S for SAS

A Beginner's Guide

Prepared by Jeomoan Kurian (Jeomoan.kurian@gecis.ge.com) strictly for the internal training use of GECF team.

Preface

SAS is one language everyone understands in Consumer Finance! That explains why an Analyst should learn it for an effective communication. This guide is an effort to equip a beginner with basic SAS skills in a week's time.

This guide attempts to:

1. Simplify SAS for the beginners, trimming the possibilities down to a bare minimum.
2. Give an introduction to the possibilities SAS offers for data extraction, analysis and reporting.
3. Customize the content to Consumer Finance environment and the way we use SAS.
4. Provide more examples and step-by-step instruction to the readers to practice SAS programs.

Target audience is expected to have basic computer skills like working with a spreadsheet or word processor. Also the user should know how to edit a program in a Windows SAS System. No statistics background is expected.

Your suggestions to improve this guide are most welcome!

Contents

Introduction

The SAS System, originally Statistical Analysis System, is an integrated system of software products provided by the SAS Institute that enables a user to perform:

- Data entry, retrieval, management, and mining
- Report writing and graphics
- Statistical Analysis and Operations Research
- Forecasting and Decision Support
- Data Warehousing (Extract, Transform, Load)

Though SAS system provides a menu driven interface, most of the interaction with SAS system in ACoE is done through writing SAS programs. SAS programs provide high level of flexibility to the user. Also, platforms like Unix and Mainframe do not provide a menu driven interface for SAS.

A SAS program is composed of two fundamental components:

DATA step(s)- the part of the program in which a structure for the data to be analyzed is created. Variables corresponding to the various elements of the data set are defined, and the data are assigned to the Variables. Data may be input manually in the body of the program, or they may be read in from a file. Additionally data may be stored in a data warehouse (for example Consumer Data Warehouse in Stoner server).

PROCs (PROCedures) - the SAS language is organized into a series of procedures, or PROCs, each of which is dedicated to a particular form of data manipulation or statistical analysis to be performed on data sets created in the DATA step. For example:

PROC PRINT: Prints the contents of a data set and create reports.
PROC FREQ: Produces frequency and cross tabulation tables on the variables specified.
PROC MEANS: Computes means, standard deviations and other summary Statistics for some or all of the variables in a data set.
PROC TTEST: Computes the 2-sample t-test for comparing the means of 2 treatments.
PROC REG: Performs regression analysis using the method of least squares.
PROC GPLOT: Constructs plots of the data as specified by the user.

A SAS program consists of one or more DATA steps to get the data into a format that SAS can understand and one or more calls to PROCs to perform various analyses on the data.

SAS Language in Nutshell

To use SAS it is necessary that you are acquainted with the scripting known as SAS language. SAS programs are written using SAS language to manipulate, clean, describe and to do data analysis.

A SAS program consists of a series of DATA, data transformation and PROCedure statements. An entry level SAS user should at least know how to use the following SAS statements to have a control over the language.

DATA;
INPUT;
CARDS;
TITLE;
LABEL;
FORMAT;
IF / THEN; ELSE;
WHERE;
SET;
SORT;
MERGE;
PROC PRINT;
PROC FREQ;
PROC MEANS;
PROC GPLOT;
PROC SQL;

In the chapters that follow, you get more familiar with these statements/procedures and various features and options that are available with each one of them.

First Steps with Data and SAS

Objectives of this chapter is to:

- Understand the terms: data value, variable, observation, and data set.
- Understand the rules for writing SAS statements and for naming variables and data sets.

Let us start with a generic example to understand how SAS reads and understands data. Most of the times, in our environment, the data is extracted from a data warehouse. But many a times we need to create our own datasets for testing certain procedures or functions so this learning comes handy.

Below is the table that provides some information about 10 accounts holders of a credit card company. Information includes account number, Account Open Date, Account Status Code and current Credit Limit. Now let us look at some data features.

Account #	Open Date	Status Code	Credit Limit
1234670	11-Sep-04	Z	2000
1234671	12-Sep-04		3000
1234672	13-Sep-04	Z	2500
1234673	14-Sep-04	T	3200
1234674	15-Sep-04		8000
1234675	16-Sep-04	D	2000
1234676	17-Sep-04		4000
1234677	18-Sep-04	S	6000
1234678	19-Sep-04	T	8000
1234679	20-Sep-04	T	2000

Data Value, Variable, Observations, and Dataset

DATA VALUE

Data value is the basic unit of information. In the field containing information about the account holder's credit limit, the DATA VALUES are 2000, 3000 etc. The DATA VALUES in the field 'status code' are 'Z','D', 'S' and 'T'.

VARIABLE

A set of data values that describes a given attribute makes up a VARIABLE. Each column of data values is a VARIABLE. For example, the first column in our data set is reserved for the VARIABLE we'll call Account #. It has all the account numbers of the sample we have for the credit card holders

SAS variables are of 2 types - numeric and character. Values of numeric variables can only be numbers or a period (.) for missing data. Character variables can be made up of letters and special characters such as plus signs, dollar signs, colons and percent signs, as well as numeric digits.

In the sample data above, account number and credit limit are numeric variables and status code is a character variable. Open date is a field that deserves special mention. In SAS, dates are stored as numeric but displayed in various format using SAS formats.

OBSERVATION

All the data values associated with a case, a single entity, a subject, an individual, a year, or a record and so on, make up an OBSERVATION. Each row of the data table (or Matrix) represents one OBSERVATION. The row below represents all the data values associated with OBSERVATION #1.

Account #	Open Date	Status Code	Credit Limit
1234670	11-Sep-04	Z	2000

DATA SET

A DATA SET is a collection of data values usually arranged in a rectangular table (or matrix).

A SAS DATA SET is the special way that SAS organizes and stores the data. For example, if we convert our sample into a SAS dataset we will have a data set with 4 columns (fields) and 10 rows with 3 numeric fields and 1 character field.

The DATA step creates the SAS data set and the PROC steps are instructions indicating how the SAS data set is to be manipulated or analyzed. There are certain procedures where the outcome of their execution results in creation of one or many datasets.

Rules for SAS names

Among the kinds of SAS names that appear in SAS statements are variables names, SAS data sets, formats, procedures, options, and statement labels.

1. Many SAS names can be 32 characters long; others have a maximum length of 8.
2. The first character must be a letter (A, B, C, . . ., Z) or underscore (_). Subsequent characters can be letters, numeric digits (0, 1, . . ., 9), or underscores.
3. You can use upper or lowercase letters. SAS processes names as uppercase regardless of how you type them.

4. Blanks cannot appear in SAS names.
5. Special characters, except for the underscore, are not allowed. In file reference, you can use the dollar sign ($), pound sign (#), and at sign (@).
6. SAS reserves a few names for automatic variables and variable lists. For example, _N_ and _ERROR_ .

Rules for SAS statements

1. SAS statements may begin in any column of the line.
2. SAS statements must end with a semicolon (;).
3. Some SAS statements may consist of more than one line of commands.
4. A SAS statement may continue over more than one line.
5. One or more blanks should be placed between items in SAS statements. If the items are special characters such as '=', '+', '$', the blanks are not necessary.

A Simple Program Explained

Objective of this chapter:

1. Learn to use the DATA statement
2. Learn to use the INPUT statement
3. Learn to use the CARDS statement
4. Learn how to use the semicolon (;)
5. Learn how to include TITLES on your output
6. Learn RUN statement.
7. Learn to use two Procedures – PRINT and FREQ
8. How to create a permanent dataset using LIBNAME
9. Learn how to run a SAS program

Let us start with a simple SAS program. This program creates a SAS dataset names Sample accounts with 4 columns. Two of them are numeric and rests of the columns are characters. This program demonstrates the use of DATA, CARDS, and INPUT statements and also demonstrates the use of two procedures viz. PROC PRINT and PROC FREQ. Each and every component in the program is explained in detail below.

```
Libname loc 'c:\mydata';

Data sample_accounts;
INPUT Account 1-7       OpenDate $ 9-17 StatusCode $ 21-22 CreditLimit 25-29;
CARDS;
1234670         11-Sep-04       Z       2000
1234671         12-Sep-04               3000
1234672         13-Sep-04       Z       2500
1234673         14-Sep-04       T       3200
1234674         15-Sep-04               8000
1234675         16-Sep-04       D       2000
1234676         17-Sep-04               4000
1234677         18-Sep-04       S       6000
1234678         19-Sep-04       T       8000
1234679         20-Sep-04       T       2000
;
run;

data rloc.sample_accounts; /* stores data permanently*/
set sample_accounts;
run;


Proc Print data = loc.sample_accouts;
Title " Account Sample";
run;

Proc Freq data =loc.sample_accounts ;
tables statuscode;
run;
```

Note: Data, input and proc statements are case insensitive. Open date is read as character variable to simplify the example.

Proc Print and Proc Freq are used to show how these two procedures used to print data. These procedures are explained in details later in this book.

LIBNAME Statement

Use: define a SAS library name

Syntax: Libname xx <folder reference>;

Libname loc 'c:\mydata';

Libname is used to declare a data library. Sas data can be stored permanently by saving it to a library. In this program, data is stored in 'C:\mydata' using this library reference.

DATA Statement

Use: Names the SAS data set

Syntax: DATA SOMENAME;

Data sample_accounts;

Result: A temporary SAS data set named sample_accounts is created

The DATA statement signals the beginning of a DATA step. The general form of the SAS DATA statement is:

DATA SOMENAME;

The DATA statement names the data set you are creating. The name should be 1-32 characters and must begin with a letter or underscore.

INPUT Statement

Use: Defines names and order of variables to SAS

Syntax: INPUT variable variable_type column(s);

INPUT Account **1-7** OpenDate $ **9-17** StatusCode $ **21-22** CreditLimit **25-29**;

Result: Input data are defined for SAS

The INPUT statement specifies the names and order of the variables in your data. Although there are three types of INPUT statements, which can be mixed, the beginning SAS user should only be concerned with learning how to use the Column Input style.

The INPUT statement should indicate in which columns each variable might be found. In addition, the INPUT statement should indicate how many lines it takes to record all of the information for each case or observation. The general form of the SAS INPUT statement is:

```
INPUT Account 1-7  OpenDate $ 9-17  StatusCode $ 21-22 CreditLimit 25-29;
```

The variables OperDate and StatusCode are character variable as indicated by the dollar sign ($) after the variable name The other variables are numeric.

CARDS Statement

Use: Signals that input data will follow

Syntax: CARDS;

Result: Data can be processed for the SAS data set

The CARDS statement signals that the data will follow next and immediately precedes your input data lines. The general form of the CARDS statement is:

```
INPUT Account 1-7 OpenDate $ 9-17 StatusCode $ 21-22 CreditLimit 25-29;
CARDS;
1234670        11-Sep-04       Z       2000
1234671        12-Sep-04               3000
1234672        13-Sep-04       Z       2500
1234673        14-Sep-04       T       3200
1234674        15-Sep-04               8000
1234675        16-Sep-04       D       2000
1234676        17-Sep-04               4000
1234677        18-Sep-04       S       6000
1234678        19-Sep-04       T       8000
1234679        20-Sep-04       T       2000
;
run;
```

Note: If the data is contained in an external file, instead of the CARDS, you will use an INFILE statement to specify where that file resides. (Example: INFILE 'c:\accounts.txt';).

SEMICOLON

Use: Signals the end of any SAS statement

Syntax:A DATA Step or PROCedure statement; (DATA;)

```
DATA sample_accounts;

INPUT Account 1-7       OpenDate $ 9-17 StatusCode $ 21-22 CreditLimit 25-29;
CARDS;

Proc Print data = sample_accouts;
Title " Account Sample"; run;
```

Result: SAS is signaled that the statement is complete

The semicolon (;) is used as a delimiter to indicate the end of SAS statements.

TITLE Statement

Use: Puts TITLES on your output

Syntax: TITLE 'some title';

Title " Account Sample";

Result: A TITLE is added at the top of each page of the output printed.

The TITLE statement assigns a title, which appears at the top of the output page.

PROC PRINT and PROC FREQ: These are two common procedures used to print the content of the dataset created and to produce a frequency table on a status code column.

RUN Statement

Use: Instruct SAS to execute the SAS program

Syntax: RUN;

Result: The statements and procedures specified in the SAS program blocks are executed.

How to Run a SAS Program

In a windows environment there are three different windows that help in successful program execution. **Program Editor** is used to compose the programs and to execute it**, Log window** displays the log of execution and **Output window** displays the output of the program.

Many times we use remote 'signon' to log on to a Unix server and in that case the program is running is a remote server. If we are using remote sign-on from windows, log and output are automatically downloaded to the local desktop.

Once the program is executed check the log for any errors or warning. Logs will normally give a clear idea whether the syntax was used correctly or the program was successfully run.

In Unix and Mainframe the programs are composed and executed differently. Please refer a relevant manual for the same. Good news is that SAS procedures and statements that are used in these platforms are same.

Getting Data In

Objective of this Chapter

- To learn data sources and methods to read data into SAS.
- To learn how to use INPUT, INFILE, PROC IMPORT and PROC SQL for reading data.

In a Consumer Finance environment, data warehouses are commonly used to store data. But its not unusual that we will have to read data from a variety of sources like spreadsheets, text files, comma separated files, MS access tables or manually entering through program editor. Data files from credit bureaus and other external data vendors are sent to us as large data files with various formats so its important to understand how to read them.

SAS understands only SAS datasets and formats so its important to convert all kinds of data into a form which SAS understands. The data so converted are called SAS datasets. This chapter explains various ways SAS reads data to make SAS DATASETS.

Reading Data Instreams and External Files using INPUT

Reading free formatted data instream(LIST INPUT)

One of the most common ways to read data into SAS is by reading the data instream in a data step - that is, by typing the data directly into the syntax of your SAS program. This approach is good for relatively small datasets. Spaces are usually used to "delimit" (or separate) free formatted data. For example:

```
DATA sample_accounts;
INPUT Account OpenDate $ StatusCode $ CreditLimit ;
CARDS;
1234670 11-Sep-04 ZX 2000
1234671         12-Sep-04 N 3000
1234672         13-Sep-04 Z     2500
12346730        14-Sep-04 TN 3200
1234674 15-Sep-04 X 8000
1234675         16-Sep-04 D 2000
12346767        17-Sep-04 N 4000
1234677         18-Sep-04 S 6000
1234678         19-Sep-04 TS 8000
1234679         20-Sep-04 T 2000
;
RUN;
```

Reading fixed formatted data instream(COLUMN INPUT)

Fixed formatted data can also be read in-stream. Usually, because there are no delimiters (such as spaces, commas, or tabs) to separate fixed formatted data,

column definitions are required for every variable in the dataset. That is, you need to provide the beginning and ending column numbers for each variable. This also requires the data to be in the same columns for each case. For example, if we rearrange the card accounts data from above, we can read it as fixed formatted data:

```
Data sample_accounts;
INPUT Account 1-7 OpenDate $ 9-17 StatusCode $ 21-22 CreditLimit 25-
29;
CARDS;
1234670 11-Sep-04 Z 2000
1234671 12-Sep-04   3000
1234672 13-Sep-04 Z 2500
1234673 14-Sep-04 T 3200
1234674 15-Sep-04   8000
1234675 16-Sep-04 D 2000
1234676 17-Sep-04   4000
1234677 18-Sep-04 S 6000
1234678 19-Sep-04 T 8000
1234679 20-Sep-04 T 2000;
RUN;
```

Reading fixed formatted data from an external file

Suppose you are working in a windows environment and you have a text file called 'sample_accounts.txt' in 'C:\Mydata' directory. Here is what the content of the 'C:\Mydata\sample_accounts.txt' look like:

```
1234670 11-Sep-04   Z   2000
1234671 12-Sep-04       3000
1234672 13-Sep-04   Z   2500
1234673 14-Sep-04   T   3200
1234674 15-Sep-04       8000
1234675 16-Sep-04   D   2000
1234676 17-Sep-04       4000
1234677 18-Sep-04   S   6000
1234678 19-Sep-04   T   8000
1234679 20-Sep-04   T   2000;
```

This file can be read into SAS by using an INFILE statement in DATA step

```
Data sample_accounts;
INFILE "C:\Mydata\sample_accounts.txt";
INPUT Account 1-7  OpenDate $ 9-17 StatusCode $ 21-22 CreditLimit 25-29;
RUN;
```

Alternately, INFILE statement can also be used to provide reference to the datafile. This is how it will look like:

```
filename sacc "D:\Mydata\sample_accounts.txt";
Data sample_accounts;
INFILE sacc ;
INPUT Account 1-7 OpenDate $ 9-17 StatusCode $ 21-22 CreditLimit 25-
29;
RUN;
```

Reading comma delimited data from an external file

Free formatted data that is comma delimited can also be read from an external file. For example, suppose you have a comma delimited file named sample_accounts.csv (.csv stands for comma separated values) that is stored in the C:\Mydata directory of your computer.

Here's what the data in the file look like:

```
1234670,11-Sep-04,Z,2000
1234671,12-Sep-04,,3000
1234672,13-Sep-04,Z,2500
1234673,14-Sep-04,T,3200
1234674,15-Sep-04,,8000
1234675,16-Sep-04,D,2000
1234676,17-Sep-04,,4000
1234677,18-Sep-04,S,6000
1234678,19-Sep-04,T,8000
1234679,20-Sep-04,T,2000;
```

We could read the data from sample_accounts.csv into SAS by using the following method:

```
Data sample_accounts;
INFILE "C:\Mydata\sample_accounts.csv" DLM =',';
INPUT Account OpenDate $ StatusCode $ CreditLimit ;
RUN;
```

Reading Tab delimited data from an external file

Free formatted data that is TAB delimited can also be read from an external file. For example, suppose you have a tab delimited file named sample_accounts.txt that is stored in the 'C:\Mydata' directory of your computer.
Here's what the data in the file look like:

```
1234670     11-Sep-04   Z     2000
1234671     12-Sep-04         3000
1234672     13-Sep-04   Z     2500
1234673     14-Sep-04   T     3200
1234674     15-Sep-04         8000
1234675     16-Sep-04   D     2000
1234676     17-Sep-04         4000
1234677     18-Sep-04   S     6000
1234678     19-Sep-04   T     8000
1234679     20-Sep-04   T     2000
```

We could read the data from 'sample_accounts.txt' into SAS by the following method:

```
Data sample_accounts;
INFILE "C:\Mydata\sample_accounts.txt" DLM='09'x
INPUT Account  OpenDate $  StatusCode $  CreditLimit ;
RUN;
```

Note: If your data is delimited by another character other than a blank or space, the DLM= option and/or DSD option on the INFILE statement will need to be specified. Some data files may also require additional INFILE statement options. If record lengths exceed 256 bytes then add the LRECL= option to the INFILE statement to specify a larger record length (. Also, TRUNCOVER may need to be specified on the INFILE statement to prevent SAS from reading more than one record at a time when reading variable length.

TRUNCOVER enables you to read variable-length records when some records are shorter than coded for on the INPUT statement.

Reading data using formatted input

Discussion on reading data into SAS would be incomplete without mentioning how to read the formatted input. Many a time the data we get are formatted and its necessary to read them as it is. This is especially true when we read data containing date and decimal places from text files.

Note:- With formatted input, an informat follows a variable name and defines how SAS reads the values of this variable. An informat gives the data type and the field width of an input value. Informats also read data that are stored in nonstandard form, such as packed decimal, or numbers that contain special characters such as commas. Have a look at the following example:

```
DATA acctinfo;
INPUT acctnum $8. date mmddyy10. amount comma9.;

CARDS;
0074309801/15/2001$1,003.59
1028754301/17/2001$672.05
3320899201/19/2001$702.77
0345900601/19/2001$1,209.61              ;
run;
```

Note that informats are specified in the INPUT statement to instruct SAS that the data following should be understood in that form (mmddyy is a date format and comma9. is a number format with commas to read easily). Also the positions to read are specified. For various informats for data and numbers please checkout the SAS help or Manual.

Reading data using PROC IMPORT

SAS uses a procedure called PROC IMPORT to read data from spreadsheets, DBMS files and other delimited files. Suppose you have an excel sheet named Sample_Accounts.xls in 'C:\mydata' directory, the following method could be used to read the data into a SAS dataset called Sample_accounts.

```
PROC IMPORT OUT= WORK.Sample_accounts
        DATAFILE= "C:\Mydata\Sample_Accounts.xls"
        DBMS=EXCEL2000 REPLACE;
    GETNAMES=YES;
RUN;
```

By default the first row in excels would be read as column names. PROC IMPORT would be useful when we need to read the files repetitively for reporting and other analytics purposes.

There are several DBMS specifications available. The below table summarizes that.

Identifier	Input Data Source	Extension
EXCEL2000	MS Excel Version 2000	.XLS
ACCESS	Microsoft Access database	.MDB
DBF	dBASE file	.DBF
WK1	Lotus 1 spreadsheet	.WK1
WK3	Lotus 3 spreadsheet	.WK3
WK4	Lotus 4 spreadsheet	.WK4
EXCEL	Excel Version 4 or 5 spreadsheet	.XLS
EXCEL4	Excel Version 4 spreadsheet	.XLS
EXCEL5	Excel Version 5 spreadsheet	.XLS
EXCEL97	Excel 97 spreadsheet	.XLS
DLM	delimited file (default delimiter is a blank)	.*
CSV	delimited file (comma-separated values)	.CSV
TAB	delimited file (tab-delimited values)	.TXT

Data Extraction using PROC SQL

In Consumer finance world, PROC SQL is most widely used to extract data from various data warehouses. Versatility of PROC SQL facilitates to use the RDBMS specific utilities and functions that make the data extraction process more efficient. For example we embed Oracle SQL Plus code within PROC SQL so that data exaction is most optimized by Oracle.

As we have seen earlier, any data should be converted into a SAS dataset before SAS can carry out any operations on them. The below sample show how to use PROC SQL to extract data from one of the ORACLE data warehouses. Further, each step is explained for a better understanding.

```
PROC SQL;

Connect To ORACLE(User=501115644 Password=mypasswd Buffsize=10000
Path=CDCIT1 Preserve_Comments );

CREATE TABLE acct_status AS SELECT * FROM Connection To ORACLE

(SELECT current_account_nbr AS account_number,
external_status_reason_code AS
ext_rcode,external_status AS estatus,
billing_cycle_day AS billing_cycle_day
FROM
ACCOUNT_DIM
WHERE CLIENT_ID='BROOK BROS'
AND
nvl(EXTERNAL_STATUS_REASON_CODE,'0') <>'98');

Disconnect From ORACLE;

Quit;
```

In the above example PROC SQL use the connect string "Connect To ORACLE(User=501115644 Password=mypasswd Buffsize=10000 Path=CDCIT1 Preserve_Comments)" to identify the oracle database (CDCITI) and use the user name and passwords specified to read data from it. Further, CREATE TABLE statement creates a SAS Dataset from the output of SELECT statement.

Note that here SELECT that used is an ORACLE SQL PLUS command. (a proprietary implementation of SQL by Oracle). So one can use all the features of ORACLE while querying.

When querying a data warehouse, SAS automatically converts the field formats in to SAS formats. For example, a date field in Oracle will be converted into SAS date and an Oracle Varchar field will be converted into character.

SAS datasets are used to store large amount of data in our environments. Some implementations of SAS Datasets like SAS SPDS are used for even data warehousing in Consumer Finance. Often we create SAS datasets and store them in shared drives and folders for future use.

Library references: A common way to read a SAS dataset is to declare the folder that holds data sets as a SAS library. The below statement creates a SAS Library reference.

```
Libname rserver '\projects\jkurian\mydata';

data cred_line_new;
set rserver.cred_line;

run;
```

The above program declares a library called 'rserver' and assigns the folder '\projects\jkurian\mydata'. Further a dataset 'cred_line' is assigned(SET) to a data set 'cred_line_new'

It is possible that there could be an older version of SAS dataset you need to read from an external source. In such situation Libname should explicitly declare the dataset version. For example the below code tells SAS that the files in library rserver is version 6.

```
Libname rserver v6 '\projects\jkurian\mydata';
```

Reading EBCDIC files using SAS

In Consumer Finance environment we often come across EBCDIC data files whenever we deal with credit bureaus data and credit card processors like FDR. EBCDIC is the character encoding system of mainframes and ASCII is the encoding system on other machines such as VAX, Windows, UNIX, and Macintosh. These two character sets represent the same data differently. For instance, the value '50'x is a '&' in EBCDIC, but a 'P' in ASCII. Credit bureaus like Experian or Equifax and processors like FDR uses mainframe systems and hence their data files are EBCDIC.

The below example reads an EBCDIC data file named f95_sep from 'C:\Mydata'.

```
DATA chm_rdc;
INFILE 'C:\Mydata\f95_sep' lrecl=32760 recfm =s370vb truncover ;
INPUT
@23 CHD_CLIENT_NUMBER $ebcdic4.
@27 CHD_SYSTEM_NO $ebcdic4.
@31 CHD_PRIN_BANK $ebcdic4.
@35 CHD_AGENT_BANK $ebcdic4.
```

```
@39 CHD_ACCOUNT_NUMBER $ebcdic16.
@130 CHD_EXTERNAL_STATUS $ebcdic1.
;
run;
```

Note that @ sign is a column pointer. @39 reads data from column 39.

Please refer the documentation on this at SAS support site for other options.
http://support.sas.com/techsup/technote/ts642.txt

Exercises:

1. Key in the following data into SAS Program Editor and create a SAS dataset. Read the date as mmddyy10. format and generate a PROC FREQ on the date variable. Convert the date into MONYY7. Format.

```
1234670 11-Sep-04 Z 2000
1234671 12-Sep-04   3000
1234672 13-Sep-04 Z 2500
1234673 14-Sep-04 T 3200
1234674 15-Sep-04   8000
1234675 16-Sep-04 D 2000
1234676 17-Sep-04   4000
1234677 18-Sep-04 S 6000
1234678 19-Sep-04 T 8000
1234679 20-Sep-04 T 2000
```

2. Create a file 'sample_data.txt' with the data below and save it in your computer. Read the file into SAS and create a SAS Dataset. Do a cross tab using PROC FREQ for account and open_date variables (First and second data fields)

```
1234670,11-Sep-04,Z,2000
1234671,12-Sep-04,,3000
1234672,13-Sep-04,Z,2500
1234673,14-Sep-04,T,3200
1234674,15-Sep-04,,8000
1234675,16-Sep-04,D,2000
1234676,17-Sep-04,,4000
1234677,18-Sep-04,S,6000
1234678,19-Sep-04,T,8000
1234679,20-Sep-04,T,2000;
```

Working with Datasets

Objectives of this Chapter:

- Learn about SAS Variables.
- Learn how to LABEL, RENAME and FORMAT data.
- Learn SAS commonly used functions for data processing.
- Learn how to MERGE datasets.
- Learn conditional processing using WHERE, IF/ELSE and DO-END

The previous chapters demonstrated how a simple SAS program looks like and how to read data from various sources into SAS. We did come across concepts like informats, formats and some procedures in SAS. Let us discuss some of these concepts in detail so that we understand some frequently used functions and methods used for data processing – to clean, format and combine/subset data to make it ready for analytics purposes- in our business environment.

SAS variables

Declaration, assignment, length, keep, drop, array, PROC contents, label, macro variables and scope of variables.

SAS variables are containers that you create within a program to store and use character and numeric values. There are two types of variables–Character and Numeric. Characters are variables of type character that contain alphabetic characters, numeric digits 0 through 9, and other special characters. Numeric variables
are variables of type numeric that are stored as floating-point numbers, including dates and times. Yes SAS stores date and time as Numbers.

To simplify, each and every field/column in a SAS dataset is a SAS variable.

These are the questions we will discuss in this chapter around SAS variables.

i) How to create a variable and type?
ii) How to decide length of a variable?
iii) How to keep or drop a variable(s) from a dataset?
iv) What are array variables and how to declare them?
v) How do we know the type of variables in an already created dataset?
vi) How to label and apply format to a variable?
vii) What's a macro variable and how to declare them?
viii) What's scope of a variable in a SAS program?

How to create a variable

There are four ways we commonly use to create a variable. 1. Using an assignment statement 2. Using and **INPUT** statement 3. Through a **LENGTH** statement and 4. As a result of a **PROC SQL/PROC IMPORT**. There are many other ways too but we limit to these four types, as they are used 90% of the times in our programs.

1. Using an assignment statement

This is the most common form of variable creation. Its not necessary that the variables should be declared well in advance.

Have a look at the following program:

```
Data var_test;
id ='JK';
NProducts= 6;
pro_price = 4.555;
tot_cost = NProducts*pro_price;
final_price = tot_cost;
run;

proc print data=var_test ;
run;

proc contents data =var_test;run;
```

The above program creates a dataset 'var_test' with five variables. As we have seen earlier, each variable will form a column/field in the dataset. 'Id' is assigned with a value of 'JK'. Nproducts and pro_price are assigned with numbers where latter is a decimal. Tot_cost is the variable that takes the value of a product of two other variables. And finally, final price variable is assigned with another variable in the dataset ie tot_cost.

PROC Print prints the data sets and all variables and this is how the output looks like:

```
                                    pro_          final_
 Obs    id    NProducts    price   tot_cost      price
 ------------------------------------------------------

  1      JK       6       4.55      27.3          27.3
```

PROC contents is the procedure used to know the data types, length and label of the variables in a dataset.

```
      -----Alphabetic List of Variables and Attributes-----

       #    Variable          Type    Len       Pos
       ------------------------------------------------
       2    NProducts         Num       8       0
       5    final_price       Num       8       24
       1    id                Char      2       32
       3    pro_price         Num       8       8
       4    tot_cost          Num       8       16
```

These outputs together tell us how the variable creations are done and what values, types and lengths are assigned by SAS. Now let us discuss some general rules of variable creation by assignment.

In a DATA step, you can create a new variable and assign it a value by using it for the first time on the left side of an assignment statement. SAS determines the length of a variable from its first occurrence in the DATA step. The new variable gets the same type and length as the expression on the right side of the assignment statement.

When the type and length of a variable are not explicitly set, SAS gives the variable a default type and length as shown in the examples in the following table.

Expression	Example	Resulting Type of X	Resulting Length of X	Explanation
Numeric variable	**a=34** **x=a;**	Numeric variable	8	Default numeric length (8 bytes unless otherwise specified)
Character variable	**a='ABCD'** **x=a;**	Character variable	4	Length of source variable
Character literal	**x='ABC';** **x='ABCDE';**	Character variable	3	Length of first literal encountered

Practical problems: Many a time the length of the variable is not sufficient to hold the value encountered during the data processing. This will lead to SAS truncating the variable in to the length of the variable created. This problem can be solved with declaring the length of the variable before assignment.

Creating variables with INPUT Statement

We have already seen how to use INPUT to read data into variables. We have also seen how to use SAS informats to tell SAS what kind of data its reading. Below reproduced is one example to show how its done.

```
DATA acctinfo;
INPUT acctnum $8. date mmddyy10. amount comma9.;

CARDS;
0074309801/15/2001$1,003.59
1028754301/17/2001$672.05
3320899201/19/2001$702.77
0345900601/19/2001$1,209.61
;
run;
proc contents data =acctinfo;run;
```

Output :

```
Alphabetic List of Variables and Attributes

    #    Variable    Type    Len    Pos
    ffffffffffffffffffffffffffffffff
    1    acctnum     Char      8     16
    3    amount      Num       8      8

    2    date        Num       8      0
```

Here INPUT statement specifies the data type next to the variable and also how many positions (length).

Specifying a New Variable in a LENGTH Statement

In practical situations, when we create new variables, the length of the variable needs to be explicitly defined. For example, when we read two character values successively into a variable, SAS assigns the length of the variable as that of the first. Suppose the second value is longer than the first, SAS reads only up to the length of first variable. So it's a good programming practice to declare the variable with a LENGTH statement so that we are sure it can hold all kinds of values the data has.

You can use the LENGTH statement to create a variable and set the length of the variable. Let us modify our earlier example:

```
Data var_test;
length id $ 10;
length NProducts 4;
id ='JK';
NProducts= 6;
pro_price = 4.55;
tot_cost = NProducts*pro_price;
final_price = tot_cost;
run;

proc contents data =var_test;run;
```

Output is :

```
Alphabetic List of Variables and Attributes

  #    Variable       Type    Len    Pos
  ffffffffffffffffffffffffffffffff
  2    NProducts      Num       4     24
  5    final_price    Num       8     16
  1    id             Char     10     28
  3    pro_price      Num       8      0
  4    tot_cost       Num       8      8
```

Output shows that now ID variable is a 10-character field so it can hold more characters. Without this explicit declaration, ID field can hold only two characters, as automatically assigned by SAS.

For character variables, you must allow for the longest possible value in the first statement that uses the variable, because you cannot change the length with a subsequent LENGTH statement within the same DATA step. The maximum length of any character variable in the SAS System is 32,767 bytes. For numeric variables, you can change the length of the variable by using a subsequent LENGTH statement.

Creating variables through PROC SQL and PROC IMPORT

We always extract data from data warehouses or import data using SAS import utilities and we find the dataset is created with all columns with various formats. Here what happens during the process is SAS identifies the best format for the database fields you are extracting and apply the same to the datasets. PROC SQL provides more flexibility in formatting the variables. This is discussed separately in the appendix-II.

KEEP and DROP statements.

KEEP and DROP statements are used often to control the number of variables (fields) read into and output into the datasets. During the data processing we create several variables but need to save only select ones in the final dataset.

If you want to restrict the number of columns in output data set, use the following method. This will ensure that output dataset is created with required variables only.

```
Data target_data (keep = var1 var2 var3 etc);
Set base_data;
Run
```

Alternately you can specify the first statement as follows:

```
Data target_data ;
Set base_data;
keep = var1 var2 var3 ;
Run
```

If you are reading a big dataset into SAS and require only a few variables from it, use the following statements in the program.

```
Data target_data ;
Set base_data((keep = var1 var2 var3 etc);
Run;
```

In the first case, SAS reads the entire data set base_data, even though you only intend to use three variables. In the second case, SAS reads from disk only the three variables you intend to keep. Please note that we have to use such efficient methods to restrict the data read into the system to optimize the system resources such as SASWORK and shared drives.

The same way DROP statement can also be specified based on the data requirement of the user.

Array Variables

Unlike other programming languages, SAS array variable is a set of similar variables grouped together with a name in an ARRAY statement. We use arrays very often when we work with datasets that are arranged as a Time Series. Let us look at simple program to understand

```
Data array_test;
Set weight_data;
array weight{50} wt1-wt50;
do i=1 to 50;
if weight{i}=999 then weight{i}= .;
end;
run;
```

As we have seen earlier one of the basic use of array is to group similar variables . In the above example 'weight_data has weight measured at fifty different growth stages and wherever data is missing that data point is updated with 999. Assume it's a very large data set and we want to replace all 999 with a '.' for missing. If we are to do this data step, we will have to write 50 statements (if wt1 =999 then wt1=. ; like that for each column) . Using array variable we can simplify this.

Line 3: Here 'array' is the key word that tells SAS the following name (weight) is an array. The closed bracket {50} specifies the number of elements in the array followed by the values of array elements*wt1, wt2, wt3….wt50). For example weight{40} = 'wt40'

Line 4: Do loop 1 to 50 to hold the record to scan through 50 different fields specified.

Line 5: Remember weight{1} will have a value wt1 and that field will be evaluated in the following IF condition to check whether it has a value of 999. Similarly when i=2, weight {2} will have a value of wt2. Like this 50 times the loop is executed for each row of data and the update is made for fifty fields (columns)

Line:6 Ending the loop .

End of the execution, all the fields are evaluated and updated with '.' for 999.

Now think about substituting fields with missing. Values with some other value.

Time Series Example

Now let us look at an example to know the way array is used in our environment (This program does not run as it requires some datasets)

```
Data Balance_due;
set TS_49;

array st_month {6} stmonth15 stmonth25 stmonth35 stmonth45 stmonth55;
array bal_due {6} baldue15 baldue25 baldue35 baldue45 baldue55 ;

do i=1 to 5;
Statement_month =st_month (i);
balance_due = bal_due(i);
output;
end;
run;

Proc summary data = Balance_due;
class Statement_month;
var balance_due ;
run;
```

The above program reads a dataset (TS_49) where the variables are arranged in a time series format. (For example, baldue15 stands for balance due for the month of Jan 2005 and so on) The user wants to see the balance due for all accounts by statement month for 5 specific months , using SAS ARRAY (s)he could achieve that easily. Further, (s)he could use PROC summary to do the summarization as data is now in a format that PROC Summary understands.

No let us look at the ARRAY statement.

```
array st_month {5} stmonth15 stmonth25 stmonth35 stmonth45 stmonth55
```

Here 'array' is the key word that tells SAS the following name (st_month) is an array. The closed bracket (5) specifies the number of elements in the array followed by the values of array elements. For example st_month(5) = 'stmonth55'

In other words, ARRAY help us to group a set of variables so that programming could be made short and flexible. If we need to do the same action repetitively on same group of variables, declaring an array would solve the same.

Variable LABELs

A SAS Label describes a variable. When labels are assigned in the data step they are available for all procedures that use that data set. When we produce reports we could print labels instead of variable names.

Let us modify our earlier example:

```
Data var_test;
length id $ 10;
length NProducts 4;
id ='JK';
NProducts= 6;
```

```
pro_price = 4.55;
tot_cost = NProducts*pro_price;
final_price = tot_cost;

LABEL       id ="Vendor Name"
            NProducts ="Number of Products"
            pro_price = "Product Price"
            tot_cost ="Total Cost"
            final_price ="Final Price"
            ;
run;

proc contents data =var_test;run;
```

Let us look at the output of PROC Contents:

```
-----Alphabetic List of Variables and Attributes-----

#   Variable     Type     Len   Pos   Label
-----------------------------------------------------------------
2   NProducts    Num      4     24    Number of Products
5   final_price  Num      8     16    Final Price
1   id           Char     10    28    Vendor Name
3   pro_price    Num      8     0     Product Price
4   tot_cost     Num      8     8     Total Cost
```

The PROC output now show 'Label', which gives more information about the variable.

When you use PROC PRINT or some other Procedures you can print the labels instead of variable names. For example the code below will print labels for the variables in var_test.

```
Proc print data = var_test label;

run;
```

Additionally, the labels can be created with PROC FORMAT procedure, which is not discussed in this guide.

Variable FORMATs

A **format** is an instruction that SAS uses to write data values. You use formats to control the written appearance of data values. Note that a format does not change the original value of a variable. Since formats are primarily used to format output, we will look at how we can use existing SAS internal formats using the FORMAT statement in PROCs and Data steps.

The following example shows how to use a format statement in a data step.

```
data format_test;
num_test = 1250;
today =today();
dollar_amt =13400.5;
Name = 'J KURIAN';
format num_test words40. name $reverj7.   dollar_amt dollar10.2
today monyy7.;
;
run;

proc print data = format_test;run;
```

The above example formats four variables – num_test with words40., name with reverj7., dollar_amt with dollar10.2 and today with monyy7. formats. Please have a look at the following table to understand what each format does to the display of the variable. The below list is not exhaustive- Do consult the SAS manual for a complete list of formats.

Syntax for the format statement is:

Format <variable name> <format name> ;

There are three categories of formats. Character, date and time and numeric. Lists of frequently used formats are provided below.

Category	Format	Description
Character	$CHAR*w.*	Writes standard character data
	$QUOTE*w.*	Writes data values that are enclosed in double quotation marks
	$REVERJ*w.*	Writes character data in reverse order and preserves blanks
	$UPCASE*w.*	Converts character data to uppercase
	$*w.*	Writes standard character data
Date and Time	DATE*w.*	Writes date values in the form **ddmmmyy** or **ddmmmyyyy**
	DATETIME*w.d*	Writes datetime values in the form **ddmmmyy:hh:mm:ss.ss**
	DAY*w.*	Writes date values as the day of the month
	DDMMYY*w.*	Writes date values in the form **ddmmyy** or **ddmmyyyy**
	DDMMYY*xw.*	Writes date values in the form **ddmmyy** or **ddmmyyyy** with a specified separator
	MMYY*xw.*	Writes date values as the month and the year and separates them with a character
	MONNAME*w.*	Writes date values as the name of the month
	MONTH*w.*	Writes date values as the month of the year

	MONYY*w.*	Writes date values as the month and the year in the form **mmmyy** or **mmmyyyy**
	QTR*w.*	Writes date values as the quarter of the year
	QTRR*w.*	Writes date values as the quarter of the year in Roman numerals
	WEEKDATE*w.*	Writes date values as the day of the week and the date in the form **day-of-week**, **month-name dd**, **yy** (or **yyyy**)
	WEEKDAY*w.*	Writes date values as the day of the week
	WORDDATE*w.*	Writes date values as the name of the month, the day, and the year in the form **month-name dd**, **yyyy**
	YEAR*w.*	Writes date values as the year
	YYMM*xw.*	Writes date values as the year and month and separates them with a character
Numeric	BEST*w.*	SAS chooses the best notation
	COMMA*w.d*	Writes numeric values with commas and decimal points
	COMMAX*w.d*	Writes numeric values with periods and commas
	DOLLAR*w.d*	Writes numeric values with dollar signs, commas, and decimal points
	FLOAT*w.d*	Generates a native single-precision, floating-point value by multiplying a number by 10 raised to the **d**th power
	NUMX*w.d*	Writes numeric values with a comma in place of the decimal point
	SSN*w.*	Writes Social Security numbers
	w.d	Writes standard numeric data one digit per byte
	WORDF*w.*	Writes numeric values as words with fractions that are shown numerically
	WORDS*w.*	Writes numeric values as words

SAS Functions in DATA steps

A SAS function performs a computation or manipulation on variables (arguments) and returns a value. Most functions use arguments supplied by the user. SAS functions are mainly used in DATA step programming statements.

```
data function_test;
Max_var= max(100,101);
Length_var = length(Max_var);
This_month = month(today());
run;
```

```
proc print;run;
```

The syntax of a function is :

Function-name (argument-1, . . .,argument-n>)

In the above example, we have shown how MAX, LENGTH and MONTH functions are used. The table below has some frequently used functions in our programs.

Function Name	Description
LENGTH	Returns the length of an argument
LOWCASE	Converts all letters in an argument to lowercase
SCAN	Selects a given word from a character expression
SUBSTR (right of =)	Extracts a substring from an argument
UPCASE	Converts all letters in an argument to uppercase
DATEPART	Extracts the date from a SAS datetime value
DAY	Returns the day of the month from a SAS date value
INTCK	Returns the integer number of time intervals in a given time span
INTNX	Advances a date, time, or datetime value by a given interval, and returns a date, time, or datetime value
MONTH	Returns the month from a SAS date value
QTR	Returns the quarter of the year from a SAS date value
TODAY	Returns the current date as a SAS date value
YEAR	Returns the year from a SAS date value
MAX	Returns the largest value
MEAN	Returns the arithmetic mean (average)
MIN	Returns the smallest value
SUM	Returns the sum of the nonmissing arguments
CALL SYMPUT	Assigns DATA step information to a macro variable
LOG	Returns the natural (base e) logarithm
MOD	Returns the remainder value
SQRT	Returns the square root of a value
RANUNI	Returns a random variate from a uniform distribution
INPUT	Returns the value produced when a SAS expression that uses a specified informat expression is read
LAG	Returns values from a queue
PUT	Returns a value using a specified format
ZIPSTATE	Converts ZIP codes to state postal codes
CEIL	Returns the smallest integer that is greater than or equal to the argument
FLOOR	Returns the largest integer that is less than or equal to the argument
ROUND	Rounds to the nearest round-off unit

TRUNC	Truncates a numeric value to a specified length
VLABEL	Returns the label that is associated with the specified variable
VNAME	Returns the name of the specified variable
VTYPE	Returns the type (character or numeric) of the specified variable

MERGE –Combining datasets

One of the features frequently used while SAS programming is combining one or many datasets. For example to combine the application data with performance data we use MERGE utility in SAS.

Most of the time a match merging is done on SAS datasets. For example, if we are interested in the performance of the accounts that are opened only in Jan 2004, we try to match only those accounts while doing the performance data merging. In consumer Finance, account number or Account Key is usually used for match merging.

Let us look at the following examples. We have two record sets that have information about credit lines of the accounts. Intial_cl holds data for all initial credit lines assigned and new_cl has all increased credit lines. Now we need to combine these into one dataset so that it will show the latest credit lines for each account.

```
DATA intial_cl;
INPUT account credit_limit;
DATALINES;
1002 2000
1003 4000
1004 3000
;

DATA new_cl;
INPUT account credit_limit;
DATALINES;
1002 3000
1004 5000
;

DATA credit_limit;
  MERGE intial_cl new_cl;
  BY account;
RUN;

PROC PRINT;
RUN;
```

Output looks like as follows:

```
                        credit_
       Obs    account     limit
```

```
1       1002       3000
2       1003       4000

3       1004       5000
```

So a simple merge statement combined the datasets and the BY statement made sure that its updated with the latest information.

What if we put 'new_cl' data set first in the MERGE statement? Note that the data overwritten on the first dataset.

Let us introduce some more complexity to reflect the way we use MERGE statement in our environments..

```
DATA credit_limit;
  MERGE  intial_cl(in=a) new_cl(in=b) ;
  BY account;
  IF  a and b;

RUN;
```

The above data step introduces a conditional processing using IF . Also a handle to the dataset is declared after the dataset name (in=a, in=b) . Using these handles we can combine the datasets conditionally.

'IF a and b' combines the dataset if the BY statement finds a match in the second dataset. Only those records with a match are combined. This is different from the first example where all records are output into the resultant dataset. Output looks like as follows

Account Sample

Obs	account	credit_ limit
1	1002	3000
2	1004	5000

There are various combinations that suits to various requirements

If a ; = combines the data with data updated for all accounts in the first dataset

If b; = combines the data with data updated for all accounts in the second data set

If a or b; combines both the datasets with data updated into the first dataset specified (Outer join)

Also it is possible to merge more than two data sets at a time. PROC SQL is also used to merge datasets. Please go through the Appendix-I for more details.

Conditional Processing with WHERE, IF-ELSE, DO-END

When working with data we come across situations where data need to be filtered or we may have to apply several conditions before we have the final data for analysis or modeling. For example, When we want to report out the number of active accounts and total $ outstanding, we need to subset the entire data so that only accounts that are active are included. Suppose our active definition says " An active account is an account where current $ outstanding is greater than 100 and Status_code is not Z". Let us look at an example now.

```
DATA account_perf;
INPUT account current_os status_code $;
cards;
1002 300   A
1003 20    A
1004 1200  .
1005 800   Z
1006 450   D
1007 560   Z
1008 450   A
1009 900   C
1110  300  C
;run;

Data perf;
set account_perf;
where current_os >100 and status_code ne 'Z';

run;
```

Here dataset 'perf' is a subset of account_perf. Conditions behind creating this subset were the activity definitions we mentioned before. Note that WHERE statement is used to evaluate two variables with and 'and' condition. The same way we can use OR also.

IF can also be used instead of WHERE . (You can use IF current_os >100 and status_code ne 'Z';). But WHERE is more efficient when we do the sub-setting.

IF ELSE conditions are commonly used to flag accounts or create new variables based on certain conditions. Suppose we need to flag the accounts into Good and Bad based on certain conditions, IF/ELSE can be used. The example below shows the usage of IF/ELSE

```
Data perf;
set account_perf;
length status $20.;
if status_code eq 'Z' then status= "Bad-Charged Off";
else if status_code eq 'C' then status = "Cancelled";
else  Status= "Good Account";
run;
```

Dataset used is same as the previous example. Here we are categorizing the data into three segments. Bad Charged off, Cancelled and Good Accounts. A new variable (status) is created based on the condition.

Obs	account	current_os	status_code	status
1	1002	300	A	Good Account
2	1003	20	A	Good Account
3	1004	1200		Good Account
4	1005	800	Z	Bad-Charged Off
5	1006	450	D	Good Account
6	1007	560	Z	Bad-Charged Off
7	1008	450	A	Good Account
8	1009	900	C	Cancelled
9	1110	300	C	Cancelled

When there is multiple processing done conditionally, then DO-END loop is used. For example we want to categorize the good and bad accounts and also want to compute the write-off amount based on a same condition ie Status_code =Z, this can be achieved using a DO-END loop. Let us have a look at the example below.

```
Data perf;
set account_perf;
length status $20.;

if  status_code eq 'Z' then do ;
status= "Bad-Charged Off";
wo_amount = current_os ;
end;

else  do;
Status= "Good Account";
wo_amount = 0 ;
end;

run;

proc print ;run;
```

The output looks like the following:

Obs	account	current_os	status_code	status	wo_amount
1	1002	300	A	Good Account	0
2	1003	20	A	Good Account	0
3	1004	1200		Good Account	0
4	1005	800	Z	Bad-Charged Off	800
5	1006	450	D	Good Account	0
6	1007	560	Z	Bad-Charged Off	560
7	1008	450	A	Good Account	0
8	1009	900	C	Good Account	0
9	1110	300	C	Good Account	0

Now let us summarize:

WHERE, IF-ELSE, DO-END are used for conditional processing in SAS data steps. WHERE and IF conditions are also used in many SAS PROCs to subset data and filter out unwanted data.

Exercises:

Create two data sets as showed below and try out the following combination of merging. Write down your observations on how each merging worked.

```
DATA intial_cl;
INPUT account credit_limit;
DATALINES;
1002 2000
1003 4000
1004 3000
;
```

```
DATA new_cl;
INPUT account credit_limit;
DATALINES;
1002 3000
1004 5000
1005 2500
;

DATA credit_limit;
  MERGE  intial_cl(in=a) new_cl(in=b) ;
  BY account;
  IF  a ;
RUN;

PROC PRINT;
RUN;

DATA credit_limit;
  MERGE  intial_cl(in=a) new_cl(in=b) ;
  BY account;
  IF  b ;
RUN;

PROC PRINT;
RUN;


DATA credit_limit;
  MERGE  intial_cl(in=a) new_cl(in=b) ;
  BY account;
  IF  a or b ;
RUN;

PROC PRINT;
RUN;
```

```
DATA credit_limit;
  MERGE  intial_cl(in=a) new_cl(in=b) ;
  BY account;
  IF  a=b ;
RUN;

PROC PRINT;
RUN;
```

Procedures for Data Insights

Objectives of this chapter:

To learn frequently used procedures in Consumer Finance environment. These procedures are used to understand the data we work with as well as to bring out business insights. We will discuss PROC DATASETS, PROC PRINT, PROC SORT, PROC TRANSPOSE, PROC DOWNLOAD, PROC FREQ, PROC MEANS and PROC GPLOT in this chapter.

PROC DATASETS

PROC DATASETS is a utility procedure that helps to manage the SAS datasets in various libraries. In a multi-user environment like ours we are constrained by the system resources like SAS Workspace or the shared folders. To remove unnecessary files and manage the datasets, Proc Datasets is often used in the main programs.

With PROC DATASETS, you can

- Delete SAS files
- List the SAS files that are contained in a SAS library
- Copy SAS files from one SAS library to another
- Rename SAS files
- List the attributes of a SAS data set, information such as the date the data were last modified, whether the data are compressed, whether the data are indexed etc.
- Append SAS data sets
- Create and delete indexes on SAS data sets

The example below demonstrates three frequently used features of Proc Contents – Delete, copy and Rename.

```
libname mylib 'D:\UKGECF';

DATA mylib.intial_cl;
INPUT account credit_limit;
DATALINES;
1002 2000
1003 4000
1004 3000
;
DATA mylib.new_cl;
INPUT account credit_limit;
DATALINES;
1002 3000
1004 5000
1005 2500
;
```

```
proc datasets library=mylib details;
   change new_cl=brand_new_cl;
   delete intial_cl;

   copy in=mylib out =work;   select  brand_new_cl;

run;
```

Line 1: specifying the library to list the detail (filenames and attributes)

Line 2: change the name 'new_cl' into 'brand_new_cl'

Line 3: delete intial_cl from mylib

Line 4: copy from 'mylib' library to 'work' library the file specified in select statement (brand_new_cl)

PROC PRINT

Proc PRINT is used to understand how the data looks and also for a variety of reporting purposes. Proc print in conjunction with ODS features can produce impressive reports. PROC Print has various options and features to control the display, filter data/variables and to do additional computation on the data.

When we work with data, it's a good practice to have a look at a sample data extract. If the dataset is large, we would want to restrict the number of column and rows in our print report. The below example show how to do that.

Assume that 'statement' is a dataset with 100 variables and 1 Million records. We want to see any 10 observations and 5 variables viz. account_code, current_balance, credit_limit, status_code and fico_score. This is how we do it.

```
Proc print data=statement (obs=10);
title 'Sample records-Statement';
var account_code current_balance credit_limit status_code
fico_score;

run;
```

Line 1: tells SAS the dataset name. Also restricts the number of observations printed through OBS statement.

Line 2: specifies a title for the report.

Line 3: Specifies the variables to be printed

What we have seen is a basic version of PROC PRINT. Some of the features we use with are: BY statement, SUM statement and PAGE statement. Here are some examples on how to use them.

BY statement: Produces a separate section of the report for each BY group.

PAGEBY: Controls page ejects that occur before a page is full. Computes the totals by page, if specified.

SUM: Computes the total for the specified variables

```
DATA account_perf;
INPUT account current_os ext_status_code $ int_status_code $;
cards;
1002 300   A  C
1003 20    A  C
1004 1200  A  D
1005 800   Z  A
1006 450   Z  A
1007 560   Z  A
1008 450   A  D
1009 900   Z  D
1110  300  Z  D
;
run;

proc sort data = account_perf;
by ext_status_code;
run;

Proc print data = account_perf;
Title "Example for SUM, BY and PAGEBY Statements";
by ext_status_code  ;
pageby ext_status_code;
sum  current_os;

run;
```

Note that BY statement in Proc Print require the data to be sorted by the variables in BY statement. PROC SORT procedure should be used to do the sorting prior to the print.

PROC SORT

PROC SORT is a very useful utility we often use when work with data and procedures. PROC SORT sorts the variable(s) in a dataset in an ascending or descending order. Additionally it performs some other operations like deleting the duplicate rows and ordering the variables for certain procedures. When we do the data cleaning, PROC SORT is used to remove all the duplicate records or duplicate observation. In the example shown below, the accounts are appearing multiple times (duplicate and non-duplicate rows) and we want to keep the record

last updated only. Using PROC SORT we can filter those accounts and create a clean SAS dataset.

Let us create a SAS dataset to demonstrate some of the capabilities of PROC SORT.

```
DATA statement;
INPUT account current_os ext_status_code $  Dt_updt mmddyy10. ;
format dt_updt mmddyy10.;
cards;
1002 300  A 03/15/2005
1003 20   A 03/15/2005
1003 20   A 03/15/2005
1004 1200 A 03/15/2005
1005 800  Z 03/15/2005
1006 450  Z 03/15/2005
1007 560  Z 03/15/2005
1002 300  Z 03/25/2005
1002 300  Z 03/25/2005
1009 900  Z 03/15/2005
1110 300  Z 03/15/2005
1004 1200 Z 03/26/2005
;

run;
```

The DATA step above creates a dataset with 12 records. Note that account number 1003 and 1002 have duplicate repords. Accounts 1002 and 1004 are repeated but they are not duplicated. In order to do any analysis or reporting, we should first clean this dataset to make sure no doble couting is done on measurement variables. Such issues are common with statement table as the customer can request to change the billing cycle for their credit card statements or problems with data loading at ETL stage.

This is how we use a PROC SORT statement:

```
proc sort data = statement out=statement1 nodup;
by account descending Dt_updt;
run;
```

Line 1: specifies the data to read in and 'out' statement specifies the dataset to be created after sorting. 'NOHUP' is a keyword tells SAS to remove all records that are identical and keep only the first one.

Line 2: BY statement specifies the variables to be used for sorting. Here account variable is sorted in an ascending order (default) and Dt_updt is sorted in a descending order. Our objective is to keep the record last updated so when SAS deletes the duplicates it keeps the first record in the order sorted, and in this case 'descending' sort keyword brings the latest record first.

Notice that we still have multiple records for couple of accounts. To remove them we use the 'nodupkey' keyword as follows:

```
proc sort data = statement1 out=statement2 nodupkey;
by account ;

run;
```

Line 1: 'nodupkey' keyword is specified to instruct SAS to remove all the records appearing repetitively for the variables in BY statement.

Line 2: BY statement specifies the variables where SAS should look for duplicates. If an account is repeated twice the fist record in the sort order is kept and rest are deleted from the output dataset.

So with these two sort steps we have a clean dataset with only latest information. It is possible to specify multiple variables in the BY statement and it is possible to control the order (ascending or descending) of the individual variables while performing the sorting.

Some Procedures use BY statements to categorize the output – For example, PROC PRINT, PROC SUMMARY, PROC UNIVARIATE. Make sure you sort the data by BY variables before you perform a procedure on it.

TIP: PROC SORT is very time/resource consuming process in Consumer Finance environment as we typically work with millions of records. We don't have to sort data for PROC SUMMARY, unless it has a BY statement.

PROC TRANSPOSE

The TRANSPOSE procedure creates an output data set by restructuring the values in a SAS data set, transposing selected variables into observations. It converts the row elements to columns.

Let us create a dataset to demonstrate an example:

```
DATA statement_summary;
INPUT Perfmonth date9. tot_accounts newacct     actives
current_balance;
format Perfmonth MONYY7. ;
label
Perfmonth ="Performance Month"
tot_accounts ="Total Number of Accounts"
actives= " #Active Accounts"
newacct     ="# New Accounts"
current_balance ="$ Current Balances";
```

```
cards;
01-Apr-03 8860303 86521 1366811     32932422.08
01-May-03 8947743 90272 1376739     30994371.23
01-Jun-03 9035228 90436 1397670     31551717.63
01-Jul-03 9123510 92157 1424469     31788023.01
01-Aug-03 9199904 79118 1417949     32329068.83
01-Sep-03 9289735 92682 1369723     33772968.41
01-Oct-03 9390095 10294 1371610     34349188.42
01-Nov-03 9493607 10673 1383296     35398736.71
01-Dec-03 9583579 93288 1427501     36525256.12
01-Jan-04 9643529 63447 1432429     39782001.88
01-Feb-04 9706194 66283 1340457     38625107.74
01-Mar-04 9723757 20907 1294083     37758060.3
;run;

proc print data = statement_summary label;run;
```

This is how the output looks like . Now we want to produce a report that displays al metric in a time series format ie. you want the Performance months as columns . Then it becomes a typical case where we use PROC Transpose.

Obs	Performance Month	Total Number of Accounts	# New Accounts	#Active Accounts	$ Current Balances
1	APR2003	8860303	86521	1366811	32932422.08
2	MAY2003	8947743	90272	1376739	30994371.23
3	JUN2003	9035228	90436	1397670	31551717.63
4	JUL2003	9123510	92157	1424469	31788023.01
5	AUG2003	9199904	79118	1417949	32329068.83
6	SEP2003	9289735	92682	1369723	33772968.41
7	OCT2003	9390095	10294	1371610	34349188.42
8	NOV2003	9493607	10673	1383296	35398736.71
9	DEC2003	9583579	93288	1427501	36525256.12
10	JAN2004	9643529	63447	1432429	39782001.88
11	FEB2004	9706194	66283	1340457	38625107.74
12	MAR2004	9723757	20907	1294083	37758060.30

This how we use a PROC TRANSPOSE

```
proc transpose data =statement_summary out = ts_statement;
id Perfmonth;
var tot_accounts newacct      actives  current_balance ;
run;
```

Line 1: Specifies the dataset and output dataset – Proc Transpose does not print the results

Line 2: Spcifies the field to be transposed through an 'ID' statement.

Line 3: Specifies the variables to be transposed

This is how it looks if we print a part of the data transposed (ts_statement)

Obs	_NAME_	_LABEL_	APR2003	MAY2003	JUN2003
1	tot_accounts	Tot Number of Accounts	860303	8947743	9035228

```
2     newacct         # New Accounts         86521   90272.   90436
3     actives         #Active Accounts      136681  1376739  397670

4     current_balance $ Current Balances    3293242 30994371 31551717
```

Note that two variables are created by SAS '_Name_ ' and '_Label_ ' . This field has all variable names and their respective labels so that we can identify them. Variable we specified in ID statement is now converted into columns.

PROC Transpose is often used to convert the variables into a time series format as shown above. It is possible to use a 'BY' statement in a PROC Transpose to transpose them in a grouped manner. When we work with SAS programs for automation, there are several instances we would do a PROC transpose to reshape the data in to a structure that helps in automation.

PROC DOWNLOAD

PROC DOWNLOAD is used to download data from a remote server, when you are working with SAS remote submit session. SAS remote sign on to a Unix server enables the user to compose and submit the program in their windows clients and see the SAS log and Output in their workstation. When you are submitting a program to remote server, the Sas datasets are created in the remote server (Stoner Unix server for example). PROC Download enables to download the data into your local windows folders.

Let us look at a program :

```
libname loc 'c:\datasets';
rsubmit;
libname user01 '/projects/walmart'
options obs =100;
Proc downlod data = user01.walmart_auth out = loc.walmart_auth;
run;

endrsubmit;
```

The above program assumes that a signon to a remote server is already estabilished. There are two librariesdeclared – 'loc' is a local SAS library and 'user01' is a remote Sas library . Options obs=100 restricts the number of observations downloaded to 100. OBS=Max should be used if the intention is to download the complete dataset.

PROC FREQ

PROC FREQ is used to produce frequency counts and cross tabulation tables for specified variables/field. It can also produce the statistics like Chi Square to analyze relationships among variables.

As a good practice, at a data processing stage we use PROC Freq on categorical fields to understand the data and their frequency distributions. For example a cross tabulation table of FICO score segments to External Status code will tell us how the accounts are distributed across various score segments and statuses. PROC FREQ offers various features to control the output and the way the tables are produced. Let us start with an example to understand them better.

```
DATA account_perf;
INPUT client $ account current_os ext_status_code $ int_status_code
$;
cards;
WalMart 1002 300   A  C
WalMart 1003 20    A  C
JCP     1004 1200  A  D
JCP     1005 800   Z  A
GAP     1006 450   Z  A
GAP     1007 560   Z  A
JCP     1008 450   A  D
GAP     1009 900   .  D
WalMart 1110 300   Z  D
;
run;
proc sort DATA =  account_perf;
BY client;
run;

proc freq DATA = account_perf;
TITLE 'Frequency of External Status Code by Client';
TABLES ext_status_code* int_status_code /missing norow nocol
nopercent;
BY client;
WHERE client ne 'JCP';
LABEL client ='Client Name';
run;
```

The data step creates a SAS dataset named 'account_perf'. Proc sort is used to sort the data by client as we use a 'BY' statement in PROC FREQ that follows.

Line 1 and 2 in PROC FREQ statement tells SAS which dataset to use and the title of the output.

Line 3 specifies SAS to generate a cross-tab of External status code and Internal Status code trough a TABLES statement. There are several options specified like missing, norow, nocol and nopercent to control the way statistics are displayed. A standard FREQ output shows a column, row and cell percentages.

Line 4: BY Statement specifies the grouping criteria. In this case, frequencies are computed for each client group.

Line 5: WHERE statement allows the conditional processing. Here all clients other than 'JCP' are taken for computation.

Line 5: Labels the output. Its also possible to use FORMAT statements to manipulate the display.

Here is a list of commonly used options with TABLES statement

Nocol - suppresses printing of column percentages of a cross tab.
Norow - suppresses printing of row percentages of a cross tab.
Nopercent - suppresses printing of cell percentages of a cross tab.

Missing - interprets missing values as non-missing and includes them in % and statistics calculations.

List - prints two-way to n-way tables in a list format rather than as cross tabulation tables

Chisq - performs several chi-square tests.

Now let us look at the following requirements:

i) The user wants the output not to be printed but to be put into a SAS data set for further processing. ii) Wants to order the values to be changed to ascending order of frequency counts (highest occurring first) and iii) The output to be listed instead of cross-tab.

The following program block does the job:

```
proc freq DATA = account_perf order=freq;
Title 'Frequency of External Status Code by Client';
TABLES int_status_code*ext_status_code  / list out = perf_freq ;
WHERE client ne 'JCP';
label client ='Client Name';
run;
```

At line : 1- please note that 'order =freq' is specified to tell SAS to order the output by descending frequency count

Line: 3- 'list' keyword is used to tell SAS to list the output and not cross-tabulate. 'Out=perf_freq' specifies the output dataset name to store the frequency output.

To sum up, PROC FREQ is a very useful and the most used of the SAS procedures. In Consumer Finance environment, PROC FREQ is used mainly in the data preparation stage and for reporting. Frequency ordering, list and output data creation using OUT are often used options.

PROC MEANS

The PROC MEANS produces descriptive statistics for numeric variables. By default, the MEANS procedure reports N (the number of non-missing observations), Mean, Std Dev (Standard Deviation), Minimum and Maximum. We can request additional statistics through the options. Let us start with a simple example:

```
proc means DATA = account_perf  ;
Title 'Summary of Current Outstanding' ;
BY client ;
var current_os;
label client ='Client Name'
      current_os= 'Current Outstanding' ;

run;
```

Line:3-Tells SAS to group the values in client field and produce the statistics separately.

Line:4- Specifies the variable for which the statistics are to be computed. Here current_os is a field in the dataset given and that contain data for current Dollar outstanding for each account in the dataset.

Now we want to request more statistic like variance, range, sum, mean, median, minimum and maximum this is how we do it.

```
proc means DATA = account_perf  var range sum  mean  median min max ;
Title 'Summary Statistics of Current Outstanding';
var current_os;
run;
```

Suppose we want to create a SAS dataset with the statistics computed for further processing, this is how we instruct SAS.

```
proc means DATA = account_perf  var  sum  mean ;
Title 'Summary of Current Outstanding' ;
var current_os;
by client;
output out = perf_mean n = count sum = total mean = avg  ;

run;
```

The above program requests three statistics are to be put into the output dataset 'perf_mean'. Total count, sum and mean for each client (BY Client) are created as count, total and avg fields, respectively, in the output dataset.

To Sum up, PROC MEANS is used to understand the numeric variables, their distributions and other statistical characteristics. Options like BY and CLASS statements enable the users to look at them by segmenting into various categories. Just like we use FREQ on categorical and character fields, Means is used on numeric data. Additionally, PROC Means is used for

summarizing large datasets with a variety of statistics for reporting purposes. In this way it can be used as a substitute for PROC SUMMARY.

PROC GPLOT

PROC GPLOT is used to produce the graphical output in SAS. PROC GPLOT is considered as an improvement over the PROC PLOT procedure as it provides good quality presentation compared to simple PROC PLOT outputs.

PROC GPLOT produces a variety of two-dimensional graphs including

- Simple scatter plots
- Overlay plots in which multiple sets of data points display on one set of axes
- Plots against a second vertical axis
- Bubble plots
- Logarithmic plots (controlled by the AXIS statement).

In conjunction with the SYMBOL statement the GPLOT procedure can produce join plots, high-low plots, needle plots, and plots with simple or spline-interpolated lines. The SYMBOL statement can also display regression lines on scatter plots.
The GPLOT procedure is useful for

- Displaying long series of data, showing trends and patterns
- Interpolating between data points
- Extrapolating beyond existing data with the display of regression lines and confidence limits.

The dataset and the program below gives a good understanding about the GPLOT options and how the graphs are created.

```
DATA account_perf;
INPUT client $ account fico_seg $  current_os tot_payment
ext_status_code $ ;
cards;
WalMart 1002 401-500 300  100 A  C
WalMart 1003 501-600 200  150 A  C
WalMart 1004 601-700 1200 180 A  D
WalMart 1005 701-800 800  190 Z  A
WalMart 1006 801-900 450  200 Z  A
GAP     1007 401-500 560  210 Z  A
GAP     1008 501-600 450  180 A  D
GAP     1009 601-700 900  145 A  D
GAP     1110 701-800 300  148 Z  D
;
run;
```

```
proc sort data = account_perf;
by client;run;

PART-I

GOPTIONS
 RESET            = global
 GUNIT            = PCT
 CBACK            = BLACK
 CTEXT            = WHITE
/* DEVICE          = GIF*/
 FONTRES     = PRESENTATION
 HTITLE           = 3
 HTEXT            = 2
 HSIZE       = 8 IN
 VSIZE       = 6 IN
 INTERPOL    = JOIN
 NOBORDER;

PART-II

SYMBOL1  VALUE = DOT      COLOR = LIME    HEIGHT = 2.5 WIDTH = 2;
SYMBOL2  VALUE = SQUARE   COLOR = VIPK    HEIGHT = 2.5 WIDTH = 2;
SYMBOL3  VALUE = TRIANGLE COLOR = STRO    HEIGHT = 2.5 WIDTH = 2;

PART-III

AXIS1 LABEL  = ('Current O/S')
      COLOR  = WHITE
      LENGTH = 60
      MAJOR  = (NUMBER = 5)
      MINOR  = NONE       ;
AXIS2 LABEL  = ('FICO SEGMENTS')
      COLOR  = WHITE
      LENGTH = 85
      MAJOR  = (NUMBER = 5)
      MINOR  = NONE;

PART-IV


PROC GPLOT DATA = Account_perf;
by client;
PLOT   current_os*fico_seg tot_payment*fico_seg/overlay haxis=axis1
vaxis=axis2;

RUN;
```

GOPTIONS: When working with GPLOT, the first step is to become familiar with the GOPTIONS. The purpose of the GOPTIONS statement is to apply levels of specific options to graphs created in the session or to override specific default values. It can be located anywhere within your SAS program; however, in order for the requested options to be applied to it, it must be placed before the graphics procedure. The GOPTIOND used above and explained below.

reset = option resets graphics options to their default values.

gunit = sets the units of character height measurement to percentage of display height.

cback = option sets the color background to black.

cfont= option sets the font color to white

device = option selects the device driver to use to create a graphic file(GIF)

htitle= option sets the text height for the first title to 3 (in percentage of display height).

htext = option sets the text height for all text to 2 (in percentage of display height).

Interpol: interpolation method (i.e., how to "connect" the points)

border option includes a border around the graphics output area.

In Part II, SYMBOL statements create SYMBOL definitions, which are used by the GPLOT procedure. These definitions control:

- The appearance of plot symbols and plot lines, including bars, boxes, confidence limit lines, and area fills
- Interpolation methods

Part III defines the Axes 1 and 2 where we can name the axis, spiffy the color, length of each axis and customize the number of major and minor divisions on it.

Having seen various options in GPLOT now let us look at the GPLOT statement in Part-IV.

Line:2 Tells SAS to produce a graph for each category in the variable specified in BY statement. Note that to use BY in GPLOT, the dataset should be sorted by the BY variable. In this example, a graph will be produced by each client in the dataset.

Line:3 specifies the X and Y axes in the graph. Here we are plotting two variables – O/S and Payments in the same Y variable (ie FICO_score) .The OVERLAY option on the PLOT statement makes sure that both plot lines appear on the same graph. Further, definitions done in the Part-III are applied using Haxis and Vaxis options.

POC GPLOT provides the flexibility to plot single or multiple variables on a same graph. Many a time we would require to name the SAS graph files and store it in a specific directory so that

we have control over the files when we do the automation. This can be achieved by DEVICE option in GOPTIONS.

In scorecard tracking or Champion-challenger strategy tracking projects often there are many segmentations (like score segments or strategy identifiers or vintage segments) and charting for various performance variables are carried out to compare the segments. This kind of complicated charting can be automated using various options of GPLOT.

SAS Powered Reporting

A lot of our work involves some form of reporting. Analytics community prefers the reports to be in a spreadsheet (MS Excel) format as they can carry out further analysis. For the regular production reporting automation is carried out to avoid manual work as much as possible. This chapter briefs some techniques and possibilities to explore for various reporting tasks.

In Consumer Finance, we use PROC EXPORT, ODS HTML or DDE methods to produce an excel or web based report. Let us look at some example to understand each one of them.

PROC EXPORT

PROC EXPORT procedure exports a SAS dataset into a MS Excel. It can also export the data into other formats like Access or Lotus but we limit our discussion to Excel only. Windows SAS provides an interface to do the data set export and when you want to do it automatically, you can specify the same as a program block as shown below.

```
PROC EXPORT DATA= work.Account_perf
            OUTFILE= "D:\ex_Account_perf.xls"
            DBMS=EXCEL2000 REPLACE;
RUN;
```

Line: 1 Specifies the SAS library and dataset name to be exported

Line: 2 Specifies the path and Excel filename where the data to be explored

Line: 3 tells SAS what data output format it is and to replace the file if existing already.

Note that not all formats applied on the variables are preserved during the export process. Also the labels are not exported by default.

Before we create reports using proc export, we may have to create the data set in SAS, which could be directly exported to excel. This method is normally used when we have to send only data tables as reports on a regular basis.

ODS HTML

Output Delivery System (ODS) statement in SAS allows the analysts to create HTML, and Excel reports that are sharable. Its possible to create templates for these reports and applies such formats on the final reports.

HTML formats are sharable across platforms and its possible to open those files in excel. So with appropriate changes in file extensions we can create XLS files through the ODS HTML statement. This is an advantage as many times customer prefers an Excel report but they cannot be produced in Unix environments.

When using the ODS HTML statement, the basic call contains two parts: the ODS call statement to tell SAS that an HTM/XLS output is requested and the ODS close statement to tell SAS to close the report. Any output operations in between call and close statements are directly written into an HTML output file.

For example, we are doing two operations- a PROC FREQ and a PROC PRINT – output of these two procedures would be written into an HTML file, if they are specified between and open and close statements. Let us look at an example

```
DATA account_perf;
format current_os dollar10.2  tot_payment words40. ;
label account="Account #"
fico_seg = "Fico Segments";
INPUT client $ account fico_seg $  current_os tot_payment
ext_status_code $ ;
cards;
WalMart 1002 401-500 300  100 A  C
WalMart 1003 501-600 200  150 A  C
WalMart 1004 601-700 1200 180 A  D
WalMart 1005 701-800 800  190 Z  A
WalMart 1006 801-900 450  200 Z  A
GAP     1007 401-500 560  210 Z  A
GAP     1008 501-600 450  180 A  D
GAP     1009 601-700 900  145 A  D
GAP     1110 701-800 300  148 Z  D
;

run;


ODS HTML FILE='D:\ods_report_test.html';
proc print data = account_perf;
by client;
title "Sample accounts with FICO Score";
run;

Proc Freq data = account_perf;
tables client/list;
Title "# of Accounts by Client";
run;

ODS HTML CLOSE;
```

The data step above creates a SAS dataset 'account_perf'. Format and labels are also applied to demonstrate later that how ODS preserves them in the final output.

Line: 1 of ODS statement specifies the path and file name of the output report.

Line: 2 – A print statement that prints data for each client separately.

Line: 6- FREQ procedure is used to display the number of accounts for each client.

Line: 10 -Tells SAS to close the ODS file opened for various output.

All the outputs of the procedure used above are written to D:\ods_report_test.html file as output tables. The standard template is used by default by SAS for the display. When writing into the html file, SAS preserves the label, format, titles that are applied on the data, which is an advantage over PROC EXPORT.

How do we produce an Excel report? One can just modify Line: 1 and change the file extension to .xls (instead of .html) to create an excel file! This file can be opened in Excel with all formats and labels preserved.

The output template provided by default can be changed by creating templates using PROC Template procedure and the same can be applied at ODS run time.

Dynamic Data Exchange (DDE)

Dynamic Data Exchange (DDE) is a MS Windows protocol for dynamically transferring data between Windows-based applications using a client/server model. Using this protocol SAS System can request data or send data and commands to other windows applications like Excel or PowerPoint.

"Could you put the data in an Excel spreadsheet, so that we could play with it?" We hear the all the time from our customers, as they really like navigating through the spreadsheet and doing additional calculations. Web based reports or SAS based reports (PROC REPORT or PROC TABULATE) do not meet this requirement of the customer but mastering DDE can help the analyst to create the customized reports in Excel

This is what you can achieve through DDE when working with Excel

i) You can have a nice template designed in Excel and direct the data into targeted cells in the template
ii) You can put data sheets in excel and make refresh the excel reports or graph
iii) You can save the report into a new name and save it in a specified folder
iv) You can run a macro you recorded in excel which does additional formatting to the output report.

Now let us look at an example to understand how DDE works in SAS. In the earlier example we had a limitation that we could not use our own excel template for reporting . Now let us assume we designed one excel template and stored it as 'D:\MyReports\DDE_test.xls' . Now we want SAS to open this template and fill in the data for various columns.

First let us invoke excel and open the template we designed:

```
options noxwait noxsync;
x '"C:\Program Files\Microsoft Office\Office\excel.exe" ';

FILENAME DDTEM DDE 'EXCEL|SYSTEM';

DATA _NULL_;
FILE DDTEM;
PUT '[OPEN("D:\MyReports\DDE_test.xls")]';
RC = SLEEP(3);

RUN;
```

Line:1- X command is used to invoke the excel short-cut in D: drive. If you don't have a shortcut created, please create one!

Line:2 – A FILENAME statement that establish a link with Excel system opened with a keyword 'DDE' DDTM is the handle defined to refer to this link later in SAS data step.

Line:3 – A temporary dataset is created to invoke the template.

Line:4 – FILE statement to tell SAS which link is established with excel to be used

Line:5 – PUT command – passes on the instruction to EXCEL system – Here the instruction is to open the template we created.

Line:6- SLEEP command to stop SAS system from processing for 3 minutes. This is to make sure that the template in Excel is opened before SAS proceeds with the data updating in the steps followed.

Now we have the template open in Excel and ready to receive data. We need to implement another link to target worksheet and define the area in the template before we tell SAS where to put the data. We will do it using the same FILENAME statement.

```
FILENAME DDE_SAS DDE "EXCEL|ACCOUNT_PERF!R2C1:R20C6" NOTAB;
```

Here, 'Account_Perf' in the template is defined as the target worksheet and the data area marked contains rows starting from 2(R2) and ending at 20 (R20) and columns 1(C1) to columns 6(C6). Please note that any data outside this defined area will be ignored. Next step is to drive the data into the spread sheet in a DATA step. Here is how we do it.

(Please note that dataset used in previous section is used here , PUT statement is used to write specified fields in to a file that is specified earlier with FILENAME statement. "09"x is a delimiter specification that tells SAS to put a TAB after each variable written into the file.)

```
DATA _NULL_;
```

```
FILE DDE_SAS;
SET  account_perf;
PUT
client $ "09"x
account  "09"x
fico_seg $ "09"x
current_os   "09"x
tot_payment  "09"x;
RUN;
```

The above code use the link established with excel and drive the data from account_perf to 'DDE_SAS'. PUT statement specifies the variable names and the order. "09"x is the delimiter (ie TAB) .

We have seen the basic form of how DDE is used in automating the Excel reports with SAS. Flexibility is that we can define the specific areas in Excel sheet and put the values from SAS dataset there. Coupled with Macros, DDE could be used as a powerful method to create EXCEL based reports.

Suppose there is an Excel macro on Client_report.xls, which does some formatting on the reports and graphs after the data was updated. To do this task analyst have to open the file and run the macro. With DDE this can also be achieved from SAS. Suppose our file had a macro called 'painter' defined and we are now running the excel macro through SAS and saving the file with a different name.

```
DATA _NULL_;
FILE DDE_SAS;
PUT '[RUN("Client_report.xls!painter")]';
PUT '[SAVE.AS("D:\MyReports\Client_report_formatted.xls")';
PUT '[QUIT()]';

RUN;
```

To conclude, SAS based reporting in our environment is limited to ODS HTML and DDE. However SAS procedures like PROC REPORT and PROC TABULATE provide lot of features to customize and format the reports.

SAS Macro Processing

The SAS macro language is help to reduce the amount of repetitive SAS code and it facilitates passing information from one procedure to another procedure. Furthermore, we can use it to write SAS programs that are "dynamic" and flexible. Generally, we can consider macro language to be composed of macro variables and macro programs. In this chapter will demonstrate how to create macro variables and how to write basic macro programs.

SAS Macro Variables

Many times we create macro variables to make them available through out various data steps and procedures. The scope of the variables is mostly limited to the data step or the procedures where they are created or used but through macro facility we can make them available throughout the program. For example its possible to declare a list of variables used in print and freq procedures repetitively as a macro variable and use it instead of a long list of variables. Also its possible to compute the data range of the data for a report and assign that to a macro variable so that all procedures and data step can access this information.

A macro variable can be created by using the %let statement. All the key words in statements that are related to macro variables or macro programs are preceded by percent sign %; and when we refer a macro variable it is preceded by an ampersand sign &. When we submit our program, SAS will process the macro variables first, substituting them with the text string they were defined to be and then process the program as a standard SAS program

There are two functions that are particularly useful when we want to get information in and out of a data step. These are symput and symget. You use symput to get information from a data step into a macro variable and symget is used when we want to get information from a macro variable into a data step.

Now let us look at some examples:

```
DATA account_perf;
format current_os dollar10.2  ;
label account="Account #"
fico_seg = "Fico Segments";
INPUT client $ account fico_seg $  current_os tot_payment
ext_status_code $ ;
cards;
WalMart 1002 401-500 300  100 A  C
WalMart 1003 501-600 200  150 A  C
WalMart 1004 601-700 1200 180 A  D
WalMart 1005 701-800 800  190 Z  A
WalMart 1006 801-900 450  200 Z  A
GAP     1007 401-500 560  210 Z  A
GAP     1008 501-600 450  180 A  D
GAP     1009 601-700 900  145 A  D
GAP     1110 701-800 300  148 Z  D
```

```
;

run;

%let varlist =  current_os tot_payment;

Proc print data =account_perf;
Title " Weekly report ";
var &varlist;
run;

proc means data = account_perf;
var &varlist;run;
```

First line after the DATA step uses %LET statement to declare a macro variable 'varlist'. Two columns are assigned to the variable.(Note that there could be more variables or string that could be assigned)

Line:4 – Var statement uses the 'varlist' variable with a '&' prefix.

Line:7- The PROC Means procedure also uses the same variable to generate the statistics.

Though this example is not a good use of macro variables, this will give some idea about the scope of variables and their availability.

The program below demonstrates the use of SYMPUT function to create macro variables from a SAS data step. This is particularly useful, as many times we have to assign values to macro variables from a SAS data step.

```
%let sweek =-1;
data _null_;
bd= INTNX('WEEK',today(), &sweek,'B');
edate = put(bd+6, date9.);
bdate = put (bd, date9.);
call symput ("edatec",edate);
call symput ("bdatec",bdate);
run;

Proc print data =account_perf;
Title " Weekly report from &bdatec to &edatec";
var &varlist;

run;
```

The program above is written for a weekly application reporting purpose where every table in the report should have a header with start date and end date. Further applications considered for these reporting should fall between these dates., finally, the report file also should have this date stamp.

To achieve this previous week start date and end dates are computed in a data step and assigned to macro variables. Now these dates are available during data extraction and reporting steps of the program.

Line 1: declares macro variable 'week' with %let statement.

Line 3: INTX function is used to compute the beginning day of previous week.

Line 4: end date is computed using the beginning date.

Lines 6&7 : Call symput statement is used to assign the value of edate and bdate to macro variables edatec and bdatec.

Line 10: The macro variables are used in the title statement of Print procedure to print start date and end date.

There are some uses of macro variables but they are extensively used in macro processing to pass on values into a SAS Macro. We will see that in the following sessions.

Macro Programs

A macro program is similar to a subroutine or a function in other programming languages. Sas programs are usually written to do a task repetitively.

A macro program always starts with the %macro statement including the user defined program name and it ends with a %mend statement.

Now let us look at a program to understand the macro processing in our environment. A sample data set is created with three clients and various score segments.

```
DATA account_perf;
INPUT client $ account fico_seg $  current_os tot_payment
ext_status_code $ ;
cards;
Lowes 1008 101-200 450  180 A
Lowes 1009 201-300 900  145 A
Lowes 1110 301-400 300  148 Z
Lowes 1002 401-500 300  100 A
Lowes 1003 501-600 200  150 A
Lowes 1004 601-700 1200 180 A
Lowes 1005 701-800 800  190 Z
Lowes 1006 801-900 450  200 Z
GAP 1008 101-200 450  180 A
GAP 1009 201-300 900  145 A
GAP 1110 301-400 300  148 Z
GAP 1002 401-500 300  100 A
GAP 1003 501-600 200  150 A
GAP 1004 601-700 1200 180 A
GAP 1005 701-800 800  190 Z
GAP 1006 801-900 450  200 Z
```

```
WalMart 1008 101-200 450  180 A
WalMart 1009 201-300 900  145 A
WalMart 1110 301-400 300  148 Z
WalMart 1002 401-500 300  100 A
WalMart 1003 501-600 200  150 A
WalMart 1004 601-700 1200 180 A
WalMart 1005 701-800 800  190 Z
WalMart 1006 801-900 450  200 Z
;
run;


%macro score_seg (client,st_code,dataset);
proc print data = &dataset;
Title "FICO Segment for External Status=&st_code and Client
=&client";
var client ext_status_code  fico_seg current_os;
where client=&client and ext_status_code =&st_code;
run;
%mend;

%score_seg('GAP','A',account_perf)
%score_seg('GAP','Z',account_perf)
%score_seg('WalMart','A',account_perf)
%score_seg('WalMart','Z',account_perf)
%score_seg('Lowes','A',account_perf)
%score_seg('Lowes','Z',account_perf)
```

Objective is to produce a set of reports to show FICO score segments and Current Outstanding based on the External Status code. The report needs to be produced individually for the clients specified. Now let us assume that the data set has more than 10 client and several status codes and a large number of records. We can write a proc print step for each and every combination but this macro simplifies that.

This task is repetitive. Only client status code and datasets are variables so they are the best candidates for parameters (macro variables) that can be passed on from a Macro call.

After the data step line 1: defines the macro with %macro statement followed by macro name '%score_seg'. This is followed by, in parenthesis, the arguments or macro variables that are passed on to the macro. These variables can take different values, every time a macro call is made .

Line 2: Proc print statement is specified and data statement takes a macro variable as the input data. The value assigned to this macro variable will be read as dataset name

Line 3: Title statement in PROC print . The macro variable passed on are used to print a meaningful title

Line 5: Where statement takes the values from macro variables passed on . This will ensure that the data is filtered as per the specifications of the user.

Line: 7 Tells SAS to end the macro with a %mend statement

Line 8: Calls the macro '%score_seg'. Values are passed on to the macro with a comma.

Line 9 onwards: the same macro is called for various combinations. Proc print is customized for the analyst requirement

What we have seen above are basic macro constructs and some simple examples. Macros are extremely useful in SAS programming environment and used in various contexts to simplify the programs or to bring efficiency to data processing.

Exercise for you: Collect 3 Macro programs from your team members written for various purposes. List down the functionality (what the mcor does) and the context they are used. Explain what way it simplifies the program.

PROC SQL and SAS

Structured Query Language (SQL) is a standardized, widely used language that retrieves and updates data in relational tables and databases.

The SQL procedure is SAS' implementation of Structured Query Language. PROC SQL is part of Base SAS software, and you can use it with any SAS data set (table). Often, PROC SQL can be an alternative to other SAS procedures or the DATA step. You can use SAS language elements such as global statements, data set options, functions, informats, and formats with PROC SQL just as you can with other SAS procedures.

PROC SQL in our environment is used:

- Retrieve data from database tables or views (Oracle or SQL Server)
- Combine SAS datasets from tables or views (MERGE)
- Create datasets and indexes
- Compute statistics and Generate reports

Data extraction with PROC SQL

We often use PROC SQL to extract data from various warehouses. Below is an example reproduced from a previous chapter.

```
PROC SQL;
Connect To ORACLE(User=501115644 Password=ypasswd Buffsize=10000
Path=CDCIT1 Preserve_Comments );
CREATE TABLE acct_status AS SELECT * FROM Connection To ORACLE
(SELECT current_account_nbr AS account_number,
external_status_reason_code AS
ext_rcode,external_status AS estatus,
billing_cycle_day AS billing_cycle_day
FROM
ACCOUNT_DIM
WHERE CLIENT_ID='BROOK BROS'
AND
nvl(EXTERNAL_STATUS_REASON_CODE,'0') <>'98');
Disconnect From ORACLE;

Quit;
```

In the above example PROC SQL use a connect string "Connect To ORACLE(User=501115644 Password=ypasswd Buffsize=10000 Path=CDCIT1 Preserve_Comments)" to identify the oracle database (CDCITI) and use the user name and passwords specified to read data from it. Further, CREATE TABLE statement creates a SAS Dataset from the output of SELECT statement.

When querying a data warehouse, PROC SQL automatically converts the field formats in to SAS formats. For example, a date field in Oracle will be converted into SAS date and an Oracle Varchar field will be converted into character.

SAS Data steps with PROC SQL

PROC SQL can perform some of the operations that are provided by the DATA step and the PRINT, SORT, and SUMMARY procedures.

Let us create a dataset and then we will see how PROC SQL works like a DATA step.

```
DATA account_perf;
INPUT client $ account fico_seg $  current_os tot_payment  ;
cards;
WalMart 1002 401-500 300  100
WalMart 1003 501-600 200  150
WalMart 1004 601-700 1200 180
WalMart 1005 701-800 800  190
WalMart 1006 801-900 450  200
GAP     1007 401-500 560  210
GAP     1008 501-600 450  180
GAP     1009 601-700 900  145
GAP     1110 701-800 300  148
;
run;

PROC SQL;
title "Summary of O/S and Payments by Client";
select client, sum(current_os) as tot_os , sum(tot_payment)as
tot_payment
from account_perf
group by client
order by tot_os descending ;
quit;
```

The above program block shows how a PROC SQL is substituting a PROC PRINT, PROC SORT and PROC SUMMARY.

Line 2: Assigns a title to the output of SELECT statement that follows
Line 3: Select statement with group function SUM used to summarize the data. GROUP BY clause is used to compute the SUM for each distinct group in the database. ORDER BY is

used to sort the output and DESCENDING keyword is to control the sort order. It is also possible to SORT by multiple variables.

A PROC SQL statement ends with 'QUIT\;' and it terminates the procedure. Output is always printed to the screen (like PROC PRINT) and it's also possible to create a SAS dataset from the output. To create a dataset from the output the above program can be modified as follows:

```
PROC SQL;
title "Summary of O/S and Payments by Client";
create table summary as select client, sum(current_os) as tot_os ,
sum(tot_payment)as tot_payment
from account_perf
group by client
order by tot_os descending ;
quit;
```

Line 3: Note the CREATE TABLE <table name> AS statement.

PROC SQL and SELECT statement

We have seen above how SELECT statements are used in PROC SQL. Most of data retrieval and data combining are done using select statement. We will see some sample select statements and how it is used conditionally to work with data. In the example above the simple SELECT statement is shown below.

```
SELECT client, sum(current_os) as tot_os , sum(tot_payment)as
tot_payment

FROM account_perf
```

The SELECT statement must contain a SELECT clause and a FROM clause, both of which are required in a PROC SQL query. Other clauses added to SLECT statements to restrict the data retrieval or conditional processing . Those clauses are WHERE, ORDER BY, GROUP BY and HAVING.

The WHERE clause restrict the data that you retrieve by specifying a condition that each row of the table must satisfy. In our example below, clients are restricted to GAP and Walmart only.

```
SELECT client, sum(current_os) as tot_os , sum(tot_payment)as
tot_payment
FROM account_perf

WHERE client in ('GAP', 'WALMART')
```

ORDER BY sorts the output in ascending or descending order as specified. In our example below total outstanding in sorted in a descending order.

```
SELECT client, sum(current_os) as tot_os , sum(tot_payment)as
tot_payment
FROM account_perf
WHERE client in ('GAP', 'WALMART')
ORDER BY tot_os descending ;
```

GROUP BY computes the statistics for each category of values in the specified variable. A summary or group function like average or SUM in SELECT statement is followed by GROUP BY clause to instruct SAS that the statistics should be computed for each group of data. Let us look at our modified example to see how total outstanding and total payments are computed client wise.

```
PROC SQL;
SELECT client, sum(current_os) as tot_os , sum(tot_payment)as
tot_payment
FROM account_perf
GROUP BY  client;
quit;
```

The HAVING clause works with the GROUP BY clause to restrict the groups in a query's results based on a given condition. PROC SQL applies the HAVING condition after grouping the data and applying aggregate functions. For example, the following query restricts the groups to include only the client GAP.

```
PROC SQL;
SELECT client, sum(current_os) as tot_os , sum(tot_payment)as
tot_payment
FROM account_perf
GROUP BY  client
HAVING Client='GAP';
quit;
```

Data retrieval Methods using SELECT

Using the dataset in above example, we will demonstrate how to retrieve data from a single table and how to create SAS datasets from resultant output.

I. **SELECT – All columns in a Table**

```
PROC SQL;
SELECT * FROM account_perf;
quit;
```

II. **SELECT- Specific columns in a Table**

```
PROC SQL;
SELECT  client,current_os  FROM account_perf;
quit;
```

III. SELECT-How to create dataset from SELECT statements

As we have seen from the previous examples, just add a CREATE TABLE <Table Name> AS before the SELECT statements. So the above example so modified would look like as follows

```
PROC SQL;
CREATE TABLE Sample_Dataset AS
SELECT  client,current_os  FROM account_perf;
quit;
```

IV. SELECT- Eliminating duplicate rows

```
PROC SQL;
SELECT  DISTINCT client  FROM account_perf;
quit;
```

V. SELECT-Computing values

```
PROC SQL;
SELECT  client,(current_os/1000)as OS_in_1000  FROM account_perf;
quit;
```

Note that row level computing can be done using the formula or multiple columns.

VI. SELECT-Assigning Column Alias and formatting it.

We have seen earlier in our examples that a new column name is formed with 'AS' statement in SELECT statement. Its also possible to specify the format of that variable in PROC SQL. Let us have a look at how OS_in_1000 variable is formed.

```
PROC SQL;
SELECT  client,(current_os/1000)as OS_in_1000 format =4.2  FROM
account_perf;
quit;
```

VII. SELECT – Conditional Assignment using CASE

Using CASE statement for conditional processing is a powerful feature of SAS Data step and PROC SQL. Here is an example where in SELECT statement CASE statement is used to create a new field Risk_Category based on certain conditions.

```
PROC SQL;
SELECT  client,current_os,
CASE
WHEN current_os <= 300 THEN 'Low Risk'
WHEN current_os <= 800 THEN 'Med Risk'
ELSE 'High Risk'
END AS Risk_category
from account_perf;
quit;
```

Note that unlike DATA step CASE constructs, in PROC SQL each line does not end with a semi column. Also 'AS' key word is logically follows after the END of the loop.

VIII. SELECT-Specifying COLUMN attributes

You can specify the following column attributes, which determine how SAS data is displayed:

FORMAT=
INFORMAT=
LABEL=
LENGTH=
If you do not specify these attributes, then PROC SQL uses attributes that are already saved in the table or, if no attributes are saved, then it uses the default attributes. Let us have look at an example:

```
PROC SQL;
SELECT  client,current_os format =4.2 label ='Current Outstanding'
FROM account_perf;
quit;
```

IX. SELECT – Using Sub queries

SUB Queries and Queries inside a Query. Instances where the WHERE clause evaluates the output of another SELECT statement, the second SLECT statement is known as a SUB Query. Here is an example:

```
PROC SQL;
SELECT  client,current_os
FROM account_perf
WHERE client IN (SELECT   distinct client  FROM account_perf
where tot_payment >180);
quit;
```

Though not a real life scenario, the above example demonstrates how a Sub-Query is used. The sub-query returns a list of clients that had at least one payment more than $180 and their current outstanding is listed for all accounts. In real life, esp when we work with multiple tables, sub queries are very useful to frame the right WHERE clauses for data retrieval. Now let us look at some conditional operators used in a WHERE clause of a SELECT Statement. Exercise for you is to frame a query using these operators. Consult some online help for syntax help.

Operator	Definition
ANY	Specifies that at least one of a set of values obtained from a sub query must satisfy a given condition
ALL	Specifies that all of the values obtained from a Sub query must satisfy a given condition
BETWEEN-AND	Tests for values within an inclusive range

CONTAINS	Tests for values that contain a specified string
EXISTS	Tests for the existence of a set of values obtained From a sub query
IN	Tests for values that match one of a list of values
IS NULL or IS MISSING	Tests for missing values
LIKE	Tests for values that match a specified pattern

X. SELECT- GROUP FUNCTIONS

There are a lot of group functions we can use in SELECT Statement of a PROC SQL. When you use an aggregate function, PROC SQL applies the function to the entire table, unless you use a GROUP BY clause. Here is an example:

```
PROC SQL;
SELECT client, avg(current_os) as avg_os , avg(tot_payment)as
avg_payment
FROM account_perf
GROUP BY  client;
quit;
```

```
PROC SQL;
SELECT client, avg(current_os) as avg_os , avg(tot_payment)as
avg_payment
FROM account_perf;
quit;
```

When you execute these program blocks, the first PROC SQL computes the averages for each client group and the second PROC SQL computes them for the entire table. Having seen how a group function is used, below given is a list of group functions you can use in a PROC SQL statement. Note that all of these are substitutes for a PROC SUMMMARY or PROC MEANS statistics.

Function	**Definition**
AVG, MEAN	Mean or average of values
COUNT, FREQ, N	Number of nonmissing values
CV	Coefficient of variation (percent)
MAX	Largest value
MIN	Smallest value
NMISS	Number of missing values
RANGE	Range of values
STD	Standard deviation
STDERR	Standard error of the mean
SUM	Sum of values
VAR	Variance

XI. SELECT – Joining the tables

When we work with multiple SAS tables we often will have to join them for data processing. We commonly use SAS DATA step and MERGE method to achieve this task. PROC SQL can be used as a simpler substitute for the MERGE data step method. Let us create another dataset so that we can demonstrate the example.

```
DATA account_perf2;
INPUT  account fico_seg $  prev_os prev_payment  ;
cards;
1002  401-500 400  100
1003  501-600 300  150
1004  601-700 400 180
1005  701-800 900  190
2009  601-700 600  145
1007  401-500 660  210
1008  501-600 750  180
2006  801-900 550  200
2009  601-700 600  145
2110  701-800 400  148
;run;
```

Now to join these tables for all common accounts (equi-join), we use DATA step and MERGE statement as follows. Note that in the data step, dataset should be sorted before we MERGE them.

```
proc sort data=account_perf out=account_perf;
by account;
run;
proc sort data=account_perf2 out =account_perf2;
by account;
run;


Data merged1;
merge  account_perf(in=a) account_perf2(in=b);
by account;
if a=b;

run;
```

Now the same results can be achieved using PEOC SQL as follows.

```
Proc SQL;
create table merged2 as Select a.*, b.*  from  account_perf a,
account_perf2 b
where
a.account=b.account;

quit;
```

The above example demonstrates that how PROC SQL can simplify the coding. Not only that we could avoid the data sort, now we can make use of the powerful WHERE clause to exactly tell SAS various conditions of merging. With the use of Sub-queries and conditional

processing (like IN, NOT IN , LIKE , CONTAIN) in WHERE clause, we can achieve any combination of data merging.

PROC SQL in SAS also provides direct merging of multiple tables with RIGHT JOIN, LEFT JOIN and FULL JOIN keywords for various Outer joins. Readers are requested to explore them as well.

Unix for SAS Analysts

Unix version of SAS is commonly used in organizations with large number of users. Additionally, PC SAS and its Remote SUBMIT features are used to connect to Unix SAS. While working with Unix SAS an analyst or SAS Programmer should be comfortable with a few Unix navigation and utility commands that enables him/her to work independently and efficiently. Here are some topics I thought would be useful.

What we do with Unix in GECF:

- All our servers are Unix based – Model (3.171.120.22), Stoner (3.174.24.73)
- All our Databases are in Unix servers – CDCI, Collections, CDW(Oracle), TS49, CTS49(SAS)
- We store Data in Unix Servers –Store SAS datasets, other files
- We execute SAS in Unix Server – Signon, Rsubmit etc.
- We upload and download files, datasets to and from Unix Servers(server folders)

Unix Server Spaces or Remote Storage

Many a times the data is permanently stored in one of the Unix servers. One should know how to use a library function to store and retrieve a dataset in unix. For example, if the allocated space is in '/projects/dual_cards/jkurian ', to save or retrieve a SAS dataset into this location, you need to declare a library name in SAS program as follows:

```
Libname rloc '/projects/dual_cards/jkurian';

proc datasets lib=rloc;run;
```

Proc datasets will list all the datasets in this location.

Note that in Unix, the '/' sing is used to separate the directories. Reverse is the case when you work with windows.

SASWORK in UNIX

'Saswork' is a location in Unix (server space) where SAS does the data processing by default. It's a shared space where each SAS Unix user is allocated with space to process their request. Typically the size of SASwork runs into 100 plus GBs but due to the number of users and volume of data used, this space gets consumed very fast. Exhausting this space can lead to terminating all the SAS program submitted by multiple users hence it's a responsibility of an analyst to monitor this space.

Let us learn 15 Unix commands listed below. These commands are commonly used in our environment and meet 90% of our requirements while working with Unix SAS.

A List of Useful UNIX Commands

pwd: to see what's the present working directory
Usage: /home/jkurian> **pwd**
ls : Lists the files and folders
ls –l : detailed listing of files and directories
Wildcard usage: ls –l *. sas7*
Usage: /home/jkurian> **ls –l**
mkdir : make a directory
Usage: mkdir <filename>
cd : change into a directory
Usage: cd <folder Name>
cp : to copy a files/directory
Usage: **cp** <filename> <folder name/filename>
rmdir: remove a directory
rm : remove a file
rm -R : Remove files and directories, recursively, empty or not
cat : to read a text file (csv, sas pgm etc) .prints the contents to the screen

chmod : Change file attributes (Read / Write / Executable for owner, group & others)
Usage: **chmod** 777 <filename/folder name>
Tips: 777- all access, 775 – all access to the group, 700- protect files

gzip : to compress a file
Usage: gzip <filename>

gunzip: Unzips the files that are compressed
Usage: gunzip <filename>

grep – a very powerful text searching command. Always used when we want to list the files created by a user or space utilization.
Usage 1: ls -l | grep jkurian – returns all files created by jkurian
Usage 2: grep 'jkurian' *.sas – lists lines where there is a 'jkurian' occurrence anywhere inside the .SAS files
ps -ef - Lists the processes currently running. Useful when we want to kill some program we submitted.
Usage: ps -ef | grep <user id> lists all the process id for that particular user. Process ID or PIDs are required to kill a particular process.

kill –9 - Kills a specified process that's submitted by the user.
Usage : kill –9 <PID> - PID Obtained by using 'ps' command.

df -k . Shows the space utilized and space available for the root folder. Helpful to estimate the space availability. Space is showed in kilobytes.

1. Steps to see the SASWORK space availability

1. Telnet/Logon to the Server (Model or Stoner)
2. Issue command 'cd /saswork' - if /saswork is the work folder
3. Issue command 'df –k .'

The space utilization would be printed to the screen. Don't submit a program unless there is enough space free (At least 10% free)

2. Steps to know how many folders are created in SASWORK

1 Telnet/Logon to the Server (Model or Stoner)
2 Issue command 'cd /saswork'
3 Issue command ls –l | grep 'jkurian'

Substitute your username instead of 'jkurian', to get of folders created by you in SAS work. Its important to clean up the dead or orphan processes in your SASwork to optimize the saswork space.

3. Steps to remove a folder in SASWORK that's no more required

1 Go to /saswork
2 Issue command ls –l | grep 'jkurian' to see the folders created by you
3 Issue command **rm –R** <foldername> to remove the entire folder.

If you want to remove only files inside a folder, do 'cd' into that folder and use 'rm <filename> ' command to remove the file.

4. FTP and How to work with it

File transfer protocol- we use ftp utility/command to transfer files (sas pgms, datasets, text outputs) between servers and also to upload and download between desktop and servers.
FTP from desktop - Go to START/RUN and type ftp <remote server name/ip> (Just like telnet)

FTP between two Unix servers (say model and stoner) - Issue ' ftp <ip address>' command in the Unix shell prompt.

Commonly Used FTP commands:

put : put <filename>
get : get <file name>
mput: mput *.sas –matches the pattern
mget: mget *.sas
prompt: turns the prompt off
bin: sets the transfer mode to binary- use it for excel, datasets etc
lcd : change local directory path
pwd : see present working directory
ls: list files and folders in the remote server.

5. Steps to run a SAS program in a Telnet Session (Unix Server)

First create a SAS program and store it in one of the Unix folders OR FTP a program from the desktop. At the shell prompt issue the following command

```
nohup sas <filename>  &
```

For example:
/home/jkurian> nohup sas scoretest.sas &
nohup: to protect from hang ups – you can close the telnet window and the program would be still running!
Sas: key word to invoke SAS program to execute the program
&: Running the program in background – you can continue work in the same terminal

You can locate the log and output files within the same directory as you submitted the program.

Sum-up With OPTIONS

Objective of this page is to discuss about various options in SAS that can be set locally to control the way SAS process data or display output.

SAS system options control many aspects of your SAS session, including the efficiency of program execution, and the attributes of SAS files and data libraries. These options come with default values supplied by SAS or set locally. Some of these options are often used in our environment.

FIRSTOBS=: causes SAS to begin reading at a specified observation in a data set. If SAS is processing a file of raw data, this option forces SAS to begin reading at a specified line of data. The default is firstobs=1.

```
Usage:  options firstobs=3;
```

OBS= : specifies the last observation from a data set or the last record from a raw data file that SAS is to read. To return to using all observations in a data set use obs=max. This is used when we wanted to read only a few records from large dataset for testing purposes.

```
Usage: options obs=10  ;  options obs=max ;
```

NODATE= : suppresses the printing of date and time to the log and output window. By default, the date and time is always printed

```
Usage: options nodate   ;
```

COMPRESS=: Compresses the SAS dataset being created. COMPRESS is a great space saver . Whenever we work with large data files make sure that this option is explicitly specified as it compressed the dataset up to 65% of its original size.

```
Usage:  options compress =yes|No ;
```

ERRORS= : controls the maximum number of observations for which complete error messages are printed. The default maximum number of complete error messages is errors=20.

```
Usage: options errors =100;
```

There are other options also that help to control the way SAS processes data. Please consult the SAS help for a complete list.

Quick Index