An Introduction to SAS

Feb 15, 2013

What is SAS?

- A programming environment and language for data manipulation and analysis
- SAS software helps companies in every industry transform their data into predictive insights about company performance, customers, markets, risks and more
- Data Warehousing - Easily access, manage and analyze data from many sources
- Analytical Solutions - Provides an integrated environment for predictive and descriptive modeling, data mining, text analytics, forecasting, optimization, simulation, experimental design and more
- Business Solutions – Empowers to solve complex business problems, manage performance to achieve measurable business objectives, drive sustainable growth through innovation and anticipate & manage change

Agenda

This course is designed to introduce basic analysis using SAS. The material presented is universal in nature, applicable for market analysis, data mining or strategic work. The course contains the following modules:

- Overview
- Introduction
  - Getting Started with SAS
  - EXL File Management System
  - Getting Familiar with SAS Datasets
- Data Access
- Data Management
- Data Analysis
  - Summarizing Data
  - Exporting Data
  - Writing Macros
- Presentation
  - ODS
  - Proc Tabulate
  - Proc Report
- Standard Code Library
- Good Coding Practices
- One-day case

Table of Contents

1. Course Goals
2. Overview
   - 2.1 Selecting The Right Data Analysis Tool
3. Introduction
   - 3.1.1 Accessing Servers Using Remote Desktop
   - 3.1.2 Accessing Servers Using Mapped Network Drive
   - 3.2 Getting Started With SAS
     - 3.2.1 Exploring SAS Libraries
     - 3.2.2 Running a SAS Program
   - 3.3 SAS Programs
     - 3.3.1 Debugging a SAS Program
     - 3.3.2 Canceling Submitted Statements
4. EXL File Management System
   - 4.1 BDA Project Life Cycle
   - 4.2 Overall Folder Structure
   - 4.3 File Management Process and Implementation
   - 4.4 Naming Convention
     - 4.4.1 Folder Naming Convention
     - 4.4.2 File Naming Convention
     - 4.4.3 SAS Files Naming Convention
   - 4.5 File Management
   - 4.6 UltraEdit

5. Data Access
    - 5.1 Getting Familiar with SAS Datasets
        - 5.1.1 Descriptor Portion of a Dataset (Proc Contents)
        - 5.1.2 Data Portion of a SAS Dataset
        - 5.1.3 Variable Names and Values
        - 5.1.4 SAS Data Libraries
    - 5.2 Using Formatted Input
        - 5.2.1 Selected Informats
    - 5.3 Importing Data (Fixed Format / Delimited)
    - 5.4 Miscellaneous importing cases
        - 5.4.1 Reading Multiple Records per Observation
        - 5.4.2 Reading “Mixed Record Types”
        - 5.4.3 Sub-setting from a Raw Data File
        - 5.4.4 Multiple Observations per Record
    - 5.5 Import Wizard
    - 5.6 Proc Import
        - 5.6.1 Delimited Text Files
    - 5.7 SAS export File
    - 5.8 Exporting Data From SAS
    - 5.9 Datalines / Cards;
    - 5.10 Importing Tips
    - 5.11 DBMS Copy
    - 5.12 Pitfalls

6. Data Management
    - 6.1 Proc Datasets
    - 6.2 Sas Options
        - 6.2.1 Compressing data
        - 6.2.2 Obs =
        - 6.2.3 Firstobs =
    - 6.3 How SAS works
        - 6.3.1 Compilation Phase
        - 6.3.2 Execution Phase
    - 6.4 Program Data Vector
    - 6.5 Automatic Variables in SAS
    - 6.6 Selecting Variables
        - 6.6.1 Keep= and Drop=
        - 6.6.2 Rename=
    - 6.7 Creating Multiple Datasets in A Single Data-step
    - 6.8 Creating & Modifying Variables
    - 6.9 Subsetting Observations
        - 6.9.1 Conditional SAS Statements
        - 6.9.2 Logical and Special Operators
        - 6.9.3 Where or Subsetting if
    - 6.10 Few efficient practices while doing variable selection

6.11 Dataset Options
6.11.1 _Null_ Datasets
6.11.2 _Last_ Dataset
6.11.3 End=
6.12 Formatting Data Values
6.13 Proc Format

7. SAS Functions
7.1 Syntax for SAS Functions
7.2 Manipulating Character Values
7.2.1 Substring
7.2.2 Right/left
7.2.3 Scan
7.2.4 Concatenation Operator
7.2.5 Trim
7.2.6 Index
7.2.7 Upcase/lowcase
7.2.8 Tranwrd
7.2.9 Compress
7.2.10 Length
7.2.11 IN

7.3 Manipulating Numeric Values
7.3.1 Round
7.3.2 Ceil
7.3.3 The Floor Function
7.3.4 The Int Function
7.3.5 Statistics Functions
7.4 Retain
7.5 SAS Dates
7.5.1 Creating SAS Date Values
7.5.2 Extracting Information
7.5.3 YRDIF Function
7.6 Random Functions
7.6.1 Random Sampling
7.7 Data Conversion
7.7.1 Automatic Character-to-Numeric Conversion
7.7.2 Input
7.7.3 Automatic Numeric-to-Character Conversion
7.7.4 Put
8. SAS by group processing
8.1 BY- Group Processing
8.1.1 Multiple BY Variables

8.2 DO Loop Processing
8.2.1 Conditional Iterative Processing
8.2.2 Iterative DO Statement with a Conditional Clause
8.2.3 Nested DO Loops
8.3 SAS Arrays
8.3.1 The Array Statement
8.3.2 Performing Repetitive Calculations
8.3.3 Creating Variables with Arrays
8.3.4 Assigning Initial Values
8.3.5 Performing a Table Lookup
8.3.6 Rotating a SAS Dataset
8.4 Proc Print
8.5 Proc Sort
8.5.1 Proc Sort with options
8.5.2 De-Duping
8.6 ProcTranspose

9. Data Merging
    - 9.1 Concatenation
    - 9.2 Interleaving
    - 9.3 Proc Append
    - 9.4 One To One Merging
    - 9.5 Match Merging
    - 9.6 IN=
        - 9.6.1 Controlling SAS merge
    - 9.7 Proc Update

10. Data Analysis
    - 10.1 Proc Freq
    - 10.2 Proc Summary
    - 10.3 Proc Means

11. Proc SQL
    - 11.1 SQL Syntax
    - 11.2 SQL Statements
    - 11.3 String Operations
    - 11.4 Nested Subqueries

11.5 Merging Using SQL
11.5.1 Cartesian Join
11.5.2 Merging Using SQL - Joins
11.5.3 Outer Union
11.5.4 Fuzzy Merges
11.5.5 Merging Files with Different Names For Variables
11.6 Copy from multiple sources into single file
11.7 Summarizing Data– Lpd Example
11.8 Comparing SQL and Datastep
12. Macros
12.1 Introduction
12.2 Map of Macro facility
12.2.1 Overview of the map
12.3 Macro Variables
12.3.1 User-defined Macro Variables
12.3.2 Displaying Macro Variable Values
12.3.3 Referencing macros
12.3.4 Macro Quoting
12.3.5 Macro Indirect Referencing
12.3.6 Automatic Macro Variables
12.3.7 Variables and Scope
12.3.8 Rules for Creating and Dereferencing of Variables
12.3.9 Enhancing Macro Programming

12.4 Understanding Macro Parameters
12.4.1 Macro Parameters
12.4.2 Invoking Macros
12.5 Macro Programming
12.5.1 %if…%then…%else and %do…%end
12.5.2 SAS Macro-Function Utilities
12.5.3 Macro Statement Nesting
12.6 Data Step Interface
12.6.1 Call Symput
12.6.2 Symget Functionality
12.7 Error Checking
12.8 Combing SQL and Macro
12.9 Storing Sas Macros
12.10 Compiling macros

13. Reporting
    - 13.1 ODS – Output Delivery System
        - 13.1.1 Creating HTML output
        - 13.1.2 Creating an RTF file
        - 13.1.3 Using ODS with Procedures
        - 13.1.4 Selecting output objects
        - 13.1.5 Proc Template
    - 13.2 Reporting With Excel
        - 13.2.1 DDE
        - 13.2.2 Exporting to Excel
14. Efficient SAS Programming
    - 14.1 Using Excel to Write Code
    - 14.2 Disk Space Saving Measures
    - 14.3 Testing
    - 14.4 Efficient SAS Programming Techniques
    - 14.5 Summary

Appendix

1. Course Goals

- This material will be taught by EXL consultants so as to provide you with a realistic perspective of how we use SAS. We hope that you will be able to leverage our experiences to raise your effectiveness in preparation for your study

- Course Goals:
- Interest all participants in the class
- Teach skills that are immediately useful through practice during training
- Share experiences
- Provide helpful "tricks of the trade“
- Emphasize basic SAS skills
- Expose participants on variety of SAS skills and learning resources

- This Course Will Not:
- Provide a definitive understanding of SAS
- Discuss complex analytical modelling methods
- Make you an expert at analysis
- Give a trailer of coming pressures
- Expose you to pressures of working with a demanding boss

2. Overview

EXL
GO NEXT. NOW.

2.1 Where does SAS fit in the project lifecycle

At EXL, SAS is used as a data management and analysis tool

	Data Access	Data Management	Analysis	Presentation
Description	▪ Access data required by the application	▪ Shape data into desired form	▪ Summarize, reduce or otherwise transform raw data into meaningful and useful information	▪ Communicate information in ways that clearly demonstrate its significance
Tool Used	▪ Excel ▪ SAS	▪ SAS dbms	▪ SAS ▪ Excel	▪ Excel ▪ PowerPoint

Data related activities account for 80% of the process in delivering meaningful information !!

Analysis related activities account for only 20% of the process

2.1 Selecting The Right Data Analysis Tool

A wide range of data analysis/management tools are available to consultants at EXL. Selecting the right tool for the problem you are about to analyse can save you hours of work and frustration.

Today's Focus

	Description	Advantages	Disadvantages	Comments
Excel	▪ Data manipulation and analysis program for which the user provides cell-based data and instructions to produce numerical and graphical output	▪ Cell-based model creation ▪ Ability to model "what-if" scenarios ▪ Comprehensive graphing and print formatting functions	▪ Basic functionality for novice users (extended greatly if you know macros)	▪ Introduction taught during Orientation ▪ Advanced modules available throughout the year
Access	▪ Data storage and retrieval program for which the user provides record and table-based data and instructions to produce numerical, textual (typically tabular) and graphical output	▪ Ability to: – Structure complex and interrelated data – Run complex queries on data – Develop customer applications	▪ Substantial setup time for structuring data and creating tables ▪ Can't easily manipulate data in individual cells	▪ Taught prior to NCT which is held in the winter ▪ Advanced modules available throughout the year
SAS	**High-power record and table-based data analysis tool for managing queries and statistical analysis**	**More powerful analytic functions**	**Limited applicability and access to program**	**Explained in this Course**

Ideal Usage

Excel: General cell- based manipulation

Access: General record-based manipulation

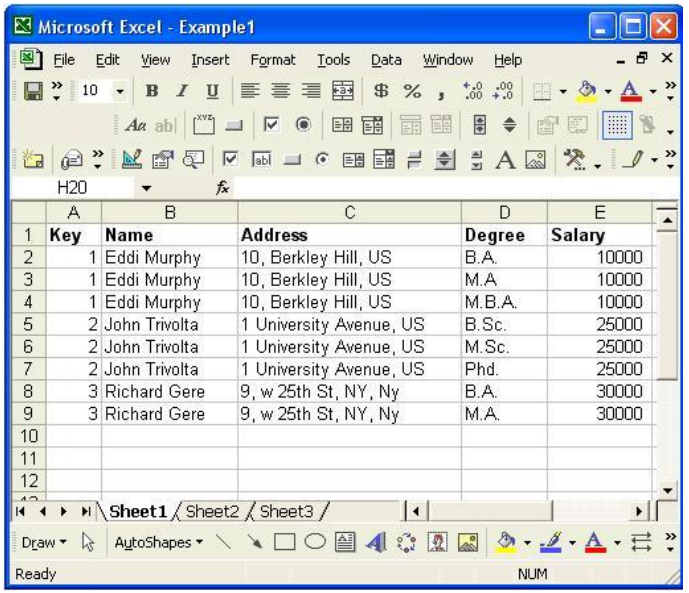
SAS: Focused manipulation of databases or statistical models

The majority of analysis and model building at EXL BDA uses Microsoft Excel and SAS. Although we will be focusing solely on SAS in today's class, it is important for you to understand SAS's capabilities and limitations. An effective consultant realises that it is often appropriate to migrate analysis between SAS and Excel, using the best each tool has to offer

Data Structures in Excel and SAS

Spreadsheet vs. Database Management System (DBMS)

Excel



	A	B	C	D	E
1	Key	Name	Address	Degree	Salary
2	1	Eddi Murphy	10, Berkley Hill, US	B.A.	10000
3	1	Eddi Murphy	10, Berkley Hill, US	M.A	10000
4	1	Eddi Murphy	10, Berkley Hill, US	M.B.A.	10000
5	2	John Trivolta	1 University Avenue, US	B.Sc.	25000
6	2	John Trivolta	1 University Avenue, US	M.Sc.	25000
7	2	John Trivolta	1 University Avenue, US	Phd.	25000
8	3	Richard Gere	9, w 25th St, NY, Ny	B.A.	30000
9	3	Richard Gere	9, w 25th St, NY, Ny	M.A.	30000
10					
11					
12					

Data can move freely between the two applications

Data should be Normalized when moving from Excel to Access

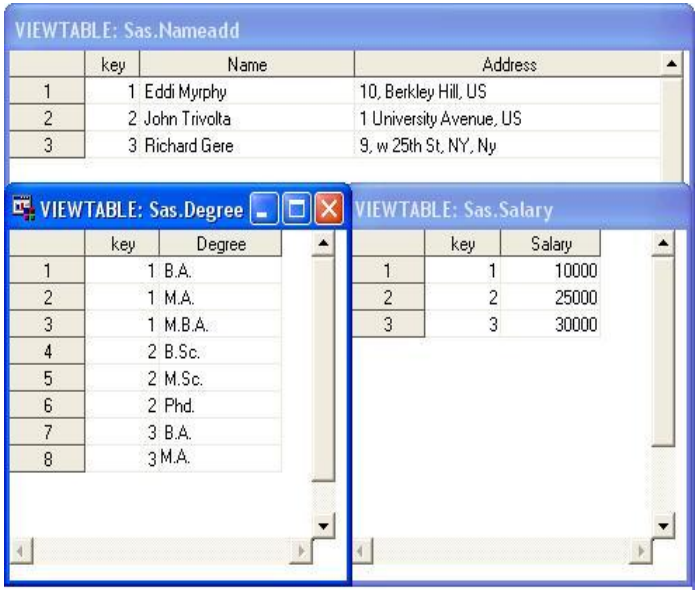
Excel provides:

- Greater functionality for "quick and dirty" text and numerical manipulation
- Greater functionality for making presentable documents and graphs

SAS provides:

- Better querying tools
- A complete developmental environment and programming language for application development

SAS



	key	Name	Address
1	1	Eddi Myrphy	10, Berkley Hill, US
2	2	John Trivolta	1 University Avenue, US
3	3	Richard Gere	9, w 25th St, NY, Ny

	key	Degree
1	1	B.A.
2	1	M.A.
3	1	M.B.A.
4	2	B.Sc.
5	2	M.Sc.
6	2	Phd.
7	3	B.A.
8	3	M.A.

	key	Salary
1	1	10000
2	2	25000
3	3	30000

3. Introduction

GO NEXT. NOW.

3.1.1 Accessing SAS Servers

- We use 3 India servers for SAS Execution and 1 India Server for training folders
  - 172.16.70.31 (Training Data, No SAS)
  - 172.16.70.75 (Login in this server for training purpose, SAS)
  - 172.16.70.76/32 (Project Data, SAS)

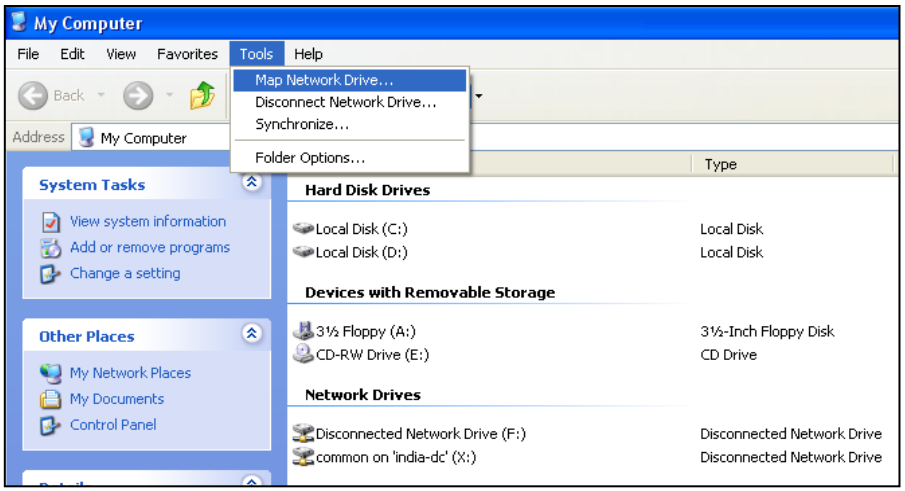
Accessing Servers using Remote Connection

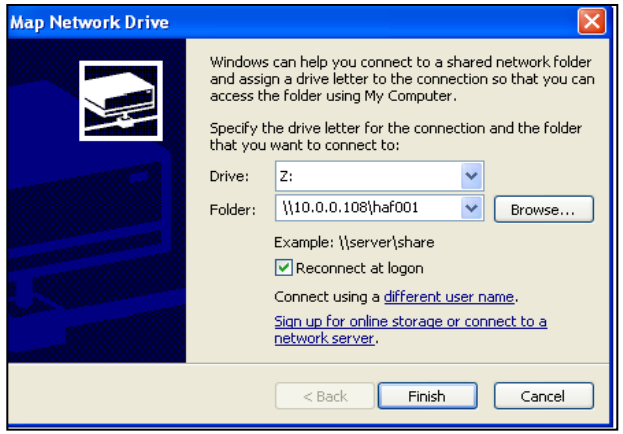
- Click start ->run and type “mstsc”
- Enter the IP Address of server on Remote Desktop Connection
- Enter login (Training id) and password to access the server

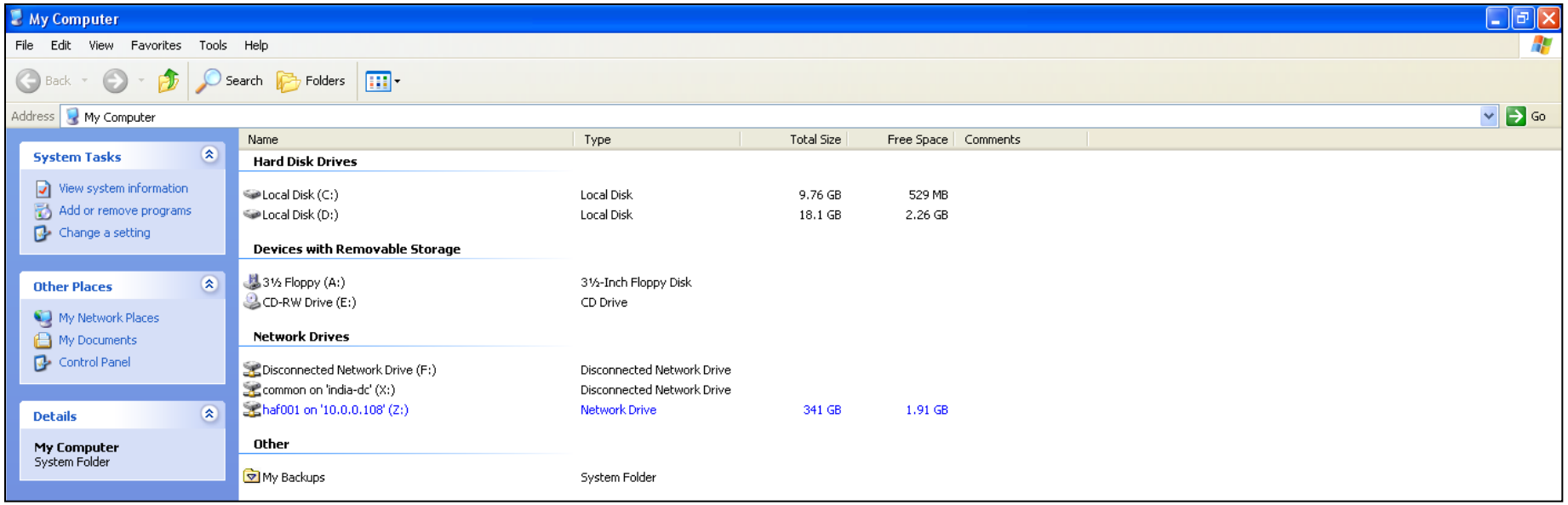
Accessing Servers using Mapped Network Drive

- Open my computer
- Click tools -> Map Network Drive
- Type \\172.16.70.31 , expand the list
- Select the folder IND004 which you want to access
- Click on connected ‘Network Drive’ to access server

Illustrated



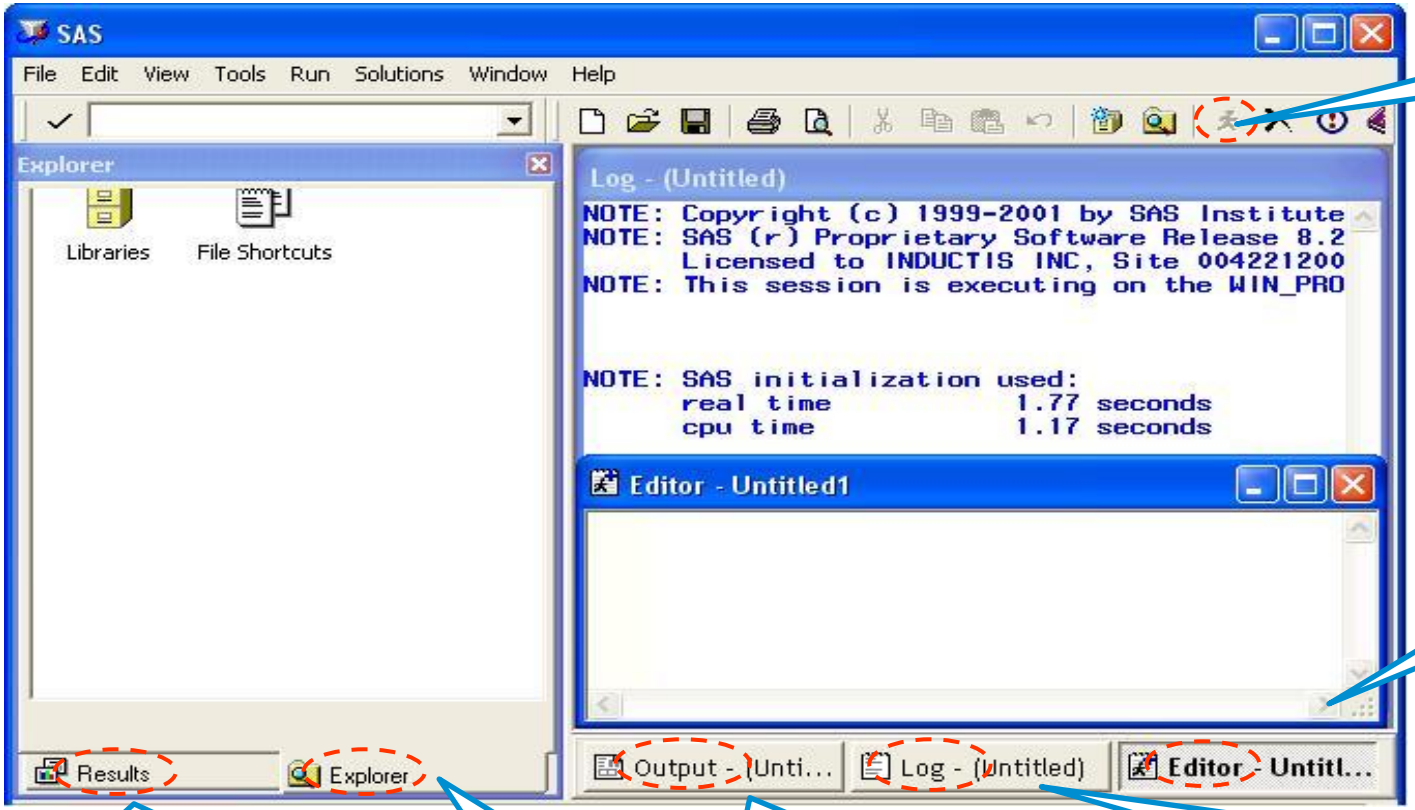




3.2 SAS Window

Interactive windows enable interface with SAS. Login to the India server and double click the SAS icon to start your SAS session

Navigating SAS Windowing Environment



Execute SAS Program

Contains information about the processing of this SAS program, including warning and error messages

Contains reports generated by SAS procedures and DATA steps

View SAS Datasets

Contains reports generated by SAS procedures and DATA steps

Contains information about the processing of this SAS program, including warning and error messages

3.2.1 Exploring SAS Libraries

Select the  Explorer tab in the SAS window bar to open the Explorer window

- Functionality of the SAS explorer is similar to explorers for window-based systems
  - Select view ------ explorer
  - Expand and collapse directories on the left. Drill-down and open specific files in the right
  - Right-click on a SAS dataset and select properties
    - Provides general information about the dataset
  - Double click on the dataset to open it in VIEWTABLE window
    - Can be used to edit datasets, create datasets and customize view of a SAS dataset

3.2.2 Running a SAS Program

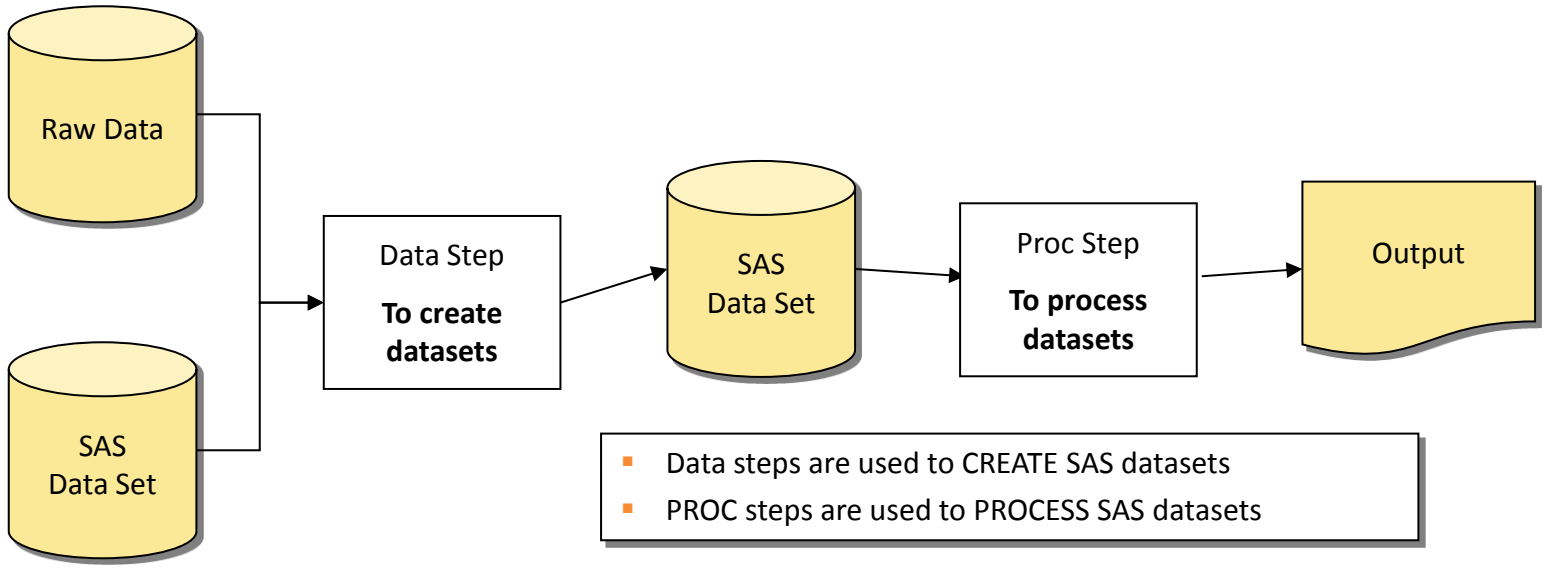
Select file --- open, select the file path D:\Projects\..... Click or  or select run to submit the program for execution

- Functionality of the SAS explorer is similar to explorers for window-based systems
  - Select view ------ explorer
  - Expand and collapse directories on the left. Drill-down and open specific files in the right
  - Right-click on a SAS dataset and select properties
    - Provides general information about the dataset
  - Double click on the dataset to open it in VIEWTABLE window
    - Can be used to edit datasets, create datasets and customize view of a SAS dataset

Enhanced Editor
▪ Access and edit existing SAS programs ▪ Write new SAS programs ▪ Submit SAS programs ▪ Save SAS programs to a file

3.3 SAS Programs

- A SAS program is a sequence of steps that the user submits for execution



SAS Statements

- Usually begin with an identifying keyword
- Always end with a semicolon
- Statements that begin with /* and end with */ are treated as comments
- An additional method used for commenting is adding a asterisk (*) before the SAS statement

SAS Syntax Rules

- SAS Statements are free-format
- One or more blanks or special characters can be used separate words
- They can begin and end in any column
- A single statement can span multiple lines
- Several statements can be on the same line

4. EXL File Management System

4.1 BDA Project Life Cycle

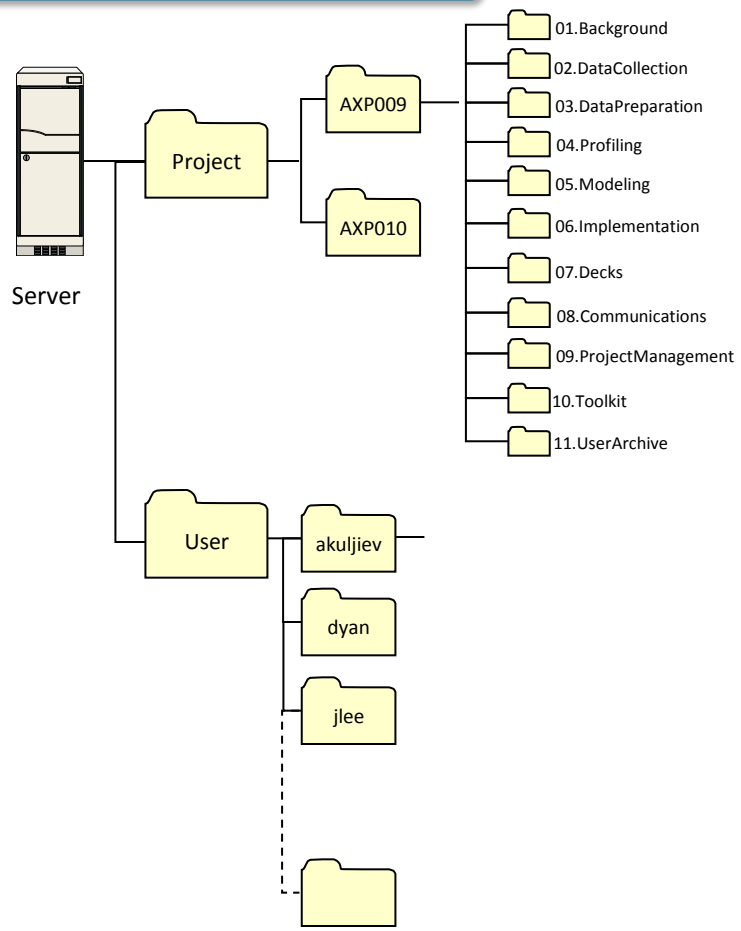
File Management is an integral part of every step of a project lifecycle, and has to be done on a continuing basis, for optimum disk usage, and ease of file access

Data Acquisition	Data Preparation	Profiling	Modeling	Sign off and Disk Management
File transfer (.sas, .txt, .mdb, .xpt, .zip)	Creation of master datasets, drivers, etc	Profiling on key variables, as well as on data sources	Creation of segment level scored / simulated datasets	Back up all files, from project directory
Layout request	EDD generation	Copy final profiling sheets, corresponding datasets and codes	Segment level EDD generation, for every run	Remove all project level user directories
Inventory update	Inventory update for the master datasets	Back-up files	Copy codes, scored datasets, estimates, lorenz curves, etc. to project folder	
Conversion to SAS datasets	Copy datasets and codes to project folder		Back up Files	
EDD Generation				
Move processed datasets to project folder				
Back-up files				

4.2 Overall Folder Structure

The designed folder structure is a generic one, which is to be applied for all BDA projects. Instructions, and guidelines are provided at a higher level , along with specific details for the execution of the same

Folder Structure



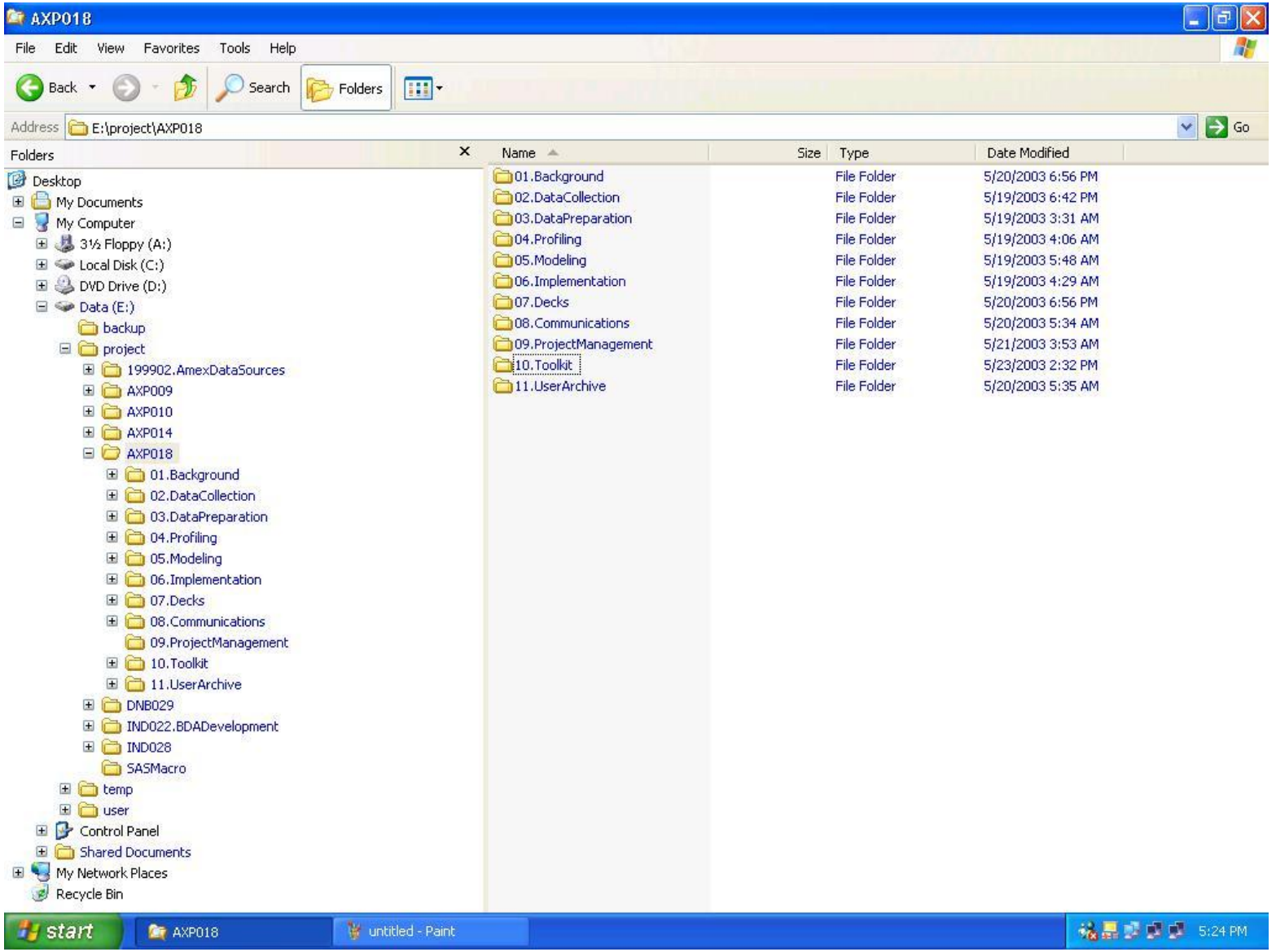
High Level Folder Creation Instructions

- On the highest level, the drive is split in “Project” and “User” folders
- “Project” folder is split by various projects. One folder being created for each project
- “User” folder is first split by various users, e.g. akuljiev, dyan and Jlee and then subsequently each user directory is split by various projects on which the user is working e.g.

  akuljiev - AXP009

  - AXP018
- Folder names at the project level take the project code and those at the user level take the combination of first alphabet of first name and the last name
- Each project level user directory completely mirrors the respective project level directory except for folder 11.UserArchive, which is replaced by 11.Personal

Illustration



4.4 Naming Convention

4.4.1 Folder Naming Convention

The folder names should be split by underscores and not by spaces. The following subfolders should be named as-

- Data Collection: For ”RawData” & “ConvertedData” folders, the following naming convention should be adopted

ProjectCode>_<DateDataAcquired>_<Data description>_<ClientContact>_<InductisContactInitials>

e.g. Folder for the application data acquired on 11th March 2003 from Ren Zang by Vineet Agrwal, would be named as AXP009_20030311_Lease_ Apps_RZang_VA

- Decks: In both the “Internal” and “External” subfolders, the following naming convention should be adopted

<DateofMeeting>_<MeetingChampion>_<MeetingDescription>

e.g. A folder for the decks for the meeting on 4th Dec ’02 with Bob Phelan to present modeling process would be named as
20021204_BobPhelan_MDL_ProcessReview

- For other subfolders, the project and file manager can decide a sub-directory structure and the corresponding names.

4.4 Naming Convention

4.4.2 File Naming Convention

All files other than SAS datasets and SAS codes should be named accordingly to the following naming convention:

- Naming of XLS

<Project Code>_<DateCreated/Changed> _<FileDescription>_<Owner>_<Version

- For the XLS files, which do not feed from a SAS dataset, <Date Created/Changed> field directly follows <Project Code>, Date Created/Changed - Format – YYYYMMDD, File Description - Separated by underscores to put more information, Owner – In case of multiple owners, of a file, all the initials should be included.

  Example: A profiling sheet created by Andrey Kuljiev and updated by Vineet Agrwal, on the modeling dataset for AXP009, on 4th december 2002, would be named as:

  AXP009_20031204_MDL_Profile_AKVA.v1.xls,

  whereas a research document created by Sujan Sreeram on specific industry report would be named as: *AXP009_20030504_IndustryReport_SS.v3.xls*.

4.4.3 File Naming Convention

- Naming of SAS Codes – <ProjectCode>_<Date>_<FileDescription>_<Created/Editedby>_<Version>
- Naming of SAS datasets – <FileDescription>

4.5 Ultra Edit

Ultra Edit is a text editor which facilitates better and quick SAS coding

Key Features

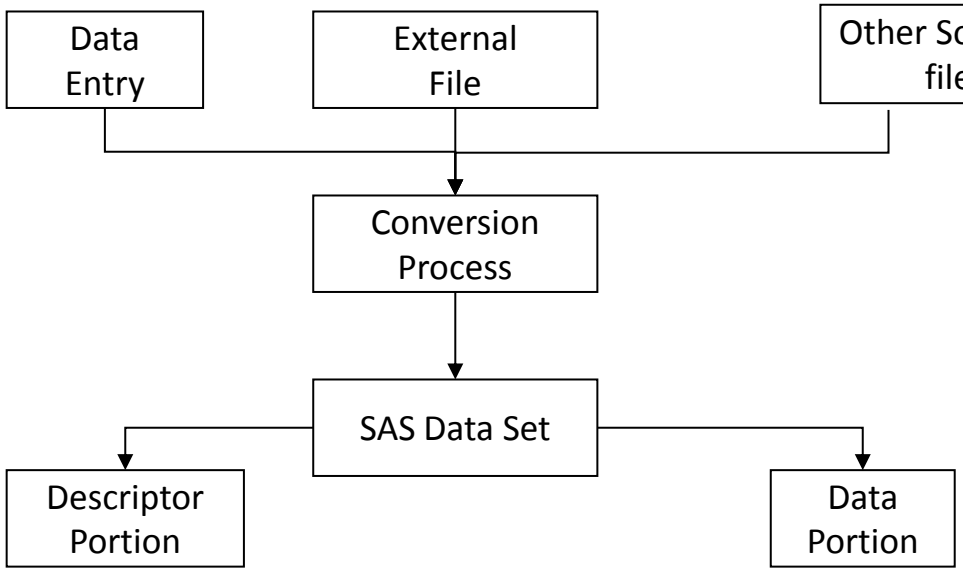
- Block copy and print
  - Press Alt-C
  - Select Block to be copied
  - Copy and paste
- Color indentation
  - Open Advanced -> Configuration -> Syntax Highlighting
  - Select “SAS” as language
  - Set “C:\Program Files\UltraEdit\wordfile.txt” as path name
- Batch submit from editor
  - Open Advanced -> Tool configuration
  - Set “C:\Progra~1\SASINS~1\SAS\V8\sas.exe "%F" -nologo -nosplash -batch -sysin %n -CONFIG h:\Progra~1\SASIns~1\SAS\V8\SASV8.CFG” as command line
  - Set “%P” as working directory
  - Select option “Windows Program” and Insert this as “batch Submit”
- Other options like Line Number, Ruler, Spacing, etc

5. Data Access

5.1 Getting Familiar with SAS Datasets

Data must be in the form of a SAS dataset to be processed by SAS procedures and DATA step statements

SAS Data Sets



5.1.1 Descriptor Portion of a Dataset

The descriptor portion of a SAS dataset contains general information about the SAS dataset (dataset name, number of observations) and variable attributes (name, type, length, position, in format, format, label)

Browsing the Descriptor Portion

General form of the CONTENTS procedure:

```
PROC CONTENTS DATA=SAS-data-set;
RUN;

PROC CONTENTS DATA=SAS-data-set out=SAS
data-set;
Run;
```

Example:

```
proc contents data=work.staff;
run;
```

Partial PROC CONTENTS Output

The SAS System

The CONTENTS Procedure

Data Set Name:	WORK.STAFF	Observations:	18
Number Type:	DATA	Variables:	4
Engine:	V8	Indexes:	0
Created:	18:09 Sunday, July 22, 2001	Observation Length:	48
Last Modified:	18:09 Sunday, July 22, 2001	Deleted Observations:	0
Protection:		Compressed:	NO
Data Set Type:			
Label:		Sorted:	NO

Alphabetic List of Variables and Attributes

#	Variable	Type	Len	Pos
2	First Name	Char	10	28
3	JobTitle	Char	8	38
1	LastName	Char	20	8
4	Salary	Num	8	0

5.1.2 Data Portion of a SAS Dataset

The data portion of a SAS dataset is a rectangular table of data values

SAS Data Sets: Data Portion

The data portion of a SAS data set is a rectangular table of character and / or numeric data values

LastName	FirstName	JobTitle	Salary
TORRES	JAN	Pilot	50000
LANGKAMM	SARAH	Mechanic	80000
SMITH	MICHAEL	Mechanic	40000
WAGSCHAL	NADJA	Pilot	77500
TOERMOEN	JOCHEN	Pilot	65000

Variable names

Variable Values

Character values

Numeric values

- **Variables (Columns) :** Correspond to fields of data, and each data column is named
- **Observations (Rows) :** Correspond to records or data lines

5.1.3 SAS Names and Variable Values

SAS Dataset Names & Variable Names

- Can be 32 characters long
- Can be uppercase, lowercase or mixed-case. Variable names are not case-sensitive.
- Must start with a letter or underscore. Subsequent characters can be letters, underscores or numeric digits (no special character)
- Examples
- Valid names:
  - Data_5
  - bad
  - Cub2c3
- Invalid names
  - Data 5
  - 1bad
  - count # 5

Variable Values

Variable Types

- Character: Contain any value, letters, numbers, special characters, and blanks. Character values are stored with a length of 1 to 32,767 bytes
- Numeric: Stored as floating point numbers in 8 bytes of storage by default
- Date is stored as a numeric variable in SAS. Conversely, any numeric variable may be interpreted as a date.
- Internally, a date value is an integer which represents the number of days since January 1, 1960
- SAS allows dates to be read and output in various format

Stored Value	Format	Displayed Value
0	MMDDYY8.	01/01/60
0	MMDDYY10.	01/01/1960
-1	DATE9.	31DEC1959
365	DDMMYY10.	31/12/1960

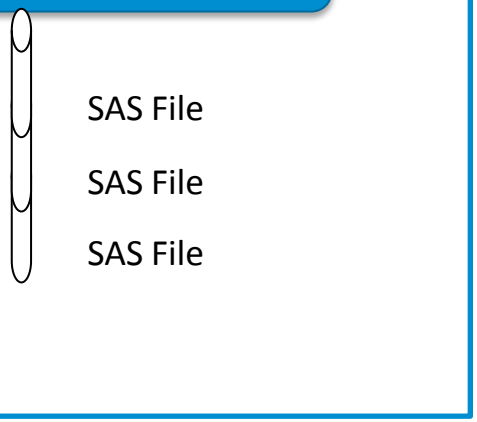
- Today ( ) function returns the current date, A date literal is specified as '<formatted date>' d e.g. '31DEC1959' d

5.1.4 SAS Data Libraries

A SAS data library is a collection of SAS files that are recognized as a unit by SAS

Libname sample “C:\mysasfiles”;

SAS Data Library – Sample



- SAS data libraries are identified by assigning a library reference name
- On invoking SAS, one automatically has access to a temporary and a permanent SAS data library
- Work - Temporary library
- SAS user - Permanent library
- One can also create and access new permanent libraries
- The work library and its SAS data-files are deleted after the SAS session ends
- SAS datasets in permanent libraries are saved after the SAS session ends

Exercise 1

1. Submit the libname statement to provide access to a permanent SAS data library
2. Check the log to confirm that the SAS data library was assigned

5.2 Importing Data (Using Formatted Input)

General form of the INPUT statement with formatted input:

INPUT *pointer-control colname informat ...;*

- Pointer control:

  @*n* moves the pointer to column *n.*

  +n moves the pointer *n* positions.

Formatted input can be used to read non-standard data values by

- moving the input pointer to the starting position of the field
- specifying a column name
- specifying an informat.

An *informat* specifies the width of the input field and how to read the data values that are stored in the field.
General form of an informat:

$informat-namew.d

$	indicates a character format.
informat-name	names the informat.
w	is an optional field width.
.	is the required delimiter.
d	optionally, specifies a decimal for numeric informats.

5.2.1 Selected Informats

7. or 7.0	reads seven columns of numeric data.
7.2	reads seven columns of numeric data and inserts a decimal point in the data value.
$5.	reads five columns of character data and removes leading blanks.
COMMA7.	reads seven columns of numeric data and removes selected nonnumeric characters, such as dollar signs and commas.
MMDDYY10.	reads dates of the form 01/20/2000.
DDMMYY8.	Reads dates of the form 01.01.04
$w.	Reads character data-trims leading blanks
$CHAR5	reads five columns of character data and preserves leading blanks

5.3 Importing Data (Fixed Format / Delimited)

Raw Files

```
Infile "X:\raw-file"
 LRECL = <length-of-observation> MISSOVER;
 Input  @<start-of-var1>     var1    <length-of-var1>.
      @<start-of-var1>     var2    <length-of-var2>.
      .
      .
      @<start-of-var1>     var3    $<length-of-var3>.;
```

To read a fixed file format raw file, one need to know the exact position from where each of the variables start and length of the variable

- For all char variable $ symbol is used while declaring its length.
- If no $ symbol is used that variable by default is taken as numeric
- The Missover option prevents SAS from loading a new record when the end of the current record is reached. If SAS reaches the end of the row without finding values for all fields, variables without values are set to missing.

Note: Quality Check should always follow converting raw data to SAS dataset

Example

Convert a fixed format file (YYY.txt) to SAS Dataset

Layout of YYY.txt	Start	End	Length	Type	Variable	Description
	1	9	9	Num	DUNS	Duns Number
	10	39	30	Char	COMP_NAME	Company Name
	40	64	25	Char	STREET	Address - Street
	65	84	20	Char	CITY	Address - City
	85	86	2	Char	STATE	Address - State
	87	89	3	Char	ZIP	Address - ZIP
	90	99	10	Num	PHONE	Telephone Number

```
data <dataset>;
 infile “X:\YYY.txt"
 LRECL = 99 MISSOVER;
 input  @1     DUNS_NUM          9.
        @10    COMP_NAME      $30.
        @40    STREET           $25.
        @65    CITY             $20.
        @85    STATE             $2.
        @87    ZIP               $3.
        @90    PHONE             10. ;
run;
```

5.3.2 Delimited

Syntax

INFILE “<Location and filename of input data>” LRECL = <width of data> MISSOVER DSD DLM=<delimiter> FIRSTOBS=n;

- DSD - It has two functions. The first function is to strip off any quotes that surround variables in the text file. The second function deals with missing values. When SAS encounters consecutive delimiters in a file, the default action is to treat the delimiters as one unit. If a file has consecutive delimiters, it’s usually because there are missing values between them. DSD tells SAS to treat consecutive delimiters separately; therefore, a value that is missing between consecutive delimiters will be read as a missing value when DSD is specified
- DLM - This option allows you to tell SAS what character is used as a delimiter in a file. If this option is not specified, SAS assumes the delimiter is a space. Some common delimiters are comma, vertical pipe, semi-colon, and tab
- FIRSTOBS= - This option indicates that you want to start reading the input file at the record number specified, rather than beginning with the first record. This option is helpful when reading files that contain a header record, since a user can specify FIRSTOBS=2 and skip the header record entirely

Example

```
FILENAME MYFILE "C:\Work\IND032\SAS_Training\Exercises_Solution_Datasets\3\comp_pipe.txt";
DATA A;
 INFILE MYFILE DLM='|' DSD LRECL=1000 MISSOVER;
 INPUT KEY1  :$100.
       KEY2 :$100.
       NAME :$100.
       PREFER :$100.
       AMOUNT :$10
RUN;
```

If the character variables are longer than 8 bytes or if there are variables such as dates, times, packed decimal, etc., one needs to use an INFORMAT and specify a WIDTH. This is called FORMATTED INPUT. Using formatted input to read a delimited file can cause problems, because SAS ignores the delimiter and reads the number of bytes specified by the INFORMAT. This problem can be corrected by using the COLON (:) FORMAT MODIFIER, which specifies modified list input. The FORMAT MODIFIER tells SAS to read up to the maximum number of bytes specified in the INFORMAT, OR until it reaches a delimiter

5.4 Miscellaneous Importing Cases

Controlling When a Record Loads

We will be looking at following complex cases where controlled reading of raw data is involved:

- Read a raw data file with multiple records per observation
- Read a raw data file with mixed record types
- Subset from a raw data file
- Read a raw data file with multiple observations per record

5.4.1 Reading Multiple Records per Observation:

A raw data file has three records per employee. Record 1 contains the first and last names, record 2 contains the city and state of residence, and record 3 contains the employee's phone number.

```
Farr, Sue
Anaheim, CA
869-7008
Anderson, Kay B.
Chicago, IL
483-3321
Tennenbaum, Mary Ann
Jefferson, MO
589-9030
```

Desired Output:

The SAS data should have one record per employee:

Lname	FName	City	State	Phone
Farr	Sue	Anaheim	CA	869-7008
Anderson	Kay B.	Chicago	IL	483-3321
Tennenbaum	Mary Ann	Jefferson	MO	589-9030

Multiple Ways of performing the above task

The INPUT Statement

The SAS System loads a new record into the input buffer when it encounters an INPUT statement.
So the task can be accomplished as:

Multiple Input Statements

```
Data Address;
 length Lname FName $ 20
        City $ 25 State $2
        Phone $8 ;
 infile 'raw-data-file' dlm;
 input Lname $ FName $ ;
 input City $ State $ ;
 input Phone $;
 run;
```

Load Record

Line Pointer Controls

You can also use line pointer controls to control when SAS loads a new record

SAS loads the next record when it encounters a forward slash.

Use of Line Pointer Controls

```
Data Address;
 length Lname FName $ 20
        City $ 25 State $2
        Phone $8 ;
 infile 'raw-data-file' dlm=',';
 input Lname $ FName $/
        City $ State $ ;/
        Phone $;
run;
```

Load Record

Load Record

Multiple Ways of performing above task..cont'd

The forward slash is known as a *relative* line pointer control because it moves the pointer relative to the line on which it currently appears.

There is also an *absolute* line pointer control that moves the pointer to a specific line. #*n* moves the pointer to line *n*.

Syntax

```
Data Example;
        infile 'raw-data-file;
         input  #1 Lname $ FName $
                #2 City $ State $
                 #3 Phone $;
 run;
```

5.4.2 Reading “Mixed Record Types”

Not all records have same format

```
101  USA 1-20-1999  3295.50
3034  EUR 30JAN1999  1876,30
101  USA 1-30-1999  2938.00
128  USA 2-5-1999  2908.74
1345  EUR 6FEB1999  3135,60
109  USA 3-17-1999  2789.10
```

Desired Output:

SalesId	Location	SaleDate	Amount
101	USA	14264	3295.50
3034	EUR	14274	1876.30
101	USA	14274	2938.00
128	USA	14280	2908.74
1345	EUR	14281	3145.60
109	USA	14320	2789.10

Say we use multiple input statements for the above case:

Example

```
input SalesID $ Location $ ;
 if Location='USA' then
                input SaleDate : mmddyy10.
                    Amount;
 else if Location='EUR' then
   input SaleDate: date9.
       Amount: comma8.;
run;
```

What's the flaw with above approach?

The flaw can be explained by the following step of execution:

Raw Data File

```
101  USA 1-20-1999  3295.50
3034  EUR 30JAN1999  1876,30
101  USA 1-30-1999  2938.00
128  USA 2-5-1999  2908.74
1345  EUR 6FEB1999  3135,60
109  USA 3-17-1999  2789.10
```

True

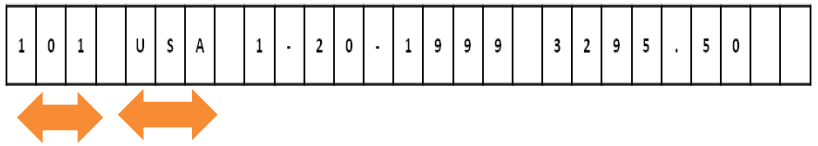
```
Data Sales;
Length SalesId $4
        Location $3;
Infile 'raw-data-file';
input SalesID $ Location $ ;
 if Location="USA" then
        input SaleDate
:mmddyy10.
                Amount;
 else if Location="EUR" then
        input SaleDate: date9.
        Amount: commax8.;
 run;
```

Raw Data File

```
101  USA 1-20-1999  3295.50
3034  EUR 30JAN1999  1876,30
101  USA 1-30-1999  2938.00
128  USA 2-5-1999  2908.74
1345  EUR 6FEB1999  3135,60
109  USA 3-17-1999  2789.10
```

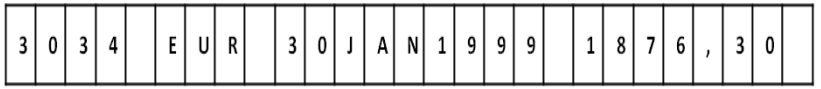
```
Data Sales;
Length SalesId $4
        Location $3;
Infile 'raw-data-file';
input SalesID $ Location $ ;
 if Location="USA" then
        input SaleDate :mmddyy10.
                Amount;
 else if Location="EUR" then
        input SaleDate : date9.
        Amount : commax8.;
 run;
```

Input Buffer



SALESID	LOCATION	PDV	SALEDATE	AMOUNT
101	USA	.	.	

Input Buffer



SALESID	LOCATION	PDV	SALEDATE	AMOUNT
101	USA		14264	3295.50

Concept of Single Trailing "@"

The single trailing @ option holds a raw data record in the input buffer until SAS

- **Executes an INPUT statement with no trailing @, or**
- **Reaches the bottom of the DATA step**

General form of an INPUT statement with the single trailing @:

Syntax

Input var1 var2 var3…@;

So using the concept of trailing, the before mentioned problem can be solved as:

Load Next Record

Hold record for next Input statement

```
input SalesID $ Location $  @;
 if Location='USA' then
    input SaleDate : mmddyy10.
          Amount;
 else if Location='EUR' then
   input SaleDate: date9.
         Amount: commax8.;
```

Note: '@' hold record for next input statement

Concept of Single Trailing "@"…cont'd

So single trailing holds the data in input buffer until it a run or an output statement is encountered.

Raw Data File

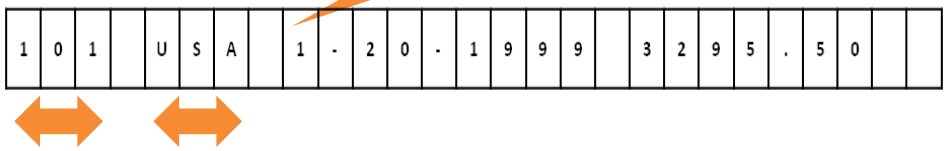
```
101  USA 1-20-1999  3295.50
3034  EUR 30JAN1999  1876,30
101  USA 1-30-1999  2938.00
128  USA 2-5-1999  2908.74
1345  EUR 6FEB1999  3135,60
109  USA 3-17-1999  2789.10
```

True

```
Data Sales;
Length SalesId $4
        Location $3;
Infile 'raw-data-file';
nput SalesID $ Location $ @ ;
if Location="USA" then
        input SaleDate :mmddyy10.
                Amount;
else if Location="EUR" then
        input SaleDate: date9.
        Amount: commax8.;
run;
```

Input Buffer

Hold Record



PDV

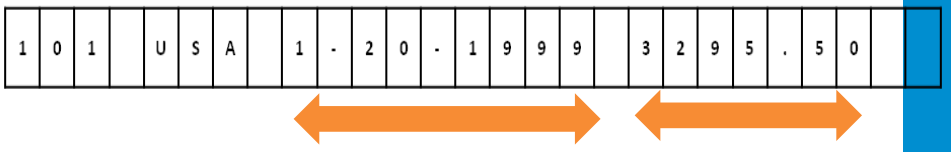
SALESID	LOCATION	SALEDATE	AMOUNT
101	USA	.	.

Raw Data File

```
101  USA 1-20-1999  3295.50
3034  EUR 30JAN1999  1876,30
101  USA 1-30-1999  2938.00
128  USA 2-5-1999  2908.74
1345  EUR 6FEB1999  3135,60
109  USA 3-17-1999  2789.10
```

```
Data Sales;
Length SalesId $4
        Location $3;
Infile 'raw-data-file';
input SalesID $ Location $@ ;
 if Location="USA" then
        input SaleDate :mmddyy10.
                Amount;
 else if Location="EUR" then
        input SaleDate : date9.
        Amount : commax8.;
 run;
```

Input Buffer



PDV

SALESID	LOCATION	SALEDATE	AMOUNT
101	USA	14264	3295.50

5.4.3 Sub setting from a Raw Data File

The Scenario uses raw data same as for the previous example:

```
101  USA 1-20-1999  3295.50
3034  EUR 30JAN1999  1876,30
101  USA 1-30-1999  2938.00
128  USA 2-5-1999  2908.74
1345  EUR 6FEB1999  3135,60
109  USA 3-17-1999  2789.10
```

Desired Output:

SalesId	Location	SaleDate	Amount
3034	EUR	14274	1876.30
1345	EUR	14281	3145.60

Use of “Subsetting if” to get to desired output:

The Subsetting IF Statement

```
data Europe;
 length SalesId $4
        location $3;
 infile ‘raw-data-file’;
 input SalesID $ Location $ @;
 if Location=‘USA’ then
    input SaleDate : mmddyy10.
          Amount;
 else if Location=‘EUR’ then
     input SaleDate: date9.
           Amount: commax8.;
  if Location=‘EUR’;
run;
```

Can you think of a better way to use the “Subsetting“ in above case ?

5.4.3 Sub setting from a Raw Data File …cont'd

The subsetting IF should appear as early in the program as possible but after the variables used in the condition are calculated

So on above basis an efficient approach for above case is:

The Subsetting IF Statement

```
data Europe;
  length SalesId $4 location $3;
  infile 'raw-data-file';
  input SalesID $ Location $ @;
  if Location="EUR" then
    input SaleDate: date9.
          Amount: comma8.;
run;
```

Because the program needs only European sales,
the INPUT statement for USA sales is not needed

If an observation does not meet the subsetting IF,

- control returns to the top of the DATA step
- the PDV is reset
- a new record is read.

The observation never reaches the bottom of the DATA step and is therefore never output.
Whereas if the subsetting IF condition is true, SAS continues processing the current observation until it reaches the bottom of the DATA step and the implicit output

5.4.4 Multiple Observations Per Record

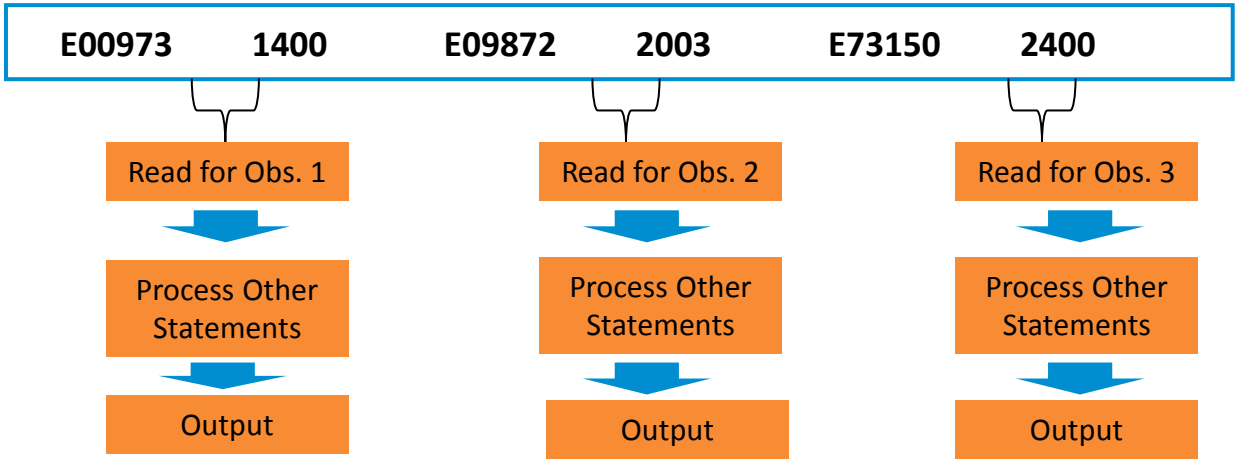
A raw data file contains each employee's identification number and this year's contribution to his or her retirement plan. Each record contains information for multiple employees.

E00973	1400	E09872	2003	E73150	2400
E45671	4500	E34805	1980		

Desired Output:

EmpId	Contrib
E00973	1400
E09872	2003
E73150	2400
E45671	4500
E34805	1980

Processing: What is Required?



Concept of Double Trailing "@@"

The double trailing @ holds the raw data record across iterations of the DATA step until the line pointer moves past the end of the line.

Syntax

Input var1 var2 var3…@@;

Note: The double trailing @ should only be used with list input. If used with column or formatted input, an infinite loop can result.

Example

```
Data work.retire;
    Length EmpId $6;
    infile 'raw-data-file';
    input EmpID $ Contrib @@;
Run;
```

Hold until end of record

Single Trailing v/s Double Trailing

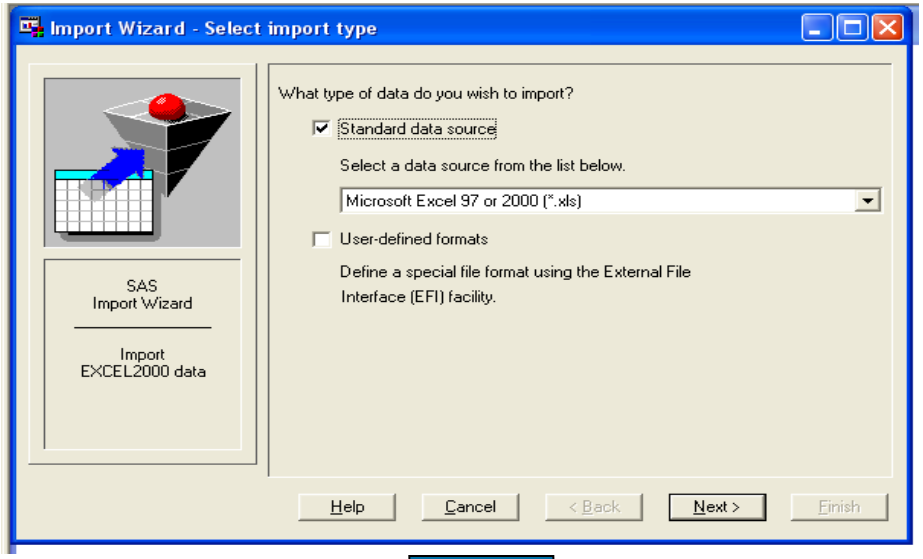
Option	Effect
Trailing @ INPUT var-1...@;	Holds raw data record until 1) An input statement with no input trailing @ 2) The bottom of the DATA step.
Double Trailing @ INPUT var-1 ...@@;	Holds raw data records in the input buffer until SAS reads past the end of the line

Note: The single trailing @ and the double trailing @ are mutually exclusive; they cannot and should not be used together. If they both appear in the same INPUT statement, the last option specified is used.
The MISSOVER option is also invalid with the double trailing @@.

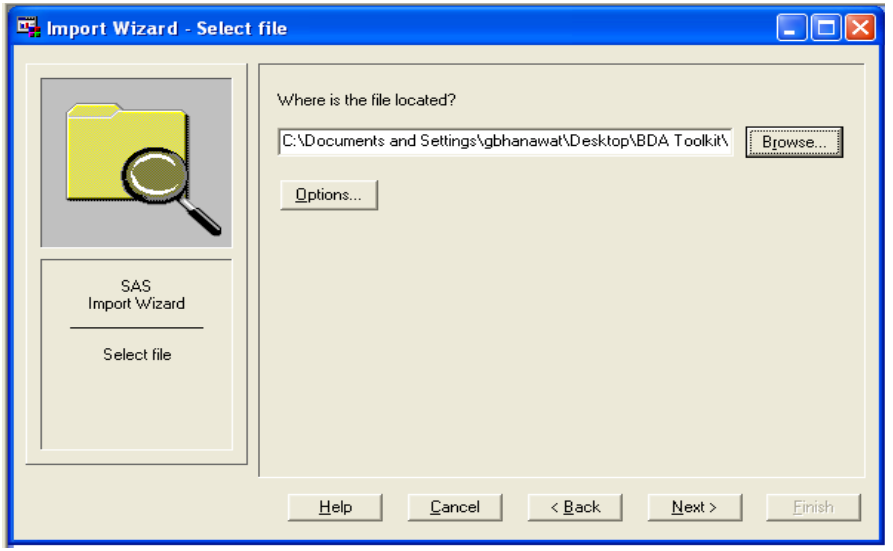
5.4.5 Import Wizard

Wizard is the SAS provided graphical interface to convert raw data file to SAS dataset. It can only convert Delimited and files to SAS files

Select the type of raw file which is to be imported

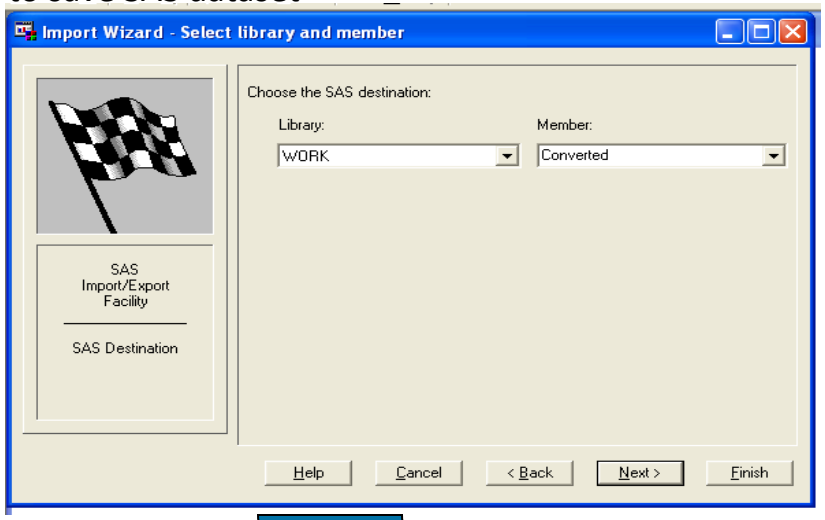


Browse to the raw file

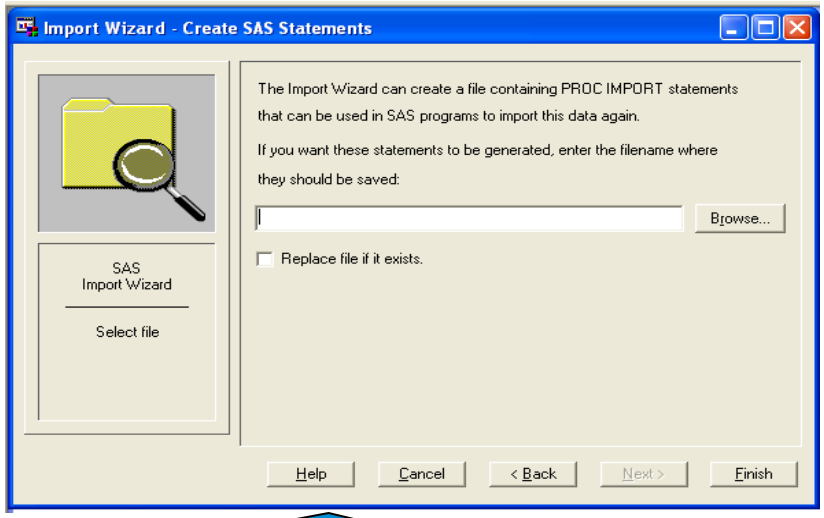


5.4.5 Import Wizard…cont'd

Enter the library name and name where you want to save SAS dataset



Press "Finish" to convert raw file to SAS dataset



Note: Import Wizard basically first generate PROC IMPORT code and then executes it. You can save the code which wizard generates as well.

5.5 Proc Import

General form of the IMPORT procedure

Syntax

```
PROC IMPORT OUT=SAS-data-set
                DATAFILE='external-file-name'
                DBMS=file-type;
                GETNAMES=YES;
RUN;
```

Example: Following code converts CSV file to SAS dataset

Example

```
PROC IMPORT DATAFILE='D:\fun\Ritesh Training\comp.csv' out=yyy
                DBMS=CSV REPLACE;
                GETNAMES=YES;
RUN;
```

How to import excel files?

5.5.1 Delimited Text File

PROC Import with slight change can read the delimited file.

General format

```
PROC IMPORT OUT=SAS-data-set
            DATAFILE='external-file-name'
            DBMS=<approptiate Delimiter>  REPLACE;
            GETNAMES=YES;
Run;
```

Example: Following code converts tab delimited file to SAS dataset

Example

```
PROC IMPORT DATAFILE = 'D:\fun\Ritesh Training\Broker comp file.txt'
            OUT=xxx
            DBMS=TAB REPLACE;
            GETNAMES=YES;
run;
```

5.6 SAS xport File

PROC CIMPORT is used to read SAS xport files

Syntax

PROC CIMPORT destination=libref | <libref.> member-name<option(s)>;

- **Destination -** identifies the file or files in the transport file as a single catalog, as a single SAS data set, or as the members of a SAS data library & can be one of the following: CATALOG | CAT | CDATA | DS | DLIBRARY | LIB | L

- **Libref | <libref. > member-name -** specifies the specific catalog, SAS data set, or SAS data library as the destination of the transport file. If the libref is omitted, Proc Cimport uses the default library as the libref, which is usually the work library.

Example

```
libname db 'D:\fun' ;
FILENAME IN1 'D:\fun\sas.xport';
PROC CIMPORT LIBRARY=db INFILE=IN1;
RUN;
```

5.7 Exporting Data From SAS

A SAS dataset can be converted into other file formats by using either “proc export” or the SAS “export wizard”.
The following code segment illustrates the use of the export procedure in SAS to output a file in the CSV format.

Syntax

```
PROC EXPORT DATA= <Name of Dataset>
        OUTFILE= <Output Filename>
        DBMS=CSV REPLACE;
            RUN;
```

Note: The output filename should be given under quotes with the full path.

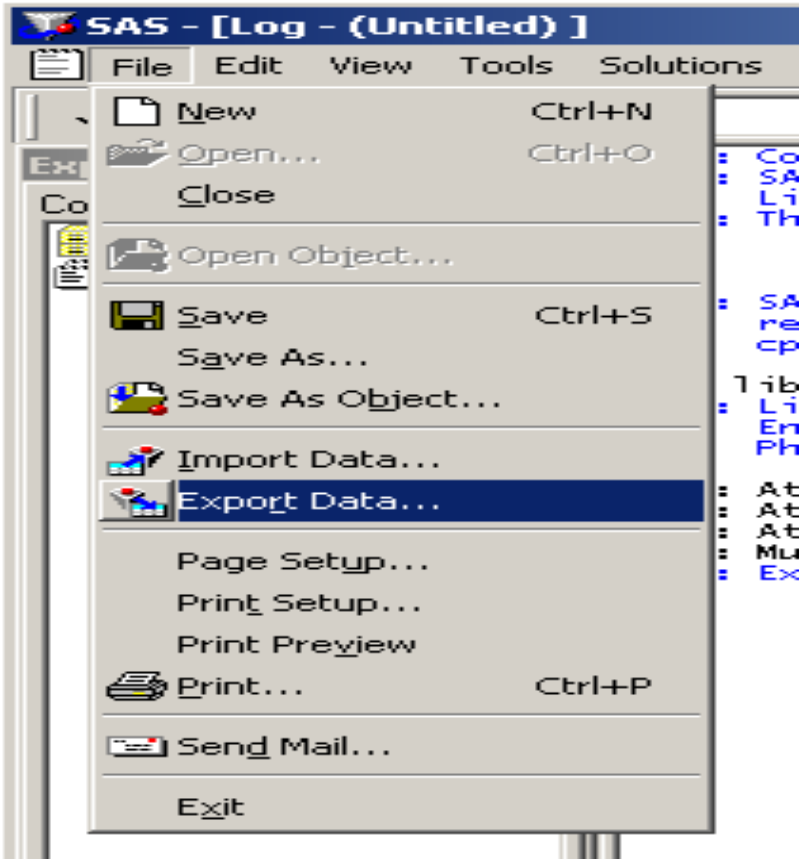
Exercise

Ex 1: Given the dataset, new residing in folder 5”, write a syntactically correct SAS code to convert it to a CSV file with the same name.

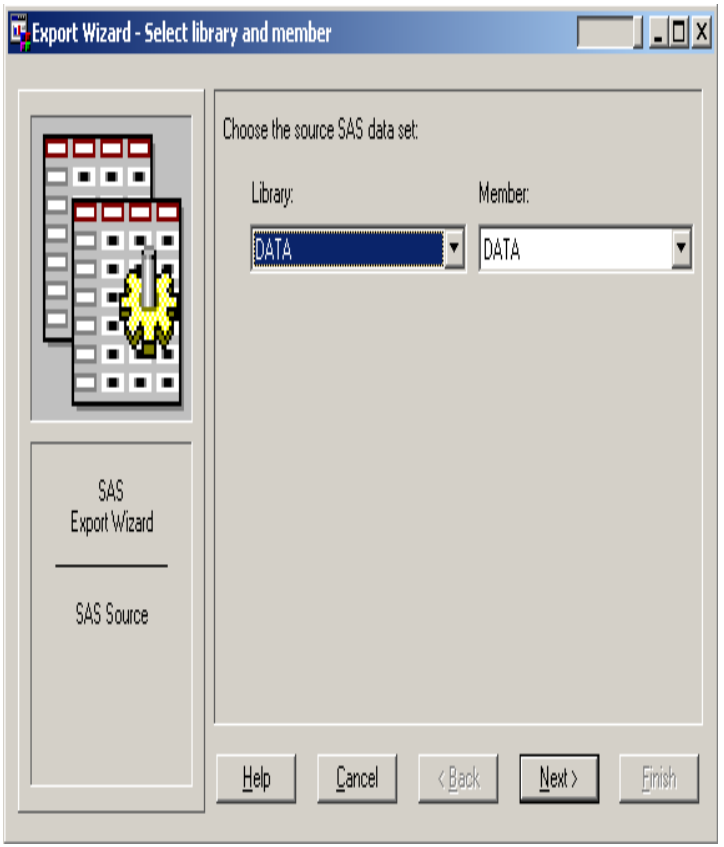
Exporting Data From SAS…cont'd

The SAS "export wizard" allows us to convert a SAS dataset into other file formats without having to write any code.

Step 1: Click on file and select "Export Data"



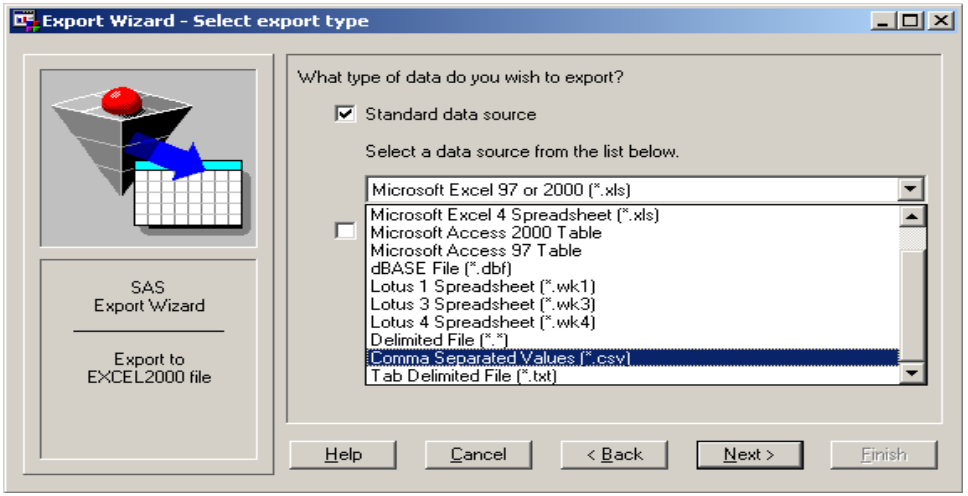
Step 2: Select the Data to be exported



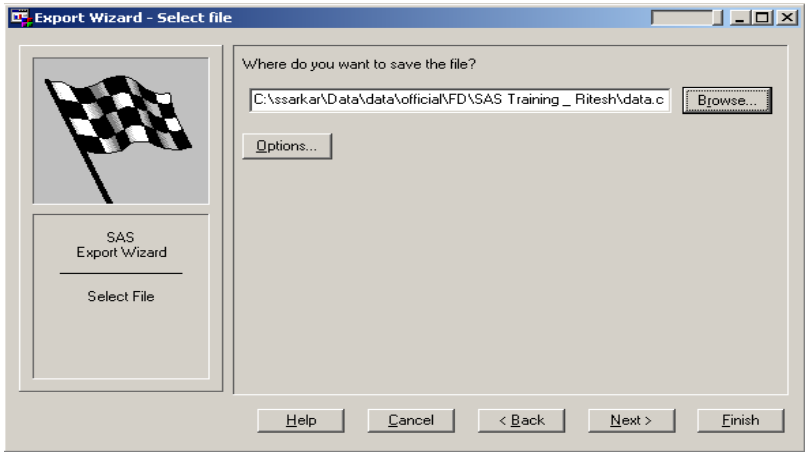
Exporting Data From SAS…cont'd

The SAS "export wizard" also allows us to save the corresponding "proc export" code

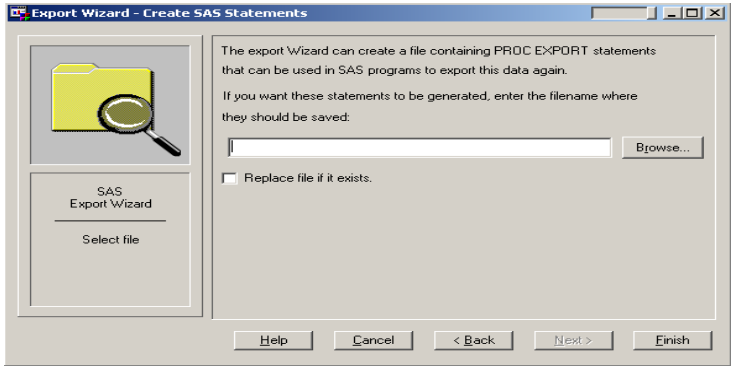
Step 3: Select the file format



Step 4: Specify the output filename and its location



Step 5: Enter the filename to save the code for export



5.8 Datelines/ Cards

- Datalines/cards is a dummy filename used to read data values from program itself.
- While reading a file using a Cards or Datalines statement, you can use the same options as those listed for the Infile statement. Simply replace the name of the file with either Cards or Datalines . You need to use the dsd option on the infile statement if two consecutive delimiters are used to indicate missing values (e.g., two consecutive commas, two consecutive tabs).

Example

```
data temp;
infile DATALINES dsd missover;
input a b c d;
CARDS;
 1, 2, 3, 4
, 3, , 5 ,
 3
 ;
run;
```

Note: Cards and Datalines are interchangeable and either word can be used in either place the data step.

5.9 Importing Tips

- **Input data can come in variety of formats**.
- **List directed input** - data must be separated by a delimiter; must read in all variables. In case of delimited data the data values are separated by a specially designated character called the delimiter. For example, in case of comma separated values, the comma separates individual data values from each other.
- **Column Input** - data in fixed columns;must know where data starts and ends; can read in selected variables. In fixed format files the data values are placed at pre-specified column addresses in the data file.
- **Informat** - alternative to column input; most flexible; must be used for special data
- **Input data can have variable names as part of the data values**. In case if the data values have the names of the variables specified in the top most row of the file, then one can use PROC IMPORT;

	Fixed Format	Delimited
Names Available	PROC IMPORT (Use Wizard)	PROC IMPORT
Raw Data	INFILE/INPUT @ signifies the start of the data value	INFILE / INPUT DLM OPTION

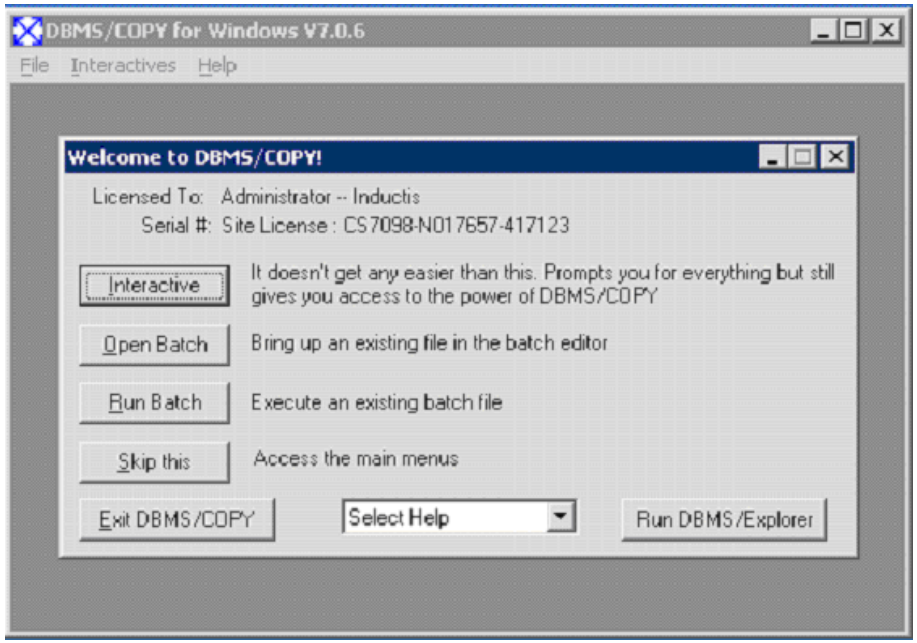
Exercise 2 :

Import all the raw data files in the folder 5

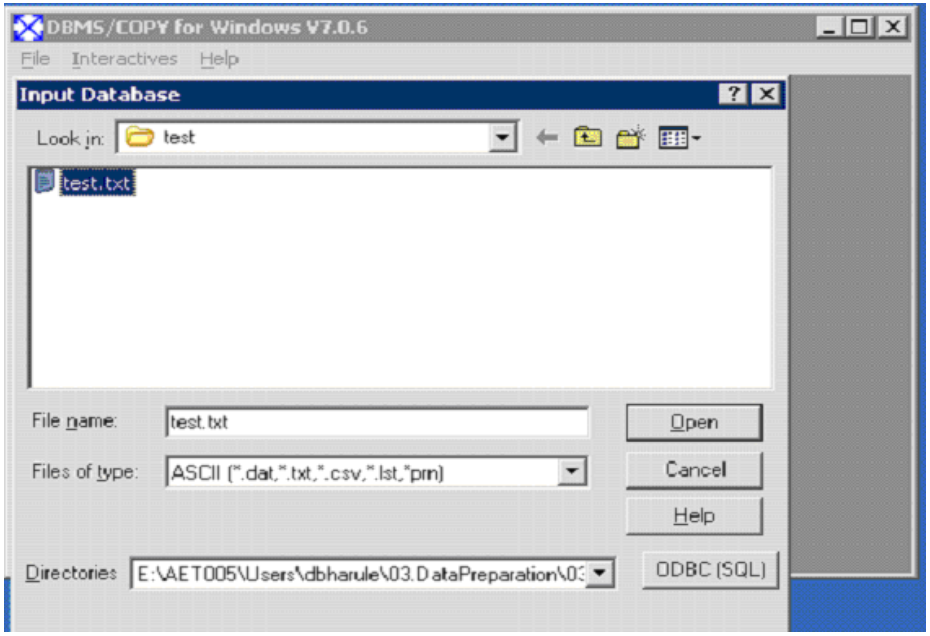
5.9 DBMS Copy

DBMS is a generic tool which can convert one file format to other. Select SAS7BDAT as destination format to convert raw file to SAS dataset

Click on “Interactive”



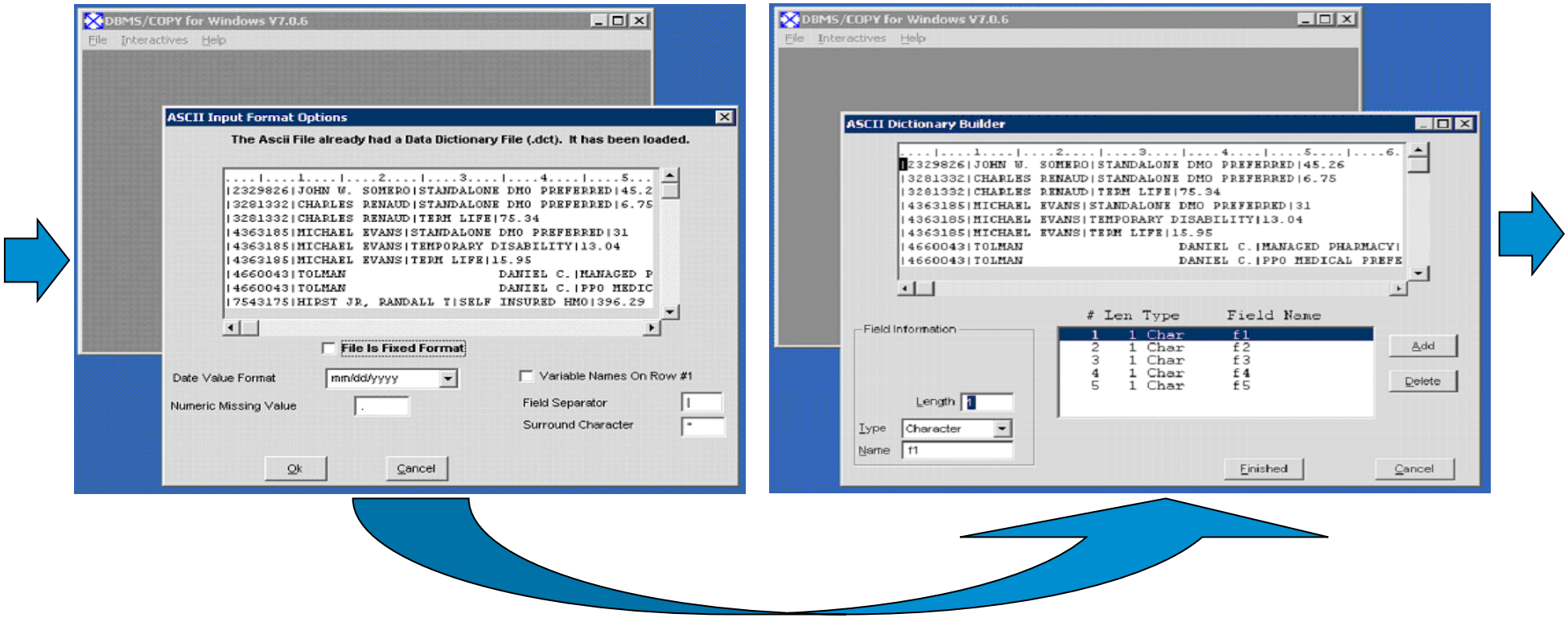
Select the text file which is to be exported



5.9 DBMS Copy…cont'd

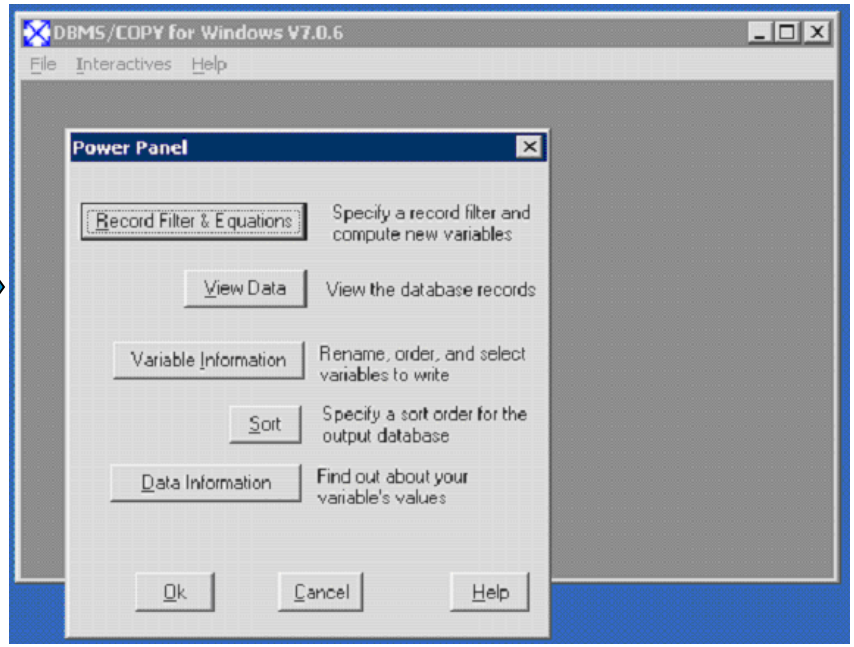
Given file is "|" delimited, select the Field Separator as "|"

Press "Add" button to until all variables in text file has been included

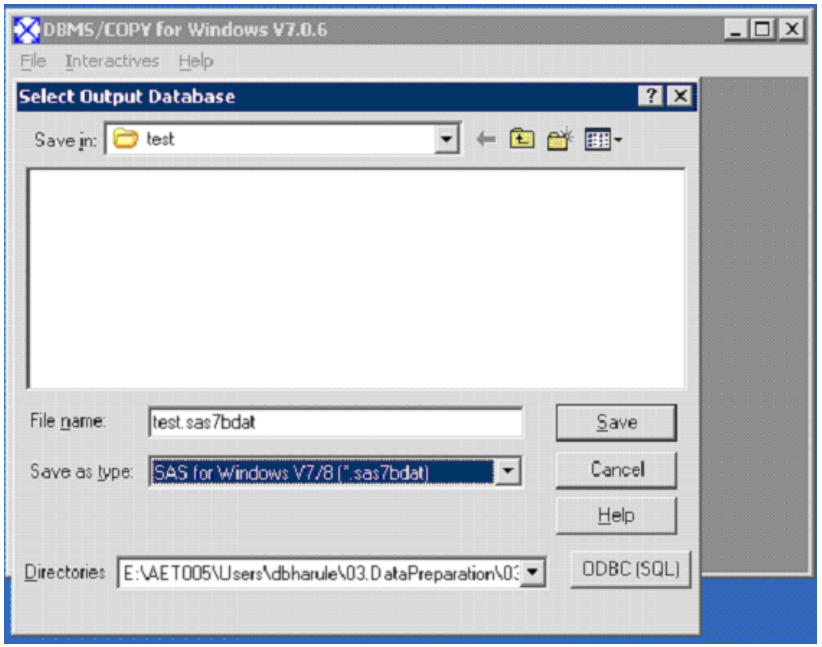


5.9 DBMS Copy…cont'd

Now you can view the data (to be exported) by clicking on "View Data"

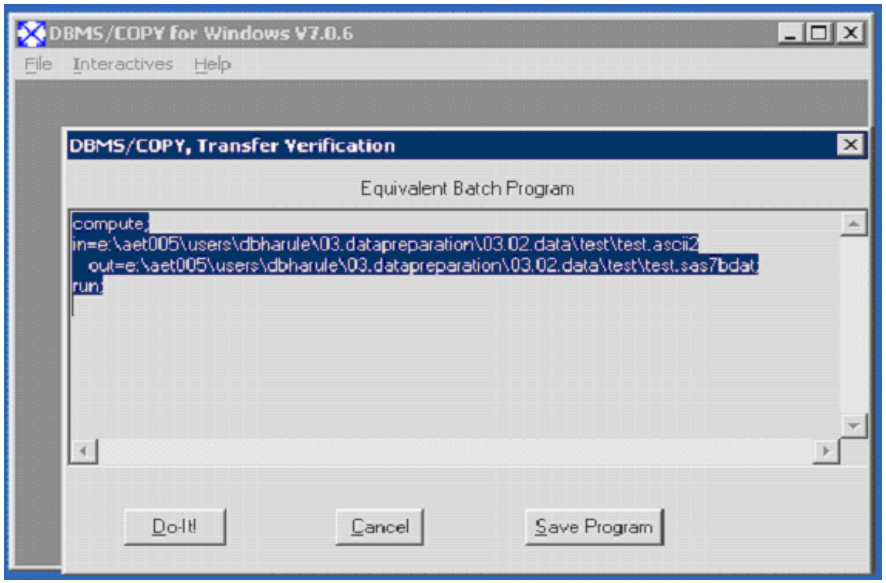


Select the output file format

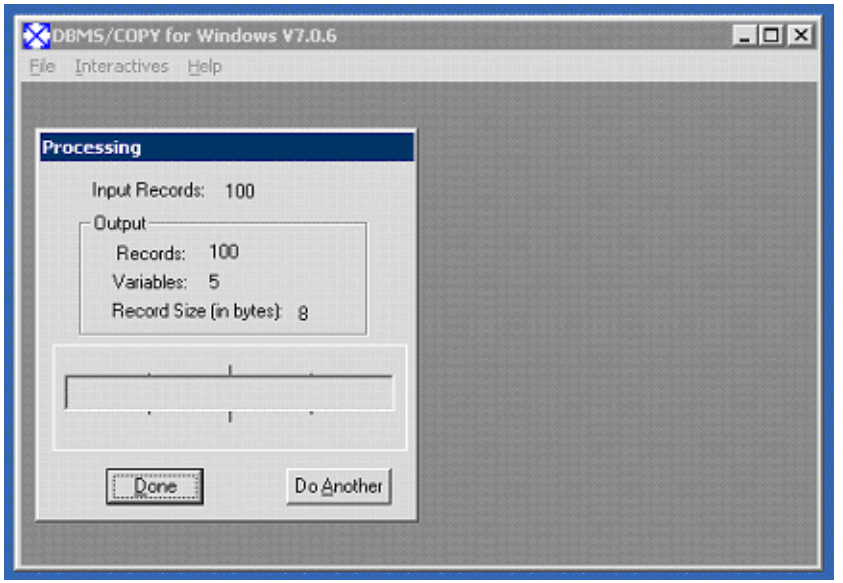


5.9 DBMS Copy…cont'd

Click on "Do-it"



Click on Done



Exercise 5

1. Medical data is stored in the raw data file BLOODTYP. The first record contains the patient's identification number and the patient's first and last names. The second record contains a code specifying the medical plan, the patient's blood type, a code indicating whether the patient has any allergies, and the number of dependants covered by the family's health plan

   a) Create a SAS data set named **work.medical** that contains the patient's identification number, first name, last name, and blood type

2. Medical data is stored in the raw data file **ALLERGY**. If the patient has an allergy (Allergy Code = Y), then the rest of the record is as follows:

Allergy Type	2-character code indicating type of allergy
Number of Dependants	Numeric field

   a) If the patient does not have an allergy (Allergy Code = N), then the rest of the record is as follows:

Number of Dependants	Numeric field

   Use conditional input to create the SAS data set named work.allergies

   b) Modify the DATA step you wrote in the previous problem to create a SAS data set named work.allergies2 that contains only patients with allergies

Exercise 5…cont'd

3. The raw data file TRANSACT contains daily bank transactions for a given account

   a) Create a SAS data set named work.transactions that contains all transactions

   b) Modify the DATA step you wrote in previous exercise to create two SAS data sets. Name the first data set work.credits; it should contain all the deposit information. Name the second data set work.debits; it should contain all the withdrawal information.

4. Read the EMPTWO raw data file to create the SAS data set **work.empinfo.**

   3.Use the absolute line pointer to control the default order of the fields, so that the variables in the SAS data set are in the following order:

   - Identification Number
   - Last Name
   - First Name
   - Division
   - Hire Date
   - Salary

6. Data Management

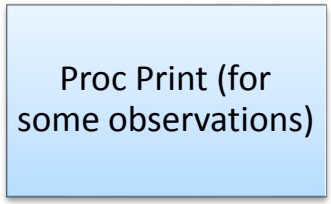
6. Things to be covered

6. Data Management
    - 6.1 How to write a code (Libraries, Body & Commenting)
    - 6.2 Creating & Modifying Datasets
        - 6.2.1 Creating Multiple datasets in one step
    - 6.3 Creating & Modifying Variables
    - 6.4 Selecting Variables
        - 6.4.1 Rename=
        - 6.4.2 Keep= and Drop=
    - 6.5 Automatic Variables in SAS
    - 6.6 Handling Multiple Datasets: Proc Datasets
    - 6.7 SAS Options
    - 6.8 Selecting Observations
    - 6.9 Subsetting Observations
        - 6.9.1 Where or Subsetting if
    - 6.10 Special and Logical Operators
        - 6.10.1 Special Operators
        - 6.10.2 Logical Operators
    - 6.11 Few efficient practices while doing variable selection
    - 6.12 Dataset Options
        - 6.12.1 _NULL_
        - 6.12.2 _LAST_
        - 6.12.3 END=
    - 6.13 Formatting Data Values

6. Things you should already know

- What you want from the code?

- Data understanding should be there. If not, use the following to have an idea of the data:



6.1 How To Write A Code

Define a library

- Library: Where one can save the datasets
- Defaults SAS Library is called “Work” (Vanishes at the end of each session)
- To create permanent datasets create your own library.

Syntax

- Libname Library_name “Path”;

Example

- Libname root_09 “\clmrsch\Auto\Root\All_2009”;

How To Use

- Root_09.Dataset_Name

6.1.1 How To Write A Code (Body)

- Body
  - Data Steps
    - Used to define and manipulate the data

Syntax

```
Data To_be_created_dataset_name;
Set To_be_used_dataset_name;
Statements;
Run;
```

Example

```
Data Final;
Set  Raw;
Statements;
Run;
```

6.1.1 How To Write A Code (Body) Continued..

- Body
- Proc Statements
  - Used to analyze and present the data

Syntax

```
Proc Proc_name data=dataset_name;
Options;
Run;
```

Example

```
Proc means Data=Final;
Var  score;
Class section;
Run;
```

6.1.1 How To Write A Code (Body) Continued..

- Body

- Commenting (Used to make code more understandable)
  - To Comment Single Line
    - Begin a line with asterisk (*) and end with semi colon (;)
  - To comment multiple lines
    - Begin with (/*) and end with (*/)

Syntax

- 1. * SAS Statement or Comment;
- 2. /* SAS Statement or Comment*/

Example

```
*Merging 2 datasets;

/*Merging 2 datasets and also putting some

Filters*/
```

6.2 Creating & Modifying datasets

- Creating a dataset from existing dataset

```
DATA <New Dataset>;
SET <Name of Existing Dataset>; Statements;
RUN;
```

- Creating multiple datasets in single step

```
DATA <New Dataset 1> <New Dataset 2> …… <New Dataset n>;
        SET <Name of Existing Dataset>;
        IF <condition 1> THEN OUTPUT <New Dataset 1>;
                ELSE IF <condition 2> THEN OUTPUT <New dataset 2>;
                ELSE IF ……
                ELSE IF ……
                ELSE OUTPUT <New Dataset n>;
              RUN;
```

6.2.1 Creating & Modifying a dataset

EXAMPLE: Create 2 separate datasets on the basis of the value “TRUE” and “FALSE”

Data True False;

Set Both;

If value = “True” then output True; else Output False;

Run;

EXERCISE

Ex 1: Given the dataset, DATA residing in folder 6, write SAS code to create two different datasets called FIRST and SECOND, containing the first five and the last five observations respectively. The new datasets should also contain a variable that should equal the positive difference of p5 and the average of p4 and p6.

6.3 Creating & Modifying Variables

- Creating new variables and modifying existing variable

SYNTAX

```
DATA <Name of New Dataset >;
        SET <Name of Existing Dataset>;
        <new var-name> = <value or code>;
                <existing var-name> = <code>;
              RUN;
```

EXAMPLE

```
DATA New;
SET Old;
Format AvgSalary dollar8.; AvgSalary=(SalQtr1+SalQtr2+SalQtr3+SalQtr4)/4;
Fname=compress(Fname);
RUN;
```

EXERCISE

Ex 2: Given the dataset, DATA residing at the located in the folder 6, write SAS code to create a variable which equals the 11^{th} power of p1.

Ex 3: Write SAS code to change the variable p10 to average of p9 and p10. Use the dataset, DATA.

6.4 Selecting Variables

- Operation on variables:
  - Rename

SAS-data-set (RENAME=(old-name-1=new-name-1))

```
Data new;
Set old(rename =(sal_m=MonthlySalary);
run;
```

  - Keep

*SAS-data-set (*KEEP=Variables*);*

```
Data new;
Set old(Keep= name address salary);
run;
```

  - Drop

SAS-data-set (DROP=variables);

```
Data new;
Set old(Drop= sal_m raw_var merge_ind);
run;
```

6.4.1 RENAME

Append 2 datasets having same kind of variables but having different name.

<Dataset1>

Name	Gender	JCode
Peter	F	FA1
Jupiter	M	FA1
Repeater	F	FA1

<Dataset2>

Name	Gender	Job
Meter	F	FA3
Heater	M	FA2

<New Dataset>

Name	Gender	JCode
Peter	F	FA1
Jupiter	M	FA1
Repeater	F	FA1
Meter	F	FA3
Heater	M	FA2

```
data <New Dataset>;
    set <Dataset1> <Dataset2> (rename=(Job=JCode));
run;
```

6.4.2 KEEP & DROP

Dataset having 2 type of expenses, finally there should be only one variable called ” Total Expense” which is a sum of both the expenses.

Solution

```
data Total;
    set Expense;
    Total=Expnse1+Expense2;
    drop Expense1 Expense2;
Run;
```

```
Data Total(drop= Expense1 Expense2);
    set Expense;
    Total=Expnse1+Expense2;
Run;
```

OR

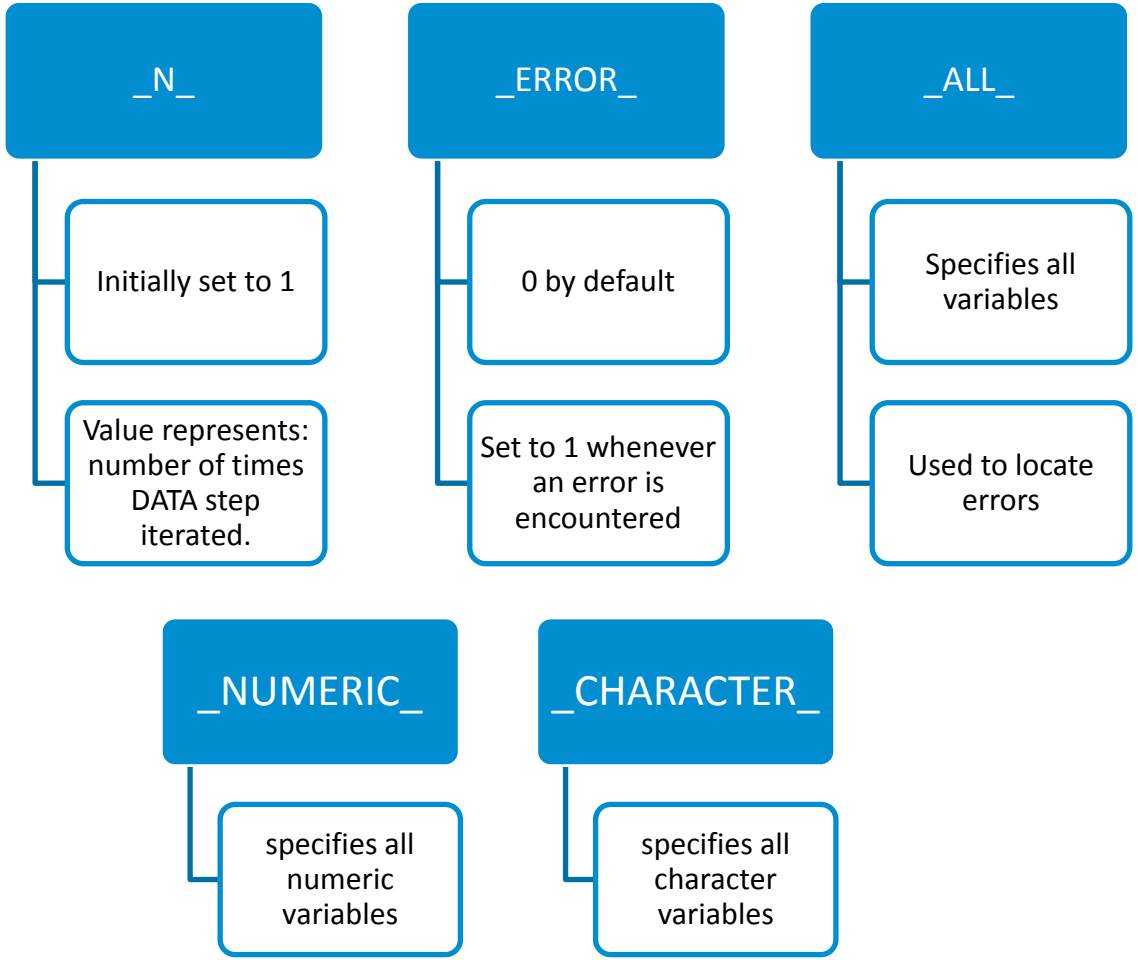
```
data Total;
    set Expense;
    Total=Expnse1+Expense2;
    Keep= Total;
Run;
```

```
Data Total(Keep= Total);
    set Expense;
    Total=Expnse1+Expense2;
Run;
```

All above are equivalent and give same output.

6.5 Automatic Variables in SAS

- The following automatic variables are created by SAS in every data-step



6.5.1 Automatic Variables in SAS (Example)

- The following automatic variables are created by SAS in every data-step

Problem Statement

Create two datasets from <Dataset> separating even numbered observations from odd numbered

```
Solution
data even odd;
   set <Dataset>;
   if (_N_ - int(_N_/2)*2=1) then output odd;
Else output Even;
Run;
```

6.6 Handling Multiple Datasets: Proc Datasets

The DATASETS procedure is a utility procedure that manages your SAS files. It allows you to

- copy SAS files from one SAS library to another,
- rename SAS files,
- delete SAS files,
- append SAS datasets,
- modify attributes of SAS datasets and variables in the datasets and
- create and delete indices.

PROC DATASETS <OPTIONS> STATEMENTS; RUN;

Options, also called Memtype options (restricts processing to one or more member types):

ALL:	All member types
DATA:	SAS data sets
PROGRAM:	Stored compiled SAS programs
VIEW :	SAS data views

6.6.1 Proc Datasets (Example)

Copies all data sets from the trg1 library to the trg2 library and lists the contents of the trg1 library. It deletes the training1 data set from the trg1 library changes the name of the training example data set to training exercise.

```
libname trg1 'address';
libname trg2 'address';

PROC DATASETS memtype=data;
                copy in=trg1 out=trg2;
run;
PROC DATASETS library=trg1 details;
delete training1;
change trainingexample=trainingexcercise;
run;
```

6.7 SAS Options

System options are also part of SAS programs. It controls how SAS handles outputs, datasets, interacts with OS and does other system level functions.

General form of the OPTIONS statement
OPTIONS *option*…….;
Example OPTIONS compress = yes;

PROC OPTIONS; run;

The log that results from running above code shows both the portable and host systems options, their settings, and short descriptions.

Selected System Options

DATE (default)	Specifies to print the date and time the SAS session began at the top of each page of the SAS output
NODATE	Specifies not to print the date and time the SAS session began
LINESIZE = width LS = width	Specifies the line size for the SAS log and SAS output
PAGESIZE = n PS = n	Specifies the number of lines (n) that can be printed per page of SAS output
NUMBER (default)	Specifies that page numbers be printed on the first line of each page of output
NONUMBER	Specifies that page numbers not be printed
PAGENO = n	Specifies a beginning page number (n) for the next page of SAS output
Compress=yes	Specifies that dataset be compressed. Results in storage space saving.

6.8 Selecting Observations

- Obs= specifies an ending point for processing an input data set

SAS-*dataset* (OBS=n)

```
data army;
    set prog2.military(obs=25);
    if Type eq 'Army' then output;
run;
```

To guarantee that SAS processes all observations from a data set

SAS-dataset (OBS=MAX)

- FIRSTOBS= dataset option specifies a starting point for processing an input data set

*SAS-data-set (*FIRSTOBS=n*);*

```
data army;
    set prog2.military(firstobs=11 obs=25);
    if Type eg 'Army' then output;
run;
```

FIRSTOBS=and OBS=are often used together to define a range of observations to be processed

6.9 Subsetting Observations

- SAS provides some conditional constructs like
  - If .. Then .. Else
  - IF Sub-setter (An IF statement without a THEN clause) and
  - DO and END statements (used to execute a group of statements based on a condition
  - Where Statement
  - Delete Statement)

If .. Then .. Else

```
DATA <Name of New Dataset>;
        SET <Name of Existing Dataset>;
        IF <expression> THEN <statement>;
              ELSE <statement>;
            RUN;
```

IF Sub-setter

```
DATA <Name of New Dataset>;
        SET <Name of Existing Dataset>;
        IF <expression>;
            RUN;
```

6.9 Subsetting Observations (Continued..)

DO and END statements

```
DATA <Name of New Dataset>;
        SET <Name of Existing Dataset>;
        IF <expression> THEN DO;
                <executable statements>;
        END;
        ELSE DO;
            <executable statements>;
        END;
      RUN;
```

Where Statement

```
DATA <Name of new dataset>;
        SET <Name of Existing Dataset>;
        WHERE <expression>;
      RUN;
```

Examples:

```
      where Salary > 25000;
      where Salary = .;
      where JobCode in ('PILOT' 'FLTAT');
```

Delete Statement

```
If <expression> THEN DELETE;
```

6.9.1 Where or Subsetting If?

WHERE and subsetting IF statements can only be used in certain PROC and DATA steps

Step and Usage	WHERE	IF
PROC step	Yes	No
DATA step (source of variable)		
INPUT statement	No	Yes
Assignment statement	No	Yes
SET statement (single data set)	Yes	Yes
SET/MERGE (multiple data sets)		
Variable in ALL datasets	Yes	Yes
Variable not in ALL datasets	No	Yes

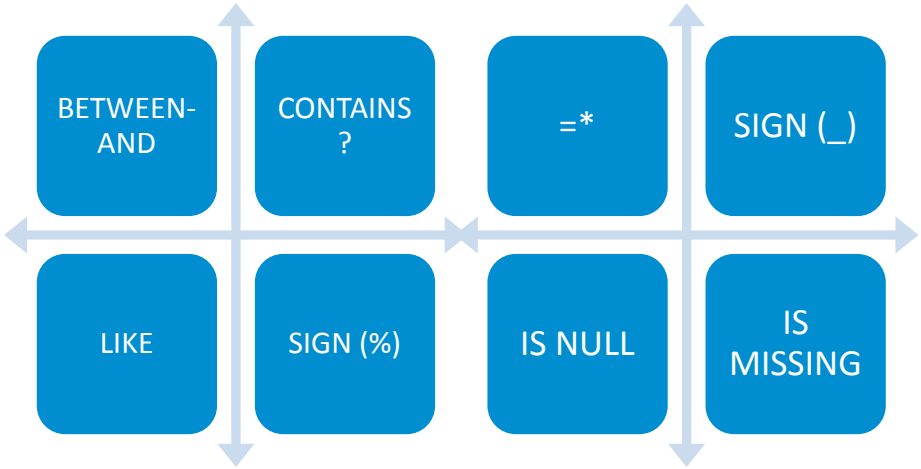
6.9.2 Subsetting Observations (Exercise)

Ex 4: Write an IF statement to create a dataset having the first five observations of DATA.
(Hint: Use an IF Sub-setter)

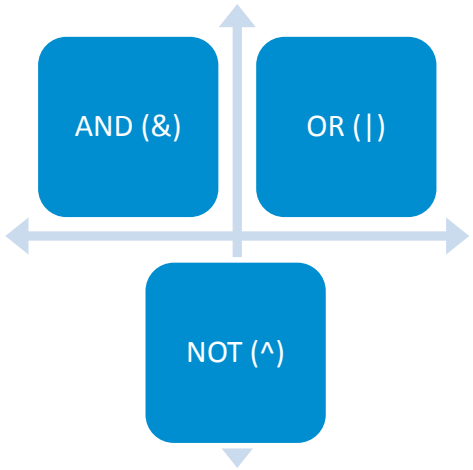
6.10 Special and Logical Operators

SAS allows usage of a host of logical and special operators to facilitate code writing.

Special Operator



Logical Operator



6.10.1 Special Operators

BETEWEEN-AND	• Selects observations in which the value of the variable falls with a range of values, inclusively where Salary between 50000 and 70000;
CONTAINS ?	• Selects observations that include the specified substring where LastName ? 'LAM'; • (LAMBERT, BELLAMY etc are selected)
LIKE	• Selects observations by comparing character values to specified patterns where JobCode not in ('PILOT','FLTAT');
SIGN (%)	• GGGGG. A percent sign (%) replaces any number of characters
SIGN (_)	• An underscore (_) replaces one character. where Code like 'E_U%'; • (Selects observations where the value of Code begins with an E, followed by a single character, followed by a U, followed by any number of characters.)
=*	• Selects observations that contain spelling variations of the word or words specified. where Name =* 'SMITH'; (Selects names like SMYTHE and SMITT).
IS NULL or IS MISSING	• Selects observations in which the value of the variable is missing. where Flight is missing;

6.10.2 Logical Operators

AND (&)	OR (|)	NOT(^)
If both expressions are true, then the compound expression is true	If either expression is rue, then the compound expression is true	Can be combines with other operators to reverse the logic of a comparison
where JobCode = 'FLTAT' and Salary >50000;	where JobCode = 'PILOT' or JobCode = 'FLTAT';	where JobCode not in ('PILOT','FLTAT');

6.11 Few efficient practices while doing variable selection

Situation 1: From 2 GB dataset, you are very sure you need only 2 out of 10 variables for certain analysis

- ✓ Read only those 2 variables
- ✗ Read 10 variables and drop 8 later
- ✗ Read 10 variables and retain them

Situation 2: You created some derived variables using raw variables

- ✓ Create indicator for different raw variables whenever possible
- ✓ Keep all the variables if dataset is big
- ✓ Drop variables if dataset is small

Situation 3: From 20 GB dataset, you think you need only 2 out 10 variables for certain analysis

- ✓ Read 10 variables and retain them
- ✗ Read 10 variables and drop 8 later
- ✗ Read only those 2 variables

Important: Reading/writing any single variable in a big data step is very costly!

6.12 Dataset Options
6.12.1 _NULL_

- Using the keyword _NULL_ as the dataset name causes SAS to execute the DATA step without writing observations to a dataset

DATA _NULL_;

```
data _null_;
      Var1 = “Funny Day”;
      Var2= “smile”;
      Put VAR1 VAR2;
run;
```

- This process can be a more efficient use of computer resources if you are using the DATA step for some function, such as report writing, for which the output of the DATA step does not need to be stored as a SAS dataset

6.12.2 _LAST_

- When you execute a DATA or PROC step without specifying an input dataset, by default, SAS uses the _LAST_ dataset

```
Options _LAST_ = <Dataset>;
```

- Some functions use the _LAST_default as well
- The _LAST_= system option enables you to designate a data set as the _LAST_dataset
- Issuing the _LAST_=system option enables you to avoid specifying the SAS dataset name in each procedure statement

6.12.3 END=

- The END=option in the SET statement creates and names a temporary variable that acts as an end-of-file indicator

  This temporary variable is initialized to 0. When the SET statement reads the last observation of the dataset listed, the value of the variable is set to 1

SAS-data-set END=variable;

Example

Create two datasets from <Dataset> separating even numbered observations from odd numbered observations

```
data _null_;
   set <Dataset> end=IsLast;
   if_N_=1 then    put “This is the first observation”;
 .
 .
   if IsLast=1 then put “This is the last observation” ;
run;
```

6.13 Formatting Data Values

- Variables can be formatted into various character, numeric or date formats using the FORMAT statement

General form of the FORMAT statement

FORMAT *variable(s) format;*

Example

FORMAT salary dollar11.2;

A format is an instruction that SAS uses to write data values. SAS formats have the following form.

<$> format<w>.<d>

<$> indicates a character format
<w> indicates the total width (including decimal places and special characters
<d> indicates number of decimal places

Selected SAS Formats

w.d 8.2	Standard numeric format Width = 8, 2 decimal places: 12234.21
$w. $5.	Standard character format Width = 5: KATHY
COMMAw.d COMMA9.2	Commas in a number Width = 9, 2 decimal places: 12,234.21
DOLLARw.d DOLLAR10.2	Dollar signs and commas in a number Width = 10, 2 decimal places: $12,234.21

7: Functions

7.1 SAS Functions

The SAS System provides a large library of functions for manipulating data during DATA step execution.

A SAS function is often categorized by the type of data manipulation performed:

- **Truncation**
- **Character**
- **Date and time**
- **Random number**
- **Arithmetic**
- **Trigonometric**
- **Sample Statistics**
- **Financial**

Syntax for SAS Functions

A SAS function is a routine that performs a computation or system manipulation and returns a value. Functions use arguments supplied by the user or by the operating environment.

General form of a SAS function:

Syntax

function-name (argument-1, argument-2,…argument-n)

Some functions accept

- Multiple arguments in any order
- A specific number of arguments in a fixed order
- No arguments

Functions that require arguments accept

- Constants
- Variables
- Expressions

SAS Variable Lists

A SAS variable list is an abbreviated method of referring to a list of variable names.

Numbered range lists	**x1-xn**	Specifies all variables from **x1** to **xn** inclusive. You can begin with any number and end with any number as long as you do not violate the rules for user-supplied variable names and the numbers are consecutive
Name range lists	**x—a**	Specifies all variables ordered as they are in the program data vector, from **x** to **a** inclusive
	x-numeric-a	Specifies all numeric variables from **x** to **a** inclusive
	x-character-a	Specifies all character variables from **x** to **a**
Name prefix lists	**Sum(of REV:)**	Tells SAS to calculate the sum of all the variables that begin with REV, such as **REVJAN**, **REVFEB**, and **REVMAR**
Special SAS name lists	**_ALL_**	Specifies all variables that are already defined in the current DATA step
	NUMERIC	Specifies all numeric variables that are currently defined in the current DATA step
	CHARACTER	Specifies all character variables that are currently defined in the current DATA step

Note: When using a SAS variable list in a SAS function, use the keyword OF in front of the first variable name in the list. If the keyword OF is omitted, subtraction is performed.

```
data <dataset>;
    set <dataset1>;
    Total=sum(of Qtr1-Qtr4);   if Total ge 50; run;
```

7.2 Manipulating Character Values

7.2.1 Substring

SAS functions and operators can be used to extract, edit, and search character values.
The SUBSTR function is used to extract substring for string or replace characters in a string

Syntax

NewVar = SUBSTR (string, start <,length>); This form of the SUBSTR function extracts characters.

SUBSTR (string, start <,length>) = string; This form of the SUBSTR function replaces characters.

STRING	can be a character constant, variable, or expression.
START	specifies the starting position
Length	specifies the number of characters to extract. If omitted, the substring consists of the remainder of the string

Note: If the length of the created variable is not previously defined with a LENGTH statement, it is the same as the length of the first argument to SUBSTR.

Example

```
data _null_;
    Var = “Funny Day”;
    NVar1 = SUBSTR(Var,1,5);
    SUBSTR(Var,1,1)=“S”;
 Put NVAR1 VAR;
run;
```

Results

Nvar1=Funny
Bar=Sunny Day

7.2 Manipulating Character Values

7.2.2 Right/Left

The RIGHT function returns its argument right-aligned. Trailing blanks are moved to the start of the value.
The LEFT function returns its argument left-aligned Leading blanks are moved to the end of the value.

Syntax

NewVar = RIGHT (argument);

NewVar = LEFT (argument);

- Argument can be a character constant, variable, or expression.

Note: If the length of the created variable is not previously defined with a LENGTH statement, it is the same as the length of the argument.

Example

```
A= Policy_Eff_Date
B=Right(Policy_Eff_Date)
Put a $110. ;
Put b $10.
```

Results

```
----+----1----+
Policy_Eff_Date
  Pollicy_Eff_Date
```

7.2 Manipulating Character Values

7.2.3 Scan

The SCAN function returns the nth word of a character value. It is used to extract words from a character value when the relative order of words is known, but their starting positions are not.

Syntax

NewVar = SCAN(string, n<,delimiters>);

STRING	can be a character constant, variable, or expression
N	specifies the nth word to extract from string
D	delimiters defines characters that delimit (separate) words

Note: If When the SCAN function is used,

- ***The length of the created variable is 200 bytes if it is not previously defined with a LENGTH statement***
- ***Delimiters before the first word have no effect***
- ***Any character or set of characters can serve as delimiters***
- ***Two or more contiguous delimiters are treated as a single delimiter***
- ***A missing value is returned if there are fewer than n words in string***
- ***If n is negative, SCAN selects the word in the character string starting from the end of string***

Example

```
data _null_;
                Var = "Funny Day";
                NVar1 = scan(Var,2," ");
                NVar2 = scan(Var,1," ");
 Put NVAR1 NVar2 ; run;
```

Results

NVAR1 = Day

NVAR12 =Funny

7.2 Manipulating Character Values

7.2.4 Concatenation Operator

The concatenation operator joins character strings

Syntax

NewVar=string1 !! string2;

Note: If the length of the created variable is not previously defined with a LENGTH statement, it is the sum of the lengths of the concatenated constants, variables, and expressions.

Example

```
data _null_;
    Var = “Funny Day”;
    NVar1 = Substr(Var,5,1);
    NVar2 = Substr(Var,8,2);
    NVar3 = compress(NVar1!!NVar2);
 Put NVAR1 NVar2 NVar3;
run;
```

Results

Nvar1=y
Nvar2=ay
Nvar3=vay

7.2 Manipulating Character Values

7.2.5 Trim

The TRIM function removes trailing blanks from its argument. To remove leading blanks, a combination of TRIM and LEFT functions can be used.

Syntax

NewVar = TRIM(argument1);

TRIM returns one blank ,if the argument is blank, TRIMN function returns a null string (zero blanks) if the argument is blank.

- ***Note: COMPBL function translates each occurrence of two or more consecutive blanks into a single blank. The value that the COMPBL function returns has a default length of 200.***

Example

```
data _null_;
    Var = “Funny Day”;
    NVar1 = Scan(Var,2,“ ”);
    NVar2 = Scan(Var,1,“ ”);
    NVar3 = TRIM(compress(NVar1!!NVar2));
 Put NVAR1 NVar2 NVar3;
Run;
```

Results

Nvar1=Day
Nvar2=Funny
Nvar3=DayFun
ny

7.2 Manipulating Character Values

7.2.6 Cats

Removes leading and trailing blanks, and returns a concatenated character string.

Syntax

CATS(item-1 <, ..., item-n>)

Item specifies a constant, variable, or expression, either character or numeric.

- **Note: In a DATA step, if the CATS function returns a value to a variable that has not previously been assigned a length, then that variable is given a length of 200 bytes.**

Example

CATS(OF X1-X4)

Equivalent Code

TRIM(LEFT(X1))||TRIM(LEFT(X2))||TRIM(LEFT(X3))|| TRIM(LEFT(X4))

7.2 Manipulating Character Values

7.2.7 Index

The INDEX function searches a character argument for the location of a specified character value and returns its location.

Syntax

Position = INDEX(target, value);

The INDEX function returns

- the starting position of the first occurrence of value within target, if value is found
- 0, if value is not found

Note: **The search for value is literal. Capitalization and blanks (leading, embedded, and trailing) are considered.**

Example

```
data _null_;
    Text = “This target contains a BULL’S-EYE.”;
    Pos = index(Text,“BULL’S-EYE ”);
    Put Pos;
run;
<can be used in place of scan, by using a
    combo of index + substr>
```

Results

Pos=24

7.2 Manipulating Character Values

7.2.8 Upcase/ Lowcase

The UPCASE function converts all letters in its argument to uppercase and LOWCASE function converts all letters in its argument to lowercase.

Syntax

NewVal = UPCASE (argument);

NewVal = LOWCASE (argument);

Use a combo of Compress + Upcase/Lowcase

Note: There is no effect on digits and special characters.

Example

```
data _null_;
  Var1="FuNNy";
  if (UPCASE(Var1) = Var1)  then put "UPCASE Active";
  if (LOWCASE(Var1) = Var1) then put "LOWCASE Active";
  if (Var1 = "FUNNY")       then put "Case doesn't matter";
  else                      put "Case Matters";
run;
```

Results

Case
Matters

7.2 Manipulating Character Values

7.2.9 Tranwrd

The TRANWRD function replaces or removes all occurrences of a given word (or a pattern of characters) within a character string.

Syntax

NewVal = TRANWRD (source, target, replacement);

- Source string which you want to translate
- Target string to be searched in source
- Replacement string that replaces target

Note: Tranwrd maintains the length of str that is replaced.
: It does not remove trailing blanks from target or replacement.
: Using this function to replace an existing string with a longer string may cause truncation of the resulting value if a LENGTH statement is not used

Example

```
data _null_;
  dessert = “Toffee Pumpkin”;
  dessert =
TRANWRD(Dessert,”Pumpkin”,”Apple”);
  put Dessert;
Run;
```

Results
Toffee
Apple

7.2 Manipulating Character Values

7.2.10 Translate

Replaces specific characters in a character string.

Syntax

TRANSLATE(source,to-1,from-1<,...to-n,from-n>)

- Source string which you want to translate
- From string to be searched in source
- To string that replaces target

Note: In a DATA step, if the TRANSLATE function returns a value to a variable that has not previously been assigned a length, then that variable is given the length of the first argument.

Example

```
x=translate('XYZW','AB','VW'); put x;
```

Results

XYZB

7.2 Manipulating Character Values

7.2.11 Compress

Removes specific characters from the string.

Syntax

Newvar = COMPRESS (argument<,chars-to-remove>)

Default character is a blank

Example

```
data _null_;
    Var = “Funny Day”;
    NVar1 = COMPRESS(Var);
    NVar2 = COMPRESS(Var,“F“);
 Put NVAR1 NVAR2;
run;
```

Results

NVAR1=Funny Day
Nvar2=unnyDay

7.2 Manipulating Character Values

7.2.12 Length

Returns the length (number of characters) of the argument.
Can also be used to define the length of a var (length var $5).

Syntax

NewVar = LENGTH(argument1);

Note: If the argument is null string (""), LENGTH returns 1

Example

```
data _null_;
    Var = "Funny Day";
    NVar1 = SCAN(Var,2," ");
    NVar2 = SCAN(Var,1," ");
    NVar3 = LENGTH(compress(NVar1!!NVar2));
    NVar4 = LENGTH(Var);
 Put NVar1 NVar2 NVar3 NVar4;
run;
```

Results

NVAR1=Day
Nvar2=Funn
y
Nvar3=8
Nvar4=9

7.2 Manipulating Character Values

7.2.13 In

Returns True or False depending on whether the value is present or not in a list of numbers or character strings

Syntax

Var IN (,arguments);

Example

```
data _null_;
  Var1="FuNNy";
  Var2=2;
  if (UPCASE(Var1) in (“SUNNY”, "FUNNY")  then put “Present 1" ;
  if (Var2 in (1,2,3,4,5,6) then  put “Present 2";
run;
```

Results

Present 1
Present 2

7.3 Manipulating Numeric Values

SAS has functions to truncate numeric values and compute statistics of numeric values.

Truncation Functions

- Round Function
- Ceil Function
- Floor Function
- Int Function

7.3 Manipulating Numeric Values

7.3.1 Round

The ROUND function returns a value rounded to the nearest round-off unit

Syntax

NewVar = ROUND (argument<,round-off-unit>);

Note: If round-off-unit is not provided, argument is rounded to the nearest integer. Round-off-unit is numeric and positive.

Example

```
data _null_;
    NVar1= ROUND(12.12);
    NVar2= ROUND(42.65,.1);
    NVar3= ROUND(6.478,.01);
    NVar4= ROUND(96.47,10);
 Put NVAR1 NVAR2 NVAR3 NVAR4;
run;
```

Results

NVAR1=12
Nvar2=42.7
Nvar3=6.48
Nvar4=100

If you change .1 to .5 ans is different ,the no has to be a multiple of .5

7.3 Manipulating Numeric Values

7.3.2 Ceil, Floor, Int

The CEIL function returns the smallest integer greater than or equal to the argument.
The FLOOR function returns the greatest integer less than or equal to the argument
The INT function returns the integer portion of the argument

Syntax

NewVar = CEIL (argument);

NewVar = Floor(argument);

NewVar = Int(argument);

Example

```
data _null_;
    Var1=6.478;
    NVar1= CEIL (Var1);
    NVar2= FLOOR(Var1);
    NVar3= INT  (Var1);
 Put NVar1 NVar2 NVar3 NVar4;
run;
```

Results

NVAR1=7
Nvar2=6
Nvar3=6

7.3 Manipulating Numeric Values

7.3.3 Statistics Function

Selected functions that compute sample statistics based on a group of values include--

- SUM Function (total of values)
- MEAN Function (average of values)
- MIN Function (lowest value)

Note: These functions

- ***Accept multiple arguments in any order***
- ***Use the same algorithm as SAS statistical procedures***
- ***Ignore missing values***

Example

```
data _null_;
 Var1=6; Var2=2; var3=1;
 NVar1 = SUM (Var1,Var2,Var3);
  NVar2 = MEAN(Var1,Var2,Var3);
 NVar3 = MIN (Var1,Var2,Var3);
 NVar4 = MAX (Var1,Var2,Var3);
 Put NVar1 NVar2 NVar3 NVar4;  *comes in log file*      run;
```

Results

NVAR1=9
Nvar2=3
Nvar3=1
Nvar3=6

7.3 Manipulating Numeric Values

7.3.4 Mathematical Functions

Selected functions that mathematical calculations based on a group of values include--

- LOG Function (Returns the natural (base e) logarithm)
- MOD Function (Returns the remainder from the division of the first argument by the second argument, fuzzed to avoid most unexpected floating-point results.)
- SQRT Function (Returns the remainder from the division of the first argument by the second argument, fuzzed to avoid most unexpected floating-point results.)

Example

```
DATA _NULL_ ;
NVAR1 =36 ;
NVAR2 =LOG(VAR1) ;
NVAR3 =MOD(VAR1,6) ;
NVAR4 =MOD(VAR1,5) ;
NVAR5 =SQRT(VAR1) ;
PUT NVAR1 NVAR2 NVAR3 NVAR4 NVAR5;
RUN ;
```

Results

NVAR1=36
NVAR2=3.58
NVAR3=0
NVAR4=1
NVAR5=6

7.4 Retain

The RETAIN statement prevents SAS from re-initializing the values of new variables at the top of the DATA step, and can be used to create an accumulating variable

Syntax

RETAIN variable-name <initial-val>…;

Previous values of retained variables are available for processing across iterations of the DATA step. The RETAIN statement initializes the retained variable to missing before the first execution of the DATA step if an initial value is not specified
RETAIN is also used to reorder the variables in the dataset.

Example

```
data <Dataset1>;
    set <Dataset2>;
 retain Mth2dte 0;
 Mth2dte = Mth2dte + SaleAmt;
run;
```

Results

SaleDate	SaleAmt	Mth2dte
01APR2001	498.49	498.49
02APR2001	946.50	1444.99
03APR2001	994.97	2439.96
04APR2001	564.59	3004.55
05APR2001	783.01	3787.56

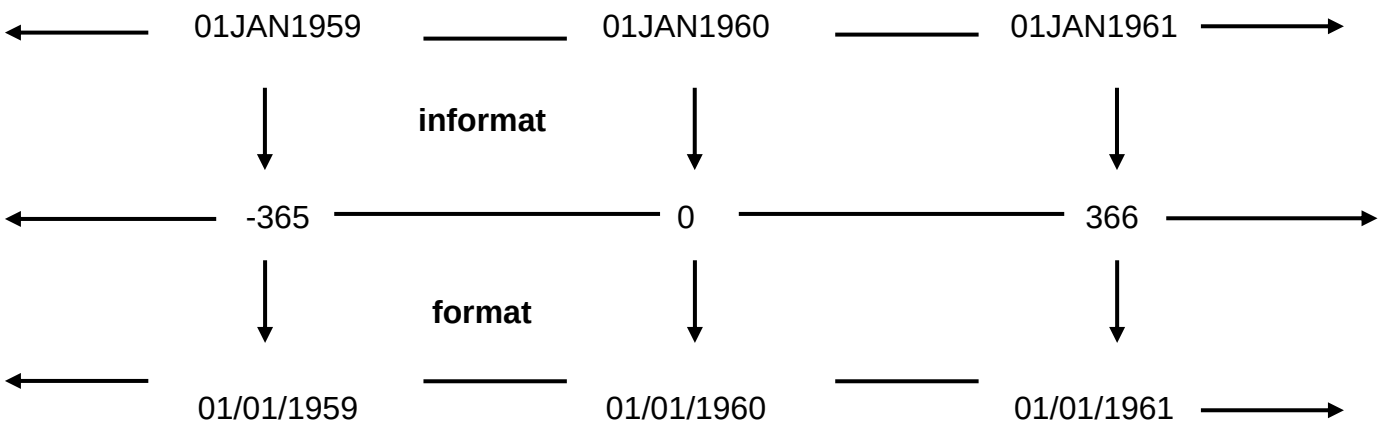
7.5 SAS Date Value

SAS has numerous informats for reading dates and formats for displaying dates.
Dates can be read with either numeric, character, or date informats.
However, only if you use one of the SAS date informats, the date is stored as a numeric variable that can be used in calculations.

A SAS date value is interpreted as the number of days between January 1, 1960, and a specific date.

Results

Raw Data Value	Informat	Converted Value
10/27/2001	MMDDYY10.	15275
031594	MMDDYY6.	12492
10/28/01	MMDDYY8.	15276
29OCT2001	DATE9.	15277
30/10/2001	DDMMYY10.	15278



SAS Date Value Continued...

Example: Imagine you were given a data file that contained a date admission to a hospital and a date of discharge.
You might be interested in computing the length stay.

Example

```
Data patients ;
Informat admit disch mmddyy8.;*Read
Format admit disch date9.;*Display
Input admit disch;
Los= disch-admit;
Datalines;
03/21/90  04/01/90
05/15/90  05/16/90;
Run;
```

Results

OBS	ADMIT	DISCH	LOS(length of stay)
1	21MAR1990	01APR1990	11
2	15MAY1990	16MAY1990	1

- ***The informat statement tells SAS to read the variables admit and disch with a date informat.***
- ***Format tells SAS to display the above variables in a date format .***

7.5.1 Creating SAS Date Values

The following functions can be used to create SAS date values-

- TODAY( ) - Returns the current date value from the systems clock as a SAS date value
- MDY (month,day,year) - Uses numeric month, day, and year values to return the SAS date value

Example

```
data _null_;
    NVar1= TODAY();
    NVar2= MDY(10,27,2001);
 Put NVAR1 NVAR2;
run;
```

Results

NVAR1=16127
*
Nvar2=15275

** 16127 is the output is today's date is 26th February, 2004*

7.5.2 Extracting Information

Some SAS Functions allow extraction of information from SAS dates

- YEAR(SAS-date) - Extracts the year from a SAS date and returns a four-digit value for year
- QTR(SAS-date) - Extracts the quarter from a SAS date and returns a number from 1 to 4
- MONTH(SAS-date) - Extracts the month from a SAS date and returns a number from 1 to 12
- WEEKDAY(SAS-date) - Extracts the day of the week from a SAS date and returns a number from 1 to 7, where 1 represents Sunday, and so on
- DAY (date)- Returns the day of the month from the SAS date

Example

```
data _null_;
    NVar1= YEAR(15275);
    NVar2= QTR(15275);
    NVar3= MONTH(15275);
    NVar4= WEEKDAY(15275);
    NVar5= DAY(15275);   Put NVar1 NVar2 NVar3 NVar4 NVar5;
run;
```

15275 is the SAS date corresponding to 27th October, 2001

Results

NVAR1=200
1
Nvar2=4
Nvar3=10
Nvar4=7
Nvar5=27

7.5.3 Yrdif Function

The YRDIF function returns the number of years between two SAS date values.
he DATDIF function returns the number of days between two SAS date values.

Syntax

NewVal = YRDIF (sdate,edate,basis);

- Sdate Specifies a SAS date value that identifies the starting date
- Edate Specifies a SAS date value that identifies the ending date
- Basis Identifies a character constant or variable that describes how SAS calculates the date difference.

The following character strings are valid

- **'ACT/ACT'** : uses the actual number of days between dates in calculating the number of years. SAS calculates this value as the number of days that fall in 365-day years divided by 365 plus the number of days that fall in 366-day years divided by 366.
- **'30/360'** : specifies a 30-day month and a 360-day year in calculating the number of years
- '**ACT/360**' : uses the actual number of days between dates in calculating the number of years. SAS calculates this value as the number of days divided by 360, regardless of the actual number of days in each year.

Note: To calculate the number of months between two dates, use the YRDIF function and multiply by 12.

15275 is the SAS date corresponding to 27th October, 2001

7.5.4 Intck Function

The INTCK function counts the number of interval boundaries between two dates or between two date time values.

Syntax

NewVal = INTCK (interval, from ,to);

Interval :is a character constant or variable containing an interval name

From : is the starting date (for date intervals) or datetime value (for datetime intervals)

To :is the ending date (for date intervals) or datetime value (for datetime intervals).

Example

```
Data test;
qtr=intck('qtr','10jan95'd,'01jul95'd);
put qtr;
Run;

Data test;
year=intck('year','31dec94'd, '01jan95'd); put year
Run;
```

Results
Qtr=2
Year=1

15275 is the SAS date corresponding to 27th October, 2001

7.5.5 Datepart Function

Extracts the date from a SAS datetime value.

Syntax

Datepart(datetime)

Example

```
conn='01feb94:8:45'dt;
 servdate=datepart(conn);
 put servdate worddate.;
```

Results
February 1, 1994

7.5.6 Intnx Function

Increments a date, time, or datetime value by a given time interval, and returns a date, time, or datetime value.

Syntax

INTNX(custom-interval, start-from, increment <, 'alignment'>)

Interval :is a character constant or variable containing an interval name such as WEEK, SEMIYEAR, QTR etc

Start From : specifies a SAS expression that represents a SAS date, time, or datetime value that identifies a starting point

Increment : specifies a negative, positive, or zero integer that represents the number of date, time, or datetime intervals.

Example

```
yr=intnx('year','05feb94'd,3);
 put yr / yr date7.;
```

Results

13515
01JAN97

7.5.5 Arithmetic Operations

The number of days which have elapsed between two points in time is easily determined by subtracting the value of one SAS date variable from another, or by subtracting a SAS date variable from a SAS date constant .The result can then be divided by an appropriate constant to obtain the desired number of time periods between the two values

Example: A common requirement is to determine how many years have elapsed between two time periods:

```
YEARS = (date2 - date1)/365.25;
```

Note: Similarly, 30.4 is frequently used to convert the number of days to the number of months

Arithmetic Operation vs. INTCK function

Example: Suppose a child is born (and therefore admitted' to the hospital) on December 28, 1994 and discharged on January 2, 1995. The child is therefore five days old at discharge.

Using arithmetic operators gives :

```
AGE = '02JAN1995'D - '28DEC1994'D .The estimated answer is 5/365.25, or .02
    years.
```

But, using the INTCK function

```
AGE=INTCK
    ('YEAR','28DEC1994'D,'02JAN1995'D)
```

Returns 1 as the result because it counts the number of time intervals which have been crossed between the from and to expressions arguments of the function

7.6 Random Functions

SAS allows generation of random numbers.

Syntax

NewVar = RANNOR (Seed);

NewVar = RANUNI (Seed);

Seed is an initial 'starting' value that is user supplied

- RANNOR returns a random variate from a normal distribution; N(0,1)
  - •-∞ <= all values <= ∞ centered around 0
  - •about 2/3 of values will be between -1 and 1
- RANUNI - returns a random variate from a uniform distribution – **model validation datasets & to test some random no. of entries**
  - •0 <= all values <= 1
  - •there is an equal probability of ANY value being assigned

Note: Random numbers can also be generated from Poisson, Binomial, Exponential and other distributions
Can be used to assign random ID
Can be used to change the order of a dataset at random

7.6.1 Random Sampling

RANNOR or RANUNI can be used to take random sample(s) of certain size from a dataset.

Example:

From a dataset with 1 million records, create a sample dataset with 1,000 records

Solution

```
data <dataset>;
  rand = ranuni();
run;

proc sort data=<dataset>;
  by rand;
run;

data <sample_dataset>;
  set <dataset>;
  if (_N_ le 1000) then output <sample_dataset>
run;
```

7.7 Data Conversion

You can convert data types

- Implicitly by allowing the SAS System to do it for you
- Explicitly with these functions:
  - •INPUT character-to-numeric conversion
  - •PUT numeric-to-character conversion

7.7.1 Automatic Character-to-Numeric Conversion

SAS automatically converts a character value to a numeric value when the character value is used in a numeric context, as in the examples given below

Don’t use the following to make numeric/ always use Input.

- Assignment to a numeric variable
- An arithmetic operation
- Logical comparison with a numeric value
- A function that takes numeric arguments

Note: the WHERE statement and WHERE= dataset option do not perform any automatic conversion in comparisons.

7.7.2 Input

The INPUT function is used primarily for converting character values to numeric values

Syntax

NumVar = INPUT (Source, informat);

- Source contains the SAS character expression to which you want to apply a specific informat
- Informat is the SAS informat that you want to apply to the source

Note:If the INPUT function is used to create a variable not previously defined, the type and length of the variable is defined by the informat.

Example

```
data <dataset>;
    set <dataset1>;
    CVar1='32000';
    CVar3='03may2008';
 Cvar4= '120181';
    NVar1=input(CVar1,5.);
    NVar3=input(CVar3,date9.);
    NVar4=input(CVar4,mmddyy6.);
run;
Proc contents data=<dataset>;
Run;
```

Results

..Alphabetic List of Variables and Attributes

#	Variable	Type	Len	Pos
1	CVar1	Char	5	32
2	CVar2	Char	6	37
3	CVar3	Char	9	43
4	CVar4	Char	6	52
5	NVar1	Num	8	0
6	NVar2	Num	8	8
7	NVar3	Num	8	16
8	NVar4	Num	8	24

7.7.3 Automatic Numeric -to- Character Conversion

SAS automatically converts a numeric value to a character value when the numeric value is used in a character context, such as

- Assignment to a numeric variable
- A concatenation operation
- A function that accepts character arguments

Note: the WHERE statement and WHERE= dataset option do not perform any automatic conversion in comparisons.

7.7.4 Put

The PUT function writes values with a specific format

Syntax

CharVar = PUT(source, format);

- Source SAS variable or constant whose value you want to reformat.
- Format contains the SAS format that you want applied to the variable or constant that is specified in the source.

Example

```
data <dataset>;
    NVar1=614;
    NVar2=55000;
    NVar3=366;
    CVar1=put (NVar1,3.);
    CVar2=put(NVar2,dollar7.);
    CVar3=put(NVar3,date9.);
run;
proc contents data=<dataset>;
run;
```

Results

…Variables Ordered by Position…

#	Variable	Type	Len
1	NVar1	Num	8
2	NVar2	Num	8
3	NVar3	Num	8
4	CVar1	Char	3
5	CVar2	Char	7
6	CVar3	Char	9

Exercise

1. From the dataset Contract, create an indicator variable - taking values of 0 and 1 for expired ever and anything else respectively. Do this using the scan function.

2. Using the dataset Contract, wherever the Contract bucket will expire after 12 months (>12MOs) replace the string/characters with “1 Year”. What would be the most efficient method to this.

3. From the dataset Contract, extract the second word from the variable and replace it with 0,1 and 2 for ‘Will', 'Expired’ and ‘Never’ respectively. The resulting variable should be a character variable. Remove the leading and trailing blanks from this variable. What are the different methods which can be used to do this.

4. Convert the variable created in question3 into a numeric variable.

5. From the dataset Collections, round off the value of Month_since_Last_Col2 up to 2 decimal places. Also, calculate the bad debts which have been reported 6 months from today till 12 months from now. Lastly, calculate the percentage of bad debts in the last 6 month to the total count of bad debts till now. If the variable takes non integer values then take the nearest integer value less than it.

6. Using the dataset Contract, calculate the date when the contract will be over (using the variable mnths_left_contract), assuming that all the information provided in the dataset is as of today. Name this variable as Contract_Over_Date. If the variable takes non integer values then take the nearest integer value greater than it. Hint: The calculation needs to be done only where the variable takes no negative values.

7. From the dataset created above, create a new dataset which consists of only 4 variables - lj_sub_id, Contract_Over_Date2, Contract_Over_Date and Contract_Over_Year in the same order. Create Contract_Over_Date2 from Contract_Over_Date as taking the value of dd (day of month) as 15 and Contract_Over_Year as the year in which the contract will expire.

8. SAS by group processing

8.1 Proc Sort

The sort procedure is used to sort (sequence) observations in a SAS dataset

Syntax

PROC SORT DATA = Input-data-set OUT = output-data-set>;
by <Descending> variable-1 <.....<Descending> variable-n>;
run;

Unsorted

VIEWTABLE: Sast.Sampleaccount

	AccountNumber	FirstName	EnrollmentYear
1	54	ABHIJIT	1996
2	205	ADAM	2001
3	114	ADELINE	2000
4	118	ADOLF	2000
5	144	ADRIAN	2000
6	215	ALAN	2001
7	239	ALAN	2000
8	43	ALEXANDR	1996
9	122	ANDREA	2000
10	99	ANDREW	2000
11	201	ANDREW	2000
12	206	ANGELINE	2001
13	174	ANNA	2000
14	181	ANTHONY	2000
15	209	ANTON	2000
16	101	APRIL	2000
17	84	BARRY	1997
18	184	BARRY	2000
19	139	BERNARD	2000
20	123	BEVERLY	2000
21	158	BEVERLY	2000
22	225	BILL	2001
23	147	BRENT	2000

Sorted data

VIEWTABLE: Sast.Sortedaccount

	AccountNumber	FirstName	EnrollmentYear
1	2	JERONIMO	1999
2	4	ROBERT	2003
3	5	GEORGE	1995
4	6	MICHAEL	1996
5	7	GEORGE	1998
6	8	DAVID	1997
7	9	F	1996
8	10	THOMAS	2000
9	11	SHEILA	1996
10	12	PATRICIA	1996
11	13	JOSEPH	1996
12	14	JAMES	2003
13	15	FRANK	1998
14	16	WALTER	1996
15	17	GERALD	2003
16	18	THOMAS	1996
17	19	LEONARD	2003
18	20	FAITH	1999
19	21	JOSEPH	1996
20	22	GREGORY	1999
21	23	JOHN	2001
22	24	JOSEPH	1998
23	25	JOHANNA	1999

8.1.1 Proc Sort with options

- **PROC sort with out options and without Out option**
  The DATA= and OUT= options specify the input and output data sets. If you don't specify the DATA= option, then SAS will use the most recently created data set. If you don't specify the OUT= option, then SAS will replace the original data set with the newly sorted version. This sample statement tells SAS to sort the data set named MESSY, and then put the sorted data into a data set named NEAT:

Syntax

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

- **Comparing Class and Proc Sort**
  CLASS statement should be used in procedures except for proc print and proc sort wherever possible instead of sorting and using a by statement. If your data is already sorted, however, a by statement is more efficient, since a class statement requires more memory.

Syntax

```
CLASS var1 var2 ... / order options
```

The CLASS statement enables you to specify variables from the input data set to be used as the classification variables You can specify one or more CLASS statements. However, a given variable may appear only once in all CLASS statements.
The class variable can be either numeric or character. The default sort order when none is specified is ASCENDING

8.1.2 De-Duping

De-duping is needed to remove multiple occurrence of a key(s) in a dataset

Syntax

PROC SORT DATA=Input-data-set <out=output-data-set> Options;
by <Descending> variable-1<……..<Descending>variable-n>;
run;

Options :-
***NodupKey**: Delete observations with duplicate BY values*
***Nodup**: Delete duplicate observations*
***Dupout**: Specify the output dataset to which duplicate observations are written*

Sorted Dataset with Duplicate Statement Dates

VIEWTABLE: Sast.Sortedstmthist

	AccountNumber	VisaPoints	statement_date
1	1	3771	17JUL2001
2	1	3771	17JUL2001
3	1	3771	17JUL2001
4	1	3819	27MAR2003
5	1	3819	27MAR2003
6	1	3822	24APR2003
7	1	3822	24APR2003
8	1	3825	29MAY2003
9	1	3825	29MAY2003
10	1	3828	26JUN2003
11	1	3828	26JUN2003
12	1	3831	24JUL2003
13	1	3831	24JUL2003
14	1	3834	28AUG2003
15	1	3834	28AUG2003

Sorted Dataset with Duplicate Records Removed

VIEWTABLE: Sast.Sortedstmthistnodup

	AccountNumber	VisaPoints	statement_date
1	1	3771	17JUL2001
2	1	3819	27MAR2003
3	1	3822	24APR2003
4	1	3825	29MAY2003
5	1	3828	26JUN2003
6	1	3831	24JUL2003
7	1	3834	28AUG2003
8	1	3841	02OCT2001
9	1	3844	27FEB2003
10	1	3939	30JAN2003
11	1	3966	30MAY2002
12	1	4018	25JUL2002
13	1	4051	25APR2002
14	1	4062	27JUN2002
15	1	4100	26SEP2002

8.2 By-Group Processing

A BY statement in a DATA step creates temporary variables for each variable listed in the BY statement.
General form of the names of BY variables in a DATA step:

Syntax

First.*BY-variable*
Last.*BY-variable*

- **First.By-variable**: has a value of 1 for the first observation in a BY group; otherwise, it equals 0
- **Last.By-variable**: has a value of 1 for the last observation in a BY group; otherwise, it equals 0

Calculate the sum of employee salary at jobcode level. Jobcode takes up 2 values.

Example

```
data work.divsal (keep=jobcode DivSal);
    set trg.empdata;
    by jobcode;
    if first.jobcode then DivSal=0;
    <additional SAS statements>
run;
```

Note: Data must first be sorted by the 'BY' variables

8.2.1 Multiple BY Variables

Multiple BY statements in a DATA step create temporary variables for each unique combination of variables listed in the BY statement

Calculate the sum of employee salary at subdivision level for each division. Division and Subdivision are a categorical variables taking 10 values each.

Example

```
data <Dataset2> (keep= Division Subdivision
    SubDivSal NumEmps);
    set <Dataset1>;
    by Division Subdivison;
  retain (NumEmps, SubDivSal);
    if First.subdivison then do;
                SubDivSal=0;
                NumEmps=0;
    end;
    SubDivSal+Salary;
    NumEmps+1;
    if Last.Subdivision;
run;
```

8.3 DO Loop Processing

Statements within a DO loop execute for a specific number of iterations or until a specific condition stops the loop.

Syntax

DATA <Dataset>;

SAS Statements;

DO Statement

iterated SAS statements

END statement

SAS statements

RUN;

DO loops can be used to

- perform repetitive calculations
- generate data
- eliminate redundant code
- execute SAS code conditionally

Iterative DO Statement Variations

DO *index-variable=start* TO *stop* <BY *increment*>;

DO *index-variable = item-1 <,…item-n>;*

e.g. do *Month = 'JAN', 'FEB', 'MAR';ca*

Note: Increment By takes only Integer Value

8.3.1 Conditional Iterative Processing

DO WHILE and DO UNTIL statements can be used to stop the loop when a condition is met rather than when the index variable exceeds a specific value

DO WHILE Statement

The DO WHILE statement executes statements in a DO loop while a condition is true.

DO WHILE (expression);

<additional SAS statements>

END;

Expression is evaluated at the top of the loop.

The statements in the loop never execute if the expression is initially false.

DO UNTIL Statement

The DO UNTIL statement executes statements in a DO loop until a condition is true.

DO UNTIL (expression);

<additional SAS statements>

END;

Expression is evaluated at the bottom of the loop.

The statements in the loop are executed at least once.

8.3.2 Iterative DO Statement with Conditional Clause

DO WHILE and DO UNTIL statements can be combined with the iterative DO statement.

DO index-variable = start TO stop <BY increment>
WHILE | UNTIL (expression);
<additional SAS statements>
END;

This is one method of avoiding an infinite loop in DO WHILE or DO UNTIL statements.
In a DO WHILE statement, the conditional clause is checked *after* the index variable is incremented.
In a DO UNTIL statement, the conditional clause is checked *before* the index variable is incremented.

8.3.3 Nested DO Loops

Nested DO loops are loops within loops.

When you nest DO loops

- Use different index variables for each loop
- Be certain that each DO statement has a corresponding END statement

8.4 Arrays

The ARRAY statement defines the elements in an array. These elements will be processed as a group. The elements of an array are referred by the array name and subscript.

Syntax

ARRAY *array-name {subscript} <$> <length> <array-elements> <(initial-value-list)>;*

- array-name: specifies the name of the array
- {subscript}: describes the number and arrangement of elements in the array.
- $: indicates that the elements in the array are character elements. The dollar sign is not necessary if the elements in the array were previously defined as character elements.
- length: specifies the length of the elements in the array that were not previously assigned a length.
- array-elements: names the elements that make up the array. Array elements can be listed in any order.
- (initial value-list): gives initial values for the corresponding elements in the array. The values for elements can be numbers or character strings. You must enclose all character strings in quotation marks.

Note:
The ARRAY statement

- must contain all numeric or all character elements
- must be used to define an array before the array name can be referenced
- creates variables if they do not already exist

8.4 Arrays continued…

Arrays can be used to simplify programs that

- perform repetitive calculations
- create many variables with the same attributes
- read data
- compare variables
- rotate SAS datasets by making variables into observations or observations into variables
- perform a table lookup

A *SAS array*

- is a temporary grouping of SAS variables that are arranged in a particular order
- is identified by an array name
- exists only for the duration of the current DATA step
- is not a variable

Note: SAS arrays are different from arrays in many other programming languages. In the SAS system, an array is not a data structure. It is simply a convenient way of temporarily identifying a group of variables.

Each value in an array is

- called an element which is identified by a subscript that represents the position of the element in the array.

When an array reference is used, the corresponding value is substituted for the reference.

8.4.1 SAS Array Processing

Consider the following problem:

Employees contribute an amount to charity every quarter. The SAS dataset *8\donate* contains contribution data for each employee. The employer supplements each contribution by 25%. Calculate each employee's quarterly contribution including the company supplement.

```
Data charity;
   set donate;
   Qtr1 = Qtr1 * 1.25;
   Qtr2 = Qtr2 * 1.25;
   Qtr3 = Qtr3 * 1.25;
   Qtr4 = Qtr4 * 1.25;
run;

Proc print data = charity;
run;
```

What if you want to similarly modify 52 weeks of data stored in *Week1* through *Week52?*

8.4.2 Performing Repetitive Calculations

We can use arrays to simplify the code for the example we saw earlier.

```
Data charity (drop = Qtr);
    set donate;
    array Contrib{4} Qtr1 Qtr2 Qtr3 Qtr4;
    do i=1 to 4;
            Contrib{i} = Contrib{i}*1.25;
    end;
Run;
```

8.4.3 Creating Variables with Arrays

Calculate the percentage that each quarter's contribution represents of the employee's total annual contribution.

```
Data percent (drop = Qtr);
  set donate;
        Total = sum(of Qtr1-Qtr4);
        array Contrib{4} Qtr1 Qtr2 Qtr3 Qtr4;
        array Percent{4};
        do Qtr=1 to 4;
                Percent{Qtr} = Contrib{Qtr}/Total;
        end;
      Run;
```

8.4.4 Assigning Initial Values

Determine the difference between employee contributions and last year's average quarterly goals of $10, $15, $5 and $10 per employee.

```
Data compare (drop = Qtr Goal1-Goal4);
                set donate;
                    array Contrib{4} Qtr1 Qtr2 Qtr3 Qtr4;
                    array Diff{4};
                    array Goal{4} Goal1 – Goal4
                    (10,15,5,10);
                    do Qtr=1 to 4;
                            Diff{Qtr} = Contrib{Qtr} –
                    Goal{Qtr};
                    end;
                Run;
```

Note: Default Initial Value in an Array is 0

8.4.5 Performing a Table Lookup

The keyword _TEMPORARY_ can be used instead of specifying variable names when an array is created to define temporary array elements.

```
Data compare (drop = Qtr);
    set donate;
    array Contrib{4} Qtr1 Qtr2 Qtr3 Qtr4;
    array Diff{4};
    array Goal{4} _TEMPORARY_ (10,15,5,10);
    do Qtr=1 to 4;
            Diff{Qtr} = Contrib{Qtr} – Goal{Qtr};
    end;
Run;
```

8.4.6 Rotating a SAS Dataset

Rotating, or transposing, a SAS dataset can be accomplished by using array processing. When a dataset is rotated, the values of an observation in the input dataset become values of a variable in the output dataset

```
Data rotate (drop = Qtr1-Qtr4);
    set donate;
    array Contrib{4} Qtr1-Qtr4;
    do Qtr=1 to 4;
            Amount = Contrib{Qtr};
            output;
    end;
Run;
```

8.5 Proc Print

General form of proc print

```
Proc print data=datasetname;
run;
```

Some Options to be used:

- **Obs** Limits observations when printing
- **n** option will print the number of observations in the data set at the end of the output.
- **by** statement, the number of observations in each BY group will be printed The proc print expects the data
- to be in order (ascending) of the BY variable. If your data is in descending order you can use by descending
- **Where = ,drop = ,keep=** : var specifies the variables to be listed and the order in which they will appear
- **Noobs** option to suppress row no. on the left side of output
- **Sum** can be used for column totals

Example

```
proc print data=trg.empdata;
by jobcode;  (breaks the print into different job codes)
sum salary;
Run;
```

Note: The data should be sorted by the by variable

8.6 Proc Transpose

The Transpose procedure transposes SAS datasets, turning observation into variables and variables into observations. PROC TRANSPOSE does not produce printed output. To print the output data set from the PROC TRANSPOSE step, use PROC PRINT

Syntax

PROC TRANSPOSE DATA=Input-data-set OUT=output-data-set;
BY variable-1…..variable-n;(grouping variable)
<ID variable>;(variables whose formatted values will become new variable names)
VAR variable-list; (variables whose values are to be transposed)
run;

Sorted and De-duped Dataset

VIEWTABLE: Sast.Sortedstmthistnodup

	AccountNumber	VisaPoints	statement_date
1	1	3771	17JUL2001
2	1	3819	27MAR2003
3	1	3822	24APR2003
4	1	3825	29MAY2003
5	1	3828	26JUN2003
6	1	3831	24JUL2003
7	1	3834	28AUG2003
8	1	3841	02OCT2001
9	1	3844	27FEB2003
10	1	3939	30JAN2003
11	1	3966	30MAY2002
12	1	4018	25JUL2002
13	1	4051	25APR2002
14	1	4062	27JUN2002
15	1	4100	26SEP2002
16	1	4180	27DEC2002
17	1	4333	28NOV2002
18	1	4365	29AUG2002
19	1	4430	24OCT2002
20	1	25370	22JAN2002
21	1	30910	27MAR2002
22	1	89870	21FEB2002
23	2	3837	28JUL1999
24	2	3840	26AUG1999

Transposed Dataset

VIEWTABLE: Sast.Transposedstmthist

	AccountNumber	NAME OF FORMER VARIABLE	_17JUL2001	_27MAR2003	_24APR2003	_29MAY20
1	1	VisaPoints	3771	3819	3822	382
2	2	VisaPoints	.	33180	4464	2314
3	4	VisaPoints	.	49210	58060	5216
4	5	VisaPoints	.	30790	20170	2870
5	6	VisaPoints	.	62340	33820	4576
6	7	VisaPoints	.	12910	5398	5491
7	8	VisaPoints	.	5470	5214	532
8	9	VisaPoints	.	17320	13640	9473
9	10	VisaPoints	.	5762	10350	1106
10	11	VisaPoints	.	14950	5907	1611
11	12	VisaPoints	.	5394	5397	540
12	13	VisaPoints	.	50810	59590	4241
13	14	VisaPoints	.	6054	40480	673
14	15	VisaPoints	.	6202	17980	669
15	16	VisaPoints	.	25880	14970	4178
16	17	VisaPoints	.	69750	70140	7294
17	18	VisaPoints	.	31350	46460	3712
18	19	VisaPoints	.	6854	6771	6774
19	20	VisaPoints	.	6912	6915	691
20	21	VisaPoints	.	34190	23890	2253
21	22	VisaPoints	.	7269	7272	727
22	23	VisaPoints	.	7356	7359	736

Exercise

1. Create a new dataset from Revenue data with unique account numbers.

2. Using the revenue data, summarize the data for aloctd_chg_amt_2 at account number.

3. Compare the interest for yearly versus quarterly compounding on a $50,000 investment made for one year at 7.5% interest. How much money will accrue in each situation?

4. Determine the number of years it would take for an account to exceed $1,000,000 if $50,000 is invested annually at 7.5%.

5. Create one observation per year for five years and show the earnings if you invest $5,000 per year with 7.5% annual interest compounded quarterly

6. From the dataset Revenue, update the variables - TOT_BILL_AMT_2, TOT_CHG_AMT_2, TOT_CR_AMT_2 and TOT_OVRG_AMT_2 as double of what they are currently. Do it using arrays. *Note: Keep and Rename variables as Amt1, Amt2, Amt3 and Amt4.*

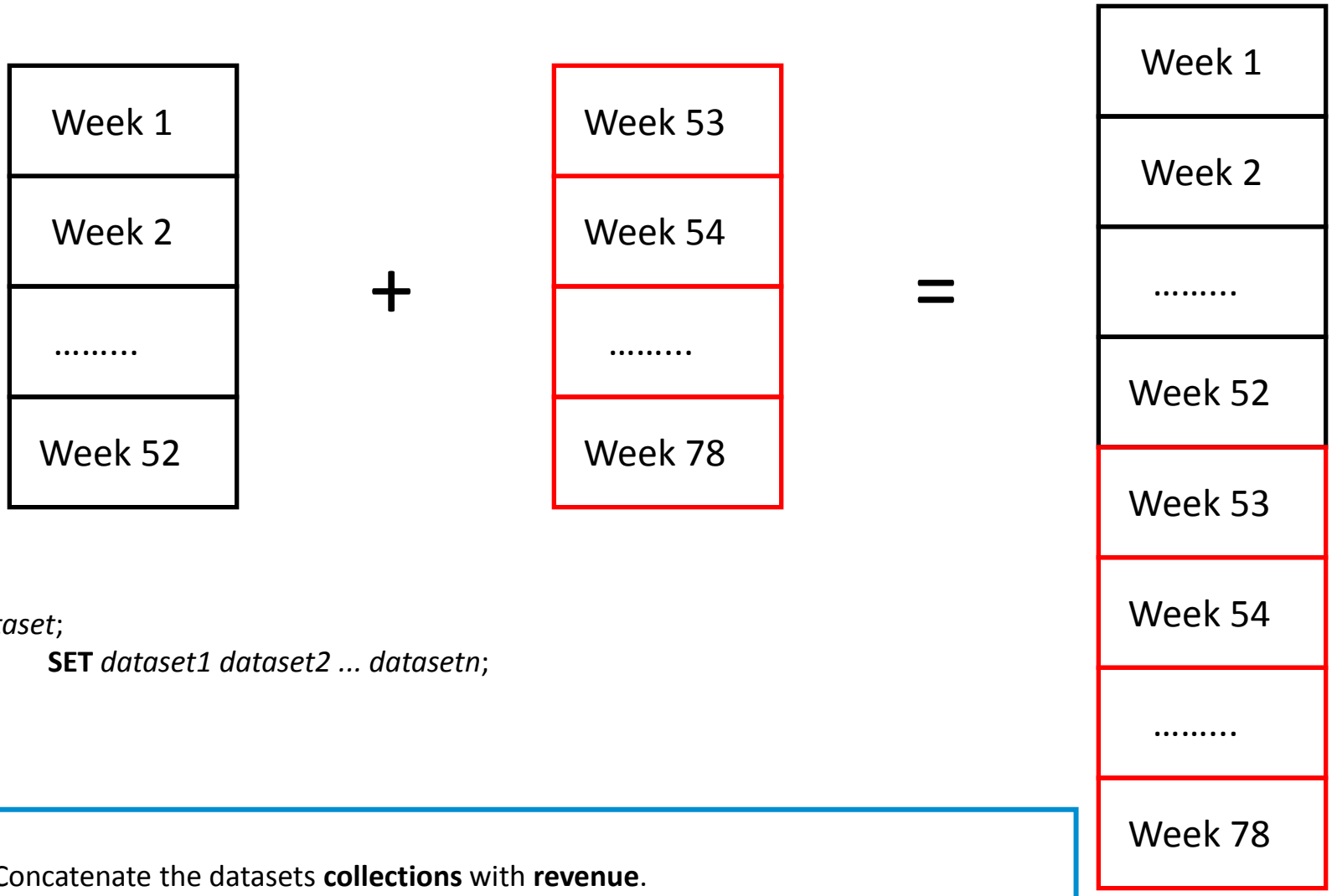
7. Determine the difference between values of the specified variables and their average amount $69, $3, $-6 and $9 respectively from dataset Revenue. Note: Keep TOT_BILL_AMT_2, TOT_CHG_AMT_2, TOT_CR_AMT_2, TOT_OVRG_AMT_2 as Amt1, Amt2, Amt3 and Amt4 from the dataset Revenue .

8. Using the contracts dataset prepare a dataset which shows the total number of ids/Accounts for each value of num_contracts_lifetime grouped by on_contract field. (*Hint: Use Proc Freq and then transpose)*

9. Print the data Revenue from 10th till 20th observation, keeping only 2 variables lj_sub_id and contract_bucket.

9. Data Merging

9.1 Concatenation




```
DATA dataset;
        SET dataset1 dataset2 ... datasetn;
RUN;
```

Exercise 1: Concatenate the datasets **collections** with **revenue**.

9.2 Interleaving

- Interleaving Data Sets
  - Similar to concatenation. The new data set will have the same number of observations but they may be in different order
- General Form

```
DATA dataname;
  SET data1 data2 (... datan);
            BY var1 (... varn);
run;
```

  - Use a ‘BY’ statement. Puts records in order of the ‘BY’ variable(s)

Exercise 2: Interleave the datasets **collections** with **revenue** on Account number

9.3 Proc Append

Append the observations of one dataset to another dataset to obtain a combined dataset

PROC APPEND *BASE=SAS-data-set <DATA=SAS-data-set> <FORCE>*

- If PROC APPEND cannot find the BASE=data set, SAS creates it
- If a DATA=data set is not specified, SAS uses the most recently created SAS data set
- If there is a system failure or other interruption during processing, the BASE=data set may not be properly updated
- If appending a large and small data set, specify the large data set as the BASE data set, and the small data set as the DATA=data set, so that SAS only reads the smaller data set.
- If appending data sets with different variables or attributes, use the FORCE option. This option must be used to prevent a syntax error
- Computationally cheaper because it will not read Base Data Set. Should be preferred

Exercise 3: Concatenate the datasets **collections** with **revenue** using Proc Append

The APPEND procedure bypasses the processing of data in the original data set and adds new observations directly to the end of the original data set. DATA step process all the observations in both data sets to create a new one.

Example:

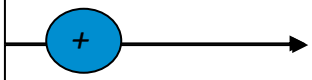
Append <Dataset2> below <Dataset1>

Dataset1

Name	Gender	JCode
Peter	F	FA1
Jupiter	M	FA1
Repeater	F	FA1

Dataset2

Name	Gender	Jcode
Meter	F	FA3
Heater	M	FA2



<Dataset1>

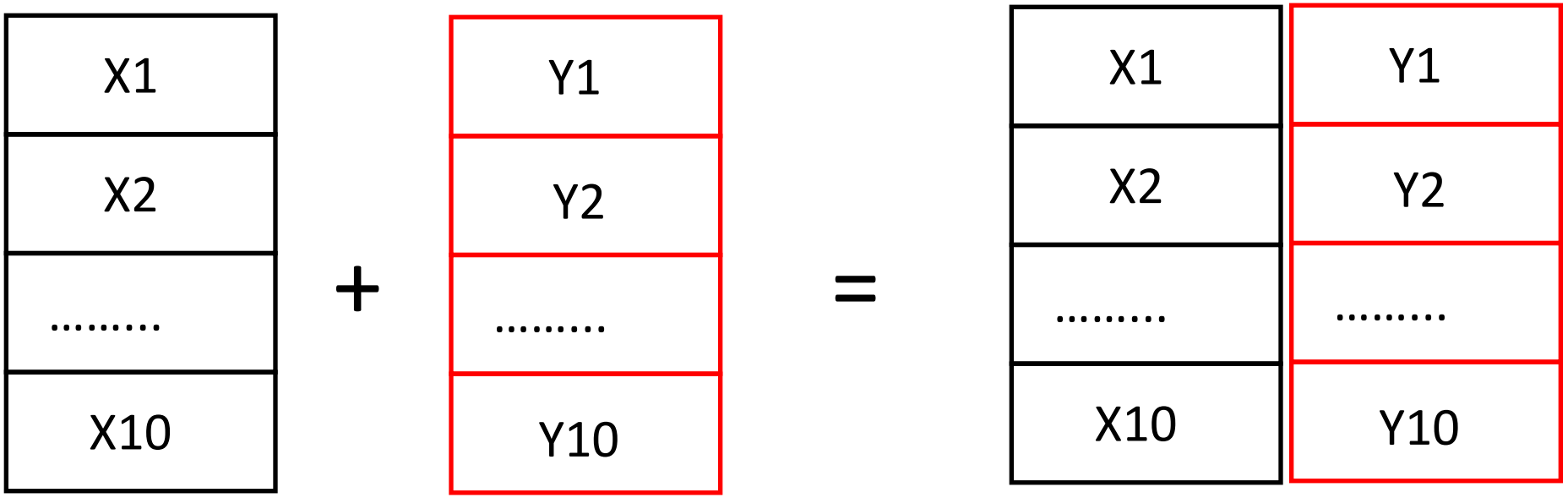
Name	Gender	JCode
Peter	F	FA1
Jupiter	M	FA1
Repeater	F	FA1
Meter	F	FA3
Heater	M	FA2

```
PROC APPEND BASE = Dataset1 DATA = Dataset2 ;
run;
```

9.4 One To One Merging

In one to one merging the no of observations in the new data set equals the no of observations in the largest dataset.

Merging datasets with duplicate values of common variables can produce undesirable results.



Merge statement is used to combine the SAS datasets with related data.

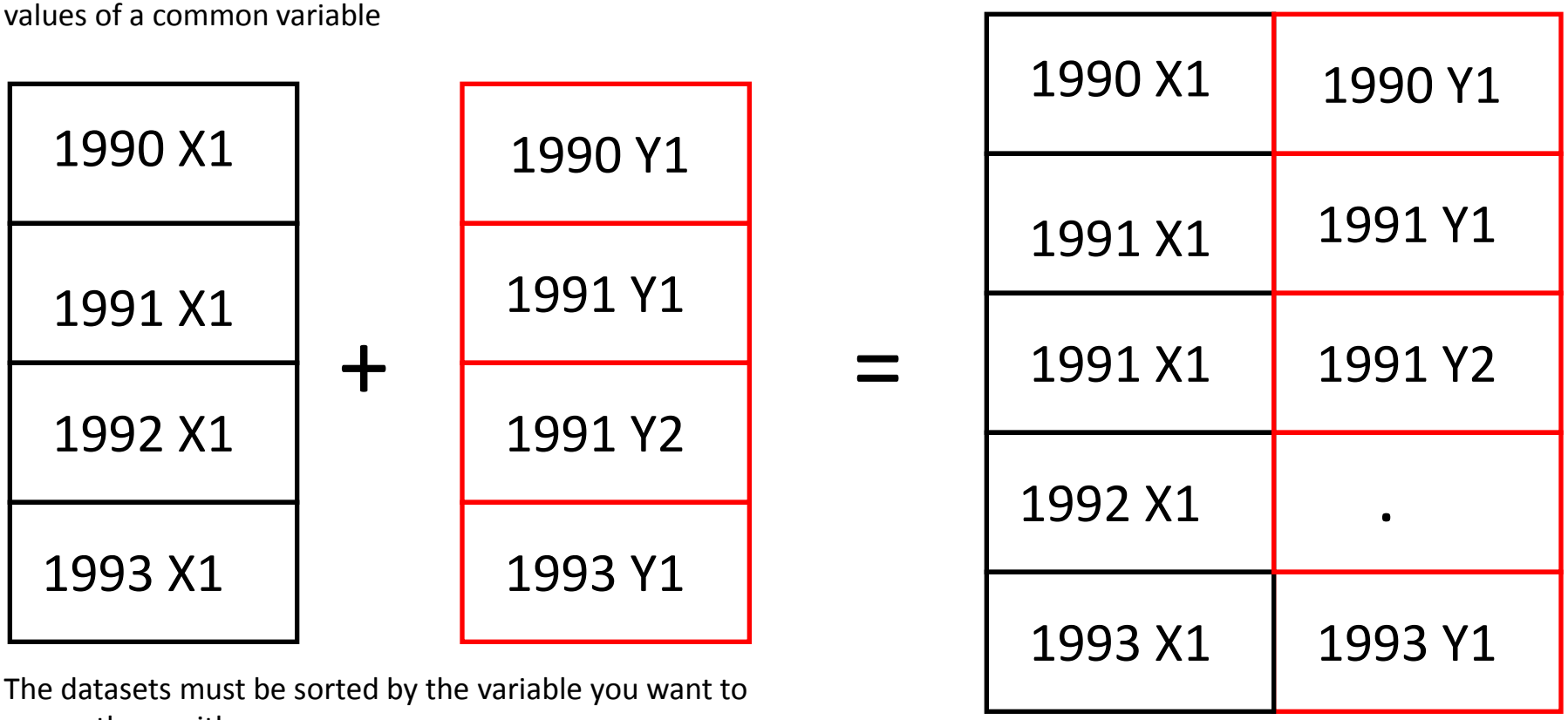
DATA *SAS-data-set….;*
MERGE *SAS-data-set-1 SAS-data-set-2….;*
<additional SAS statements>
RUN;

- **Always sort the dataset on the key before merging**
- **Common variables get overwritten**

Exercise 4: Merge the datasets collections with Revenue

9.5 Match Merging

It combines observations from 2 or more datasets into a a single observation into a new dataset according to the values of a common variable



The datasets must be sorted by the variable you want to merge them with.

DATA *SAS-data-set….;*
MERGE *SAS-data-set-1 SAS-data-set-2….;*
BY *BY-variable-1….;*
<additional SAS statements>
RUN;

Exercise 5: Merge the three datasets **collections, revenue and contract**

9.6 IN= (To identify left, right, outer or inner merge)

Creates a temporary (0/1) variable that identifies which dataset contributed to a record

*Set SAS-data-set (**IN=**NewVar);*

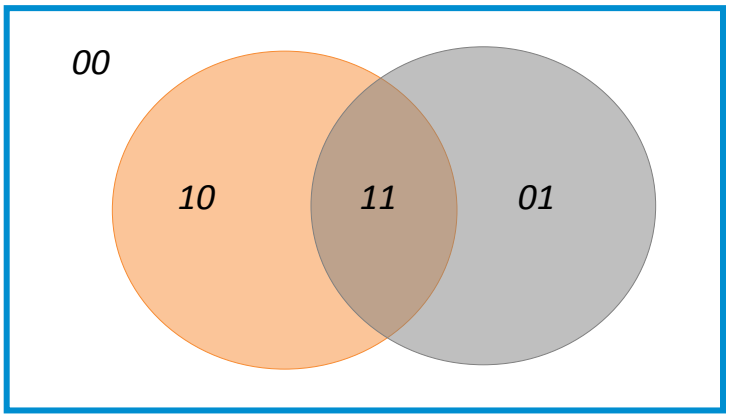
*Merge SAS-data-set (**IN=**NewVar);*

It is always a good practise to do quality check after merging datasets using merge indicator

Exercise 6: Merge contents of dataset **B** into **A**

Example:

```
data <New Dataset>;
  Merge
  <Dataset1> (IN=a)
  <Dataset2> (IN=b);
  Merge_ind = compress(a!!b);
run;
```



9.6.1 Controlling SAS Merge

```
PROC SORT DATA=trg.A;
        by I;
run;

PROC SORT DATA=trg.B;
BY I;
RUN;

DATA trg.ALL;
MERGE trg.A (IN=FROMA) trg.B(IN=FROMB);
BY I;
If FROMA and FROMB;
RUN;
```

10. Data Analysis

10.1 Proc Freq

A SAS dataset typically contains observations that carry information for an individual along various dimensions. Summarizing datasets allows us to obtain an overall or collective view of the data along selected dimensions that is easily comprehensible. PROC FREQ gives the number of observations for each value taken by a variable.

Example

The following is the dataset summary example which shows the spend, revolve and balance behavior of 10 different accounts of a month

Account	Account_type	Card_type	Spend	Revolve	Balance
10001	old	red	200	100	300
10002	old	yellow	10	200	210
10003	new	red	100	50	150
10004	new	blue	0	10	10
10005	new	yellow	20	320	340
10006	old	yellow	200	250	450
10007	old	red	300	0	300
10008	new	blue	5	5	10
10009		red	.	.	.
10010	new	blue	50	0	50

Make a sample dataset from this screenshot and complete the exercises ?

Exercises

- **Ex 1 : How many accounts have a blue, yellow and red card type respectively?**
- **Ex 2 : How many old accounts have a blue-card type?**
- **To obtain summaries that are simple counts by one or more category variable, one can use proc freq. The general syntax of a proc freq statement is**

```
proc freq data = <datasetname>;
tables <variable(s)>/ <options*>
Output <options**>
out= <outputdatasetname>;
run;
```

- **The options list and missing are commonly used with a freq procedure:**
- **list : displays two-way to n-way tables in list format**
- **missing treats missing values as nonmissing**
- ****commonly used stat function is ALL which gives the CHISQ, MEASURES, CMH, and the number of nonmissing subjects**

Ex 1

```
proc freq data = exam.summ_example;
tables card_type/list missing;
run;
```

Output

The FREQ Procedure

Card_type	Frequency	Percent	Cumulative Frequency	Cumulative Percent
blue	3	30.00	3	30.00
red	4	40.00	7	70.00
yellow	3	30.00	10	100.00

Ex 2

```
proc freq data =  exam.summ_example;
tables account_type*card_type/ missing;
run;
```

Output

The FREQ Procedure

Table of Account_type by Card_type

Account_type
Card_type

Frequency Percent Row Pct Col Pct	blue	red	yellow	Total
	0 0.00 0.00 0.00	1 10.00 100.00 25.00	0 0.00 0.00 0.00	1 10.00
new	3 30.00 60.00 100.00	1 10.00 20.00 25.00	1 10.00 20.00 33.33	5 50.00
old	0 0.00 0.00 0.00	2 20.00 50.00 50.00	2 20.00 50.00 66.67	4 40.00
Total	3 30.00	4 40.00	3 30.00	10 100.00

Notes:

1. The column frequency contains the number of observations in the category. Hence *proc freq* is often referred to as taking frequencies or counts.
   It is a common practice to use the freq procedure on the merge indicator after merging two datasets to ensure the correctness of the merge.
2. If a variable list is specified with a tables statement, the procedure produces one-way frequencies for each variable on the list.

- NOCOLS does not give column percentage
- NOPERCENT option suppresses printing of the cell percentages and cumulative percentages
- OUTPUT creates a SAS data set with the statistics that PROC FREQ computes for the last TABLES statement request. The variables contain statistics for each two-way table or stratum, as well as summary statistics across all strata.

  OUTPUT statistic-keyword(s) <OUT=SAS-data-set>;

  Note: using out option without output statement will only give the freq, percent, row percent and column percent in the dataset. With output command, we can get other parameters like chisqr, binomial stats etc.

Exercise 3

Repeat ex 1 and 2 but now create output datasets instead of LST files using out= option and note the difference.

10.2 Proc Summary

Proc summary **(and** ***proc means*****)** statements allow us to get more detailed summaries of a dataset. They allow us to summarize specified numerical variables by given 'class' variables

The general syntax of a proc summary statement is as follows:

```
proc summary data = <datasetname> <options>;
class <class_variables>;
var  <var_variables>;
output out = <output datasetname> <requested statistics>;
run;
```

- Variables specified as 'class' variables are typically categorical variables which take a few unique values across the dataset
- Variables specified on var statement must be defined numerical
- A proc summary groups together the observations by distinct combinations of class variables
- For each group so defined, the numerical variables specified in the var statement are summarized by the requested statistics
- The statistics that may be requested on a var variable are
  - N= : # observations with non-missing values
  - Nmiss= : # observations with missing values
  - sum= : sum
  - max= : max
  - min= : min
  - mean= : average over non-missing observation
- If an output statement is not specified with a proc summary then no output dataset is produced. In such cases an explicit print option to print the results must be specified.

10.2.1 Options

The *nway* and *missing* options:
It is advisable to specify the 'nway' and 'missing' options with a proc summary. They ensure that the result and output is 'MECE'

- When the *missing* option is not specified then observations with missing values for a *class* variable are ignored
- When the *nway* option is not specified and there are *n* class variables, all the 2^n ways of grouping these variables are considered.

```
proc summary
data=exam.summary_example
nway missing;
class  account_type card_type;
output out = tmp1;
run;
```

Output

Obs	Account_ type	Card_ type	_TYPE_	_FREQ_
1		red	3	1
2	new	blue	3	3
3	new	red	3	1
4	new	yellow	3	1
5	old	red	3	2
6	old	yellow	3	2

```
proc summary
data=exam.summary_example
nway;
class  account_type card_type;
(categorical)
var  balance;(non categorical)
output out = tmp2 ;
run;
```

Output

Obs	Account_ type	Card_ type	_TYPE_	_FREQ_
1	new	blue	3	3
2	new	red	3	1
3	new	yellow	3	1
4	old	red	3	2
5	old	yellow	3	2

```
proc summary
data=exam.summary_example
missing;
class  account_type card_type;
var  balance;
output out = tmp3 ;
run;
```

Output

Obs	Account_ type	Card_ type	_TYPE_	_FREQ_
1			0	10
2		blue	1	3
3		red	1	4
4		yellow	1	3
5			2	1
6	new		2	5
7	old		2	4
8		red	3	1
9	new	blue	3	3
10	new	red	3	1
11	new	yellow	3	1
12	old	red	3	2
13	old	yellow	3	2

Note:

1. When 'nway' is specified, the_type_variable takes the same value for all observations
2. If in a *proc summary,* an *output* statement is specified without a *var* statement, then no statistics should be requested

The Types and Ways statements:

The *types* and *ways* statements override the *nway* option in a proc summary statement. These help output only specific combinations of the class variables to the output.

To summarize a dataset by class variables A, B, C and D, a simple code as follows would be used.

```
proc summary data=D1 nway
missing;
Class A B C D;
output out = tmp1;
run;
```

This approach considers all the 4-way combinations of the values assumed by the variables A, B, C and D and blows up the number of observations in the resulting dataset.

If we are only interested in combinations of variables A, B, C and D two at a time, *ways* statement can be used

```
proc summary data=D1 missing;
Class A B C D;
output out = tmp1;
ways 2; (4c2)
run;
```

This produces summaries by the combinations A*B, A*C…..C*D.

The *ways* statement will override the *nway* option even if it is specified.

An even more customized output can be obtained by using the *types* statement which allows one to specify the specific combinations of the class variables. For example, to obtain summaries by A, and combinations (A,C) and (A,B,D), use the code below

```
proc summary data=D1 missing;
Class A B C D;
Types A A*C A*B*D;
output out = tmp1;
run;
```

This produces summaries by the combinations A*B, A*C…..C*D.

The *ways* statement will override the *nway* option even if it is specified.

Note: Types statement can be also be written using associative grouping, as A*(() C B*D)

Examples

What is the total balance of old and new accounts with a red, blue or yellow card? Store this information in a SAS dataset

/* var tells the non categorical variables on which categorical var have to be evaluated*/

Program to print results

```
proc summary data=exam.summary_example print
sum nway missing;
class account_type card_type;
var spend balance;
run;
```

Output

The SUMMARY Procedure

Account_ type	Card_type	N Obs	Variable	Sum
	red	1	Spend	.
			Balance	.
new	blue	3	Spend	55.0000000
			Balance	70.0000000
	red	1	Spend	100.0000000
			Balance	150.0000000
	yellow	1	Spend	20.0000000
			Balance	340.0000000
old	red	2	Spend	500.0000000
			Balance	600.0000000
	yellow	2	Spend	210.0000000
			Balance	660.0000000

Program to store and print results

```
proc summary data=exam.summary_example nway
missing;
class account_type card_type;
var spend balance;
output out=work.summ1 sum= a b mean = c d;
run;
proc print data = work.summ1;
run;
```

Output

Obs	Account_ type	Card_ type	_TYPE_	_FREQ_	Spend	Balance
1		red	3	1	.	.
2	new	blue	3	3	55	70
3	new	red	3	1	100	150
4	new	yellow	3	1	20	340
5	old	red	3	2	500	600
6	old	yellow	3	2	210	660

Notes:

1. The _freq_ and _type_variables are automatically generated by the summary procedure. *_freq_* holds the number at observations that fall within a group
2. It is possible to drop/rename the _freq_variables and type variables and _type_variables using the *drop* or *rename* option in the output statement

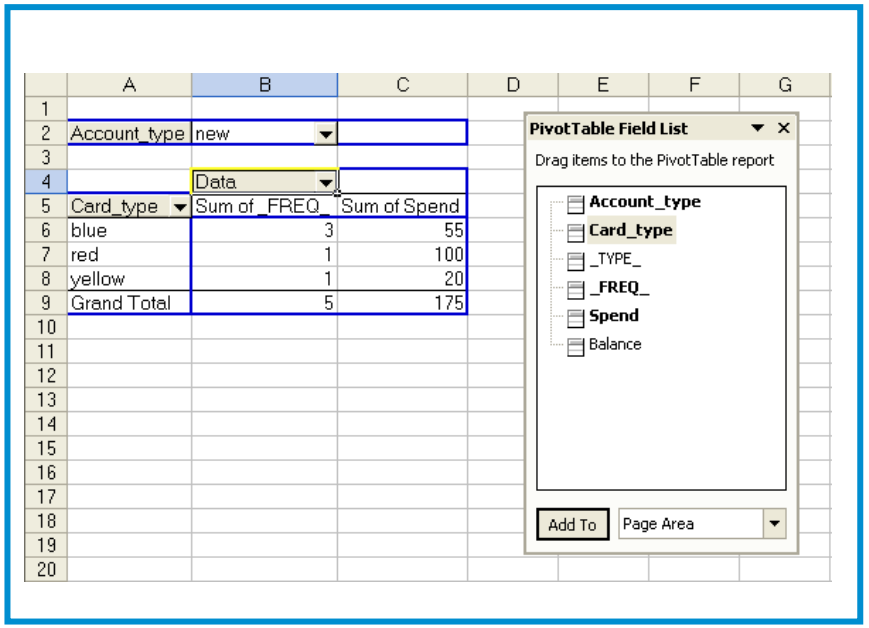
Transfer Analysis to Excel

With *proc summary* we can bring down the size of data to a level amenable for analysis using excel. For example the output set created using Ex 1 can be exported to an Excel file using *DBMS copy or proc export*. A pivot can be generated in this Excel file to obtain different statistics

Summary Dataset

Account_type	Card_type	_TYPE_	_FREQ_	Spend	Balance
	red	3	1	.	.
new	blue	3	3	55	70
new	red	3	1	100	150
new	yellow	3	1	20	340
old	red	3	2	500	600
old	yellow	3	2	210	660

Excel Pivot



10.3 Proc Means

- The *proc means* statement is similar to the *proc summary* statement except that
- When a var statement is not specified, proc summary does not compute any statistics whereas proc means computes default statistics on all numeric variables. The default statistics are N, mean, std dev, min, max
- The default action in proc means is that it places the output in the output window whereas in proc summary the default is to create an output data set.

General Syntax of *proc means*

```
proc means data = <datasetname> <option>;
class <class_car>;
var <var_name>;
run;
```

```
proc summary data = exam.summary_example nway
missing print;
run;
```

Output

The SUMMARY Procedure

N Obs
10

```
proc means data = exam.summary_example
nway missing;
run;
```

Output

The MEANS Procedure

Variable	N	Mean	Std Dev	Minimum	Maximum
Account	10	10005.50	3.0276504	10001.00	10010.00
Spend	9	98.3333333	109.5445115	0	300.0000000
Revolve	9	103.8888889	122.6727806	0	320.0000000
Balance	9	202.2222222	157.8589384	10.0000000	450.0000000

When to use Summary and when to use Means?

10.3.1 Options

- **CHARTYPE OPTION** converts _type_ function to the character format composed of a series of zeroes and ones corresponding to the variables in the class statement. You can create multiple output datasets from proc means using where close together with chartype. Unlike the ways option with proc summary the chartype option allows the creation on new datasets.
- **AUTONAME** – with the autoname option, SAS places the _keyword after the variable name in the output. This is important when requesting for multiple analysis otherwise proc means incomplete results.

```
proc means data= trg.summary_example
chartype missing;
class account_type card_type;
var spend revolve;
output out=trg.new1 (where = (_type_ in ('01' ,'10'))) sum=;
output out=trg.new2 (where = (_type_ in ('11'))) mean= sum=/autoname;
run;
```

10.3.2 Proc means for EDD

Proc means is used in the EDD generation macro. Apart from the default statistics, one can specify the following statistics with *proc means* statement

Nmiss :observations with missing value
median :median value
p1, p5, p10, p25, p50, p75, p90, p95, p99 where
p<n> is the nth percentile value

Example

```
proc means data = exam.summary_example nway
missing N Nmiss min p1 p10 median p90 p99 max ; (gets reflected in the output file only)
run;
```

Output

The MEANS Procedure

Variable	N	N Miss	Minimum	1st Pctl	10th Pctl	Median	90th Pctl	99th Pctl	Maximum
Account	10	0	10001.00	10001.00	10001.50	10005.50	10009.50	10010.00	10010.00
Spend	9	1	0	0	0	50.0000000	300.0000000	300.0000000	300.0000000
Revolve	9	1	0	0	0	50.0000000	320.0000000	320.0000000	320.0000000
Balance	9	1	10.0000000	10.0000000	10.0000000	210.0000000	450.0000000	450.0000000	450.0000000

Notes

1. The median for the *Account* variable is a value that none of the observations in the dataset takes
2. In general it is a good practice to define 'index' variables as character. Thus *Account* here could have been defined as a character variable. No statistics would have been computed for *Account* in that case.

Exercise 4

1. **Find the number of subscribers in the various bins in contract_bucket for the Contract Dataset**
2. **Find the mean and standard deviation values for num_contracts_lifetime(Number of Contracts a subscriber has ever been on) for the individual bins in contract_bucket for the Contract Dataset**
3. **For Question 2 above output the values in a dataset**
4. **Find the mean.p25,p50 and p75 for mnths_left_contract in the Contract dataset for the subscribers on contract (on_contract=1) and not on contract(on_contract=0)**

11. Proc SQL

11.1 PROC SQL Syntax

PROC SQL is SAS' implementation of the Structured Query Language (SQL). It is a powerful Base SAS Procedure that combines the functionality of DATA and PROC steps into a single step.

PROC SQL <options>**;**
CREATE TABLE Dataset-Name **AS**
SELECT <column(s)(* ,distinct)>
FROM <table-name | view-name>
WHERE <expression>
GROUP BY <column(s)>
HAVING <expression>
ORDER BY <column(s)>**;**
QUIT;

Note: The above order of SQL steps is important.

- The **CREATE TABLE** statement provides the ability to create a new dataset (same as DATA statement in SAS).
- The purpose of the **SELECT** statement is to name the columns that will appear on the dataset and the order in which they will appear.
- The **FROM** clause is used for specifying the input dataset name (same as SET statement of SAS).
- The **WHERE** clause selects the rows meeting certain condition(s) (same as WHERE statement in SAS).
- The **GROUP** statement is used for aggregating/summarizing the data.
- The **ORDER BY** clause returns the data in sorted order of specified column(s) (same as PROC SORT in SAS).

SAS Data/Proc steps v/s SQL

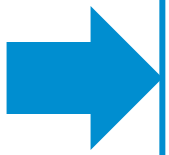
- SAS looks at a dataset one record at a time, using an implied loop that moves from the first record to the last record. SQL looks at all the records, as a single object, thus SQL built-in functions work over the entire dataset
- Among the significant differences is that PROC SQL can use SAS data step functions, SAS macros and macro variables. SQL Proc can do a many to many merge. It does not require to sort data set before merging. It can also easily join on variables with different names. SQL can perform a Cartesian product very easily. SQL also allows merging where the condition of match is not equality.
- Variable separator is a comma in contrast to SAS data/proc steps where a blank is used
- It is recommended that one should not use SQL for large datasets

11.2 PROC SQL Statements

Using dataset FLTATTND

- **List all information in the table**:

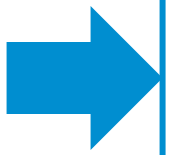
  An asterisk on the SELECT statement will select all columns from the dataset.



```
PROC SQL;
SELECT *
FROM trg.fltattnd;
QUIT;
```

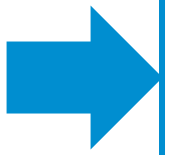
- **Limiting information on the SELECT**:

  To specify certain variables in the output dataset, the variables are listed and separated on the SELECT statement by a comma(,). The SELECT statement does NOT limit the number of variables read.



```
PROC SQL;
CREATE TABLE new AS
SELECT LastName, FirstName, Salary
FROM trg.fltattnd;
QUIT;
```

- **SAS features work with PROC SQL**



```
PROC SQL;
CREATE TABLE new AS
SELECT *
FROM trg.fltattnd (DROP =JobCode);
QUIT;
```

The SELECT statement is mandatory on every PROC SQL query.

11.2 PROC SQL Statements .. Contd.

- **Creating new variables**:

  Variables can be dynamically created in PROC SQL.

```
PROC SQL;
CREATE TABLE new AS
SELECT Salary, Salary*0.05 AS Tax
FROM trg.fltattnd;
QUIT;
```

- **Sorting the data using PROC SQL**:

  If the data is already in sorted order, PROC SQL will print a message in the LOG stating the sorting utility was not used.

  Group by will work as order by(Sort) when no computations are done with a note in the log.

```
PROC SQL;
CREATE TABLE new AS
SELECT *
FROM trg.fltattnd
ORDER BY JobCode, Salary DESCENDING;
QUIT;
```

- **Sub-setting using the WHERE**:

  The WHERE statement will process a subset of data rows before they are processed.

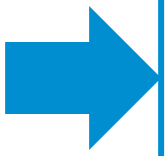
  The WHERE clause cannot reference a computed variable from the same dataset

```
PROC SQL;
CREATE TABLE new AS
SELECT Salary, HireDate
FROM trg.fltattnd
WHERE Salary > 20000;
QUIT;
```

11.2 PROC SQL Statements .. Contd.

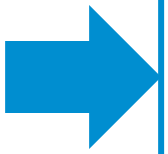
- **Removing duplicates:**

NoDup : Removing duplicate rows and keeping only unique



```
PROC SQL;
CREATE TABLE new AS
SELECT DISTINCT JobCode, Location
FROM trg.fltattnd;
QUIT;
```

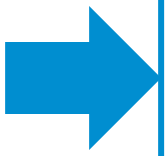
NoDupkey : **Removing duplicate rows and keeping only unique on some or combination of variables**



```
PROC SQL;
CREATE TABLE new AS
SELECT DISTINCT JobCode, MAX(HireDate) AS HireDate
FROM trg.fltattnd
GROUP BY JobCode;
QUIT;
```

- **Using FORMATS:**

SAS-defined or user-defined formats can be used to improve the appearance of variables in a dataset.



```
PROC SQL;
CREATE TABLE new AS
SELECT HireDate, Salary, Salary*0.05 as Tax
FORMAT = DOLLAR10.2
FROM trg.fltattnd
WHERE Salary > 10000
ORDER BY Salary, HireDate DESCENDING;
QUIT;
```

Any of the DATA step functions can be used in an expression to create a new variable except ALG, DIF, and SOUND.

11.2 PROC SQL Statements .. Contd.

- **The CALCULATED option** :

The CALCULATED component must refer to a variable created within the same SELECT statement.

```
PROC SQL;
SELECT Country, Salary*0.05 AS Tax, (Salary*0.05)*
.01 AS Rebate
FROM trg.fltattnd;
QUIT;
```

Alternatively , by using calculated option

```
PROC SQL;
SELECT Country, Salary*0.05 AS Tax,
CALCULATED Tax * .01 AS Rebate
FROM trg.fltattnd;
QUIT;
```

- **The CASE Expression**:

The CASE expression can be used to create a new variable that is a “re-categorization” of the values of another variable. Coding the WHEN in descending order of probability will improve efficiency because SAS will stop checking the CASE conditions as soon as it finds the first true value.

```
PROC SQL;
CREATE TABLE new AS
SELECT LastName, Salary
      CASE
        WHEN Salary <= 20000 THEN ‘Low Sal’
        WHEN Salary <= 30000 THEN ‘High Salary’
        ELSE ‘Very High’
      END as SalType LENGTH = 12
  FROM trg.fltattnd
  ORDER BY Salary;
  QUIT;
```

PROC SQL Examples

```
PROC SQL;
 CREATE TABLE new AS
 SELECT COUNT(*) AS n, ROUND(MEAN(Salary),0.01) FORMAT = 8.2 AS SalaryMean, PUT(Salary,12.) AS Salary FORMAT
 $12.
 FROM trg.fltattnd;
 QUIT;
```

```
PROC SQL;
 CREATE TABLE new AS
 SELECT JobCode, COUNT(JobCode) AS n, Salary, SUM(Salary) AS SalaryMax, ROUND(Salary/(CALCULATED
 SalaryMax)*100,0.01) FORMAT 6.2 as Salpct
 FROM trg.fltattnd
 GROUP BY jobcode;
 QUIT;
```

Note:

- Summary functions are restricted to the SELECT and HAVING clauses only.
- All the above functions will be calculated for each jobcode group not for each observation
- GROUP BY Clause requires at least 1 summary function in the SELECT statement, else it get transforms to a ORDER BY clause.

PROC SQL Examples

```
PROC SQL;
  CREATE TABLE new AS
  SELECT JobCode, COUNT(JobCode) AS n, HireDate FORMAT = date7., Salary, MAX(Salary) as SalaryMax
  FROM trg.fltattnd
  GROUP BY JobCode
  HAVING Salary = CALCULATED SalaryMax
  ORDER BY CALCULATED SalaryMax DESCENDING;
  QUIT;
```

Note: Predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups.

PROC SQL Examples

Example Code

Proc SQL can be used either to create new variables or to merge datasets.

```
        PROC SQL;/* create mean of a variable */
    CREATE TABLE <new dataset> AS
    SELECT *, mean(<var-name>) AS <new var-name>
    FROM <existing dataset>;
QUIT;

        PROC SQL;/* merge two datasets by a variable */
          CREATE TABLE <new dataset> AS
          SELECT * FROM <dataset 1>, <dataset 2>
          WHERE <dataset 1>.<Var 1> = <dataset 2>.<Var 2>;
        QUIT;
```

Note: One limitation of PROC SQL is that it can merge only up to 16 tables (datasets) at a time. The normal limit in SAS is 100. Also note that if you use PROC SQL to merge two or more datasets, you need not sort any dataset.

Exercise 1

Ex 1:Merge datasets Contracts and Revenue on the common field

11.3 String Operations

- SQL supports a variety of string operations such as
  - concatenation (using "||")
  - converting from upper to lower case (and vice versa)
  - finding string length, extracting substrings, etc.

- SQL also includes a **string-matching operator** for comparisons on character strings. Patterns are described using two special characters:
  - percent (%). The % character matches any substring.
  - underscore (_). The _ character matches any character.

- Find the names of all customers whose street includes the substring "Main".

SELECT *name*
FROM *customer*
WHERE *street* ***LIKE*** **'**%Main%'

11.4 Nested Sub queries

- SQL provides a mechanism for the nesting of sub queries.
- A sub query is a **select-from-where** expression that is nested within another query.
- A common use of sub queries is to perform tests for set membership, set comparisons, and set cardinality.

Example :Find all customers who have both an account and a loan at the bank.

Example:

SELECT DISTINCT *customer-name*
FROM *borrower*
WHERE *customer-name IN* (SELECT *customer-name* FROM depositor)

Exercise 2

Using Dataset CONTRACT compute:

Summary Commands

- What are the total number of records.
- What are the number of distinct values in variable “contract_bucket”.
- Check for duplicates on Subscriber Number(“ lj_sub_id”) using SQL.

Functions

- What is the maximum and minimum value of num_contracts_lifetime.
- Make 2 bins dividing the data into 2 equal groups(a new variable containing the categories) on the variable num_contracts_lifetime.

Conditional

- Make a dataset containing subscribers who have some months remaining for the contract using mnths_left_contract variable
- Calculate the years left on the contract using the months left on contract and filter the customers who have 2 or more years left on the contract using the calculated variable
- Create a dataset containing records which have cntrc_lgth_qty>=24 and on_contract=1.

Grouping

- Compute average num_contracts_lifetime by contract_bucket and make a dataset containing average num_contracts_lifetime >2.

11.5 Merging and Concatenation Using SQL

11.5.1 Cartesian Join

- A Cartesian join is when you join every row of one table to every row of another table. You can also get one by joining every row of a table to every row of itself.

- Example. Run the following code on the dataset for SQL and note the results

```
PROC SQL;
  CREATE TABLE matrix AS select * FROM
   (SELECT ans AS ans0001 FROM trg.forsql WHERE var='0001'),
   (SELECT ans AS ans0006 FROM trg.forsql WHERE var='0006'),
   (SELECT ans AS ans0003 FROM trg.forsql WHERE var='0003')
   ORDER BY ans0001, ans0006, ans0003;
QUIT;
```

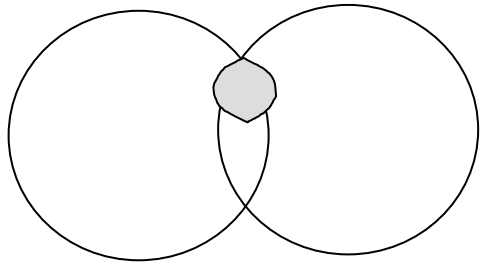
- Cartesian Joins allow for combining tables to self.
- Ex. Find the names of all branches that have greater assets than some branch located in Brooklyn.

```
PROC SQL;
SELECT DISTINCT T.branch-name
                FROM branch AS T, branch AS S
                WHERE  (T.assets > S.assets) AND S.branch-city = ‘Brooklyn’;
```

11.5.2 Merging Using SQL - Joins

- **Inner Join: Inner joins retrieve all matching rows from the WHERE** clause

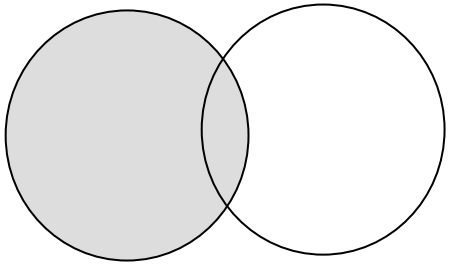
```
PROC SQL;
SELECT A.Zip, A.State, B.zip
FROM States AS A, zip AS B
WHERE A.Zip = b.Zip;
QUIT;
```



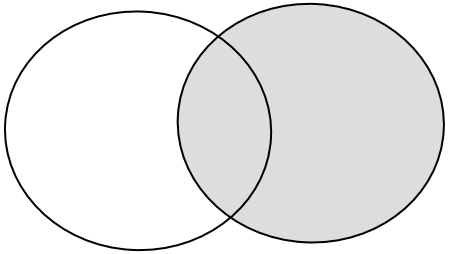
Inner Join

- **Outer Join**: There are also joins called *outer joins* which can only be performed on two tables at a time. They retrieve all matching rows plus any non-matching rows from one or both of the tables depending on how the join is defined. There are three common types

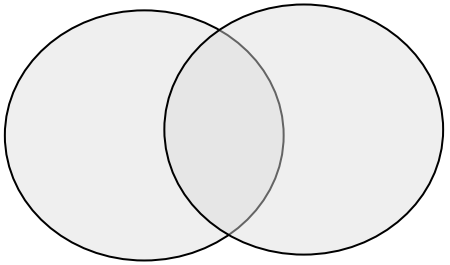
Left join



Right join



Full join



Note: by default proc sql prints the result of every query .use the no print option to suppress the printing.

Examples

Example 1: Solve the exercise of merging contents of B into A using SQL merge.

```
PROC SQL;
CREATE TABLE all AS
SELECT b.i, a.x, b.y
FROM trg.a AS p LEFT JOIN trg.b AS q ON p.i = q.i
QUIT;
```

NOTE: by default, SAS implements inner join.

Example 2: Solve the exercise of merging contents of A into B using SQL merge.

Example 3: Solve the exercise of merging contents of A and B into C using SQL merge.

11.5.3 Concatenation : Outer UNION

OUTER UNION is equivalent to concatenation done on Base SAS.

The CORRESPONDING, or CORR, keyword causes PROC SQL to match the columns in datasets by name and not by ordinal position. Without the use of CORRESPONDING, matching columns from the contributing tables with the same name would be created in separate columns in the result table.

```
PROC SQL;
TITLE 'A and B: OUTER UNION';
SELECT *
FROM trg.a OUTER UNION CORR
SELCT *
FROM trg.b
QUIT;
```

11.5.3 Concatenation Operators

Operator	Description
Union	Matches by **column position (not column name) and** drops duplicates
Union All	Matches by **column position but does not drop** duplicates
Union Corresponding	Matches by **column name and drops duplicates**
Union All Corresponding	Matches by **column name and does not drop duplicates**
Except	Matches by **column name and drops rows found in** both tables
Intersection	Matches by **column name and keeps unique rows in** both tables

Exercise 3:

- Do a CARTESIAN Join on datasets Collections and Revenue for first 20 records(Take only the first 20 rows in separate datasets).

- Inner Join
  - Do an inner join on account number(lj_acct_no) on Collections and Revenue datasets

- Outer Joins

  - Left Join
    - Bring the PAST_DUE_AMT information from collections data on account numbers into the revenue data by left join
  - Right Join
    - Do a right join of Contract dataset with revenue dataset to bring “on_contract” information and output only on_contract=1 data.

11.5.4 Fuzzy Merges

- Fuzzy merges allow matching using non-equality conditions. It is used in two different ways. In case of dates we can use it as
  - "where date is between begdate and enddate"

- It can also be done as
  - where a.id = b.id and date is between begdate and enddate

- In another cases where you want to merge on identifiers but are not sure of the correctness. You can apply a fuzzy merge condition like

- where sum (substr(a.SSN,1,1) = substr(b.SSN,1,1) , substr(a.SSN,2,1) =substr(b.SSN,2,1) ,

  ....

  substr(a.SSN,9,1) =

  substr(b.SSN,9,1)

) >= 7

Fuzzy Merge Example

Ex. Merge the records from benefit table with scheme dates. Each customer has purchased a scheme offered to her given in Scheme Offers Table. The merged file should have record for each customer, purchase date of scheme, accumulated points, scheme name

Benefit Table (Benefits_table)

Customer	Purchase Date	Points
1	4/14/2003	12
2	4/10/2003	3
3	4/4/2003	99

Schemes Offer Table (Scheme_table)

Customer	Scheme Start Date	Scheme End Date	Scheme
1	4/13/2003	4/15/2003	GOLD
1	4/19/2003	4/21/2003	SILVER
2	3/22/2003	3/25/2003	BRONZE
2	4/9/2003	4/11/2003	GOLD
3	3/1/2003	3/3/2003	BRONZE
3	5/9/2003	5/11/2003	PLATINUM

```
proc sql;
  select c.customer,
       c.purchase_date,
       c.points, d.Scheme
   from benefits_table c,
       Scheme_table d
                  where c.customer=d.customer and (d.start_date le c.purchase_date  le d.end_date);
quit;
```

11.5.5 Merging Files with Different Names For Variables

Ex 4. Make 2 sample datasets given in the format and description below by using the datalines statement. There are various departments in IIT. There are two different attendance records formats available. We are required to create a single file having information from both the departments.

Record Layout 1	Record Layout 2
year	Year
Cno	Course
Rnum	Rollno
S1-S10	S1-S10

You are required to convert the source variables having analogous information and copy information from these two files in a single file using SQL.

11.6 Copy from multiple sources into single file

```
PROC SQL;
CREATE TABLE attendance
(
year                    char(4),
course                  char(5),
Rollno                  char(3),
S1                      char(1),
S2                      char(1),
…
S10                     char(1)
)
;
QUIT;
```

```
%MACRO insert(course, roll, file);
 PROC SQL;
  INSERT INTO attendance
   SELECT year,
        &course.,
        &roll.,
        S1-S10,

  FROM &file.;
 QUIT;
%MEND insert;
```

Function definition

```
%insert(cnum, rnum, elecatd);
%insert(course_no, roll_no, compatd);
```

Function calls

11.7 Summarizing Data Using PROC SQL – LPD Example

- Create summary records from master data set based on different “where clause” conditions in separate queries and insert them into one summary data set.
  - Patient ID
  - Date Dispensed
  - Product ID
  - Prescriber ID
  - Days Supply
  - Quantity Dispensed
  - New Brand Start (NBS)
  - Switch From
  - Yearmo
- A summary table with one record per patient is desired to be used in statistical analyses
- You want the summary data set to be created with the following characteristics:
  - One record per prescriber
  - “Product ID” (8 class levels) to be categorized to create the new variable, “Zetia Status (2 Levels),
  - Summary variable “Mean_Quantity” to be created as the mean quantity supplied per patient,
  - Records with missing Prescriber ID to be excluded,
  - Persons with more than one value for Zetia_Status were excluded (to eliminate patients with indistinguishable status)

Create Summary File:

```
Proc SQL;
        Create Table drugsummary
        (
        patient_id                      char(11),
        Zetia_Status                    char(4),
        Mean_qs                         num
        )
;
Quit;
```

Insert Values

```
Proc  Sql;
INSERT INTO DrugSummary
 Select  Distinct Patient_id,
  Case
  When Product ID = ‘1’
   Then ‘1’
   Else ‘0’
  End As Zetia_Status,
  Mean(quantity_supplied) As Mean_qs
 FROM BrandKPM
 WHERE PrescriberID Is Not Missing
   And DateDispensed between  ’01Jan04’d and ’01jan03’d
   And Count(distinct zetia_status) = 1
 GROUP BY PatientID;
Quit;
```

- The count function in this WHERE clause is an example of PROC SQL code that can accomplish in one step, what would take more than one step and several more lines of regular SAS code

```
WHERE PrescriberID Is Not Missing
 And DateDispensed between  '01Jan02'd and '01jan03'd
 And Count(distinct   zetia_status) = 1
GROUP BY PatientID;
```

- The case expression can be used to create a new variable that is a "re-categorization" of the values of another variable

```
    Case
            When Product ID = '1'
                    Then '1'
                            Else '0'
            End As Zetia_Status,
```

11.8 Comparing Proc SQL and Data step

Function	Non-SQL Base SAS STATEMENT OPTION	PROC SQL STATEMENT/CLAUSE
Create a Table	DATA	CREATE TABLE
Create a table from raw data	INPUT	INSERT
Add columns	Assignment statement	SELECT….as….
Drop columns	DROP KEEP	SELECT
Add rows	OUTPUT	INSERT INTO
Delete rows	WHERE IF/THEN DELETE	DELETE FROM
Sorting	PROC SORT	ORDER BY
De-dupe records	NODUPLICATE	DISTINCT
Establish a connection with a RDBMS	LIBNAME	CONNECT DISCONNECT
Send a RDBMS-specific non-query SQL statement to a RDBMS		CONNECTION TO
Concatenating	SET	OUTER JOIN
Match merging	MERGE/SET BY IF in1 IF in2 IF in1 &in2	FROM LEFT JOIN RIGHT JOIN FULL JOIN WHERE/ON
Rename column	RENAME	AS
Displaying resultant table	PROC PRINT	SELECT
summarizing	PROC/BY	GROUP BY

12. Macros

EXL
GO NEXT. NOW.

Things to be covered – Day 1

12.1 Introduction

12.2 Map of Macro facility

- 12.2.1 Overview of the map

12.3 Macro Variables

- 12.3.1 User-defined Macro Variables
- 12.3.2 Displaying Macro Variable Values
- 12.3.3 Referencing macros
- 12.3.4 Macro Quoting
- 12.3.5 Macro Indirect Referencing
- 12.3.6 Automatic Macro Variables
- 12.3.7 Variables and Scope
- 12.3.8 Rules for Creating and Dereferencing of Variables
- 12.3.9 Enhancing Macro Programming

Things to be covered – Day 2

12.4 Understanding Macro Parameters

- 12.4.1 Macro Parameters
- 12.4.2 Invoking Macros

12.5 Macro Programming

- 12.5.1 %if…%then…%else and %do…%end
- 12.5.2 SAS Macro-Function Utilities
- 12.5.3 Macro Statement Nesting

12.6 Data Step Interface

- 12.6.1 Call Symput
- 12.6.2 Symget Functionality
- 12.6.3 Call Execute

12.7 Error Checking

12.8 Combing SQL and Macro

12.9 Storing Sas Macros

12.10 Compiling macros

12.1 Introduction

- MACRO facility is provided as a part of BASE SAS. It consists of a Macro Language having its own Statements, Syntax, Functions and Macro Variables.
- Macro language enables one to create generalized and flexible code by
  - Create and resolve *macro variables* anywhere in a SAS program
  - Allows nesting and iterating.
  - Implement the notion of a ‘function’ or ‘method’ as in a conventional programming language.
  - Pass information between SAS steps
  - Dynamically create code at execution time
  - Conditionally execute DATA and PROC steps

Example

From the collections dataset example, we want to find out how does the mean amount due change if we exclude the highest amount in first case, second highest amount in the second case and vice versa.

```
PROC SORT DATA=library.collections OUT=collections1;
BY past_due_amt;
RUN;
```

```
%MACRO compute;
  %DO i=1 %to 10;
                DATA temp;
                    set collections1;
                    if _n_ ne &i;
       run;
                proc means data=temp;
           var past_due_amt;
       run;
       %End;
%mend compute;
% compute;
```

Initialize Macro

Macro Statements

Macro Variable Reference

SAS Statements

End Macro

Execute Macro

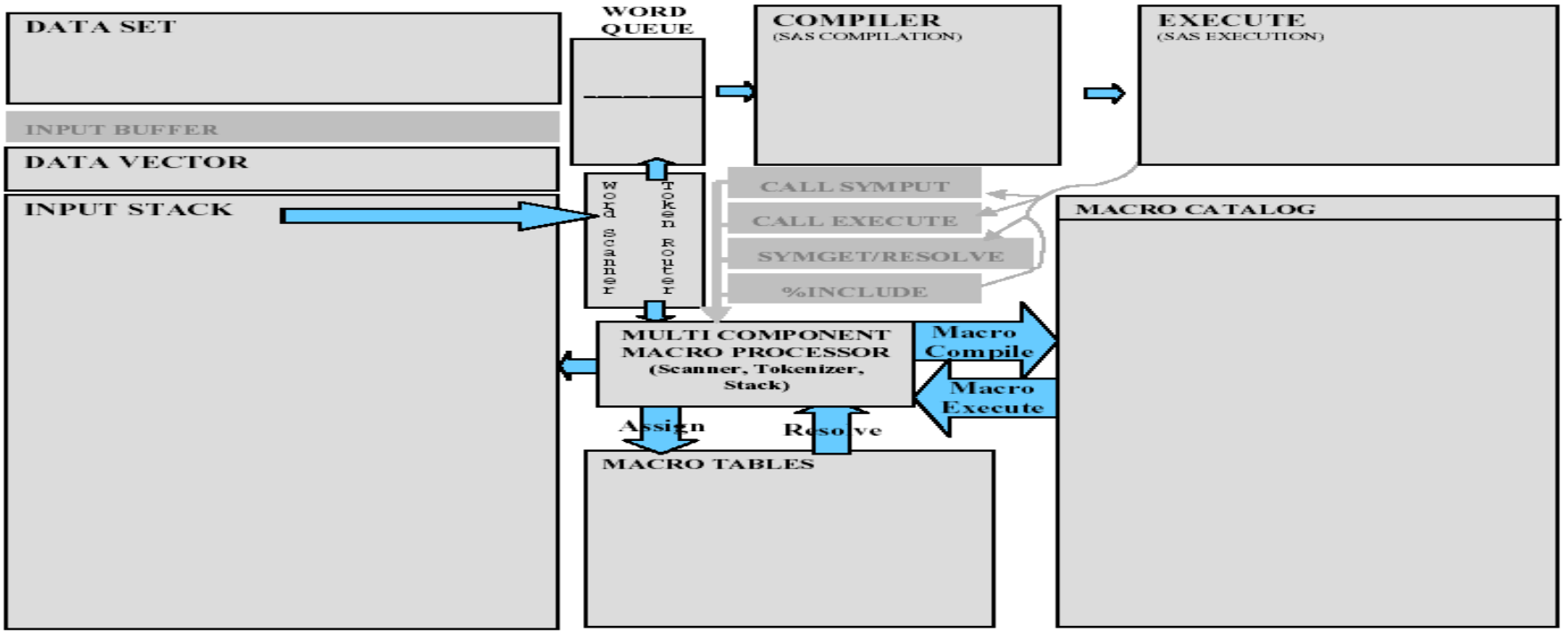
Points to remember:

1. % is put against all those commands which do not come inside a data or a proc statement (in this case, DO, to and End)(if instead of 10, we wanted a generalized n, we will have to write i=1 %to &n)

12.2 Macro Architecture

Macro architecture presents overview of Macro Processing and associated program and data flows among various components of SAS system

Diagram



Compilation/Execution flow Details

- **Data Source,** the first component of the system, can be either a SAS Data Set or Input Text. If the data source is a text file, the data will flow from the text file into the Input Buffer and then to the Program Data Vector (PDV). The input buffer holds one line of unparsed data from an input text file in preparation for passing it to the PDV.
- **Program Data Vector (PDV)** can be thought of as a one-row spreadsheet where data is read into from the input Buffer or SAS data set. Data Step processing is performed in the PDV and, when a data step has completed, the values in the PDV are written to the output file.
- **Input Stack** is a holding area which holds the code for pre-processing before it goes to the compiler.

Continued …

- **Word scanner** performs two functions:
  1. It takes characters off the Input Stack and assembles the characters into tokens (groups of characters that the compiler can process).
  2. It also decides if the token should go to the word Queue or to the Macro Processor.
- **Word Queue** holds six tokens and allows the SAS supervisor to access "previously assembled" tokens to build context for the token currently being assembled in the word scanner.
- **SAS compiler** requests tokens from the word scanner until it is passed a token that indicates a step boundary (e.g. run, quit, proc). When the SAS compiler receives a step boundary token, it takes total control of the system and attempts to compile code. If the code is correct it is compiled and passed to SAS execute module.
- **SAS Execute** module runs the job after compilation, if the job has no run errors (e.g. data mismatch etc)

Points to Remember:

During the processing of a SAS program

- Macro syntax is processed separately from regular SAS syntax
  - Macro triggers are % and & followed by non blank character
- The word scanner intercepts macro syntax before it reaches the SAS compiler
- The macro processor manipulates the character strings it receives to construct SAS statements, parts of SAS statements, constants or text. It then passes them to the SAS System to be compiled and executed
- Moving text back and forth between the storage areas and the Input Stack is the function of the Macro Processor. By using statements like %if you can control whether a certain section of code gets recalled and moved to the Input Stack. Using %do allows you to control how many times a section of code is recalled from macro storage areas and moved to the Input Stack.

- The Macro catalog is where SAS stores macro definitions i.e. any code between %macro and %mend.
- The Macro Table is used to store strings that you want to recall later in your code e.g. the %let statement.

12.3 Macro variables

12.3.1 User-defined Macro Variables

Method 1. –
The %LET statement enables one to define a macro variable and assign it a value. Their values can be displayed in the SAS log by specifying the _USER_ argument in the %PUT statement. It only gives the global variables.

Rules for the % LET Statement

- *Variable* can be any name following the SAS naming convention
- If *variable* already exists in the symbol table, *value* replaces the current value
- If either *variable* or *value* contains a macro trigger, the trigger is evaluated before the assignment is made.
- *Value* can be any string
  - Maximum length is 64K characters, minimum length is 0 characters
  - Numeric tokens are stored as character strings
  - Mathematical expressions are not evaluated
  - The case of *value* is preserved

Examples: User-defined Macro Variables

Determine the values assigned to macro variables by the following %LET statements

%put _global_ ;

%let name = Ed Norton;

%let name2= ‘ Ed Norton ‘;

%let title=“Joan’s Report”;

%put _user_;

%let start=;

%let total=0;

%let sum=3+4;

%let total=&total+∑

%let x=varlist;

%let x=name age height;

%put _user_ ;

%put _global_ ;

First ‘PUT’ Statement:

Ed Norton

‘ Ed Norton ‘

“Joan’s Report”

Second ‘PUT’ Statement:

0

3+4

0+3+4

varlist

name age height

Quotes are also assigned along with the character string

Exercise: Note the difference between the output of first and second “%put _global_” statement.

12.3.2 Displaying Macro Variable Values

To display your own messages to the SAS log, use the %PUT statement.

General form of the PUT statement:

%PUT text;

e.g. %put The value of the macro variable AMOUNT is: &amount; generates

Output: The value of the macro variable AMOUNT is: 975

Points to remember:

The %PUT statement

- ***writes to SAS log only***
- ***always writes to a new log line starting in column one***
- ***writes a blank line if text is not specified***
- ***does not require quotes around text***
- ***resolves macro triggers in text before text is written***
- ***can be used inside or outside a macro definition***

12.3.3. Referencing macros

- **text**&*variable*
- *&variable***text**
- **text***variable***text**
- *&variable&variable*
- The word scanner recognizes the end of a macro variable name when it encounters a character that cannot be part of the name token.

A *period* (.) is a special character that is treated as part of the macro variable reference and does not appear when the macro variable is resolved.

%let var1 = perm;

Data=&var1.dataset; will be resolved as data=permdataset;

Data=&var1..dataset; will be resolved as data=perm.dataset;

The first period is treated as a macro variable name delimiter. The second period is simply text.

```
%let dsn1=year1991;
%let dsn2=year1992;
%let dsn3=year1993;
%macro test1;
        %do I=1 %to  3;
        %put &dsn&I;
        %put year199&I;
%end;
%mend;
%test1;
```

Note: dsn will not be resolved because you cannot construct a macro variable while referencing

```
%macro test2;
        %do I=1 %to 3;
        %let dsn&I=year199&I;
%end;
%put &dsn1 &dsn2 &dsn3;
%mend test2;
%test2;
```

12.3.4 Macro Quoting

- Suppose a macro variable needs to contain a special character such as ; . Then this must be protected using the %str function
  %str masks the following special tokens + - * / , < > = ;
  %nrstr also masks macro triggers & % apart from those masked by %str function
- To include quote tokens that normally occur in pairs, such as ‘ “”) (precede them by a % sign while using %str
- To mask commas, use %Qsysfunc

Example

```
%let aa = z = x + y ; ;
%put &aa;

%let aa1 = z = x + y %str(;);
%put &aa1;
%let comp = AT&T;
%put &comp;
%let comp1 = AT%nrstr(&)T;
%put &comp1;
```

```
%macro test1;
Resolved
%mend test1;

%put "%test1";
%put '%test1';
```

Multiple ways of using %str

e.g. %let prog=data new; x=1; run;

1) %let prog = %str(data new; x=1; run;);
2) %let prog=data new%str(;) x=1%str(;)run%str(;)
3) %let s=%str(;);
e.g. %let prog=data new&s x=1&s run&s;

e.g. To assign text the value - Joan’s Report
1) %let text=%str(Joan%’s Report);
2) %let text=Joan%str(%’)s Report;

e.g. To assign June 12, 2000
1) %let text = %sysfunc(left(%qsysfunc(today(),worddate.)));

Macro triggers (& and %) are resolved when enclosed by double or no (“s) but are not when enclosed by single quotes (‘)

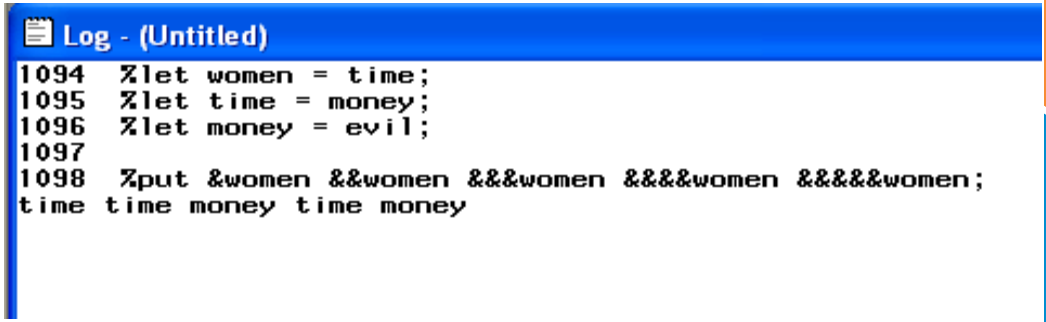
12.3.5 Macro Indirect Referencing

- In SAS macro-variable names can themselves be assigned to other macro-variables, creating the effect of 'indirecting'
- SAS allows a limited level of indirection and the rules are not consistent
- To dereference such variables, an appropriate number of &s should be prefixed to the name. The rules are
  - Two &s (&&) resolve to one &
  - Four or more &s are resolved from left to right
- It is advisable not to use indirection beyond the 2nd level
- Study the following example

Example

```
%let women = time;
%let time = money;
%let money = evil;
%put &women &&women &&&women &&&&women
&&&&&women;
```

Output[1]



Exercise 1

Study the macro assignments given along side. Predict what the following would resolve to

%let VarHeader = Bal_; %let VarTail = 11; %let Bal_11 = 11000;

&VarHeader&VarTail	&VarHeader.&VarTail	&&VarHeader&VarTail	&&VarHeader.&VarTail
&&&VarHeader&VarTail	&&&VarHeader.&VarTail	&&&VarHeader&&VarTail	&&&VarHeader.&&VarTail

Verfiy your answers using the %put statement

[1]*This proves that women might be time, or money but they are not evil!* ☺

12.3.6 Automatic Macro Variables

System-defined macro variables are global, created at SAS invocation, and can be assigned values by the user in some cases. Their values can be displayed in the SAS log by specifying the _AUTOMATIC_ argument in the %PUT statement

Variables with static values that are set at SAS invocation

Name	Value	Application
SYSDATE	Date of SAS invocation (DATE7.)	Check the current date to execute programs on certain days of the month
SYSDATE9	Date of SAS invocation (DATE9.)	
SYSDAY	Day of the week of SAS invocation	Check the value to run a given job on a certain day of the week
SYSTIME	Time of SAS invocation	
SYSDAYSYSENV	FORE (interactive execution), BACK (batch execution)	Check the execution mode before submitting code that requires interactive processing
SYSSCPSYSTIME	Abbreviation for the operating system used such as OpenVMS, WIN, HP 300	
SYSVER	Release of SAS software being used	Check for the release of SAS software being used before executing a job with newer features
SYSJOBID	Identifier of current SAS session or batch job. Mainframe systems: the userid or job name Other systems: the process ID (PID)	Check who is currently executing the job to restrict certain processing or issue commands specific to a user.

Variables with dynamic values based on submitted statements

Name	Value
SYSLAST	Name of the most recently created SAS dataset in the form of *libref.name*
SYSPARM	Text specified at program invocation

12.3.7 Variables and Scope

Local and Global Variables

Broadly there are two types of variables 'Global' and 'Local' . Local variables are those that have been created inside a macro

- Global variables are visible throughout the program session
- Local variables are visible only in their scopes on the scopes nested within
- Local symbol table
  - A local symbol table is not created until a request is made to create local variables
    - A Local table is initialized if a macro has any parameters
    - A **call symput**[1] can create local variables only if the local table already exists
- A macro variable can be forced to be created as local or global explicitly by the % local and % global statement respectively. Their syntaxes are

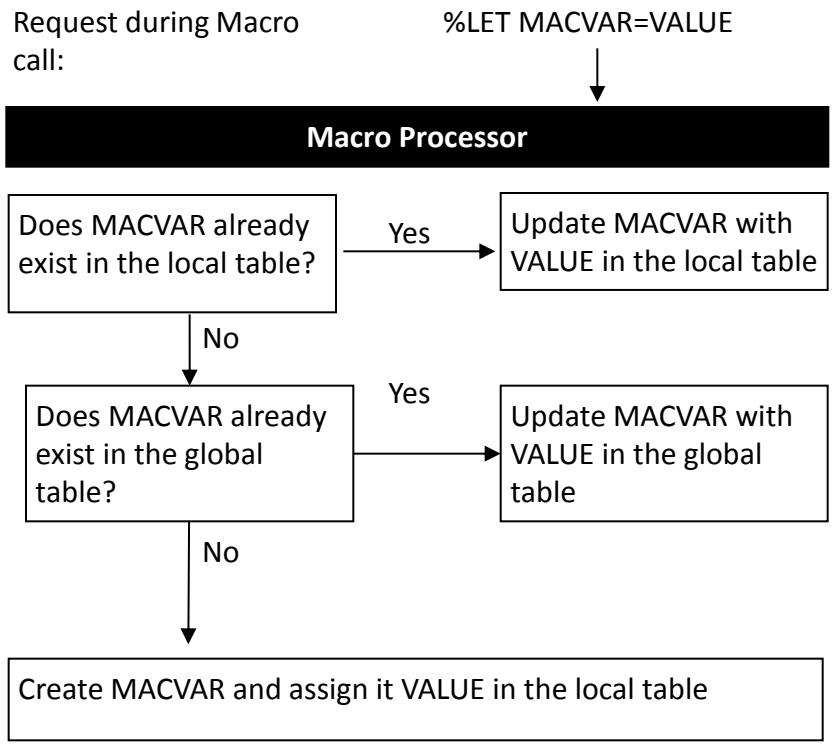
```
% local <varlist>;
```

```
% global <varlist>;
```

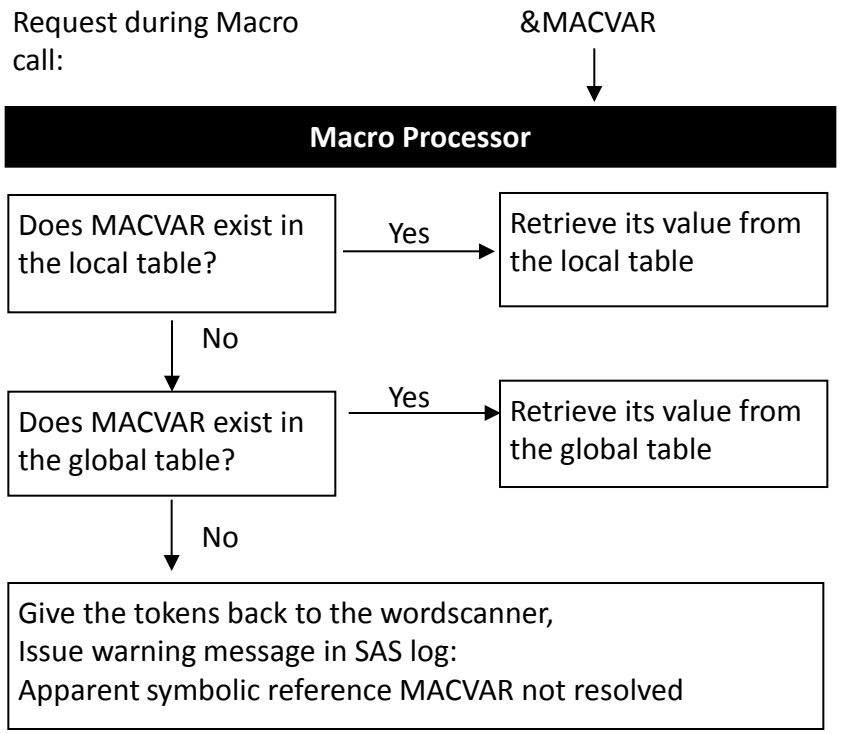
[1] Discussed in detail further

12.3.8 Rules for Creating and Dereferencing of Variables

When the macro processor receives a request to create or update a macro variable during macro execution, the macro processor follows these rules



To resolve a macro variable reference during macro execution, the macro processor follows these rules:



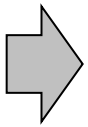
Illustration

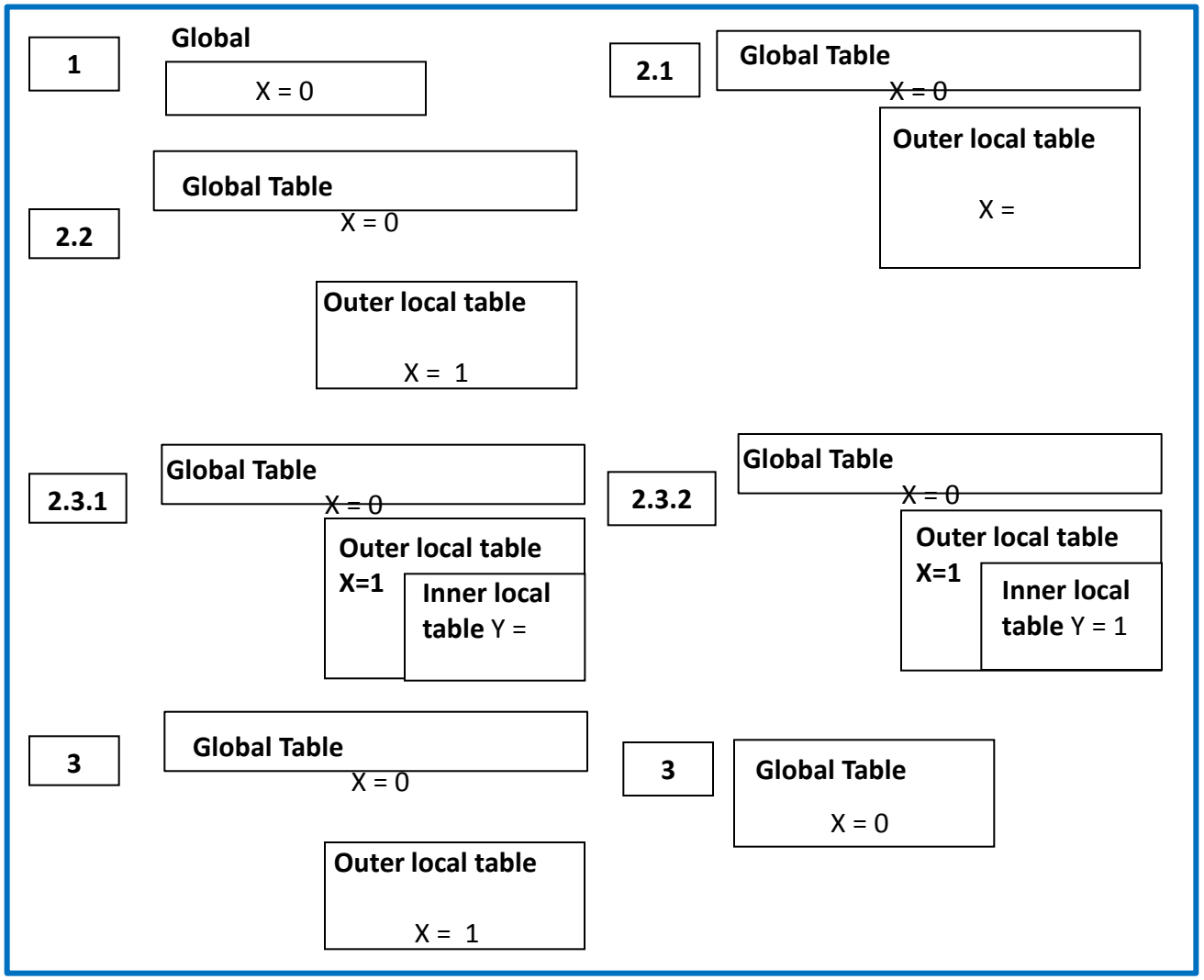
The following illustrates a program and the global and local symbol tables at the various stages of execution.

```
        % macro outer;
2.1     % local x;
2.2     % let x = 1;
2.3     % inner
        % mend outer;

        %macro inner;
2.3.1       %local y;
2.3.2       %let y = &x;
        %mend inner;

1       % let x = 0;
2       % outer;
3
```





The outer local table variables are accessible in the inner macro but not vice versa

12.3.9 Enhancing Macro Programming

options mprint; options nomprint;

- To have the macro displayed after the variables have been filled in set mprint or to turn off the printing use nomprint.

options mlogic; options nomlogic;

- If you have logical checks on macro variables you can turn on mlogic and you will get true or false values for every logical check.

options symbolgen; options nosymbolgen;

- If you want to see what values a macro variable “resolve to” during the running of a macro you can use symbolgen.

options mfile mprint;
filename mprint "File Path and Name";

- You can also ask SAS to saved the "resolved" version of a macro program in a text file by including these lines before your macros:
- This is particularly useful if you inherent complex macro code.

Options errorabend;

- Stop Execution on Error

Options merror;

- Controls whether SAS issues a warning message when a macro-like name (of the form %name) does not match a macro keyword.

- Write and debug the desired SAS program without any macro coding
- Make sure the SAS program runs with hard-coded programming constants on a fixed set of data
- Generalize the program by removing hard-coded programming constants and substituting macro variable references
- Initialize the macro variables with %LET statements and try different values for the macro variables
- Use the SYMBOLGEN system option to assist in debugging or confirmation.

Exercise

The following is a macro to draw a right – angled triangle with a height and base of 10 each.

What is the mistake? Run and see if you can spot the same. Correct the same

```
options nomprint nosymbolgen nomlogic;
%macro nstar(n);
%do ii = 1 %to &n;
*
%end;
%mend nstar;


%macro triang;
%do ii = 1 %to 10;
%put %nstar(&ii);
%end;
%mend triang;
%triang;
```

```
options nomprint nosymbolgen nomlogic;
%macro nstar(n);
%local ii;
%do ii = 1 %to &n;
*
%end;
%mend nstar;
%macro triang;
%do ii = 1 %to 10;
%put %nstar(&ii);
%end;
%mend triang;
%triang;
```

Inside a macro, * cannot comment a single line. Use a %* to comment everything between it and the next ;

Example

Observe the following dataset contract

lj_sub_id	on_contract	cntrc_lgth_qty	mnths_left_contract
342001	1	24	18
4251021	1	5	1
593001	0	12	-52
8741021	1	24	19
105103	1	24	15
12081021	1	24	22
14650021	1	24	13
16010021	1	24	15

Q. You have to create a variable called Days ft for contract which is defined as the product of months left for contract * 30. Write a program to compute No. of days using macros.

Program

```
%let  var1 =mnths;
%let var2 =days;
%let var3 = left_contract;

data macros_exmpl;
Set  library.contracts;
&var2_&var3 = &var1_&var3 * 30;
run;
```

12.4 Understanding MACRO Parameters

12.4.1 Macro Parameters

- A macro can contain two types of parameters
  - Positional parameters (parameter values are separated by commas, and can be null values, text, macro calls, or macro variable references. These are substituted for the parameter variable using a one-to-one correspondence)
  - Keyword parameters (parameter values are followed by an = sign, separated by commas, can be specified in any order, and can be omitted from the call – takes on a default value)
  - Mixed Parameters (all positional parameters must be listed before any keyword parameter variables)

Positional Parameter Macro

Definition

```
%macro postnmacro(par1,par2,...,parn);
        Code …
%mend postnmacro;
```

Invoking Call

```
%postnmacro(arg1,arg2...,argn)
```

Keyword Parameter Macro

Definition

```
%macro kwdmacro(par1=,par2=,...,parn=);
        Code ..
%mend kwdmacro;
```

Invoking Call

```
%kwdmacro(par1=arg1,par2=arg2...,parn=argn)
```

Mixed Parameters macro

Definition

```
%marco postnmacro(Ppar1,Ppar2,...,Pparm, Kpar1=, Kpar2=,…,Kparn=);
        Code …
%mend postnmacro;
```

Invoking Call

```
%postnmacro(Parg1,Parg2...,Pargm,Kpar1=Karg1,…,
Kparn=Kargn)
```

Rules for Macros with Parameters

- Positional parameters as substituted for the parameter-variables using a one-to-one correspondence
- During a call values must be provided for all the positional parameters and in the same order
- Values can be null values, text, macro-variables, or macro call
- Keyword parameters in a macro definition must be followed by an = sign
- They must be assigned a default value. Null value is counted as a valid assignment
- During a call, the keyword parameter name has to be explicitly assigned a value as indicated in the syntax
- The keyword parameters can be omitted from the call. In this case the parameter takes the default value. If the parameter is specified in the call, then the value assigned in the call is used
- For macros with mixed parameter-lists, the positional parameters must be listed before keyword parameters both in the definition as well as the invoking cal
- If a macro with keyword parameter list needs to be invoked with its default values, then an empty bracket must be specified in the call

Exercise:

Write a program to randomize the order of observations using a specified random seed. Use system time as a default random seed

```
%macro ransort(inp=, outp=, seed =  %sysfunc(int(%sysfunc(time()))));
data &outp;
set &inp;
ran_key = ranuni(&seed);
run;
proc sort data=&outp; by ran_key; run;
%mend ransort;

%ransort(inp=macroex.drop1, outp = macroex.exv);
%ransort(inp=macroex.drop1, outp = macroex.exv2, seed = 1000);
```

Exercise

- **Write a macro that produces a sorted copy of a dataset by a key. The dataset should be sorted only when the key is specified i.e is non-empty**

```
%macro sortcopy(inp=, outp=, key=);

%if (&key ~= ) %then %do;
proc sort data = &inp out =&outp; by &key; run;
%end;

%else %do;
data &outp;
set &inp;
run;
%end;
%mend sortcopy;

%sortcopy(inp=macroex.drop1, outp = macroex.exv2);
%sortcopy(inp=macroex.drop1, outp = macroex.exv2, key = Account);
```

12.4.2 Invoking Macros

Macros can be invoked by a simple code. It is very easy to use macros written by others.

- The % INCLUDE statement retrieves SAS source code (including macros) from an external file

% INCLUDE "*file-specification*" </SOURCE2>;

file-specification describes the location and name of the SAS code to be inserted

SOURCE2 causes the inserted SAS statements to appear in the SAS log

- The following are two formats in which a macro program can be defined and the respective ways in which they can be invoked

Positional parameter macro definition

```
%macro postnmacro (par1, par2);
Code ….
%mend postnmacro;
```

Keyword parameter macro definition

```
%macro kwdmacro (par1=, par2=);
Code…
%mend kwdmacro;
```

Invoking a positional parameter macro

```
%Postnmacro (arg1, arg2);
```

Invoking a keyword parameter macro

```
%kwdmacro(par2=,par1=);
```

Exercise

Copy the rename macro and invoke it in the different ways as indicated. Observe the log file and the output dataset.

Macro

```
%macro rename(varlist=,add=,option=);

%local varcnt this_var;

 %let this_var=%scan(&varlist,1,%str(" "));
 %let varcnt=1;

 %do %while (&this_var ne);
    %if (%upcase("&option") eq "P") %then %do ;
              rename &this_var=&add._&this_var;
    %end ;
    %if (%upcase("&option") eq "S") %then %do ;
              rename &this_var=&this_var._&add;
    %end;
  %let varcnt=%eval(&varcnt+1);
  %let this_var=%scan(&varlist,&varcnt,%str(" "));
 %end;

%mend rename;
```

Program to Invoke the Macro

```
options mprint mlogic symbolgen;

data rename2;
set exam.summary_example;
%rename (
              varlist=balance account,
              add=new,
              option=S
              )
run;

proc print data = slabs2;
var Account_new Balance_new slab_balance;
run;
```

12.5 Macro Programming

12.5.1 %if…%then…%else and %do…%end

Macros are used to reduce the code-writing effort. Two features that allow this are conditional processing and iterative or repetitive processing features. These are provided by the **%if...%then...%else** construct and the **%do...%end** construct respectively

- The %if… %then and %do…%end constructs are available only within a macro-scope. Unlike the %let statement, these constructs cannot be used outside a macro i.e. in the open code
- The %if... %then...%else and %do...%end constructs may be nested

Conditional execution

%if (macro condition) % then <statement>;
%else <statement>;

%if (macro conditions) % then %do; <statement list>; % end;
%else %do;
<statement-list>
%end;

Loop *(Counter Version)*

%do <macro variable> = <start-value> %to <end-value>
%by <step>;
<statement-list>
% end;

Loop *(Condition Version)*

%do %while (macro condition);
<statement-list>
% end;

- The conditions in the %if or %do % while statements may involve only macro parameters and variables. The counter variable in the %do loop also must be macro variables
- The statement list may contain either a macro-statement or a non-macro statement
- There is no macro-counterpart of the do <var> in <value-list> ...end construct
- All macro comparisons are case sensitive

12.5.2 SAS Macro-Function Utilities

- **%eval** is used to evaluate numerical expressions involving macro variables
- **%sysevalf** is used to evaluate numerical expressions with decimal values involving macro variables

Example

```
%let y = 5;
%let z = 5.2;
%let y1 = &y + 5;
%let y2 = %eval(&y + 5);
%let z1 = %eval (&z + 5);
%let z2 = %sysevalf (&z + 5);
%put &y;     %put &y1;    %put &y2;
%put &z;     %put &z1;    %put &z2;
```

- **%upcase, %substr, %scan, %length, %index** etc work the same way as their non macro counterparts

Example

```
%let exmpl_strng = 5 apples : 6 bananas : 8 mangoes;

%put &exmpl_strng;          %put %upcase(&exmpl_strng);          %put %substr(&exmpl_strng, 5, 16);

%put %scan(&exmpl_strng, 2);          %put %scan(&exmpl_strng, 1, :);          %put %length(&exmpl_strng);
```

%Sysfunc

- Some other non-macro functions can be used in a macro context using the %sysfunc utility
- Enables you to use standard SAS functions not normally available to SAS Macros.

Example

```
title "%sysfunc(date(),worddate.)";
title "July 24, 2002";
%let date=%sysfunc(today());
title
 "%sysfunc(intnx(MONTH,&date,0),date9.)";
title "01JUL2002";
%macro printds(dset);
  %if %sysfunc(exist(&dset))=1 %then %do;
    proc print data=&dset;
    run;
  %end;
  %else
    %put The data set &dset does not
         exist.;
%mend printds;
```

Example

```
%put %sysfunc(today(), worddate.);

%let strng = A      long            word;

%put A %substr(&strng, 2, 16) word;
%put A %sysfunc(trim(%substr(&strng, 2, 16))) word;
```

Exercise

Before beginning these exercises turn on the options mprint, symbolgen and mlogic in your session using the statement

```
options mprint symbolgen mlogic;
```

Write a macro that compares two numbers by their absolute values, and returns the number with the smaller absolute value

```
%macro smallabs(n1, n2);

%if (%sysfunc(abs(&n2)) < %sysfunc(abs(&n1)))
%then &n2; %else &n1;

%mend smallabs;


%put %smallabs(9,-8.9);
```

There are 10 datasets by the name set0 to set9. Each of them needs to be sorted by the variable Acct. Write a macro to do this

```
%macro sortall;

%do ii = 0 %to 9;

proc sort data = macroex.set&ii out =
macroex.srtd_set&ii; by Acct;

run;

%end;

%mend sortall;


%sortall
```

Exercise

1) Write a macro to count the number of tokens in a space delimited list

2) Write a macro that takes a list of variables as an input and generates code for renaming these variables by appending '_old' to the original name

3) Modify the above macro so that the macro generates code to drop those variables in the list which contains 'old' in their names

1

```
%macro cnttknsinlst (list);
%local cnt;
%let cnt=0;
%do %while (%scan(&list, %eval(&cnt+1),%str(" ")) ~= );
%let cnt = %eval(&cnt+1);
%end;
&cnt;
%mend cnttknsinlist;
%put "There are %cnttknsinlst(Apples Bananas Grapes Oranges Sweetlimes Coconut watermelons) fruits in the list
'Apples Bananas Grapes Oranges Sweetlimes Coconut watermelons' ";
```

Exercise

2

```
%macro renold(varlist);
%local cnt this_var;
%let this_var = %scan (&varlist,1,%str(" "));
%let cnt = 1;
%do %while (&this_var ~= );
   rename &this_var = &this_var._old;
%let cnt = %eval(&cnt+1);
%let this_var = %scan (&varlist,&cnt,%str(" "));
%end;
%mend renold;

data macroex.ren1;
set macroex.Mcex1;
%renold (spend balance);
run;
```

3

```
%macro dropold(varlist);
%local cnt this_var;
%let this_var = %scan (&varlist,1,%str(" "));
%let cnt = 1;
%do %while (&this_var ~= );
 %if (%index(&this_var,old) > 0) %then drop
&this_var; ;
%let cnt = %eval(&cnt+1);
%let this_var = %scan (&varlist,&cnt,%str(" "));
%end;
%mend dropold;

data macroex.drop1;
set macroex.ren1;
%dropold (spend_old balance_old revolve);
run;
```

12.5.3 Macro Statement Nesting

- Macros can be nested within themselves. i.e. A macro can be defined within another macro
- Data steps and proc steps can be nested within a macro
- The **%if... %then** constructs and **%do {%while}...%end** constructs cannot occur in open code. Thus in general a macro cannot be nested within a data or a proc step unless the step itself is nested within a macro

Example

Write a macro to sort 10 different data sets named set-0 to set-9 (This has a data step within a macro)

```
%macro sortall;
%do ii = 0 %to 9;
        proc sort data = macroex.set&ii out = macroex.srtd_set&ii;
        by Acct;
        run;
%end;
%mend sortall;

%sortall
```

Exercise

Try the following program and observe the log. The program tries to find if there can be a triangle with specified lengths

```
%macro iftriang(l1,l2,l3);
        %macro max3(a1,a2,a3);
                %sysfunc(max(&a1,&a2,&a3))
        %mend max3;
        %if (&l1 + &l2 + &l3 > 2* %max3(&l1,&l2,&l3)) %then
                %put There exists a traingle with sides
                whose lengths are &l1, &l2, &l3. ;
        %else
                %put There cannot be a triangle with lengths
                &l1, &l2, &l3.;
%mend iftriang;


%iftriang(3,4,6);
%put The maximum of the nos 8,7,9 is %max3(8,7,9);
```

Exercise

Observe the following program. The intent is to create 6 indicator variables ipresent1 to ipresent6 which are set to 1 or 0 depending on whether the variables bal1 to bal6 are non-missing or missing.

Can you tell why this wouldn't work? 1) What modifications can be made to make this work? 2) Suggest a method to achieve the same without using macros

```
data macroex.missingbalinds;
set macroex.misssingbal;
%do i = 1 %to 6;
        if (bal&i=.) then ipresent&i = 0 ;else ipresent&i=1;
%end;
run;
```

Solution 1

```
%macro encap;
data macroex.missingbalinds;
set macroex.missingbal;
%do i = 1 %to 6;
        if (bal&i=.) then ipresent&i = 0 ;else ipresent&i=1;
%end;
run;
%mend encap;
%encap
```

Solution 2 – Use arrays

Exercise

Write a program to merge an unknown number of datasets by given common keys. Assume the datasets would be given sorted by the keys and would be named in an array like fashion. Do not forget to create the merge indicator

```
%macro mrg(data_array=, num =, key=, outp=);
data &outp;
merge %do ii = 1 %to &num; &data_array&ii (in = in_&ii) %end;;
by &key;
length merge_ind $ &num;
%do ii = 1 %to &num;
merge_ind = compress(merge_ind!!in_&ii);
%end;
run;
%mend mrg;


%mrg(data_array=macroex.Srtd_set, num=5, key=Acct, outp=macroex.mrgd_sets);
```

Exercise

Convert JoiningDate = “August 2 2004” to a SAS date?

12.6 Data Step Interface

12.6.1 Call Symput Functionality

- The macro-statements are precompiled or pre-processed. Thus all the macro-variables and statements are resolved prior to actual execution
- How does one create macro-variables whose value depends on a dataset contents? How would such variables be created, assigned values and resolved?
  - It is possible to create macro-variables that depend on dataset contents
  - The call symput routine is provided by SAS to create macro variables that are assigned during a data step execution
  - An executable SAS program consists distinct program elements or steps. Each of these steps is either a **data step** or a **proc step.**
  - The steps are executed in a sequential fashion or on a step by step basis
  - SAS first resolves or dereferences the macro references associated with a program step. It then executes that step and proceeds to dereference of resolve macro references associated with the next program step
  - Using call symput, one can create a macro variable or assign value to an existing macro variable during a 'data' program step
  - These variables can then be resolved with their values so assigned in the subsequent 'program steps'
  - It is however erroneous to reference this variable before the creating data step is completed.

Call Symput *(expression1, expression2);*

Expression1 evaluates to a character value that is a valid macro variable name. This value should change each time you want to create another macro variable. This can also be dataset dependent and allows creation of multiple macro variables in a single data step

Expression2 is the value to be assigned to the specified macro variable

Example

Given below is a program which attempts to set the macro-variable Revolver_found if the account 10002 in the dataset McEx1 has more then 75% of its balance as revolve. Does it work? Why?

```
data _null_;
set Macroex.Mcex1 (where = (Account=10002));
if (Revolve > 0.75*Balance) then
do;
        %let iRevolver = 1;
end;
else
do;
        %let iRevolver = 0;
end;
run;
%put &iRevolver;
```

Wrong: Because before execution compiler assigns iRevolver value 1 (%Let is a compiler statement which is not dependent on the execution of data step) then rewrites that value to 0 and will always output 0

Macro variables can be used even outside a data step, a data set variable cannot be used without opening that data set.

Example

Now run the program shown below and observe the output. Reason how it works, especially the creation and assignment of macro variables

```
data _null_;
set Macroex.Mcex1 (where = (Account=10002));
if (Revolve > 0.75*Balance) then
do;
        call symput ("iRevolver",1);
end;
else
do;
        call symput ("iRevolver",0);
end;
run;
%put &iRevolver;
```

Call symput also creates a macro variable but its value is not assigned before the execution of the code.

That macro variable cannot be used in the same data step.

Exercise

Write a macro that prints a dataset only if it has less than 50 observations

Solution 1

```
%macro condprint(inp);
data _null_;
set &inp nobs = tot_obs;
%let obs=tot_obs;
run;
%if (&obs<50) %then %do;
proc print data = &inp;
run;
%end;
%mend condprint;
```

%***condprint*** (Macroex.McEx1);

Solution 2

```
%macro condprint2(inp);
data _null_;
set &inp nobs = tot_obs;
if (_N_=1) then call symput('obs',tot_obs);
run;
%if (&obs<50) %then %do;
proc print data = &inp;
run;
%end;
%mend condprint2;
```

%***condprint2*** (Macroex.McEx1);

Run the programs above and observe the outputs. Can you identify the problem with Solution 1?

Use call symput when macro variables will take value depending on the values inside a data set

Exercise

For each variable in the dataset example create a macro variable which is named <var>_type, which holds the type of the variable, whether it is a char or a num

```
proc contents data=Macroex.example out=valist; run;
proc format;
value typef
1 = 'Numeric'
2 = 'Char';
run;
data _null_;
set valist(keep = NAME TYPE);
call symput(trim(left(NAME))!!'_TYPE',put(TYPE,typef.));
run;
```

Exercise

The following is a program that tries to produce a space-delimited list of account numbers with balances >100. Why would it not work? Execute the program and confirm your answer. Can you modify the program to make it work?

```
data _null_;
set Macroex.McEX1;
if (Balance>100) then
call symput('Acctlist',"&Acctlist"!!' '!!trim(left(Account)));
run;
 %put &Acctlist;
```

Hint:

- Initialize Acclist
- Use Call symget

12.6.2 The Symget Functionality

Just as call symput allows us to create or assign macro variables is a data step, symget allows us to resolve a macro expression during program execution. The general-form of the symget function is

Dataset-variable=symget (“macro-variable”)

The macro variables specified in symget are resolved during the program-execution. They may be data dependent expressions and will be resolved to different variables for different observation of a data step.

Example: Using call symput and symget in conjunction to emulate a look-up table functionality

You are familiar with the card holder dataset listed below:

There is another dataset called APRs which specifies the APRs for each card type. The intent is to add a column to the original dataset called ‘APR’ based on the values specified this dataset

Card holder dataset

VIEWTABLE: Macroex.Example

	Account	Account_type	Card_type	Spend	Revolve	Balance
1	10001	old	red	200	100	300
2	10002	old	yellow	10	200	210
3	10003	new	red	100	50	150
4	10004	new	blue	0	10	10
5	10005	new	yellow	20	320	340
6	10006	old	yellow	200	250	450
7	10007	old	red	300	0	300
8	10008	new	blue	5	5	10
9	10009		red	.	.	.
10	10010	new	blue	50	0	50

APRs dataset

VIEWTABLE: Macroex.Aprs

	card_type	apr
1	blue	18
2	green	12
3	red	15
4	yellow	24

Example Solution

```
data _null_;
set Macroex.APrs;
call symput(trim(left(card_type))!!'_APR',APR);
run;

data Macroex.Acct_Aprs;
set Macroex.example;
APR = symget(trim(left(card_type))!!"_APR");
run;
```

Exercise

Solve the example problem without using macro variables

12.7 Error Checking

```
%macro update_db;
 libname trg "D_Projects\Training" access=readonly;

 %if &syslibrc ne 0 %then %goto err_lib;
 proc append base=trg.yrsales data=trg.weeklist;
 run;
 %if &syserr ne 0 %then %goto err_append;
 %goto finish;
%err_lib:
 %put Library failure;
 %goto finish;

 %err_append:
 %put Append failure;
 %goto finish;
 %finish:
%mend update_db;
Error codes are &syserr, &syslibrc, &sysfilrc, &syslckrc (0=success, +ve=error, -ve=warning)
```

Syslckrc –error when file already in use.

Example (Counting number of observations)

```
%macro obsnvars(ds);
%global dset nvars nobs;
%let dset=&ds;
%let dsid = %sysfunc(open(&dset));
%if &dsid %then %do;
   %let nobs =%sysfunc(attrn(&dsid,NOBS));
   %let nvars=%sysfunc(attrn(&dsid,NVARS));
   %let rc = %sysfunc(close(&dsid));
%end;
%else %put Open for data set &dset failed - %sysfunc(sysmsg());
%mend obsnvars;
%obsnvars(trg.filename) %put &dset has &nvars variable(s) and &nobs
      observation(s).;
```

Note: **ATTRC** (data-set-id,attr-name)returns the value of a character attribute for a SAS data set

ATTRN (data-set-id,attr-name)returns the value of a numeric attribute for the specified SAS data set

open assigns a non zero value in case it can open the data set.

Nobs can not be used anywhere inside a data step. It can be used with a set command or it can be used as shown above.

Sysmsg() stores the errors/warnings

12.8 Combing SQL and Macro

- A great feature of Proc SQL is that you can load a value or values from a SQL statement into a macro variable
- Can put a specific value into a macro variable for use throughout your program
- Coupled with the SQL functions, you can load calculated values into a macro variable

```
PROC SQL;
  SELECT MEAN(AGE)          /* Create Mean of a variable */
      INTO :MEANAGE
      FROM <datasetname>
      WHERE AGE IS NOT NULL;
  QUIT;
%put The mean age is: &meanage;
```

For generating a comma separated list of account no. with some condition :

Proc sql;

Select account into :acclist separated by “,”

from lib.dataset

where …… ; quit;

12.9 Storing SAS Macros

Porting SAS Macros

Using the **%include** statement it s possible to reuse a single macro code in different places and keep the definition and the use of macros separate.

- Example - Macro definition in an external file

```
%include 'D:\Data\Work\SAS_Training\Macros\programs\slabvar.sas';
data macroex.ex1;
set macroex.Mcex1;
%slabvar(var=balance, vals=0 100 200);
run;
```

Example - Macro definition in a source catalog

```
libname macro "D:\Data\Work\SAS_Training\Macros\programs";
filename mcrsrc catalog 'macro.std_code_lib_db2';
%include mcrsrc(edd_gen2);
%include mcrsrc(edd_num);
%include mcrsrc(edd_char);
%edd (libname= MacroEx, dsname=example,
edd_out_loc=D:\Data\Work\SAS_Training\Macros\programs\Mcex1_example, NUM_UNIQ=Y );
```

Exercise

Write a macro that creates n number of datasets based on n observations in a dataset. Ex:

Name
Sachin
Yuvraj
Sehwag
Dhoni
Kohli

5 output datasets should be created with the names above. This should work across for any number of observations

12.10 Compiling Macros

- Using the **mstore** option it is possible to compile the macro before calling it

Example - Macro compiled as a catalog file

```
libname sasmacros “C:\sasmacro”;
options nosymbolgen nomlogic mstore sasmstore=sasmacros;
%macro nstar(n) / store des='compiled macro’ secure;
%do ii = 1 %to &n;
*
%end;
%mend nstar;
```

- mstore- SAS option for compiling and storing the macro
- sasmstore- Gives location of library where the sas7bcat file is created
- des- Allows us to give a small description of the compiled macro which can be viewed
- MPRINT- Specifies whether SAS statements that are generated by macro execution are displayed.
- MFILE- Specifies whether or not MPRINT output is directed to an external file.
- MLOGIC- Controls whether SAS traces execution of the macro language processor. If MLOGIC is specified, the trace information is written to the SAS log.

13. Reporting

GO NEXT. NOW.

13.1 ODS – Output Delivery System

The Output Delivery System is a set of SAS commands that allow users to manage the output of their programs and procedures.

Using **ODS**, one can:

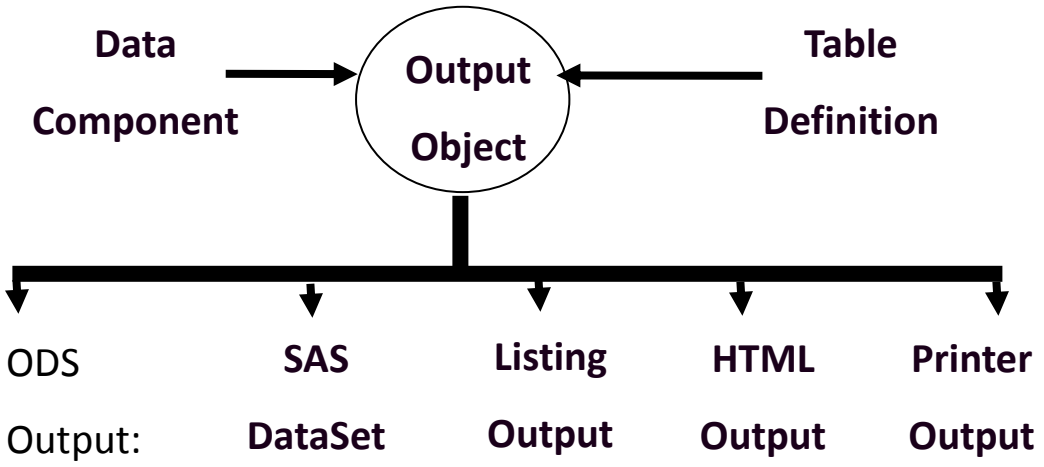
- create HTML and postscript files
- select and exclude output tables from display
- manipulate the layout and format of output tables
- create **SAS** datasets directly from output tables
- ODS can produce in the following formats HTML, PDF, RTF, CSV, JPEG, GIF, Postscript, XML, etc.

Sources of ODS include:

- SAS Procedures (including SAS/GRAPH)
- The Data Step (ODS option in the FILE statement)

…

```
ODS LISTING CLOSE;
ODS HTML BODY='OUTPUT1.HTM';
PROC PRINT …;
RUN;
ODS HTML CLOSE;
ODS LISTING;
```



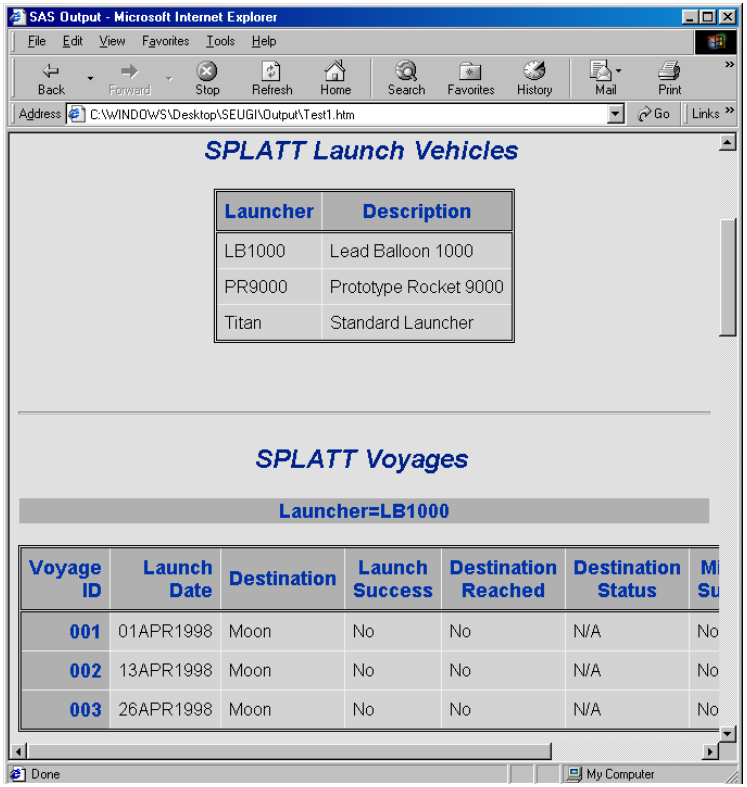
- ODS Trace On/Off

An object can be identified by its name, label, or path. To find out what objects are used by a given procedure we use trace on and off option.

- ODS Listing / Close;
- Whether or not one wants to print the output to the LST file or not

13.1.1 Creating HTML output

```
ODS SELECT BasicMeasures
      TestsForLocation  ExtremeObs;
ODS HTML body = 'listingOutput.html'
contents = 'tableOfContents.html'
frame = 'frameTiesItTogether.html'
page = 'tableOfPages.html';
proc univariate normal data=trg.fltattnd;
var x;
run;
ODS HTML close;
```



ODS Example

```
ods html body="%sysfunc(pathname(trg))\Test1.htm";

proc print data=trg.fltattnd noobs label;
  title 'Incomes';
run;

proc sort data=trg.fltattnd;
by location jobcode;
run;

proc print data=trg.fltattnd heading=h noobs label;
  id empid; (highlights empid, makes it the first column)
  by location jobcode;
  pageby jobcode;
  title 'ODS Testing-II';
run;

ods html close;
```

NOTE: Entire sample code is located on the Notes page for this slide..

ODS Example

```
libname trg 'C:\Documents and Settings\vgupta\My Documents\sas
training\SAS 2004 July\Exercises\6.4';

ods html file='Test2-body.htm'
        contents='Test2-contents.htm'
        frame='Test2-frame.htm'
        body='Test2-page.htm'
style=barreteblue;
pattern1 c=red v=solid;
pattern2 c=green v=solid;

proc gchart data=trg.fltattnd;
  title 'ODS Testing';
  vbar Salary / group=jobcode ;
run;
quit;

ods html close;
```

Note: The HTML destination can produce four different types of files: body, contents, Page (V8), and frame files. The body file contains the output, the other files are merely auxiliary HTML code.

13.1.2 Creating an RTF file

```
ods rtf file="fltattnd.rtf";
proc univariate data=trg.fltattnd;
var salary;
run;
ods rtf close;
```

This creates “prettier” output that can be read by MS Word and other word processing programs.

13.1.3 Using ODS with Procedures

To use ODS OUTPUT to create an output data set, you need to name the output object

For example:

```
ODS OUTPUT OutputObjectName=DataSetName;
ODS OUTPUT Summary=MyMeans;
```

How do you find this output object name?

```
ODS OUTPUT Moments=trg.moments;
proc univariate data=trg.fltattnd;
  var salary;
run;

ODS OUTPUT CLOSE;
```

13.1.4 Selecting output objects

- Once output object names and paths have been identified ,a subset of output objects can be selected for output. Either the **ODS** SELECT or the **ODS** EXCLUDE statement accomplishes this task. When the output object name is used, all output objects with that name are selected. When a path is used, a single unique output object is selected

```
ODS SELECT BasicMeasures ExtremeObs;
proc univariate data= trg.fltattnd;
var salary;
run;
```

13.1.5 Modify ODS Templates – Add a logo to Output

```
PROC TEMPLATE;
DEFINE STYLE DefaultLogo;
PARENT=Styles.Default;
REPLACE Body from Document
   /preimage="C:\Documents and
   Settings\vgupta\My Documents\My
   Pictures\inductis.gif";  END;
RUN;
ods html
   body="%sysfunc(pathname(trg))\Test1.htm
   " style=defaultlogo;
```

13.2 Reporting With Excel

- CSV and Tab delimited files
  - 'Do it yourself' approach
- File Export
  - Very ordinary
- ODS and File Open in MS Excel
  - Possibly can be automated using DDE
- DDE and OLE
  - Microsoft standards for data interchange between Windows applications

13.2.1 DDE

- Dynamic Data Exchange is the older standard
- SAS supports both, but OLE requires SAS/AF
- DDE is relatively easy to use, but has some limitations
- DDE can also be used to write to other applications, MS Word and IBM Lotus 123

Dynamic Data Exchange (DDE) is a method of dynamically exchanging information between Windows applications. DDE uses a client/server relationship to enable a client application to request information from a server application. In Version 8, the SAS System is always the client. In this role, the SAS System requests data from server applications, sends data to server applications, or sends commands to server applications.

You can use DDE with the DATA step, the SAS macro facility, SAS/AF applications, or any other portion of the SAS System that requests and generates data. DDE has many potential uses, one of which is to acquire data from a Windows spreadsheet or database application.

Syntax:

FILENAME *fileref* DDE '*DDE-triplet*' <DDE-options> where:

fileref

is a valid fileref

DDE

is the device-type keyword that tells the SAS System you want to use Dynamic Data Exchange.

'*DDE-triplet*'

is the name of the DDE external file.

13.2.2 Exporting to Excel

- Under Windows, you can use *dynamic data exchange* (*DDE*) to output SAS datasets directly to Excel as shown below.
- Open the Excel spreadsheet into which you want to output the SAS data. If you are creating a new spreadsheet, then open Excel and save a blank spreadsheet with your desired filename.
- Each column in Excel will receive one variable and each row will receive one record from SAS. Only variables (columns) that you wish to import from SAS should be included in your SAS syntax (see example below). Prior to running the SAS syntax, format all numeric columns in Excel as general (select "Format, Cells, Number, General"). Leave the Excel file open during the dynamic data exchange procedure.

Example

```
libname trg "D:\Exercises\6.5N";
filename data dde "Excel|[dde.xls]sheet1!r1c1:r10c2";
data _null_;
set trg.fltattnd;
file data;
put hiredate  lastname ;
run;
```

14. Efficient SAS Programming

14.1 Using Excel to Write Code

- For coding the same statement repetitively, MS Excel is a very handy tool
- Consider the following example: you have to map one variable (Product_Desc) to another (Product_Group). Product_Desc takes more than 20 values, which you have to combine into 6 groups, using If-Then statements

Product_Desc	Product_Group
1	A
2	A
3	A
4	A
5	A
6	B
7	B
8	B
9	C
10	C
11	C
12	C
13	D
14	D
15	D
16	E
17	E
18	R
19	R
20	R

Output
If Product_Desc="1" then Product_Group="A";
If Product_Desc="2" then Product_Group="A";
If Product_Desc="3" then Product_Group="A";
If Product_Desc="4" then Product_Group="A";
If Product_Desc="5" then Product_Group="A";
If Product_Desc="6" then Product_Group="B";
If Product_Desc="7" then Product_Group="B";
If Product_Desc="8" then Product_Group="B";
If Product_Desc="9" then Product_Group="C";
If Product_Desc="10" then Product_Group="C";
If Product_Desc="11" then Product_Group="C";
If Product_Desc="12" then Product_Group="C";
If Product_Desc="13" then Product_Group="D";
If Product_Desc="14" then Product_Group="D";
If Product_Desc="15" then Product_Group="D";
If Product_Desc="16" then Product_Group="E";
If Product_Desc="17" then Product_Group="E";
If Product_Desc="18" then Product_Group="R";
If Product_Desc="19" then Product_Group="R";
If Product_Desc="20" then Product_Group="R";

Note:If the INPUT function is used to create a variable not previously defined, the type and length of the variable is defined by the informat.

14.2 Disk Space Saving Measures

- Use “options compress = yes” at the beginning of every code. This compresses all datasets produced
- Use “options compress=binary” for even more compression. Then data set can only be viewed in SAS interactive.
- As far as possible, create temporary datasets. Create permanent datasets for required output
- Use the “Keep” and “Where” statements to only read data that is required

14.3 Testing

The code should be tested as much as possible

- Always batch submit code to automatically generate log and output files
- Run code on a sample of a few thousand observations and review logs and output
- Code should be reviewed with your TPM/PM
- Once you’re satisfied with your test runs, batch submit code on the complete dataset

14.4 Efficient SAS Programming Techniques

Some commonly used good coding practices are:

- Use KEEP or DROP statements to retain desired variables.
- Read only the data that is needed
- Create and use indexes with large datasets.
- Utilize macros for redundant code.
- Use IF-THEN/ELSE statements to process data.
- Use the SQL procedure to consolidate the number of steps.
- Avoid unnecessary sorting and Use CLASS statements in procedures wherever possible.
- Use data compression for large datasets

14.5 Summary

- Know the problem
- Design the solution
- Distribute and split logic and code units among files, macros and blocks
- Write comments
- Fill in the code around it
- Indent and layout well
- Get it reviewed
- Test it
- Use it