, it is incumbent for the SAS programmer to have a working knowledge of how to move information from SAS to XML and from XML to SAS.

The XML engine (Section 1.6.2) was first introduced in Version 8 of SAS. Later the ODS XML destination was added; however, currently the functionality of the XML destination has been built into the ODS MARKUP destination (see Section 1.6.1).

Because XML is text based and each row contains its own metadata, the files themselves can be quite large.

SEE ALSO

A very nice overview of XML and its relationship to SAS can be found in (Pratter, 2008). Other introductory discussions on the relationship of XML to SAS include: Chapal (2003), Palmer (2003 and 2004), and in the SAS documentation on “XML Engine with DATA Step or PROC COPY”.

1.6.1 Using ODS

You can create an XML file using the ODS MARKUP destination. The file can contain procedure output in XML form, and this XML file can then be passed to another application that utilizes /

reads XML. By default the MARKUP destination creates a XML file.

```
title1 '1.6.1 Using ODS MARKUP';
ods markup file="&path\data\E1_6_1Names.xml"; ❶

* create a xml file of the report; ❷
proc print data=advrpt.demog;
  var lname fname sex dob;
run;
ods markup close; ❸
```

❶ The FILE= option is used to designate the name of the file to be created. Notice the use of the XML extension.

❷ The procedure must be

within the ODS 'sandwich.'

❸ The destination must be closed before the file ❶ can be used outside of SAS.

MORE INFORMATION

If the application that you are planning to use with the XML file is Excel, the EXCELXP tagset is a superior choice (see Section 11.2).

SEE ALSO

The LinkedIn thread

http://www.linkedin.com/groupItem?view=&srctype=discussedNews&gid=70702&item=74453221&type=member&trk=eml-anet_dig-b_pd-ttl-cn&ut=34c4-P0gjofkY1

follows a discussion of the generation of XML using ODS.

1.6.2 Using the XML Engine

The use of the XML engine is a process similar to the one shown in Section 1.6.1, and can be used to write to the XML format. XML is a markup language and XML code is stored in a text file that

can be both read and written by SAS. As in the example above, an engine is used on the LIBNAME statement to establish the link with SAS that performs the conversion. A *fileref* is established and it is used in the LIBNAME statement.

```
filename xml1st "&path\data\E1_6_2list.xml";

libname toxml xml xmlfileref=xml1st; ❶

* create a xml file (E1_6_2list.xml);
data toxml.patlist; ❷
  set advrpt.demog(keep=lname fname sex dob);
run;

* convert xml to sas;
data fromxml;
  set toxml.patlist; ❸
run;
```

❶ On the LIBNAME statement that has the XML engine, the XMLFILEREf= option is used to point to the

fileref either containing the XML file or, as is the case in this example, the file that is to be written.

```

<?xml version="1.0" encoding="windows-1252" ?>
- <TABLE>
- <PATLIST> ❸
  <lname>Adams</lname>
  <fname>Mary</fname>
  <sex>F</sex>
  <dob>1951-08-12</dob>
</PATLIST>
- <PATLIST>
  <lname>Adamson</lname>
  <fname>Joan</fname>
  <sex>F</sex>
  <dob missing="." />
</PATLIST>
... the remaining observations are not shown ...

```

has been written using the `missing=` notation.

SEE ALSO

Hemedinger and Slaughter (2011) briefly describe the use of XML and the XML Mapper.

❷ The *libref* TOXML can be used to both read and write the XML file. The name of the data set (PATLIST) is recorded as a part of the XML file ❸. This means that multiple SAS data sets can be written to the same XML file.

The selected variables are written to the XML file. Notice that the variables are named on each line and that the date has been re-coded into a YYYY-MM-DD form, and that the missing DOB for ‘Joan Adamson’

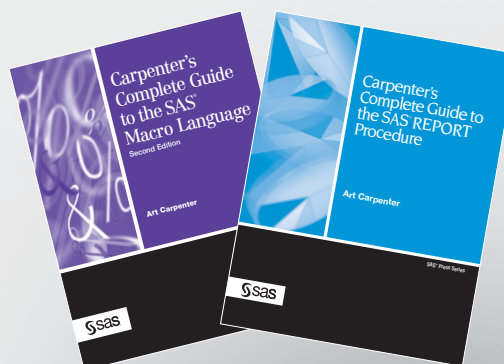
Do you like what you just read?

Buy *Carpenter's Guide to Innovative SAS® Techniques* today!



Learn more about esteemed SAS expert and writer Art Carpenter, read free chapters from all of his books, and access example code and data on [Art Carpenter's Author Page](#).

Find [additional books](#) that are just right for you.



Browse and search [free SAS documentation](#) sorted by release and by product.

Email us: sasbook@sas.com

Call: 1-800-727-3228



THE
POWER
TO KNOW®