FAA

Fractal Academy of Analytics

Data Manipulation in SAS

Agenda

- SET statement
- Operators
- Subsetting data
  - Subsetting IF
  - Where
- IF – THEN
- IF – ELSE
- Select groups
- Output statement
- Length
- Functions
- Dataset options & statements
  - Keep and Drop
  - Rename
  - Label
- Sum Statement
- Retain
- Sorting Datasets
- By group processing
- Combining datasets

Reading data from an existing dataset

SET statement:

A set statement is used to copy data from an existing dataset into a new dataset.

```
data work.test;
SET sashelp.class;
run;
```

By running the above code, all the contents of class dataset are copied to test

Operators

- Operators are generally used in a condition to compare variables or expressions.

Operator	Meaning	Example
= or eq	Equal to	Name = “Fractal”
^= or ne	Not equal to	ID ne 212
> or gt	Greater than	Income > 20000
< or lt	Less than	Salary < 10000
>= or ge	Greater than or equal to	Premium >= 200
<= or le	Less than or equal to	Claim <= 1000
in	List of values	Age in (13,20)

- Logical operators are used to combine two or more conditions

Operator	Meaning
AND or &	and, both. If both expressions are true, then the compound expression is true.
OR or |	or, either. If either expression is true, then the compound expression is true.

Sub-setting data – Using “IF”

- Subsetting “If ” condition is used to pull a smaller part of data from the original dataset.
  - Example: We need data for males/females only from the mother dataset. So we use a subsetting if condition to separate the data.

```
data work.young;
set sashelp.class;
if age < 15;
run;
```

- The above step pulls the data only when the If condition is true!

Sub-setting data – Using “Where”

- Where condition performs a similar task of imposing a condition while pulling the data.
  - Using the same Example: We need data for males/females only from the mother dataset. Here we are using a where condition to separate the data.

```
data work.young;
set sashelp.class;
where age < 15;
run;
```

The Difference: WHERE and IF

There are some important differences between Where and Subsetting IF.

- WHERE conditions are applied before data enters into the input buffer. IF conditions are applied after the data enters the PDV.
  - WHERE condition is faster than IF because not all observations have to be read!
- WHERE can be used in PROC's to impose conditions
- Automatic conversion of variables is not possible in WHERE statement.
- Automatic variables like _n_, first.by, last.by can be used only in IF statement

Quick Exercise – 1 (Time: 15 mins)

- Import card, customer and transaction data
- Create a subset from customer data for those customers whose age is between 20 and 50

Creating multiple datasets

- Output statement is very useful when we need to create multiple datasets from the main dataset.

  Syntax:

  ```
  data work.male work.female;
  set sashelp.class;
  If sex = “M” then output work.male;
  If sex = “F” then output work.female;
  run;
  ```

- The above code creates two datasets, one each for the two genders.

Quick Exercise – 2 (Time: 5 mins)

- Subset data for each card type into separate datasets

Dataset Options: Keep, Drop and Rename

- Dataset options help us manipulate the variables inside a dataset.
- DROP = option allows us to select those variables which we do not want in the new dataset.
- KEEP= option allows us to select those variables which we want in the new dataset
  - These options can be specified in the SET and MERGE statements
  - Syntax:

```
Data work.test;
set sashelp.class (KEEP = Name Age);
run;
```

- If you want to rename a variable you can use RENAME= dataset option.
  - Syntax:

```
Data work.test;
set sashelp.class (Rename = (Weight = Wt_kgs));
run;
```

Dataset Statements: Keep, Drop, Rename and Label

- These work similar to the keep and drop options but are specified inside a dataset.
- DROP statement allows us to drop those variables which we do not want.
- KEEP statement allows us to keep those variables which we want
- RENAME statement allows us to rename a variable
- LABEL statement allows us to label a variable
  - These options can be specified in the SET and MERGE statements
  - Syntax:

```
Data work.test;
set sashelp.class;
DROP height weight;
RENAME sex = Gender;
LABEL Name = "Name of the Student";
run;
```

Quick Exercise – 3 (Time: 5 mins)

- Create a subset of card data such that it only has the following 3 variables
  - Card_id
  - Card_type
  - Card_limit
- After sub-setting the data, rename “card_limit” to “Limit”

Conditional Processing: If – Then

- If – Then is used to specify a condition and perform a task when the condition(s) is true.
  - Example:

```
data work.if_then;
set sashelp.class;
length group $ 8;
if height <= 60 then group = “short”;
if height > 60 then group = “tall”;
run;
```

- The above code creates a new variable group. This variable is filled accordingly as per the If – Then condition specified.

Conditional Processing: If – Else

- If – Else is used to nest two or more If – Then statements.
- If – Else ignores all the other statements in the nest once an IF condition is satisfied
- Example:

```
data work.if_else;
set sashelp.class;
length age_grp $ 8;
if  age <= 30 then age_grp = “young”;
else if 30 < age <= 60 then age_grp = “mid-age”;
else age_grp = “old”;
run;
```

- The above code creates various age groups based on the conditions specified.

The Difference: IF-THEN and IF-ELSE

– For example:

```
data work.test;
set sashelp.class;
length group $ 10;
if age < 14 then group = “A”;
if age <=18 then group = “B”;
run;
```

– If the above code is executed, what will be the group of a student whose age is 13?

The Difference: IF-THEN and IF-ELSE

Though IF-THEN and IF-ELSE perform the same task of imposing condition to execute a task. They have some important differences in the way they work!

– Writing the same code using IF-ELSE

```
data work.test;
set sashelp.class;
length group $ 10;
if age < 14 then group = “A”;
else if age <=18 then group = “B”;
else group = “NA”;
run;
```

– If the above code is executed, what will be the group of a student whose age is 13?

The Difference: IF-THEN and IF-ELSE

- What's happening?
  - When IF-THEN is used, SAS executes all the IF-THEN statements without taking into consideration if the previous condition was satisfied or not!
    - In the first case, when age = 13, it satisfies both the conditions. When first IF-THEN is executed the group is set to A
    - But when the second statement is executed, the group variable is overwritten to B since the condition is true!
  - But, when IF-ELSE is used, SAS stops executing the chain of IF-ELSE when one of the if condition is satisfied.
    - So in the second case, after the first condition is satisfied for age = 13, SAS stops executing the rest of the chain.
    - So the group is set to A and is NOT overwritten since rest of the nested IF statements are not executed!!

Conditional Processing: SELECT Groups

- We can also use SELECT groups in DATA steps to perform conditional processing
  - Syntax:

```
SELECT (age);
        WHEN (12,13,14) group = “A”;
        WHEN  (15,16) group = “B”;
        OTHERWISE group = “NA”;
END;
```

Using a SELECT group is slightly more efficient than using a series of IF-THEN or IF-THEN/ELSE statements because CPU time is reduced.

- SELECT groups also make the program easier to read and debug.

Quick Exercise – 4 (Time: 5 mins)

- In the Education field of customer data Postgraduate is spelled as Postgradraduate. Change the values to make this right.

Do Statement

- Loops are generally used to perform repetitive tasks on a group of variables.
- There are 3 forms of Do statement in SAS.

Form	Meaning
Do Statement	Executes statements between DO and END statements repetitively based on the value of an index variable. The iterative DO statement can contain a WHILE or UNTIL clause.
Do Until	Executes statements in a DO loop repetitively until a condition is true, checking the condition after each iteration of the DO loop. Do Until will always execute at least once! This is because the condition is evaluated at the end of the loop
Do While	Executes statements in a DO loop repetitively while a condition is true, checking the condition before each iteration of the DO loop In Do While the condition is checked at the beginning of the loop, so the DO statement executes only if the condition is true!

Do Statement

- It is used to execute a set of statements conditionally.
  - Example:

```
data work.test;
set sashelp.class;
length group $ 10 score 8;
if sex = “F” then DO;
    group = “Female”;
    BMI =  weight / (height**2);
END;
run;
```

  - In the above code when the condition is true, all the statements inside the DO – END are executed.

Quick Exercise – 5 (Time: 5 mins)

- Create a (1,0) tag variable for silver cards and change the card_limit of the silver cards to 9000.

Length Statement

- LENGTH statement is used to specify the length of a variable.
  - Syntax: LENGTH *varname* ($) *length;*
  - Example: LENGTH id_num 6;
    LENGTH name $ 20; /* $ is specified for character variables*/

- When LENGTH statement is not used to specify length, SAS automatically assigns a length depending on the first occurrence of the variable in the submitted code.

- It is very important to assign length especially to character variables to avoid truncation of data.

Length Statement

Example:

```
data work.test;
set sashelp.class;
if height > 60 then group = “tall”;
if height <= 60 then group = “short”;
run;
```

- What will be the value of group for those who have height less than 60?

Example:

```
data work.test;
set sashelp.class;
length group $ 5;
if height > 60 then group = “tall”;
if height <= 60 then group = “short”;
run;
```

Quick Exercise – 6 (Time: 5 mins)

- Create a Income_group variable based on the customer’s monthly income as follows:
  - If monthly income is less than 3000 then the group is LOW
  - If monthly income is more than 6000 then group is HIGH
  - Otherwise group must be MED

Delete Statement

- Delete statement is generally used with an IF statement to delete observations conditionally.
  - Syntax: IF *condition* THEN DELETE;

- This permanently deletes an observation when the IF condition is true.

- Using DELETE is not advised, because it deletes the data from the original dataset. Instead we can subset the required data into a separate dataset.

- It is always a good practice NOT to modify the original dataset.

FUNCTIONS

Functions - Introduction

- Functions are used to modify, extract or create new variables.

- Functions can be broadly divided into three types:

  1. Character functions – Used to modify character variables

  2. Numeric functions – Used to modify numeric variables

  3. Date functions – Used to modify date variables

Character Functions

Function	Description
SCAN	Returns a specified word from a character value.
SUBSTR	Extracts a substring or replaces character values.
TRIM	Trims trailing blanks from character values.
CATX	Concatenates character strings, removes leading and trailing blanks, and inserts separators.
INDEX	Searches a character value for a specific string.
FIND	Searches for a specific substring of characters within a character string that you specify.
COMPRESS	To remove specified characters from a variable
COMPBL	To replace all occurrences of two or more blanks with a single blank character.
COMPGED	To compute the similarity between two strings, using a method called the generalized edit distance
UPCASE	Converts all letters in a value to uppercase.
LOWCASE	Converts all letters in a value to lowercase.
PROPCASE	Converts all letters in a value to proper case.

Changing the Case

- UPCASE Function
  - converts all letters in a character expression to uppercase.

    UPCASE(*argument*)
- LOWCASE Function
  - converts all letters in a character expression to lowercase.

    LOWCASE(*argument*)
- PROPCASE Function
  - converts all words in an argument to proper case (so that the first letter in each word is capitalized).

    PROPCASE(*argument,<delimiter(s)>)*
  - Delimiters specifies one or more delimiters that are enclosed in quotation marks. The default delimiters are blank, forward slash, hyphen, open parenthesis, period, and tab.

SCAN Function

- The SCAN function
  - Enables you to separate a character value into words and to return a specified word.
  - It uses **delimiters**, which are characters that are specified as word separators
  - Syntax: SCAN(argument,n,delimiters)
- Specifying Multiple Delimiters
  - When using the SCAN function, you can specify as many delimiters as needed to correctly separate the character expression.
  - The SCAN function treats two or more continuous delimiters, such as the parenthesis and slashes below, as one delimiter. Also, leading delimiters have no effect.
- Default Delimiters
  - If you do not specify delimiters, default delimiters are used. They are

    blank . < (+ | & ! $ *) ; ^ - / , %
- Note that the SCAN function assigns a length of 200 to each target variable.

SUBSTR Function

- The **SUBSTR** function can be used to
  - extract a portion of a character value
  - replace the contents of a character value.
  - The SUBSTR function enables you to extract any number of characters from a character string, starting at a specified position in the string.

General form, SUBSTR function:

SUBSTR(*argument,position,<n>*)

- When “n” is not provided, SAS will extract till the end of the string

SUBSTR Function

- Extract a portion of a character value
- When the function is on the **right side** of an assignment statement, the function returns the requested string.

  E.g. `MiddleInitial=substr(middlename,1,1);`

- Replace the contents of a character value.
- if you place the SUBSTR function on the **left side** of an assignment statement, the function is used to modify variable values.

  E.g. `substr(region,1,3)='NNW';`

SCAN Function Compared with SUBSTR Function

- Both the SCAN and SUBSTR functions can extract a substring from a character value:
  - **SCAN** extracts words within a value that is marked by delimiters.
  - **SUBSTR** extracts a portion of a value by starting at a specified location.

- The SUBSTR function is best used
  - When you know the **exact position** of the substring that you want to extract from the character value.
  - The substring does not need to be marked by delimiters.
- The SCAN function is best used when
  - You know the order of the words in the character value
  - The starting position of the words varies
  - The words are marked by some delimiter.

Quick Exercise – 7 (Time: 5 mins)

- Create a variable(Marital_code) which will work as a single letter code for a customer's marital status (extract the first letter of the marital status variable and give it a name)

- Extract the second word from the occupation field.

TRIM Function

- The **TRIM** function enables you to remove trailing blanks from character values.

- The concatenation operator (||) enables you to concatenate character values.

  E.g. `NewAddress=address||', '||city||', '||zip;`

- Whenever the value of a character variable does not match the length of the variable, SAS pads the value with trailing blanks. The TRIM function enables you to remove trailing blanks from character values.

  E.g. `NewAddress=trim(address)||', '||trim(city)||', '||zip;`

CATX Function

- CATX function enables you to concatenate character strings, remove leading and trailing blanks, and insert separators.

- Results of the CATX function are usually equivalent to those that are produced by a combination of the concatenation operator and the TRIM and LEFT functions.
  - Syntax: CATX(separator,string-1 <,...string-n>)
    - separator specifies the character string that is used as a separator between concatenated string
  - Example:

```
NewAddress = catx(', ',address,city,zip);
```

Quick Exercise – 8 (Time: 5 mins)

- Concatenate marital_code and education with underscore (_) as a separator using:
  - Concatenation operator (||)
  - CATX function

Final variable must look like “M_Graduate”

INDEX Function

General form, INDEX function:

INDEX(*source*,*excerpt*)

- The **INDEX** function
  - Enables you to search a character value for a specified string.
  - Searches values from left to right, looking for the first occurrence of the string. It returns the position of the string's first character;
  - If the string is not found, it returns a value of ***0***.

E.g. index(job,'word processing')

FIND Function

General form, FIND function:

FIND(*string,substring,<modifiers>,<startpos>*)

- The **FIND** function
  - Enables you to search for a specific substring of characters within a character string that you specify.
  - It searches a string for the first occurrence of the substring, and returns the position of that substring.
  - If the substring is not found in the string, FIND returns a value of *0*.
  - The FIND function is similar to the INDEX function.
- The *modifiers* argument enables
  - The modifier **i** causes the FIND function to ignore character case during the search.
  - The modifier **t** trims trailing blanks from *string* and *substring*

Quick Exercise – 9 (Time: 5 mins)

- Find out if the customer is a graduate or not, if yes create a binary variable (1 = yes, 0 = no)

Compress Function

- Removes specified character values from a string value. Removes blanks if compress list is not provided.
  - Syntax: COMPRESS(*character-value, <'compress-list'>, modifiers)*;
  - Example:
    1. CHAR = “A C123XYZ”

       COMPRESS(CHAR) returns “AC123XYZ”

       COMPRESS(CHAR,"0123456789“) returns “A CXYZ”
    2. PHONE = “(908) 777-1234”

       COMPRESS(phone, “ (-)“) returns “9087771234”

- There are many modifiers which can be used with compress. The modifier “k” is used to retain the compress list and remove other characters.

COMPBL Function

- Replaces all occurrences of two or more blanks with a single blank character.
- No effect on single blanks
- This is particularly useful for standardizing addresses and names where multiple blanks may have been entered.
  - Syntax: COMPBL(character value);
- If a length has not been previously assigned, the length of the resulting variable will be the length of the argument.
  - Example:

    CHAR = “A   B   C   DEF”

    COMPBL(char) will return a value of “A B C DEF”

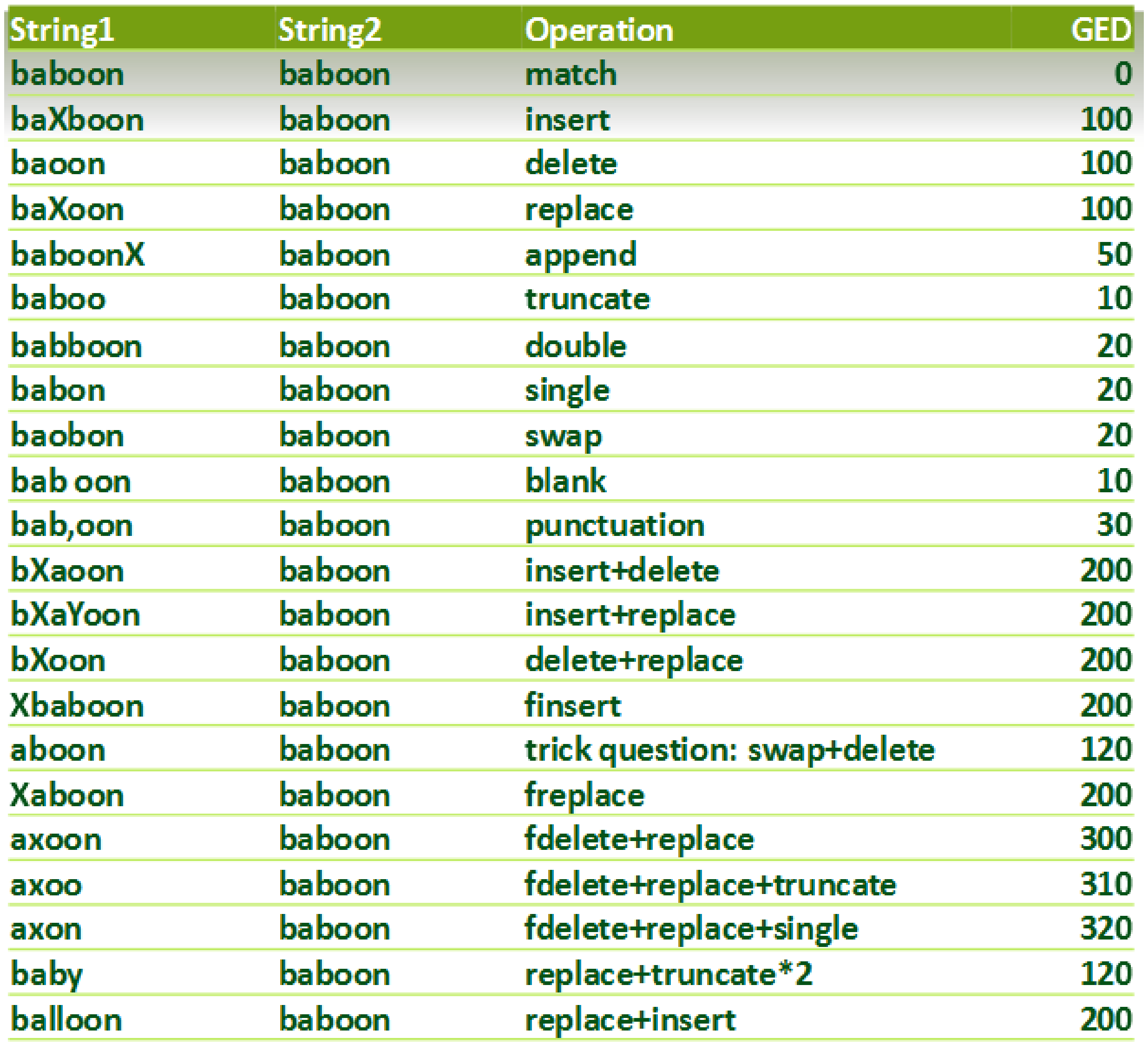
COMPGED Function

- A SAS function that **COMP**utes the **G**eneralized **E**dit **D**istance between two character strings
- COMPGED calculates a number representing how much work it takes to make the second string exactly like the first
- The higher the computed GED, the less likely the two strings match
- Zero = perfect match
- COMPGED allows fuzzy matching when used with PROC SQL
  - Syntax: COMPGED(STRING1, STRING2);
  - Example:

    COMPGED(SAME,SAME) returns 0

    COMPGED(SAME,same) returns 500

COMPGED Function



String1	String2	Operation	GED
baboon	baboon	match	0
baXboon	baboon	insert	100
baoon	baboon	delete	100
baXoon	baboon	replace	100
baboonX	baboon	append	50
baboo	baboon	truncate	10
babboon	baboon	double	20
babon	baboon	single	20
baobon	baboon	swap	20
bab oon	baboon	blank	10
bab,oon	baboon	punctuation	30
bXaoon	baboon	insert+delete	200
bXaYoon	baboon	insert+replace	200
bXoon	baboon	delete+replace	200
Xbaboon	baboon	finsert	200
aboon	baboon	trick question: swap+delete	120
Xaboon	baboon	freplace	200
axoon	baboon	fdelete+replace	300
axoo	baboon	fdelete+replace+truncate	310
axon	baboon	fdelete+replace+single	320
baby	baboon	replace+truncate*2	120
balloon	baboon	replace+insert	200

Numeric Functions

- INT Function
  - To return the integer portion of a numeric value, use the **INT function**. Any decimal portion of the INT function argument is discarded.
  - **INT**(*argument*)

- ROUND Function
  - To round values to the nearest specified unit, use the **ROUND function.**
  - **ROUND**(*argument, round-off-unit*)
  - If a round-off unit is not provided, a default value of 1 is used, and the argument is rounded to the nearest integer.

Functions For Descriptive Statistics

Function	Syntax	Calculates
SUM	sum(argument, argument,...)	Sum of values
MEAN	mean(argument, argument,...)	Average of non-missing values
MIN	min(argument, argument,...)	Minimum value
MAX	max(argument, argument,...)	Maximum value
VAR	var(argument, argument,...)	Variance of the values

- When functions are used, SAS ignores the missing values. Consider the values of V1, V2, V3 to be 3,.(missing),5 respectively
  - X = SUM(V1, V2, V3); will return 8 as output
- But when a calculation is done manually, a missing value input returns a missing value as the output of the calculation.
  - X = V1 + V2 + V3; will return .(missing value) as output

DATE FUNCTIONS

Manipulating SAS Date Values with Functions

Function	Typical Use	Result
DATE CONSTANTS	'ddmmm<yy>yy' D or "ddmmm<yy>yy" D	Creates a SAS date using date constants
MDY	date=mdy(mon,day,yr);	Creates a SAS date based on the input values

TODAY DATE	now=today();
	now=date();
		Returns today's date as a SAS date

TIME	curtime=time();	current time as a SAS time
DAY	day=day(date);	day of month (1-31)
QTR	quarter=qtr(date);	quarter (1-4)
WEEKDAY	wkday=weekday(date);	day of week (1-7 where 1 represents sunday)
MONTH	month=month(date);	month (1-12)
YEAR	yr=year(date);	year (4 digits)

INTCK	x=intck('day',d1,d2);
	x=intck('week',d1,d2);
	x=intck('month',d1,d2);
	x=intck('qtr',d1,d2);
	x=intck('year',d1, d2);
		days from D1 to D2
		weeks from D1 to D2
		months from D1 to D2
		quarters from D1 to D2
		years from D1 to D2

INTNX	x=intnx('interval',start- from,increment);	date, time, or datetime value
DATDIF YRDIF	x=datdif('date1',date2, ACT/ACT);x=yrdif('date1',date2, ACT/ACT);	days between date1 and date2 years between date1 and date2

Date Constants

- Assign date values to variables in assignment statements by using date constants.
  - Syntax:

‘ddmmm<yy>yy’D

or

"ddmmm<yy>yy" D

  - *dd is a one- or two-digit value for the day*
  - *mmm is a three-letter abbreviation for the month (JAN, FEB, and so on)*
  - *yy or yyyy is a two- or four-digit value for the year, respectively.*

Note: Be sure to enclose the date in quotation marks

Example: TestDate='01jan2000'd;

MDY Function

- MDY function is used to create date value by inputting date, month and year values
  - Syntax: MDY(month,day,year);
  - Month can be a variable that represents month, or a number from 1-12
  - Day can be a variable that represents day, or a number from 1-31
  - Year can be a variable that represents year, or a number that has 2 or 4 digits.

YEAR, QTR, MONTH, and DAY Functions

- General form, YEAR, QTR, MONTH, and DAY functions:

- YEAR(*date*)
  QTR(*date*)
  MONTH(*date*)
  DAY(*date*)

Function	Description	Form	Sample Value
YEAR	Extracts the year value from a SAS date value.	YEAR(date)	2005
QTR	Extracts the quarter value from a SAS date value	QTR(date)	1
MONTH	Extracts the month value from a SAS date value.	MONTH(date)	12
DAY	Extracts the day value from a SAS date value.	DAY(date)	5

Quick Exercise – 10 (Time: 5 mins)

- Create a date (10th September, 2012) using
  - Date constant
  - MDY
- Create the transaction date for all the transactions

INTCK Function

General form, INTCK function:

INTCK('*interval*',*from*,*to*)

- The **INTCK** function
  - The INTCK function counts intervals from fixed interval beginnings, not in multiples of an interval unit from the *from* value.
  - Partial intervals are not counted.
  - Intervals that can be used: days,weeks,months,quarters,years

SAS Statement	Value
Weeks = intck ('week','31 dec 2002'd,'01jan2003'd);	0
Months = intck ('month','31 dec 2003'd,'01jan2003'd);	1
Years = intck ('year','31 dec 2003'd,'01jan2003'd);	1

Quick Exercise – 11 (Time: 5 mins)

- Check the interval between card activation date and the card expiry date for all the cards.

INTNX Function

General form, INTNX function:

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

- The **INTNX** function is similar to the INTCK function.
- The INTNX function applies multiples of a given interval to a date, time, or datetime value and returns the resulting value.
- You can use the INTNX function to identify past or future days, weeks, months, and so on.

INTNX Function

- <*alignment*> argument: it lets you specify whether the date value should be at the beginning, middle, or end of the interval.

- In the INTNX function, use the following arguments or their corresponding aliases
  - BEGINNING ’B’
  - MIDDLE ‘M’
  - END ‘E’
  - SAMEDAY ‘S’

SAS Statement	Date Value
MonthX=intnx('month','01jan95'd,5,'b');	12935 (June 1, 1995)
MonthX=intnx('month','01jan95'd,5,'m');	12949 (June 15, 1995)
MonthX=intnx('month','01jan95'd,5,'e');	12964 (June 30, 1995)

Quick Exercise – 12 (Time: 5 mins)

- Assuming that all the cards expire after 4 years from the card activation date, calculate the new expiry date.

DATDIF and YRDIF Functions

General form, DATDIF and YRDIF functions:

DATDIF(*start_date,end_date,basis*)
YRDIF(*start_date,end_date,basis*)

- The **DATDIF** and **YRDIF** functions calculate the difference in days and years between two SAS dates, respectively.
- Both functions use a *basis* argument that describes how SAS calculates the date difference.

Character String	Meaning	Valid In DATDIF	Valid In YRDIF
'30/360'	Specifies a 30 day month and a 360 day year	yes	yes
'ACT/ACT'	Uses the actual number of days or years between dates	yes	yes
'ACT/360'	Uses the actual number of days between dates in calculating the number of years (calculated by the number of days divided by 360)	no	yes
'ACT/365'	Uses the actual number of days between dates in calculating the number of years (calculated by the number of days divided by 365)	no	yes

Explicit Data Conversion

- General form, INPUT function: (Character to Numeric)
  INPUT(*source,informat*)

- General form, PUT function: (Numeric to Character)
  PUT(*source,format*)

- Note that the INPUT function requires an informat, whereas the PUT function requires a format
- To remember which function requires a format versus an informat, note that the **IN**PUT function requires the **in**format.
- This conversion occurs before the value is assigned or used with any operator or function.

Automatic Character to Numeric Conversion

- What happens if you omit the INPUT function or the PUT function when converting data?

- Automatic Character-to-Numeric Conversion
  - If you reference a **character variable in a numeric context** SAS tries to convert the variable values to numeric.
  - This conversion is completed by creating a temporary numeric value for each character value of Variable. This temporary value is used in the calculation.
  - The character values of Variable are **not** replaced by numeric values.
  - Whenever data is automatically converted, a message is written to the SAS log stating that the conversion has occurred.

Automatic Character to Numeric Conversion

- The automatic conversion
  - Uses the *w.d* informat, where *w* is the width of the character value that is being converted
  - Produces a numeric missing value from any character value that does not conform to standard numeric notation (digits with an optional decimal point or leading sign).

- Restriction for WHERE Expressions
  - The WHERE statement **does not** perform automatic conversions in comparisons.
  - Mismatch of data types prevents the program from processing the WHERE statements, the program stops, and error messages are written to the SAS log.

LAG Function

General Form

LAG<n>(argument)

- n specifies the number of lagged values.

- The LAG functions, LAG1, LAG2, ..., LAG*n* return values from a queue.
- LAG1 can also be written as LAG.
- A LAG*n* function stores a value in a queue and returns a value stored previously in that queue.
- Each occurrence of a LAG*n* function in a program generates its own queue of values.
- Memory Limit for the LAG Function
  - When the LAG function is compiled, SAS allocates memory in a queue to hold the values of the variable that is listed in the LAG function.

RETAIN STATEMENT & MERGING

Sum Statement and Retain

- Used to produce column totals for numeric variables
  - Syntax: *new_accumulator_variable + existing_dataset_variable;*
- The accumulator variable is assigned an initial value of zero automatically.
- The value of the accumulator variable is retained across all the data step iterations.

- We can also create accumulator variables using a RETAIN statement.
  - Syntax: RETAIN *variable initial_value;*
  - RETAIN is a compile-time only statement that creates variables if they do not already exist
  - It initializes the retained variable to missing before the first execution of the data step if you do not supply an initial value

Proc Sort

- PROC SORT is a procedure used to sort the data on a group of selected variables and also to remove any duplicates.
    - Syntax:

      PROC SORT DATA = input data OUT = output data (OPTIONS);

      BY <DESCENDING> variables;

      RUN;

  – DATA = option specifies the dataset to be read

  – OUT = option specifies the output dataset which contains the data in sorted order

  – OPTIONS: We can either specify NODUPREC or NODUPKEY
    - NODUPREC – Removes duplicate records (observations)
    - NODUPKEY – Removes duplicate "KEY"s

  – DESCENDING is specified when we need the data to be sorted in descending order

Proc Sort

- If OUT = is not specified, SAS rearranges the observations and replaces the original dataset
- When OUT= is specified, a new SAS data set that contains the rearranged observations is created
- Can sort on multiple variables (BY var1 var2….;)
- Can sort in ascending or descending order (by default ascending). To sort in descending specify DESCENDING keyword before that particular variable
- Does not generate printed output
- Treats missing values as the smallest possible values

By Group Processing

- When you use the BY statement with the SET statement,

```
data work.test;
set sashelp.class;
BY age;
run;
```

  - The data sets that are listed in the SET statement must be sorted by the values of the BY
  - The DATA step creates two temporary variables for each BY variable.
    - FIRST.*variable,* where variable is the name of the BY variable, and the other is named LAST.*variable.* Their values are either 1 or 0. They identify the first and last observation in each BY group respectively.

This Variable	**Equals**
FIRST.*variable*	1 for the FIRST observation in a BY group
	0 for any other observation in a BY group
LAST.*variable*	1 for the LAST observation in a BY group
	0 for any other observation in a BY group

By Group Processing

- Using a sum statement and the automatic by group variables, we can find the cumulative sum of each by group.

Example:

```
proc sort data=work.test out=work.temp;
by age;
run;
data work.temp;
set work.temp;
by age;
if first.age then Payroll=0;
payroll + salary;
if last.age;
drop name sex;
run;
```

Sorts the data

Creates the BY group variables first.age and last.age

For the first observation in the by group, payroll is equated to 0

Outputs the data when it is the last observation in the BY group

Quick Exercise – 13 (Time: 5 mins)

- Find out the total income earned across occupations in the customer data

Simple Match-Merging

- Combine observations from two or more data sets into a single observation in a new data set according to the values of a common variable.
- When you match-merge, you use a MERGE statement rather than a SET statement to combine data sets.

General form, basic DATA step for match-merging:

DATA *output-SAS-data-set;*
MERGE *SAS-data-set-1 SAS-data-set-2*;
BY <DESCENDING> *variable(s);*
RUN;

Simple Match-Merging

- While Merging the datasets
  - Each input data set in the MERGE statement **must** be sorted in order of the values of the BY variable(s), or it must have an appropriate index.
  - Each BY variable must have the same type in all data sets to be merged.

- By default, DATA step match-merging combines all observations in all input data sets.

- To exclude unmatched observations from your output data set, you can use the **IN= data set option** and the **subsetting IF statement** in your DATA step.

IN = option

- IN = option assign temporary reference variables to the datasets being merged.
- Using an IF condition and the temporary variables, we can control the data we get in the output dataset.
  - Example:

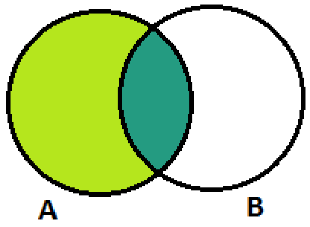
```
data work.sample;
merge work.test1(in = a) work.test2(in = b);
by name;
IF a;
run;
```

IN = option

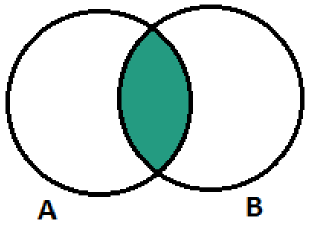
- We can use various combinations in the IF condition to produce different outputs.

Condition	Output
IF a (equivalent to “IF a = 1”)	Output contains all the observations from reference a
IF a and b	Output contains observations which are common in a and b
IF ~b (equivalent to “IF not b”)	Output contains observations which are present “only” in a
IF a or b	Output contains observations which are present in either of the datasets

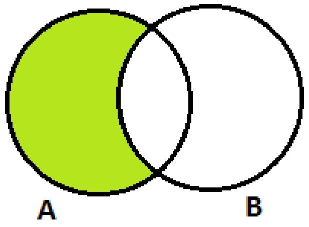
IF a



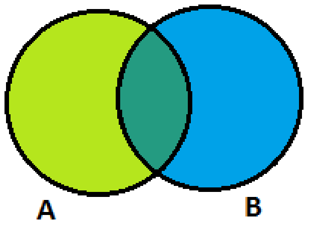
IF a and b



IF not b



IF a or b



Checks to be performed before Merging

- Check that the merge key is unique. The merge key **MUST NOT** repeat in more than one of the datasets that are being merged. This will avoid "many to many" join
- Treat appropriately (rename, drop, etc.), the common variables other than the merge keys in the merging datasets
- Lengths and informats of merge keys should be the same, for appropriate matching of the By variable values
- Do the appropriate join - inner or left join or full join

Exercise on Merging:

- Create a master dataset at the **Customer** level.
  - Which data set will you use as a base?
  - What are the checks that you have to perform before proceeding with merging?
  - In which sequence will you merge? (If any)
  - What are the different types of merge you will use at every stage?

- Create a master dataset at the **Card** level.
  - Which data set will you use as a base?
  - What are the checks that you have to perform before proceeding with merging?
  - In which sequence will you merge? (If any)
  - What are the different types of merge you will use at every stage?