OUTLINE of FUNDAMENTALS of the DATA STEP [1]

I. SAS datasets (ds) and the DATA set-up

- A. Role and nature of SAS ds in SAS
- B. Overview of how the DATA step works
- C. Naming SAS ds
- D. Processing incoming data into SAS ds
    - 1. Four types of incoming data
        - a. "Cards"
        - b. Text files
        - c. SAS ds
        - d. Excel and Access data files
    - 2. **INPUT** statement

II. Useful statements used in the DATA step

- A. File handling statements
    - **DATA**
    - **CARDS** or **DATALINES**
    - **INFILE**
    - **INPUT**
    - **SET**
    - **FILENAME**
    - **LIBNAME**
- B. Functions
- C. Expressions
- D. Assignments
- E. Other useful statements
    - **IF** - **THEN** ; **ELSE**
    - **DELETE** , subsetting **IF**
    - **RETAIN** and sum statements
    - **FORMAT**
    - **LABEL**
    - **ATTRIB**
    - **DROP**
    - **KEEP**

==

[1] The Department of Statistics and Data Science has an excellent tutorial on this material at **https://stat.utexas.edu/images/SSC/documents/SoftwareTutorials/SAS_GettingStarted.pdf**.

I. SAS DATASETS (DS) AND THE DATA SET-UP

SAS program has two parts: DATA steps and PROCedure steps.
The DATA step creates SAS datasets.
The PROCedure step analyzes SAS datasets.

A. Role and nature of SAS dataset (ds)

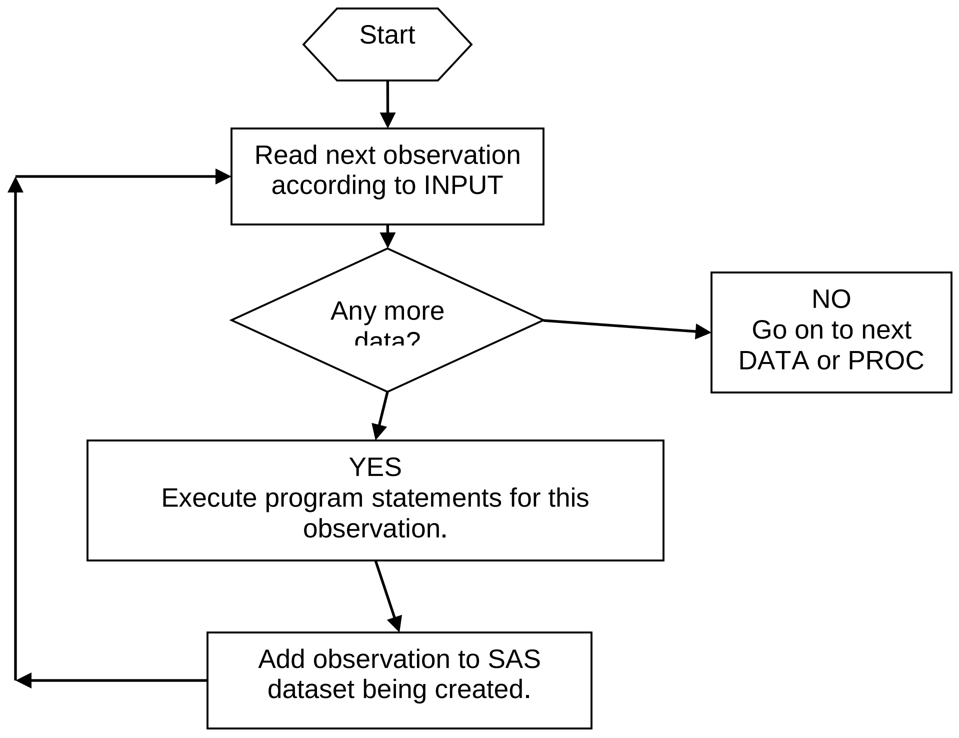
The "purpose in life" of a SAS DATA step is to create a SAS dataset.
A SAS ds is a rows-and-columns worksheet written in a special binary code.
SAS ds stores the data, variable names, lengths, types, formats, statements that created it, time of creation, etc.
Important: A set of data is not a *SAS dataset* until SAS has processed the data into the special form of a SAS ds (usually by means of a DATA step).
Normally, each DATA step creates one SAS ds.
Main importance of SAS ds: PROCs will work only on SAS ds.
So you create a SAS ds in order to change the raw data into a form that PROCs can use.

B. How the DATA step works

The "purpose in life" of a SAS DATA step is to create a SAS dataset.
The key to understanding how the DATA step works:

1. It is an implied loop.
2. It processes data one observation (row) at a time.

Simple flowchart of DATA step workings:


Start
Read next observation according to INPUT
Any more data?
NO
Go on to next DATA or PROC
YES
Execute program statements for this observation.
Add observation to SAS dataset being created.

More detail:
At Start of DATA step, SAS does bookkeeping:

- Scans entire DATA step
- Checks for syntax errors
- Sets up Program Data Vector (PDV), a 1 row x N column storage buffer for holding the current values of the N variables used in the DATA step

SAS reads the values of the variables in each new observation and stores them in the PDV.[2]
Program statements in the DATA step change the values in the PDV.
SAS writes the contents of PDV to the SAS ds being created at end of each loop of the DATA step.
So the DATA step can be viewed as a mechanism for making changes in the PDV prior to adding the PDV contents (as the next row) to the SAS dataset that is being created.

C. Naming the SAS ds
You can have more than one SAS ds (worksheet) per program.
Refer to the SAS ds by name, where needed, to select the ds you want.
A SAS ds is known to SAS as a two-name file. The first name is the *libref* (or library reference), which is a nickname for the location where the file is stored. The second name is the *file name*, which is the unique name of the file. The two names are separated by a period.
Ex: **`LIBREF.FNAME`** In SAS v9, the libref must be 8 or fewer letters and numerals, with the first being a letter; but the file name may be up to 32 characters.
A SAS ds is named in DATA statement. A SAS ds can be named by any of 3 different methods:

DATA statement	Resulting SAS ds
1. **`DATA;`**	**`WORK.DATA1`**
2. **`DATA RECORDS;`**	**`WORK.RECORDS`**
3. **`DATA EMPLOYEE.RECORDS;`**	**`EMPLOYEE.RECORDS`**

Method 1 invokes a default name for both the first and last names. The default first name is "WORK". The default last name is "DATA<n>" where <n> is a sequence number. Method 2 invokes a default name for the first name only. Method 3 uses explicit names for both first and last names.
Having a name permits you to specify explicitly the particular SAS ds to work with.
Ex: **`PROC MEANS DATA=RECORDS;`** analyzes **`WORK.RECORDS`**. You do not have to write the word WORK when using method 2. WORK is understood to be the libref, if omitted. WORK is a nickname for a temporary file storage area.
Effects of different ways of naming SAS ds:

- Any SAS ds with libref of WORK (methods 1 & 2) is erased when you terminate the SAS session (i.e., close and exit in Windows or sign out of SAS OnDemand). Storage for files in the WORK directory is only temporary.
- Any SAS ds with libref other than WORK is saved "permanently" on your disk. A libref other than WORK is a nickname for a permanent file storage area.

Method 3 in addition requires a **LIBNAME** statement to define the nickname as a file storage location; methods 1 and 2 do not – they use default temporary file storage area.

[2] For gaining an advanced understanding of SAS, it is very helpful to note that before reading each new observation, SAS (re)initializes the PDV by setting character variables blank and numeric variables missing (.)

Ex1:
```
LIBNAME EMPLOYEE 'C:\MYFOLDER';
DATA EMPLOYEE.RECORDS;
. . .
```
This LIBNAME statement defines the libref **EMPLOYEE** as a synonym of a folder on your computer where the SAS dataset can reside indefinitely. The LIBNAME tells SAS that whenever it needs to read or write a SAS ds with the libref of EMPLOYEE, it should read or write the file in C:\MYFOLDER. Thus, in this example, SAS will save **EMPLOYEE.RECORDS** as **C:\MYFOLDER\RECORDS.SAS7BDAT**, with the extension .SAS7BDAT [3] being understood in Windows 7, Vista, XP, 2000, and NT. The file known to SAS as **EMPLOYEE.RECORDS** will be known to Windows as **C:\MYFOLDER\RECORDS.SAS7BDAT**. Same file. What happened to the "EMPLOYEE" part of the name? In SAS the libref is not a permanent part of the name of a SAS dataset. The libref is only a temporary internal SAS "nickname" for the folder where the SAS ds is stored. The libref in SAS merely points to the external directory location where the SAS ds is to be read or saved. Thus the libref used in a SAS program is arbitrary (except for WORK [4]). To expand on this point a bit: suppose, for example, that the following program had been written and run instead of our example Ex1:

Ex2:
```
LIBNAME NEWREF 'C:\MYFOLDER';
DATA NEWREF.RECORDS;
. . .
```
Then the SAS ds created would also have been named **C:\MYFOLDER\RECORDS.SAS7BDAT** and the contents of the SAS ds created would have been identical to that of our example Ex1. In spite of the use of a different LIBNAME, the Windows name of the SAS ds would be the same.[5]

A **LIBNAME** is also required if a permanent SAS ds is to be read.

Ex:
```
LIBNAME EMPLOYEE 'C:\MYFOLDER';
PROC MEANS DATA=EMPLOYEE.RECORDS;
```
will read and analyze the Windows file **C:\MYFOLDER\RECORDS.SAS7BDAT** -- again, "EMPLOYEE" is arbitrary; any nickname could be used.

RULE: If your program uses a SAS ds with a libref other than WORK, you must also use a **LIBNAME**. If you do not use LIBNAME when reading or writing a permanent SAS ds, your DATA step or PROC will fail and an error message will be generated.

---

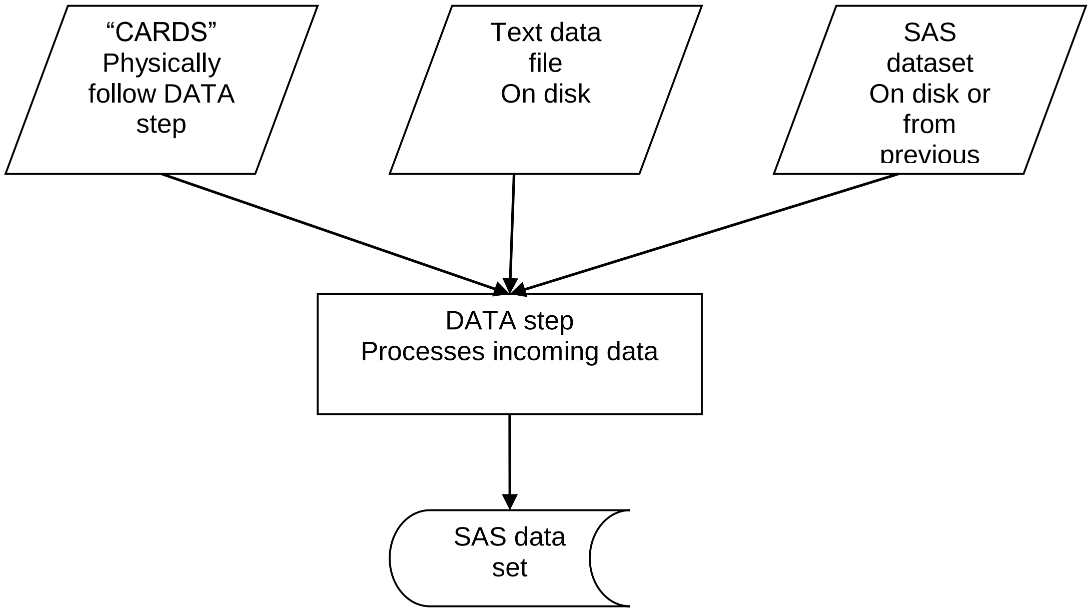
[3] In earlier versions of SAS, the extension for SAS datasets was .SD2.

[4] The libref WORK also points to a folder: the folder called SAS Temporary Files located by default in your TEMP directory (e.g., C:\Temp\SAS Temporary Files).

[5] But what if Ex1 and Ex2 were not alternatives, but were both to be run, consecutively, one after another? Then Ex1 would create **C:\MYFOLDER\RECORDS.SAS7BDAT** and Ex2 would write over it, erasing the original and replacing it. Note that you would lose the original version of this dataset.
[The following is a more advanced continuation of this note]: That is what would happen under the default SAS system option REPLACE. You can change this system option by the global statement **OPTIONS NOREPLACE;**. With NOREPLACE in effect, the result of running Ex1 followed by Ex2 would be that Ex1 would create **C:\MYFOLDER\RECORDS.SAS7BDAT**, but Ex2 would not be permitted to write over it, so the attempted creation and writeover in Ex2 would be abandoned, with a warning message placed on the LOG. The default setting can be changed. `NOREPLACE` has no effect on WORK files. Making this still more complicated is that SAS v9 permits different version numbers of the same SAS dataset (by default, this feature is disabled, but can be enabled with the GENMAX dataset option when the dataset is created). `NOREPLACE` effectively cancels new version numbers.

RULE: If any SAS statement or procedure needs a SAS ds and you omit it the name, the most recently created SAS ds will be used. This is the default.
RECOMMENDATION: Always provide a name for your SAS ds (method 2 or 3). Make the file names unique (but non-unique librefs are OK).

D. Processing incoming data into SAS ds


"CARDS" Physically follow DATA step
Text data file On disk
SAS dataset On disk or from previous
DATA step Processes incoming data
SAS data set

1. There are four different kinds of incoming data:
    a. "Cards" requirements (data included in program)
        INPUT to describe structure of the raw data
        CARDS[6] immediately precedes data.
            Example:

```
DATA;
  INPUT NAME $ AGE SALARY;
CARDS;
BILL 36 30000
MARY 48 36000
JIM 32 27000
```

[6] A synonym for **CARDS** is **DATALINES**.

b. Text [7] file requirements (data in separate ASCII/text file)

These three statements in this order:

FILENAME to nickname the data source

INFILE to point SAS to the nickname

INPUT to describe structure of the raw data

Example: (assume C:\MYFOLDER\RECORDS.DAT is a Windows text file containing the data)

```
FILENAME INFO 'C:\MYFOLDER\RECORDS.DAT';
DATA;
  INFILE INFO;
  INPUT NAME $ AGE SALARY;
```

(Note: You can combine the FILENAME and INFILE statements into a single statement: `INFILE 'C:\MYFOLDER\RECORDS.DAT';` for short.)

c. SAS ds requirements (data are already a SAS ds)

SET to identify the SAS ds.

Syntax: `SET <previously created SAS ds>;`

Ex: `SET RECORDS;`

or `SET EMPLOYEE.RECORDS;`

(The latter would also require a LIBNAME.)

Ex: (assume data are in a SAS ds known to Windows as `C:\MYFOLDER\REC.SAS7BDAT`), then

```
LIBNAME ALIAS 'C:\MYFOLDER';
DATA;
  SET ALIAS.REC;
```

(Note that no INPUT or INFILE is used because SAS already knows the structure of the data.)

d. Excel and Access data files

In the Program Editor window, click **FILE → IMPORT DATA …** to invoke a "wizard" to guide you through the process of importing Excel and Access data files.[8] The wizard gives you the useful option of creating a SAS program file to store the import commands that it creates. The imported SAS dataset tends to be bloated in size. For best results, imported Excel files should be simple and clean (e.g., first row is variable names, second row begins the data, no mixed formatting within the same column, no merged cells, etc.) SAS scans *part* of your Excel data and *guesses* how to translate it into SAS format. ***I recommend that you compare the SAS data set to your input data to see whether or not the import worked properly.***

[7] Text files are also often called "flat" files, i.e., they are free of special formatting.

[8] You may also import data in Excel, Access and other data programs by using **PROC IMPORT**.

D.2. INPUT

Purpose of INPUT is to describe to SAS the structure of your data.
Need INPUT with "Cards" and text files – it is an error to use INPUT for SAS ds.
Three kinds of INPUT modes:

1. List input mode

Ex: `INPUT NAME $ AGE SALARY;`

- `INPUT` followed by list of variables in order of appearance in the data, with character variables followed by $
- the data for the variables separated by at least one blank space
- character variables ≤ 8 characters, no imbedded blanks
- missing values represented by periods (both numeric and character data) [9]

2. Column input mode

Ex: `INPUT NAME $ 1-4 AGE 6-7 SALARY 9-13;`

- `INPUT` followed by list of variables in the order to be read, character variables followed by $, and the location of the variables as keystroke ranges counting from the left edge of the file. In the above example, NAME is up to four characters long, beginning flush left in the file, AGE is found in the 6th and 7th keystrokes from the left edge of the file, and SALARY occupies 5 keystrokes beginning in the 9th keystroke from the left edge.
- input data fields must be aligned in same keystroke columns in all obs
- variables need not be separated by blanks
- missing values are represented by blanks or periods
- character variables may have > 8 characters and imbedded blanks
- variables can be read in any order

3. [Advanced] Formatted/pointer input mode

Ex: `INPUT @1 NAME $4. @6 AGE 2. @9 SALARY 5.;`

- `INPUT` followed by list of variables in order to be read, character variables followed by $, followed by a format modifier, and (perhaps) preceded by a pointer control (@). The "at" sign (@) directs SAS to a keystroke column, counting from the left edge of the file. The format modifiers in the example ($4., 2., 5.) are common character ($) and numeric ***informats*** that tell SAS how many keystrokes long each variable is and what their input formats are.

[9] SAS represents a missing value for numeric data by a period, for character data by a blank. But the blank cannot be used to represent missing character data in list input mode. The reason is that in list input mode, blanks are used to separate data values, rather than as values in themselves. Therefore, SAS will read a period for character data as a missing value and translate it into a blank.

<u>Miscellaneous INPUT notes:</u>

You can move the pointer to next line of input data by ordinary forward slash **/**
(useful when each obs requires more than one physical line to record all vars)
For most jobs, list or column input mode is sufficient.
INPUT modes can be mixed in the same INPUT statement.

II. (SOME) USEFUL STATEMENTS USED IN THE DATA STEP

In general, most statements that are legal in the DATA step are illegal in PROCs.

<u>Ex</u>: `IF SEX = 'M' THEN DELETE;` is legal in a DATA step. But the same statement in a PROC MEANS (trying to calculate statistics only for females), as below, is illegal:

<u>Ex</u>:
```
PROC MEANS;
     IF SEX = 'M' THEN DELETE;  * ILLEGAL!
     VAR SALARY; [10]
```

Conversely, most legal PROC statements are illegal in the DATA step.

<u>Ex</u>:
```
PROC SORT;
     BY NAME;
```
is a legal way to alphabetize a SAS dataset. However, putting this into a DATA step, in an attempt to sort the data while reading the data, as below, is illegal:
```
DATA RECORDS;
  INPUT NAME $ AGE SALARY;
  PROC SORT; * ILLEGAL!
    BY NAME; * ILLEGAL!
  NEWSAL = SALARY*1.10;
CARDS; [11]
 . . .
```
You should think of a DATA step as a block of statements completely separate from the block of statements that constitute a PROC step. Do not put a DATA step inside a PROC or vice-versa.

A. File handling statements

`DATA`, `CARDS`, `INFILE`, `INPUT`, `SET`, `FILENAME`, `LIBNAME` -- see above

B. Functions

A function has a name and one or more arguments

<u>Ex</u>. `SUM(X1,15,Z)` -- the function name is `SUM`, the arguments are `X1`, `15`, and `Z`

A function calculates a value based on the values of the arguments

<u>Ex</u>. `SUM(X1,15,Z)` calculates X1 + 15 + Z

The values of variables used as function arguments are taken from the Program Data Vector

[10] A correct way to accomplish the intent of this code is to replace the `IF` statement by `WHERE SEX = 'M';`

[11] A correct way to accomplish the intent of this code is to finish creating the SAS dataset in the DATA step, and then sort the newly created SAS dataset in a PROC SORT.

Ex: in `SUM(X1,15,Z)` the values of `X1` and `Z` are taken from the current values in PDV as it is at the time the function is encountered.

Propagation of missing values: Generally, if any argument of a function has a missing value, the result of the function is missing.

Ex: if `X1` = 10 and `Z` = ., then `SQRT(Z) = .` and `SQRT(X1 + Z) = .`

Exception: "Statistical" functions like `SUM`, `MAX`, `MEAN`, etc. return the value calculated from the *non*-missing arguments.

Ex: `SUM(X1, 15, Z) = 25`, but `SUM(Z) = .`

`OF` argument modifier: Statistical functions may use the special argument modifier `OF`, which permits you to list the arguments without putting commas between them, and to list a range of variables by using a dash.

Ex: If `X1=10` and `Z = .`, then `SUM(OF X1 15 Z) = 25` and `SUM(OF X1-X4 Z) = X1+X2+X3+X4+Z`, with missing values being omitted from the calculation by this statistical function.

Important: A function is not a variable. A function is not part of the PDV. But you can assign the value of a function to a variable (see below).

Functions are used in the DATA step, not in (most) PROCs. Thus functions calculate across data rows, not down data columns.

TIP: Avoid programming: Set up your data so that the PROCs do most of the statistical calculations, operating on the columns as units.

Ex: Use a DATA step to put 100 students' test scores in one column, with each student in a different row, so PROC MEANS can take the mean of the entire column of scores. This is much better than setting up a different DATA step variable for each student and calculating `MEAN(OF STUDENT1-STUDENT100)`

A few functions:

`SQRT(arg)` equals the square root of the argument

`RANNOR(arg)` generates a random value from a standard normal distribution. Arg is the seed for the random number generator; arg=0 uses the clock time as seed.

`RANUNI(arg)` is like `RANNOR` but generates a random value from a uniform distribution between 0 and 1.

`LOG(arg)` equals the natural logarithm (to base e=2.71...) of the arg.

`EXP(arg)` equals e raised to the power arg.

SAS also has character functions.

C. Expressions

An expression consists of variables, constants, and functions combined with operators.

Examples:

`X1 - SQRT(Z+1) / 3.14` is a numerical expression.

`(X > 2) & NAME IN ('Bill','Mary','Jim')` is a logical expression.

Common arithmetic operators: `+  -  /  *  **`

Common comparison operators: `EQ  NE  GT  LT  GE  LE  =  ^=  >  <  >=  <=`

Common logical operators: `AND  OR  NOT  &  |  ^  IN`

Parentheses may be used to group operations. Ex: `(X1+Z)*(4+Y)`

D. Assignment statement

Purpose: creates a new SAS variable and assigns it a value or reassigns a new value to an existing variable. Changes the content of the PDV for the (re)assigned variable.

Syntax: `var = SAS expression ;`

The SAS expression on the right-hand-side is built according to II.C out of SAS functions, variables, constants, operators.

Examples:

`NEWSAL = SALARY * 1.10;` (numerical expression)

`TYPE = 'Hourly ';` (character expression)

`TEST = (X > Y);` (logical expression, = 1 if true, = 0 if false)

`SCORE = .5*(SCORE + 50);` (reassignment)

Important: Functions cannot be assigned, because functions are not variables -- a variable must appear on the left-hand-side of an assignment.

Ex: `LOG(Z) = Y;` will produce an error. But `Y = LOG(Z);` will work.

`LOGZ = Y;` will also work, because `LOGZ` is a legitimate name for a SAS variable -- `LOGZ` is not a function. Functions have parentheses.

Missing values "propagate": Ex: `X = A + B; Y = C + X; Z = D + Y`; if A is missing, then X will be missing, so then Y will be missing, and hence Z will be missing.

Creation of character vars.

Ex. `NAME = 'BILLY BOB PRUITT';`

E. Other useful statements

1. `IF ... THEN...; ELSE ...;`

Purpose: to take some action only if a condition is true

Syntax: `IF <logical expression> THEN <command>; ELSE <command>;`

(Note: The `ELSE` part is optional.)

Examples:

```
IF TAXINC > 30000 THEN TAX = 5000 + .3*(TAXINC-30000);
IF BAL < 0 THEN GOTO OVERDRAW; ELSE PUT NAME BAL;
IF SCORE > 90 THEN GRADE = 'A'; ELSE IF SCORE > 80 THEN
     GRADE = 'B'; ELSE GRADE = 'C';
```

`IF SEX = 'M' THEN DELETE;` When `DELETE` is encountered, the following happen:

- The current loop of the DATA step stops.
- No more statements in the DATA step are executed for the current obs.
- SAS returns to the top of the DATA step and begins a new loop for the next obs.
- The current obs is not added to the SAS ds being created.

The effect of `IF SEX = 'M' THEN DELETE;` is to remove all obs with SEX code 'M' from the SAS ds.

Subsetting `IF`: A special construction of the `IF` statement has syntax `IF <logical expression>;` It is used to screen or sieve the data for certain characteristics. If the logical expression is true, SAS continues the DATA step. But if the logical expression is false, SAS terminates the DATA step for the current obs, behaving as though a `DELETE` had been encountered. The result is that the only records ending up in the SAS ds are those for which the logical expression is true.

Examples:

`IF SEX NE 'M';` is equivalent to `IF SEX = 'M' THEN DELETE;`

`IF BAL < 0 AND TEST = 'Closed';` is equiv to `IF BAL >= 0 OR TEST NE 'Closed' THEN DELETE;`

`IF SCORE < 10 OR SCORE > 90;` is equiv to `IF SCORE >= 10 AND SCORE <= 90 THEN DELETE;`

2. RETAIN and sum statements

`RETAIN`

Purpose: Prevents RETAINed vars from being reinitialized at start of each DATA step loop

Syntax: `RETAIN <var list>;`

Ex: `RETAIN N;`

Permits "passing" values of variable values from one DATA loop (observation) to the next.

The "sum" statement
Purpose: Accumulates totals from one observation to the next (creates a running total variable)
Syntax: `var + expression;`
Examples:

`TOTPAY + PAY;` `TOTPAY` becomes a running total of `PAY` from obs to obs.

`N + 1;` The variable `N` becomes an observation counter.

`TOTBAL + (DEPOSIT - CHECK);` `TOTBAL` is a running checkbook balance.

A sum variable is automatically RETAINed: It is not reinitialized at the beginning of each execution of the DATA step but retains its value from the previous execution.

The initialization of RETAIN and sum variables before the first loop through a DATA step is to the value zero, rather than to missing.

3. `FORMAT`
Purpose: to improve the printout appearance of the values of variables in the SAS listing
Syntax: `FORMAT var <format>;`

Examples:

`FORMAT (X Y) (6.1);` If X=4331.75 and Y=3520, they will be expressed as 4331.8 and 3520.0 -- note the parenthetical grouping to create a format list.

`FORMAT WAGE DOLLAR11.2;` results in expressing a WAGE of 15323.457 as $15,323.46.

`FORMAT NAME $5. START MMDDYY10.;` results in a NAME of " William" [note 2 blanks at beginning] becoming "Wil" and a START date of Feb 23, 1996 as 02/23/1996.

SAS date vars are actually integer numeric vars [12] and can be subtracted (for example, to obtain durations) like other numeric vars. This is similar to the treatment of dates in Excel.

Formats assigned to variables in a DATA step remain attached to those vars in all PROCs.

Formats may also be assigned to variables in PROCs,[13] but such PROC formats remain in force only during the particular PROC.

A format does not change the value as it is stored in SAS. A format changes only the way the value is displayed. Operations with formatted values use the stored values, not the displayed values.[14] This is like Excel.

Ex: If the format 5.1 applies to 6.24 and 7.53 and their sum, the three values will be displayed as 6.2 and 7.5 and 13.8; but their sum will be stored as 13.77, rather than as 13.7 or 13.8.

[12] SAS stores 1/1/1960 as 0, 1/2/1960 as 1, 1/3/1960 as 2, etc., counting by days. Dates before 1960 are negative counts: 12/31/1959 is -1, 12/30/1959 is -2, etc. Spreadsheets (like Excel) also store dates as integer counts. For dealing with years after 2000 entered as two-digit numbers in input data, SAS has the `YEARCUTOFF` option to allow reading such two-digit years as 21st century rather than as 20th century dates.

[13] Thus, `FORMAT` is one example of a DATA step statement that may also be used in PROCs.

[14] Similarly, Excel uses the unrounded values stored in its cells for calculations, rather than rounded displays.

4. **`LABEL`**
Purpose: to provide descriptors for variables on SAS output
Syntax: `LABEL var = '<description>';`
Ex: `LABEL PAY = 'Pay For This Week';`
Where space permits, the descriptor for a LABELed var is printed on the SAS listing when the variable is analyzed. SAS respects upper and lower case for labels.

5. **`ATTRIB`**
Purpose: to assign both a label and a format to a variable
Syntax: `ATTRIB var FORMAT=<format> LABEL='<label>';`
Ex: `ATTRIB PAY FORMAT=DOLLAR11.2 LABEL='Pay for Feb 1 - Feb 15';`

6. **`DROP`**
Purpose: to prevent DROPped vars from being included in a SAS ds under construction
Syntax: `DROP <var list>;`
Ex: `DROP AGE SALARY;`
`DROP` removes a column, **`DELETE`** removes a row from the SAS ds.
`DROP` has no effect on the PDV until the PDV is ready to be written to the SAS ds being created.
`DROP` may occur anywhere in the DATA step and has the same effect wherever it appears.
Important: A DROPped variable remains in the PDV! You may continue to use a DROPped var in the DATA step -- use it in expressions, calculate with it, change its value, etc. -- even after the DROP statement appears. DROP's only effect is on which variables are written to the SAS dataset being created by the DATA step.
Ex:
```
DATA REC;
  INPUT X Y;
  DROP X;
  Z = X + Y;
```
X is still available to calculate Z, because X is dropped only when the PDV is written to WORK.REC.
Note alternative usage and syntax as dataset modifier:
Ex: `DATA REC (DROP=AGE SALARY);`

7. **`KEEP`**
Purpose: to prevent un-KEEPed vars from being in the SAS ds
Syntax: `KEEP <var list>;`
Ex: `KEEP AGE SALARY;`
`KEEP` is the opposite of **`DROP`**. A variable not KEEPed is DROPped.
Note alternative usage and syntax as dataset modifier:
Ex: `DATA REC (KEEP=AGE SALARY);`
If variables are both DROPped and KEEPed, they are kept.
Important: Non-KEEPed vars remain in the PDV -- they just are not written to the SAS ds being created. (This corresponds to the treatment in DROP).