Rotman

Master of Management Analytics

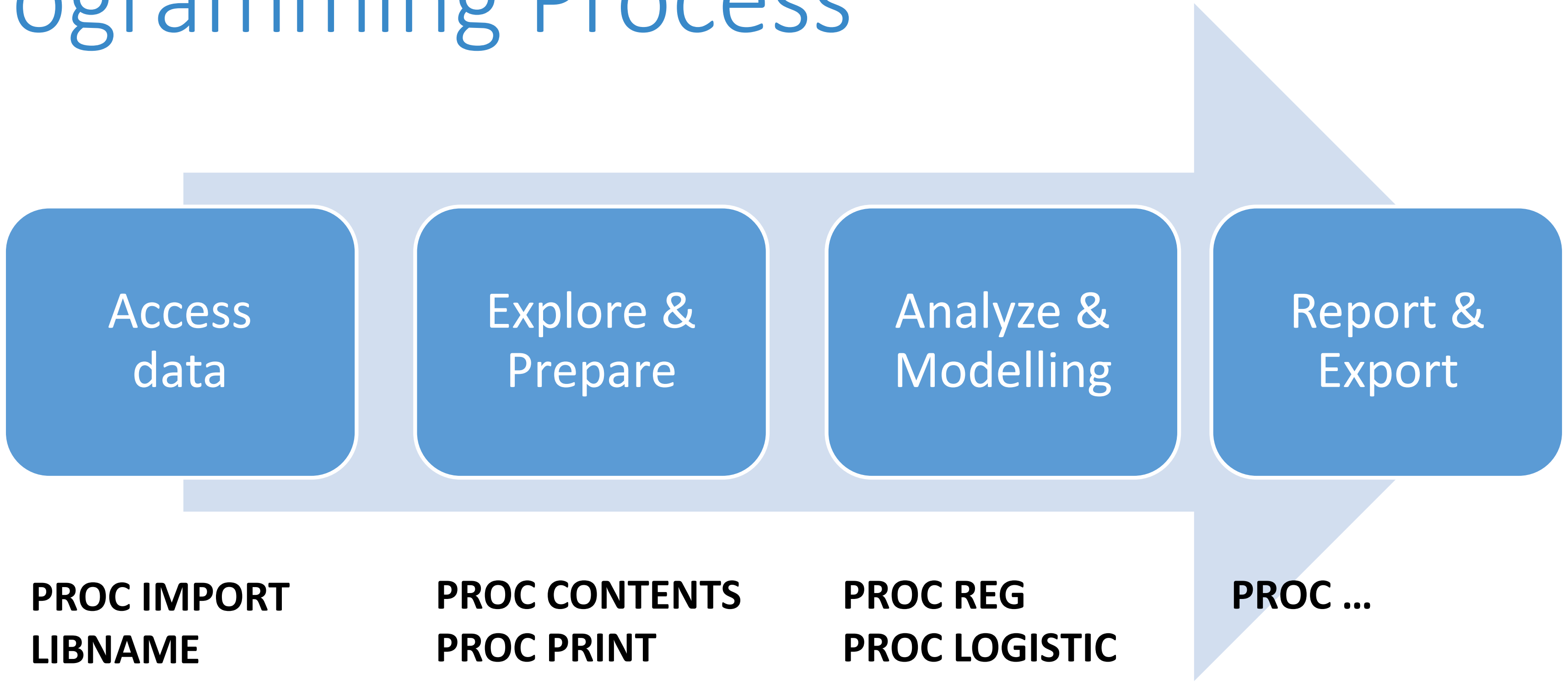
# INTRO TO SAS PROGRAMMING

Bootcamp SAS 2 (4 hours) – Part 2

August 22, 2019 Prepared by Jay Cao / TDMDL

Rotman School of Management
UNIVERSITY OF TORONTO

# SAS Programming Process



This workshop

PROC IMPORT
LIBNAME

PROC CONTENTS
PROC PRINT
PROC MEANS
PROC UNIVARIATE
PROC FREQ

DATA STEP
PROC SQL

PROC REG
PROC LOGISTIC

PROC …

PROC EXPORT

Ref. SAS Programming 1: Essentials

## DATA STEP and PROC SQL

- DATA STEP is the SAS way of manipulating data
  - Almost like a mini language itself (has it’s own conditional and loop syntax)
  - Think row-wise when using DATA STEP
  - Allan’s class

- We just learned SQL, so let’s leverage our SQL knowledge too
  - Think column-wise when using PROC SQL

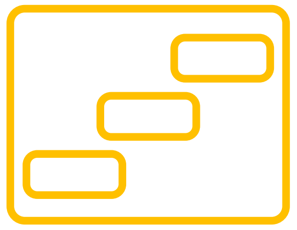
Ref. 1) DATA STEP document; 2) PROC SQL document

## PROC SQL (hands-on)

- Q1: Find all (unique) baseball division and league
  - Use **`sashelp.baseball`** dataset

- Q2: Calculate revenue by year and country
  - Import **`Orders.csv`** and **`OrderDetails.csv`** dataset
  - Use PROC SQL to merge the two imported table and do the calculation

## Data Step



```
DATA output-table;
   . . .
RUN;
```

filter rows and columns

compute new columns

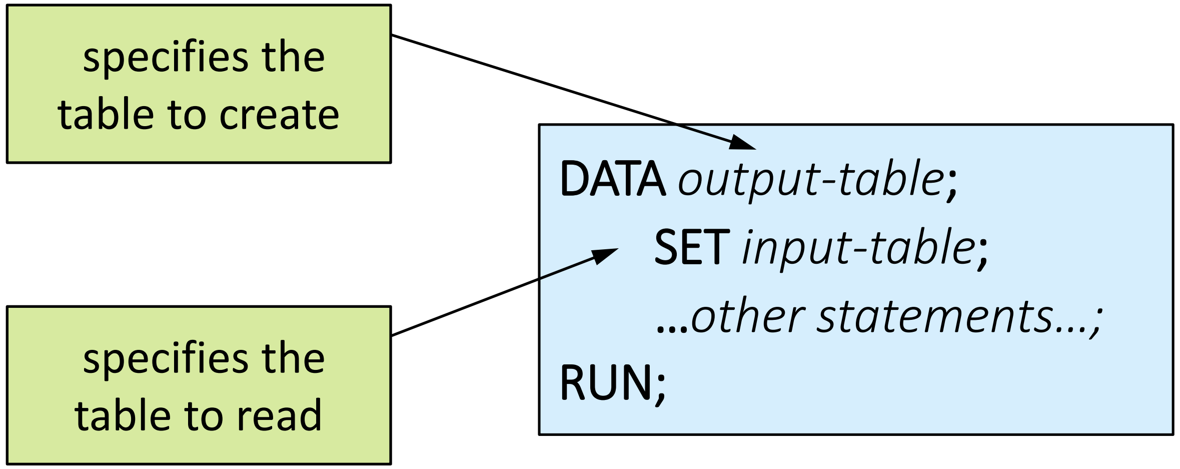
conditionally process

and much more …

The DATA step is a powerful tool to create, clean, and prepare your data!

Ref. SAS Programming 1: Essentials

# Data Step: Input and Output tables



DATA *output-table*;
    SET *input-table*;
    ...*other statements...;*
RUN;

```
data myclass;
    set sashelp.class;
run;
```

Ref. SAS Programming 1: Essentials

# Data Step Process

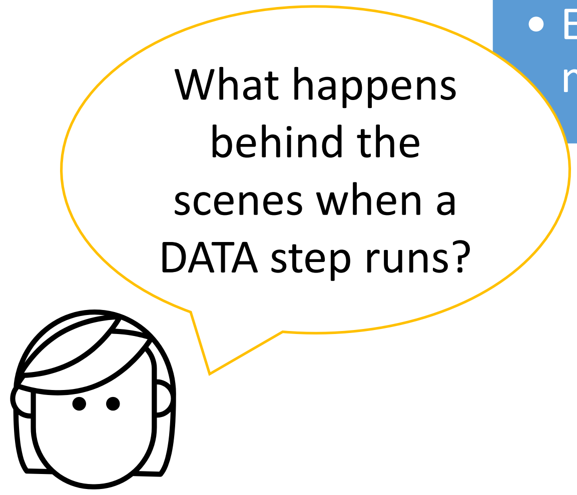
**Compilation**

- Check syntax for errors.
- Identify column attributes.
- Establish new table metadata.

→

**Execution**

- Read and write data.
- Perform data manipulations, calculations, and so on.



Ref. SAS Programming 1: Essentials

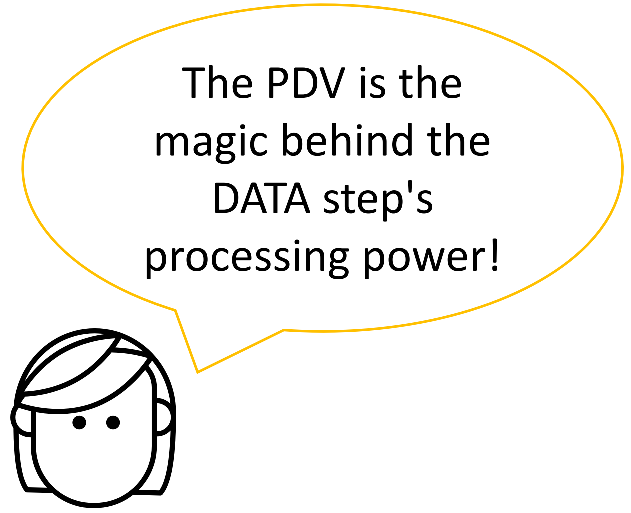
# Data Step Process: Compilation

**Compilation**

1) Check for syntax errors.
2) Create the *program data vector (PDV)*, which includes all columns and attributes.
3) Establish the specifications for processing data in the PDV during execution.
4) Create the descriptor portion of the output table.

**PDV**

| Season<br>N 8 | Name<br>$ 25 | StartDate<br>N 8 | Ocean<br>$ 8 |
|---|---|---|---|
| | | | |


The PDV is the magic behind the DATA step's processing power!

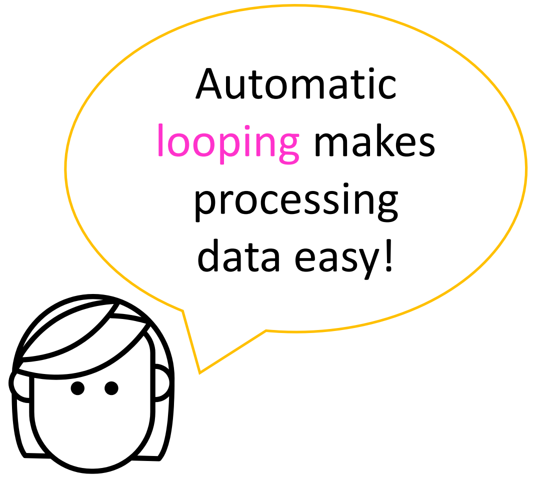
Ref. SAS Programming 2: Data Manipulation

## Data Step: Execution (Row-wise operation)

**Execution**

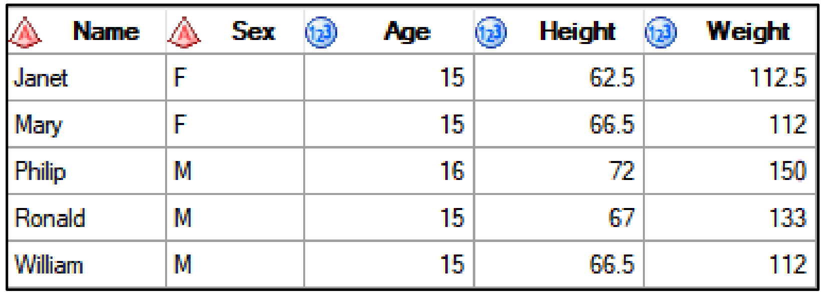
1) Initialize the PDV.
2) Read a row from the input table into the PDV.
3) Sequentially process statements and update values in the PDV.
4) At the end of the step, write the contents of the PDV to the output table.
5) Return to the top of the DATA step to read the next row.

```
data myclass;
     set sashelp.class;
     ...other statements...
run;
```



Ref. SAS Programming 2: Data Manipulation

# Filtering Rows (1)

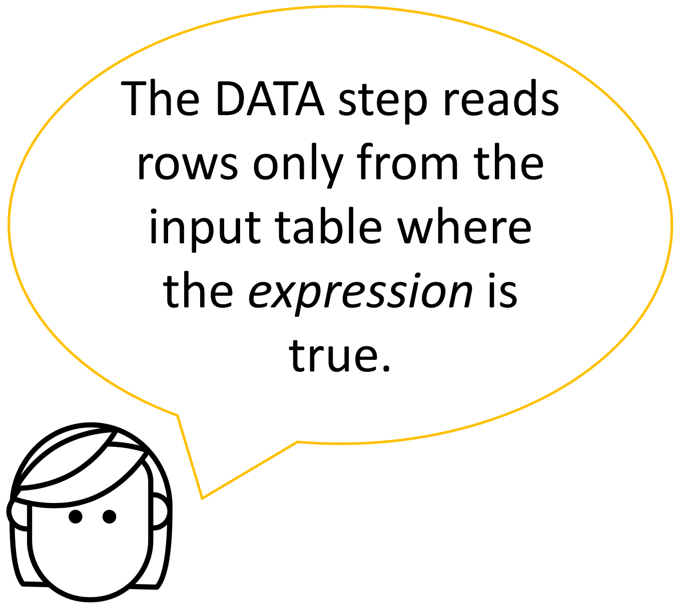


| Name | Sex | Age | Height | Weight |
|---|---|---|---|---|
| Janet | F | 15 | 62.5 | 112.5 |
| Mary | F | 15 | 66.5 | 112 |
| Philip | M | 16 | 72 | 150 |
| Ronald | M | 15 | 67 | 133 |
| William | M | 15 | 66.5 | 112 |

DATA *output-table*;
    SET *input-table*;
    WHERE *expression*;
RUN;

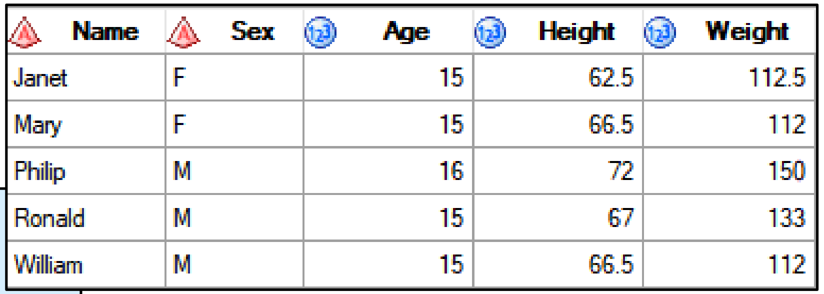
filters rows based on the expression

```
data myclass;
    set sashelp.class;
    where age >= 15;
run;
```


The DATA step reads rows only from the input table where the *expression* is true.

Ref. SAS Programming 1: Essentials

## Filtering Rows (2)

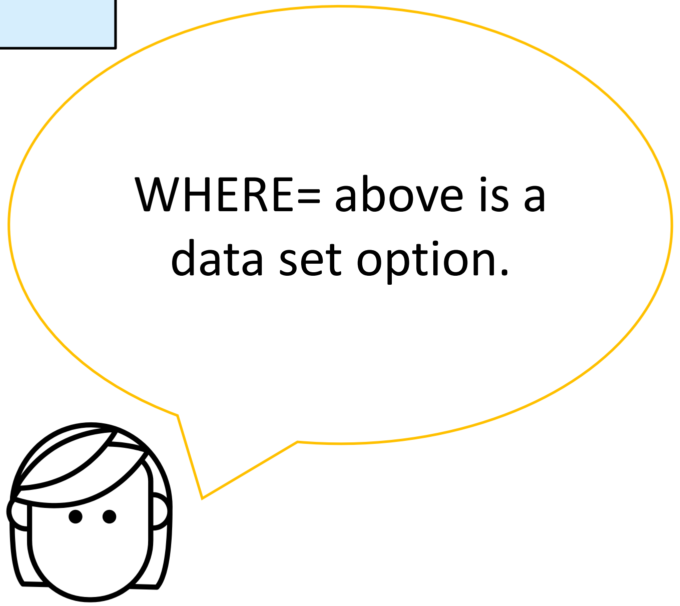


| Name | Sex | Age | Height | Weight |
|---|---|---|---|---|
| Janet | F | 15 | 62.5 | 112.5 |
| Mary | F | 15 | 66.5 | 112 |
| Philip | M | 16 | 72 | 150 |
| Ronald | M | 15 | 67 | 133 |
| William | M | 15 | 66.5 | 112 |

DATA *output-table*;
    SET *input-table (WHERE=(expression))*;
RUN;

Dataset WHERE= option: filters rows based on the expression

```
data myclass;
    set sashelp.class(where=(age>=15));
run;
```

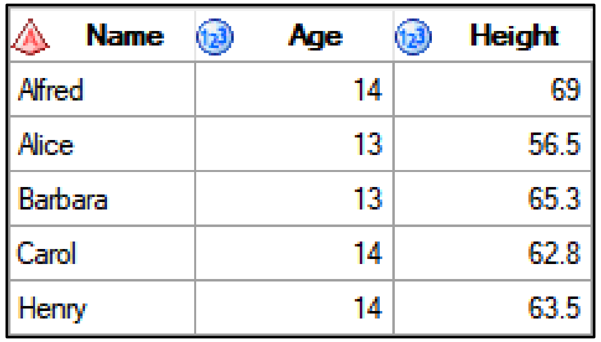


Ref. Ref. 1) SAS Programming 2: Data Manipulation; 2) WHERE= data set option

# Subsetting Columns (1)

DROP *col-name <col-name>*;

KEEP *col-name <col-name>*;

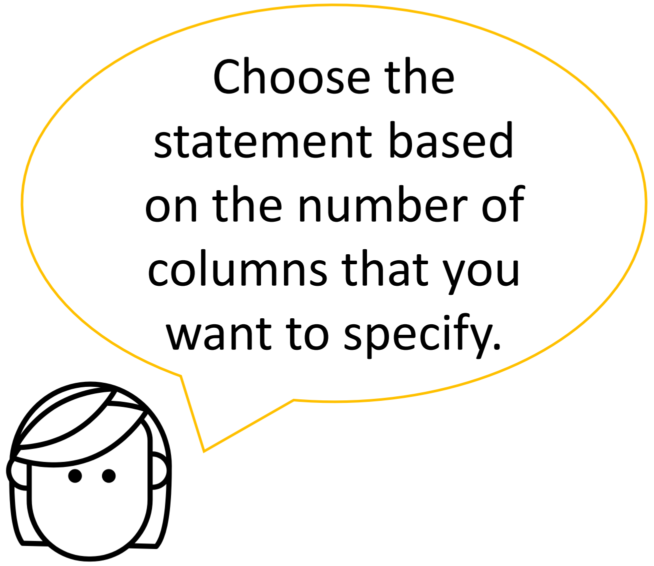


| Name | Age | Height |
|---|---|---|
| Alfred | 14 | 69 |
| Alice | 13 | 56.5 |
| Barbara | 13 | 65.3 |
| Carol | 14 | 62.8 |
| Henry | 14 | 63.5 |

```
data myclass;
    set sashelp.class;
    keep name age height;
    drop sex weight;
run;
```

These statements have the same result in the output table.

or


Choose the statement based on the number of columns that you want to specify.

Ref. 1) SAS Programming 1: Essentials; 2) KEEP and DROP statement

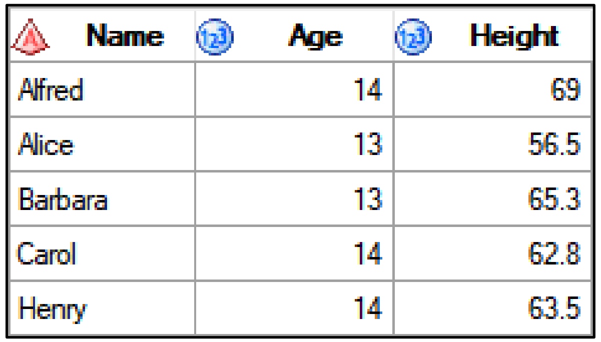
# Subsetting Columns (2)

SET *table*(DROP=*col1 col2...*)

SET *table*(KEEP=*col1 col2...*)

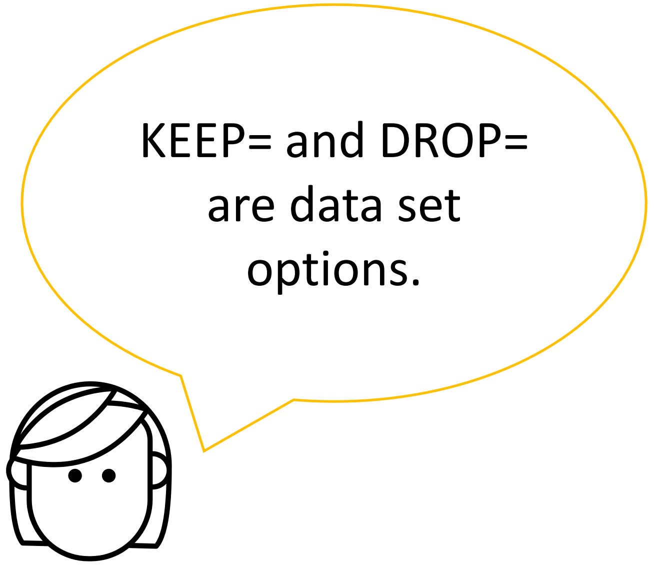
DATA *table*(DROP=*col1 col2...*)

DATA *table*(KEEP=*col1 col2...*)



| Name | Age | Height |
|---|---|---|
| Alfred | 14 | 69 |
| Alice | 13 | 56.5 |
| Barbara | 13 | 65.3 |
| Carol | 14 | 62.8 |
| Henry | 14 | 63.5 |

```
data myclass(drop=(sex weight));
    set sashelp.class;
run;
```



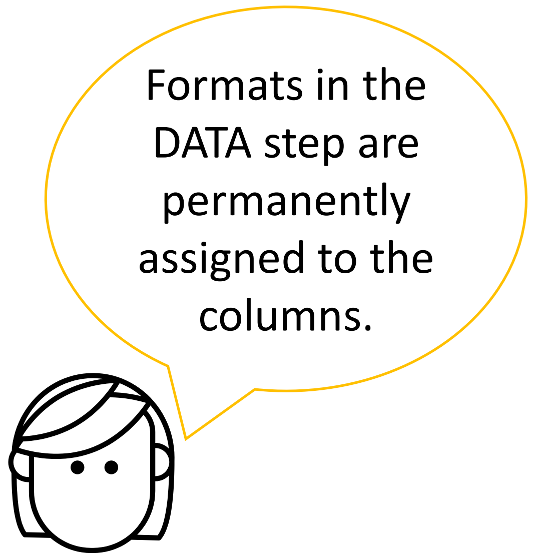
Ref. 1) SAS Programming 2: Data Manipulation; 2) KEEP= and DROP= data set options

## Formatting Columns (1)

```
DATA output-table;
        SET input-table;
        FORMAT col-name format;
RUN;
```

name of the column that you want to format

name of the format that you want to apply


Formats in the DATA step are permanently assigned to the columns.

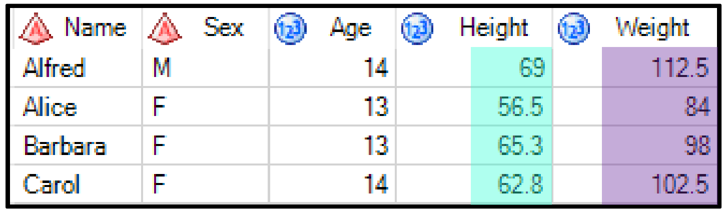
Ref. 1) SAS Programming 1: Essentials; 2) FORMAT statement

# Formatting Columns (2)

```
data myclass;
    set sashelp.class;
    format height 4.1 weight 3.;
run;
```

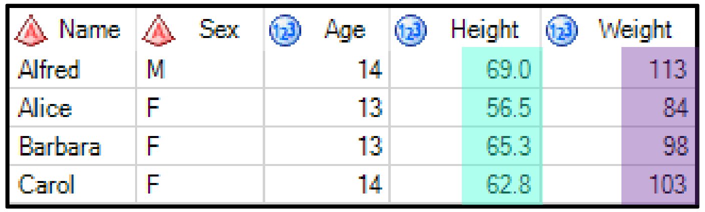
rounds values of height to one decimal place and weight to the nearest whole number

**sashelp.class**



| Name | Sex | Age | Height | Weight |
|---|---|---|---|---|
| Alfred | M | 14 | 69 | 112.5 |
| Alice | F | 13 | 56.5 | 84 |
| Barbara | F | 13 | 65.3 | 98 |
| Carol | F | 14 | 62.8 | 102.5 |

**myclass**



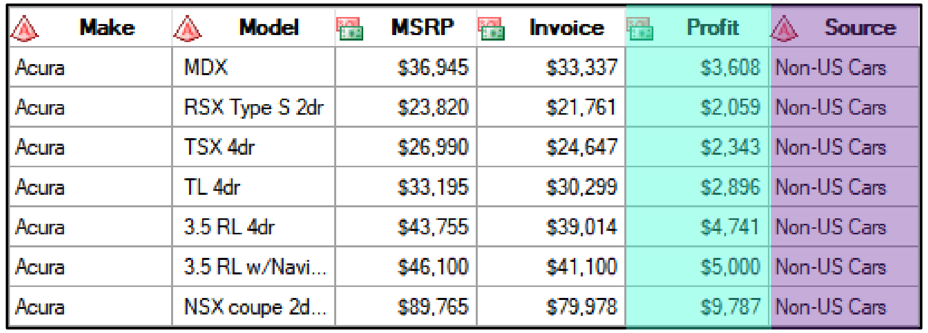
| Name | Sex | Age | Height | Weight |
|---|---|---|---|---|
| Alfred | M | 14 | 69.0 | 113 |
| Alice | F | 13 | 56.5 | 84 |
| Barbara | F | 13 | 65.3 | 98 |
| Carol | F | 14 | 62.8 | 103 |

Ref. 1) SAS Programming 1: Essentials; 2) FORMAT statement

# Filter Rows and Subset Cols (hands-on)

- Q: list Intel stock monthly open price after 2003
  - Use the **`help.stocks`** dataset
  - Only display **`stock`**, **`date`**, and **`open`** columns
  - Filter out date before 2004 (hint: **`Date>="01Jan2004"d`**)

# Creating New Columns (1)



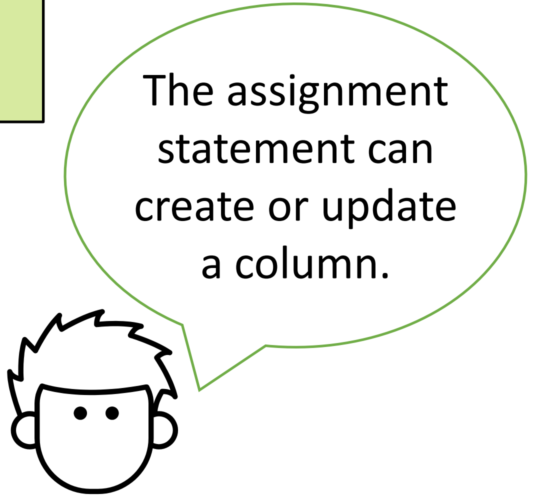
| Make | Model | MSRP | Invoice | Profit | Source |
|---|---|---|---|---|---|
| Acura | MDX | $36,945 | $33,337 | $3,608 | Non-US Cars |
| Acura | RSX Type S 2dr | $23,820 | $21,761 | $2,059 | Non-US Cars |
| Acura | TSX 4dr | $26,990 | $24,647 | $2,343 | Non-US Cars |
| Acura | TL 4dr | $33,195 | $30,299 | $2,896 | Non-US Cars |
| Acura | 3.5 RL 4dr | $43,755 | $39,014 | $4,741 | Non-US Cars |
| Acura | 3.5 RL w/Navi... | $46,100 | $41,100 | $5,000 | Non-US Cars |
| Acura | NSX coupe 2d... | $89,765 | $79,978 | $9,787 | Non-US Cars |

DATA *output-table*;
        SET *input-table*;
        *new-column* = *expression*;
RUN;

arithmetic expression or constant

assignment statement

```
data cars_new;
    set sashelp.cars;
    where Origin ne "USA";
    Profit = MSRP-Invoice;
    Source = "Non-US Cars";
    format Profit dollar10.;
    keep Make Model MSRP Invoice Profit Source;
run;
```



Ref. SAS Programming 1: Essentials

## Creating New Columns (2)

| Functions |
|---|
| SUM (*num1, num2, ...*) |
| MEAN (*num1, num2, ...*) |
| MEDIAN (*num1, num2, ...*) |
| RANGE (*num1, num2, ...*) |
| MIN (*num1, num2, ...*) |
| MAX (*num1, num2, ...*) |
| N (*num1, num2, ...*) |
| NMISS (*num1, num2, ...*) |
| ... |

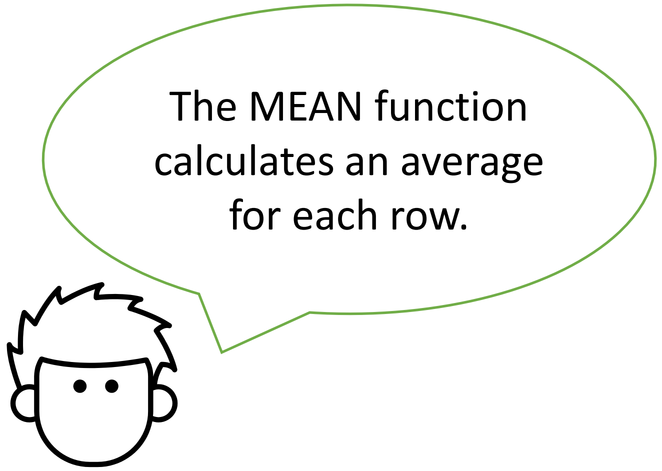
```
DATA output-table;
      SET input-table;
      new-column=function(arguments);
RUN;
```

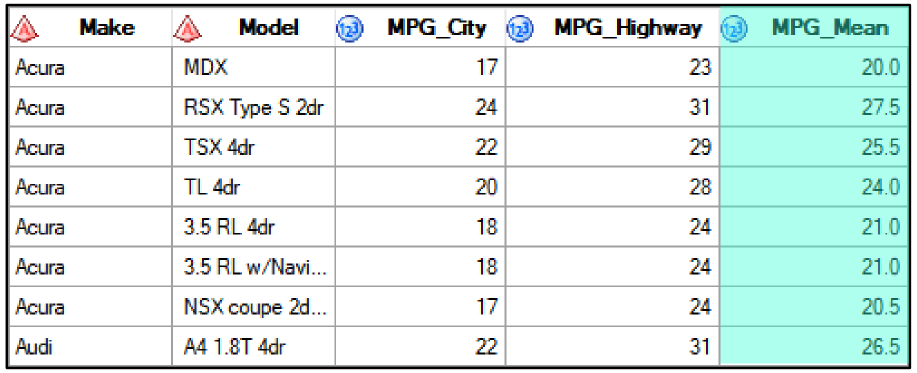
Functions can be used in an assignment statement to create or update a column.

Ref. 1) SAS Programming 1: Essentials; 2) Most Commonly Used Functions

# Creating New Columns (2)

```
data cars_new;
    set sashelp.cars;
    MPG_Mean=mean(MPG_City, MPG_Highway);
    format MPG_Mean 4.1;
    keep Make Model MPG_City MPG_Highway MPG_Mean;
run;
```





| Make | Model | MPG_City | MPG_Highway | MPG_Mean |
|---|---|---|---|---|
| Acura | MDX | 17 | 23 | 20.0 |
| Acura | RSX Type S 2dr | 24 | 31 | 27.5 |
| Acura | TSX 4dr | 22 | 29 | 25.5 |
| Acura | TL 4dr | 20 | 28 | 24.0 |
| Acura | 3.5 RL 4dr | 18 | 24 | 21.0 |
| Acura | 3.5 RL w/Navi... | 18 | 24 | 21.0 |
| Acura | NSX coupe 2d... | 17 | 24 | 20.5 |
| Audi | A4 1.8T 4dr | 22 | 31 | 26.5 |

Ref. SAS Programming 1: Essentials

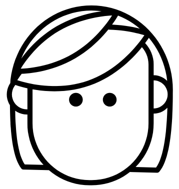
## Conditional (IF-THEN)

IF *expression* THEN *statement*;

defines a condition

any single SAS action

If *expression* is true, then execute the *statement*.



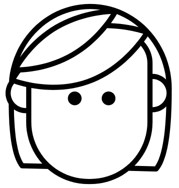
Ref. 1) SAS Programming 1: Essentials; 2) IF-THEN/ELSE Statement

## Conditional (IF-THEN/ELSE)

```
IF expression THEN statement;
<ELSE IF expression THEN statement;>
<ELSE IF expression THEN statement;>
ELSE statement;
```

If the others are not true, execute this statement.

If *expression* is true, then execute the *statement* and skip the rest.



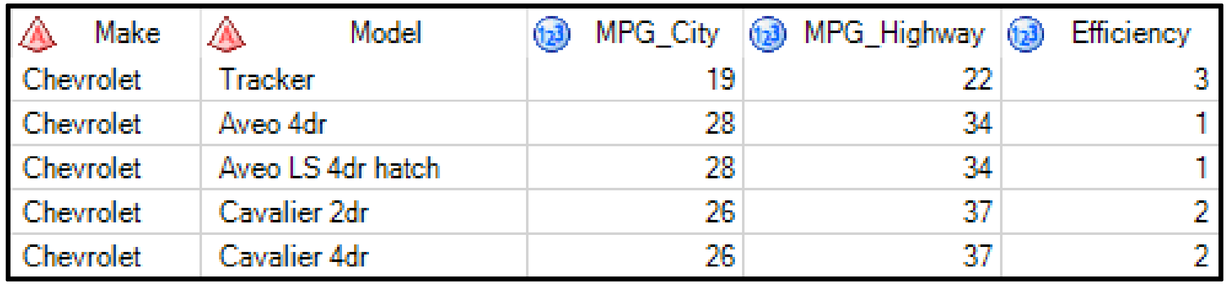
Ref. 1) SAS Programming 1: Essentials; 2) IF-THEN/ELSE Statement

# Conditional (IF-THEN/ELSE)

```
data cars2;
    set sashelp.cars;
    if MPG_City>26 and MPG_Highway>30 then Efficiency=1;
    else if MPG_City>20 and MPG_Highway>25 then Efficiency=2;
    else Efficiency=3;
    keep Make Model MPG_City MPG_Highway Efficiency;
run;
```

AND Both conditions must be true.

OR One condition must be true.

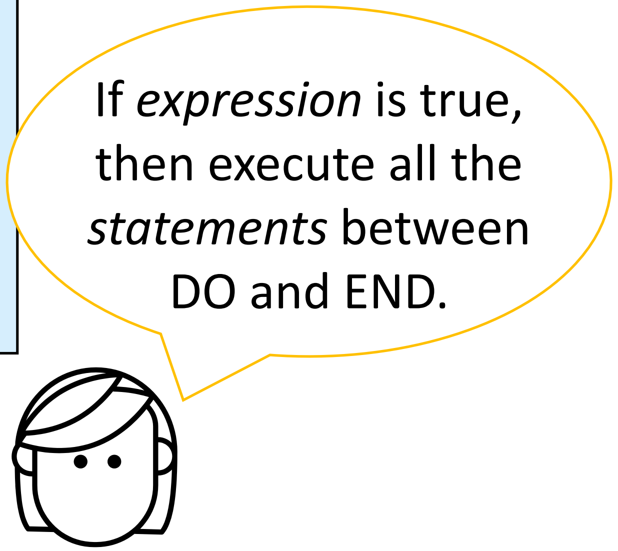


| Make | Model | MPG_City | MPG_Highway | Efficiency |
|---|---|---|---|---|
| Chevrolet | Tracker | 19 | 22 | 3 |
| Chevrolet | Aveo 4dr | 28 | 34 | 1 |
| Chevrolet | Aveo LS 4dr hatch | 28 | 34 | 1 |
| Chevrolet | Cavalier 2dr | 26 | 37 | 2 |
| Chevrolet | Cavalier 4dr | 26 | 37 | 2 |

Ref. SAS Programming 1: Essentials

## Conditional (IF-THEN/ELSE/DO)

```
IF expression THEN DO;
   <executable statements>
END;
ELSE IF expression THEN DO;
   <executable statements>
END;
ELSE DO;
   <executable statements>
END;
```


If *expression* is true, then execute all the *statements* between DO and END.

Ref. SAS Programming 1: Essentials

## Conditional (IF-THEN/ELSE/DO)

create two tables

DATA *table1 table2...*

```
data under40 over40;
     set sashelp.cars;
     keep Make Model MSRP Cost_Group;
     if MSRP<20000 then do;
        Cost_Group=1;
        output under40;
     end;
     else if MSRP<40000 then do;
        Cost_Group=2;
        output under40;
     end;
     else do;
        Cost_Group=3;
        output over40;
     end;
run;
```

conditionally output to one table

OUTPUT *table*;

Ref. SAS Programming 1: Essentials

# Conditional (IF-THEN/ELSE/DO)

```
data under40 over40;
     set sashelp.cars;
     keep Make Model MSRP Cost_Group;
     if MSRP<20000 then do;
        Cost_Group=1;
        output under40;
     end;
     else if MSRP<40000 then do;
        Cost_Group=2;
        output under40;
     end;
     else do;
        Cost_Group=3;
        output over40;
     end;
run;
```

create two tables

DATA *table1 table2...*

conditionally output to one table

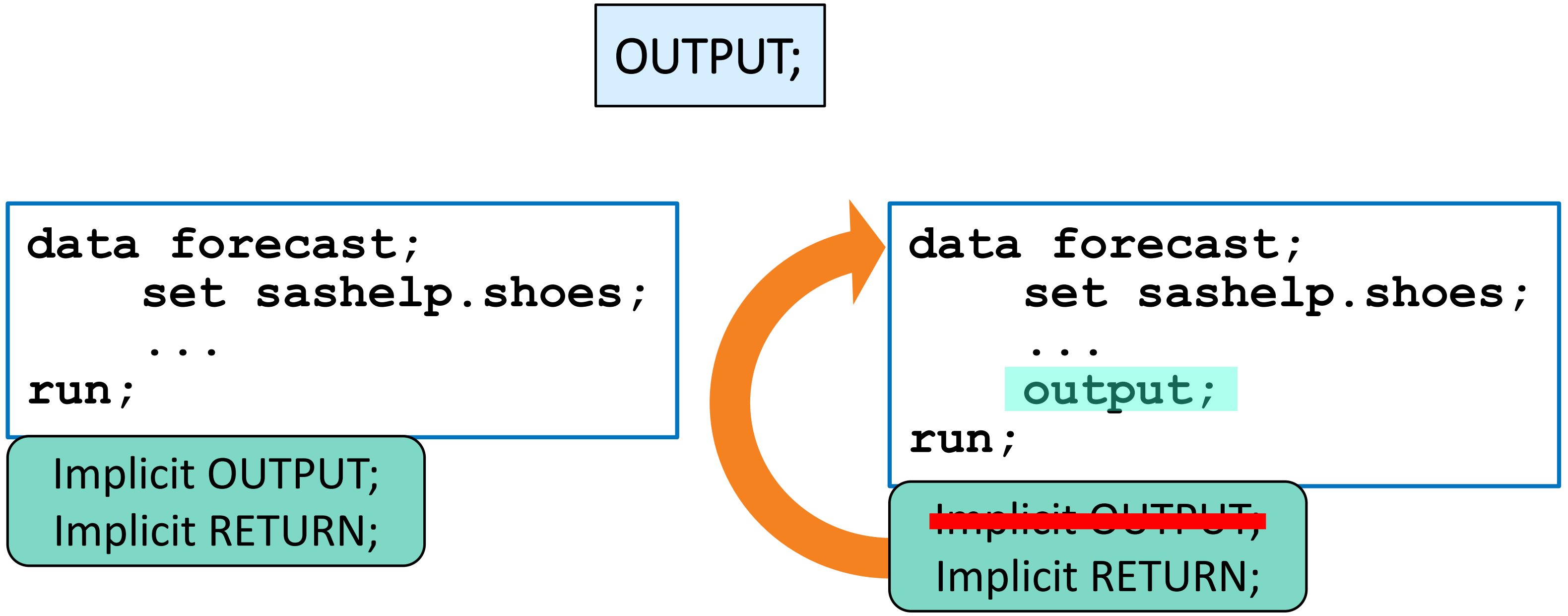
OUTPUT *table*;

Ref. SAS Programming 1: Essentials

## Conditional (IF-THEN/ELSE/DO) (hands-on)

- Try the **`sashelp.cars`** example

## Explicit Output



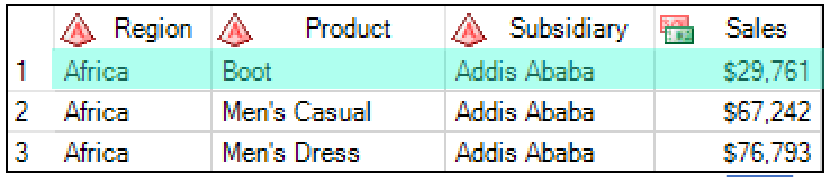
```
data forecast;
    set sashelp.shoes;
    ...
run;
```

```
data forecast;
    set sashelp.shoes;
    ...
    output;
run;
```

Ref. SAS Programming 2: Data Manipulation

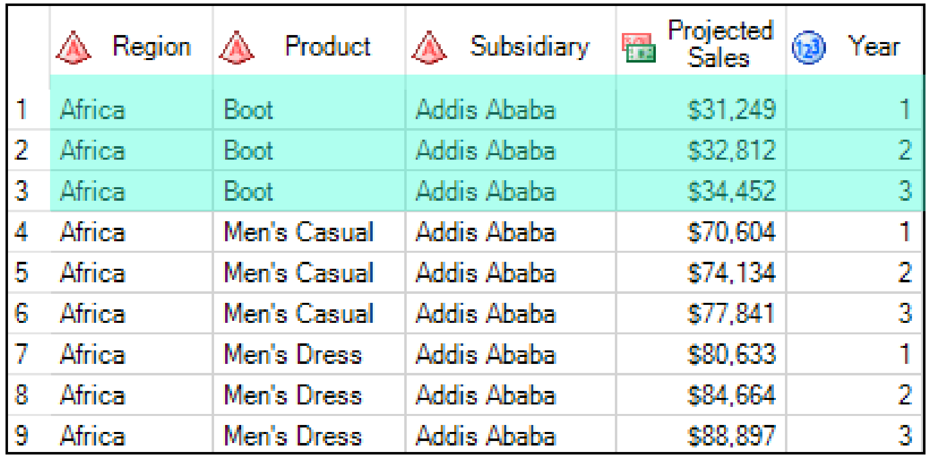
# Controlling Output

**sashelp.shoes**



| | Region | Product | Subsidiary | Sales |
|---|---|---|---|---|
| 1 | Africa | Boot | Addis Ababa | $29,761 |
| 2 | Africa | Men's Casual | Addis Ababa | $67,242 |
| 3 | Africa | Men's Dress | Addis Ababa | $76,793 |

**forecast**



| | Region | Product | Subsidiary | Projected Sales | Year |
|---|---|---|---|---|---|
| 1 | Africa | Boot | Addis Ababa | $31,249 | 1 |
| 2 | Africa | Boot | Addis Ababa | $32,812 | 2 |
| 3 | Africa | Boot | Addis Ababa | $34,452 | 3 |
| 4 | Africa | Men's Casual | Addis Ababa | $70,604 | 1 |
| 5 | Africa | Men's Casual | Addis Ababa | $74,134 | 2 |
| 6 | Africa | Men's Casual | Addis Ababa | $77,841 | 3 |
| 7 | Africa | Men's Dress | Addis Ababa | $80,633 | 1 |
| 8 | Africa | Men's Dress | Addis Ababa | $84,664 | 2 |
| 9 | Africa | Men's Dress | Addis Ababa | $88,897 | 3 |

Use the DATA step to create a table that includes a sales forecast for each of the next three years.

Ref. SAS Programming 2: Data Manipulation

## Controlling Output

```
data forecast;
    set sashelp.shoes;
    keep Region Product Subsidiary Year ProjectedSales;
    format ProjectedSales dollar10.;
    Year=1;
    ProjectedSales=Sales*1.05;
    output;
    Year=2;
    ProjectedSales=ProjectedSales*1.05;
    output;
    Year=3;
    ProjectedSales=ProjectedSales*1.05;
    output;
run;
```

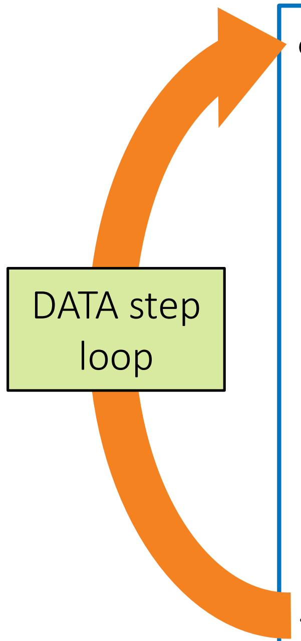
3*395 =
1,185 rows

Ref. SAS Programming 2: Data Manipulation

## Controlling Output (hands-on)

- Try the **`sashelp.shoes`** example
  - Do you find part of the code is quite repetitive? Can we do better?

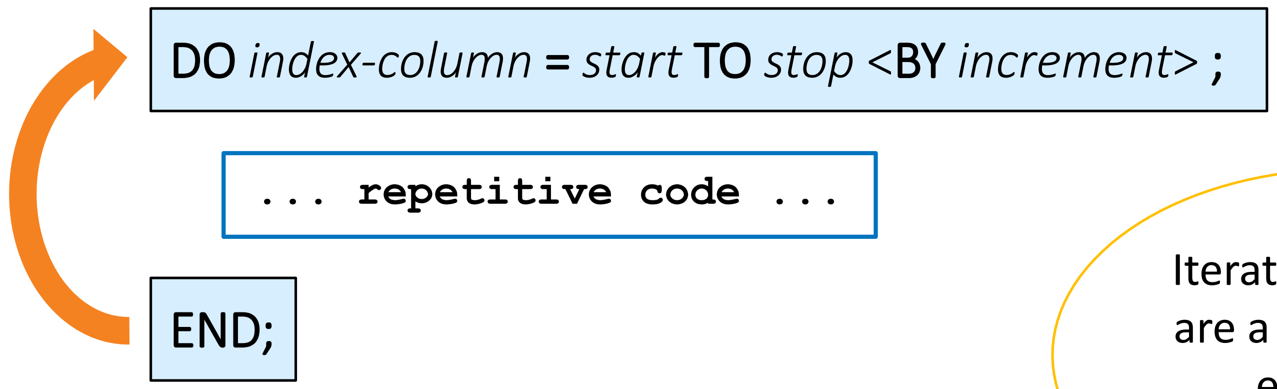
## Processing Repetitive Code



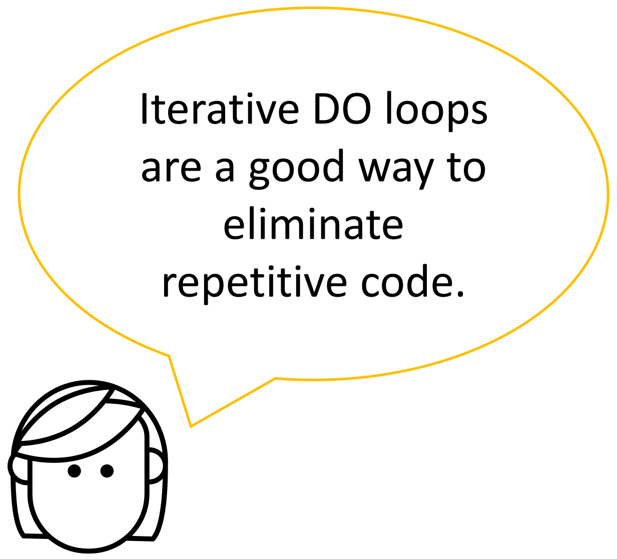
```
data forecast;
    set sashelp.shoes;
    keep Region Product Subsidiary Year ProjectedSales;
    format ProjectedSales dollar10.;
    Year=1;
    ProjectedSales=Sales*1.05;
    output;
    Year=2;
    ProjectedSales=ProjectedSales*1.05;
    output;
    Year=3;
    ProjectedSales=ProjectedSales*1.05;
    output;
run;
```

Ref. SAS Programming 2: Data Manipulation

# Iterative Do Loop



```
DO index-column = start TO stop <BY increment> ;
    ... repetitive code ...
END;
```


Iterative DO loops are a good way to eliminate repetitive code.

Ref. SAS Programming 2: Data Manipulation

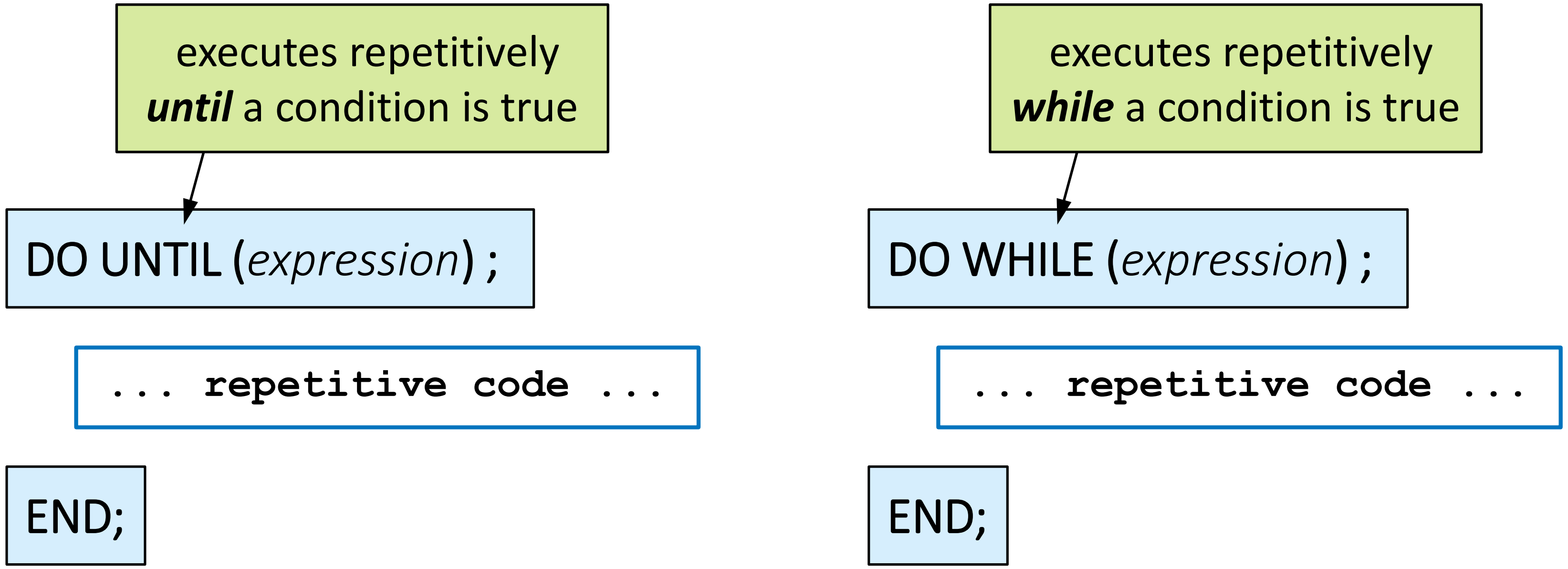
## Executing an Iterative Do Loop

rename= is a dataset option

```
data forecast;
    set sashelp.shoes(rename=(Sales=ProjectedSales));
    keep Region Product Subsidiary Year ProjectedSales;
    format ProjectedSales dollar10.;
    do Year = 1 to 3;
       ProjectedSales=ProjectedSales*1.05;
       output;
    end;
run;
```

Ref. SAS Programming 2: Data Manipulation

# Conditional Do Loops


executes repetitively *until* a condition is true
executes repetitively *while* a condition is true

```
DO UNTIL (expression) ;
    ... repetitive code ...
END;
```

```
DO WHILE (expression) ;
    ... repetitive code ...
END;
```

Ref. SAS Programming 2: Data Manipulation

## Iterative Do Loop (hands-on)

- Try the **`sashelp.shoes`** example with Do Loop

## Creating an Accumulating Column

RETAIN *column <initial-value>*;

```
data houston2017;
    set pg2.weather_houston;
    retain YTDRain 0;
    YTDRain=YTDRain+DailyRain;
run;
```

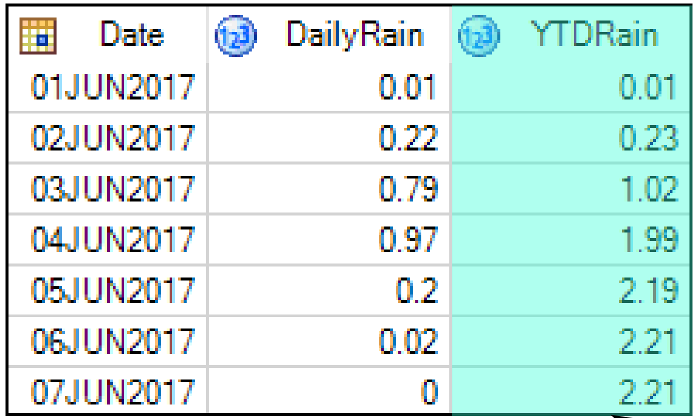
**Retain statement**

1) retains the value each time that the PDV reinitializes

2) assigns an initial value

An **assignment statement**

- creates **YTDRain** and sets the initial value to 0
- retains **YTDRain**
- adds **DailyRain** to **YTDRain** for each row
- **In the next iteration of data step, the retain statement is skipped**



| Date | DailyRain | YTDRain |
|---|---|---|
| 01JUN2017 | 0.01 | 0.01 |
| 02JUN2017 | 0.22 | 0.23 |
| 03JUN2017 | 0.79 | 1.02 |
| 04JUN2017 | 0.97 | 1.99 |
| 05JUN2017 | 0.2 | 2.19 |
| 06JUN2017 | 0.02 | 2.21 |
| 07JUN2017 | 0 | 2.21 |

Create a new column that stores an accumulating total.

Ref. SAS Programming 2: Data Manipulation

# Creating an Accumulating Column

*The sum function: Sum();*

```
data houston2017;
    set pg2.weather_houston;
    retain YTDRain 0;
    YTDRain=sum(YTDRain, DailyRain);
run;
```

*The sum statement: Column + expression;*

```
data houston2017;
    set pg2.weather_houston;
    YTDRain + DailyRain;
run;
```

**No retain statement here!**

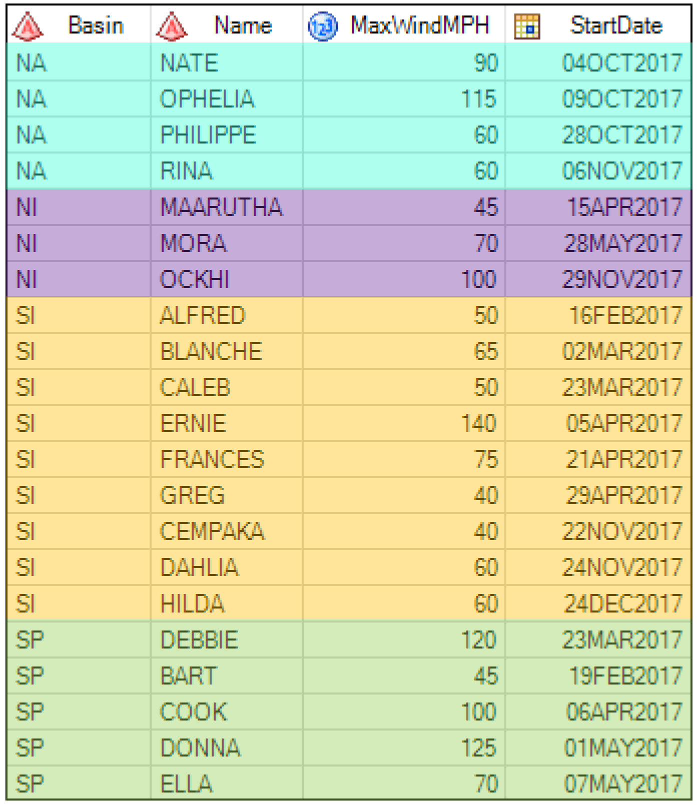
- creates **YTDRain** and sets the initial value to 0
- retains **YTDRain**
- adds **DailyRain** to **YTDRain** for each row
- ignores missing values

Ref. SAS Programming 2: Data Manipulation

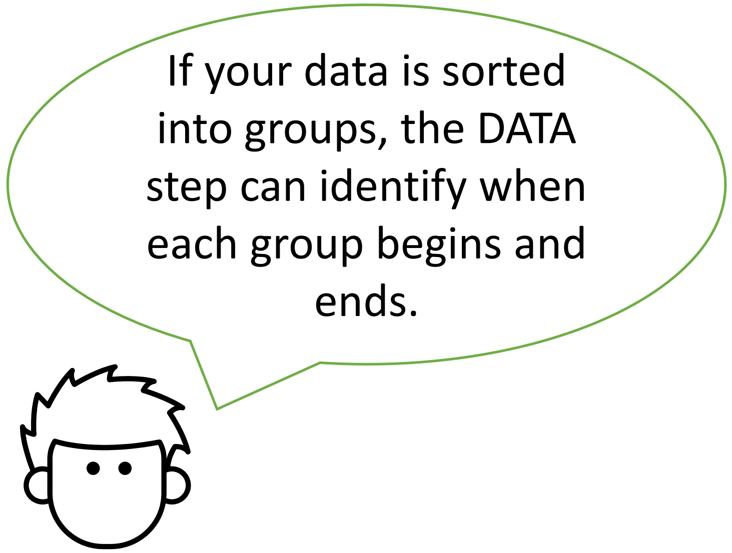
## Creating an Accumulating Column (hands-on)

- Calculate IBM stocks monthly accumulated volumes in 2005
  - Use **`sashelp.stocks`** data set
  - Filter out IBM stock in 2005 and sort by date
  - Calculate monthly accumulated volumes

# Processing Data in Group



| Basin | Name | MaxWindMPH | StartDate |
|---|---|---|---|
| NA | NATE | 90 | 04OCT2017 |
| NA | OPHELIA | 115 | 09OCT2017 |
| NA | PHILIPPE | 60 | 28OCT2017 |
| NA | RINA | 60 | 06NOV2017 |
| NI | MAARUTHA | 45 | 15APR2017 |
| NI | MORA | 70 | 28MAY2017 |
| NI | OCKHI | 100 | 29NOV2017 |
| SI | ALFRED | 50 | 16FEB2017 |
| SI | BLANCHE | 65 | 02MAR2017 |
| SI | CALEB | 50 | 23MAR2017 |
| SI | ERNIE | 140 | 05APR2017 |
| SI | FRANCES | 75 | 21APR2017 |
| SI | GREG | 40 | 29APR2017 |
| SI | CEMPAKA | 40 | 22NOV2017 |
| SI | DAHLIA | 60 | 24NOV2017 |
| SI | HILDA | 60 | 24DEC2017 |
| SP | DEBBIE | 120 | 23MAR2017 |
| SP | BART | 45 | 19FEB2017 |
| SP | COOK | 100 | 06APR2017 |
| SP | DONNA | 125 | 01MAY2017 |
| SP | ELLA | 70 | 07MAY2017 |


If your data is sorted into groups, the DATA step can identify when each group begins and ends.

Ref. SAS Programming 2: Data Manipulation

## Processing Data in Group

```
PROC SORT DATA=input-table
              <OUT=sorted-output-table>;
      BY <DESCENDING> col-name(s);
RUN;
```

sorts the table into groups

```
DATA output-table;
      SET sorted-output-table;
      BY <DESCENDING> col-name(s);
RUN;
```

processes the data in the sorted table by groups

Ref. SAS Programming 2: Data Manipulation

## Processing Data in Group

```
data storm2017_max;
    set storm2017_sort;
    by Basin;
run;
```

First.*bycol*

Last.*bycol*

The BY statement creates **First./Last.** variables in the PDV that can be used to identify when each BY group begins and ends.

**PDV**

| *...other columns...* | Basin | D First.Basin | D Last.Basin |
|---|---|---|---|
| | | | |

Ref. SAS Programming 2: Data Manipulation

# Processing Data in Group

**PDV**

| *...other columns...* | Basin | D First.Basin | D Last.Basin |
|---|---|---|---|
| | NA | 1 | 0 |

first row where **Basin** is *NA*

**PDV**

| *...other columns...* | Basin | D First.Basin | D Last.Basin |
|---|---|---|---|
| | NA | 0 | 0 |

subsequent rows where **Basin** is *NA*

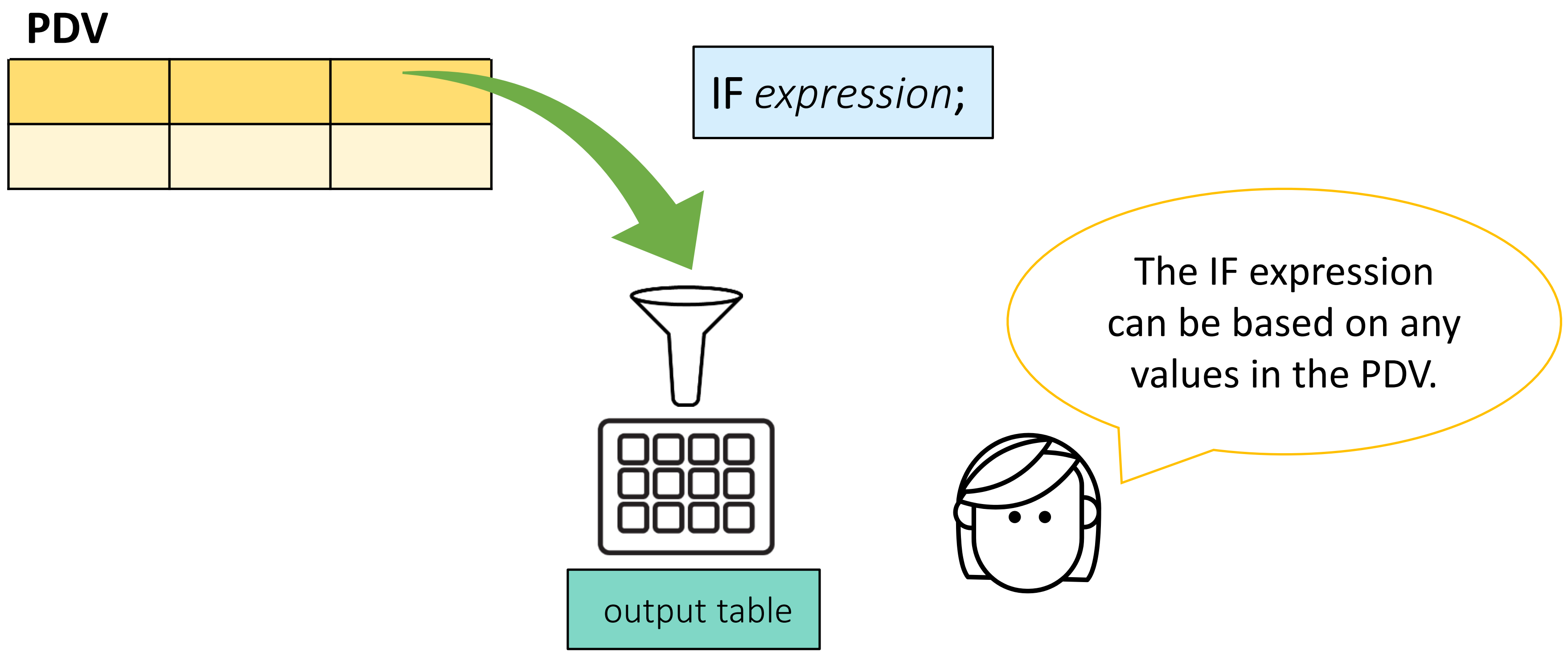
**PDV**

| *...other columns...* | Basin | D First.Basin | D Last.Basin |
|---|---|---|---|
| | NA | 0 | 1 |

last row where **Basin** is *NA*

Ref. SAS Programming 2: Data Manipulation

# Processing Data in Group



# Processing Data in Group

true

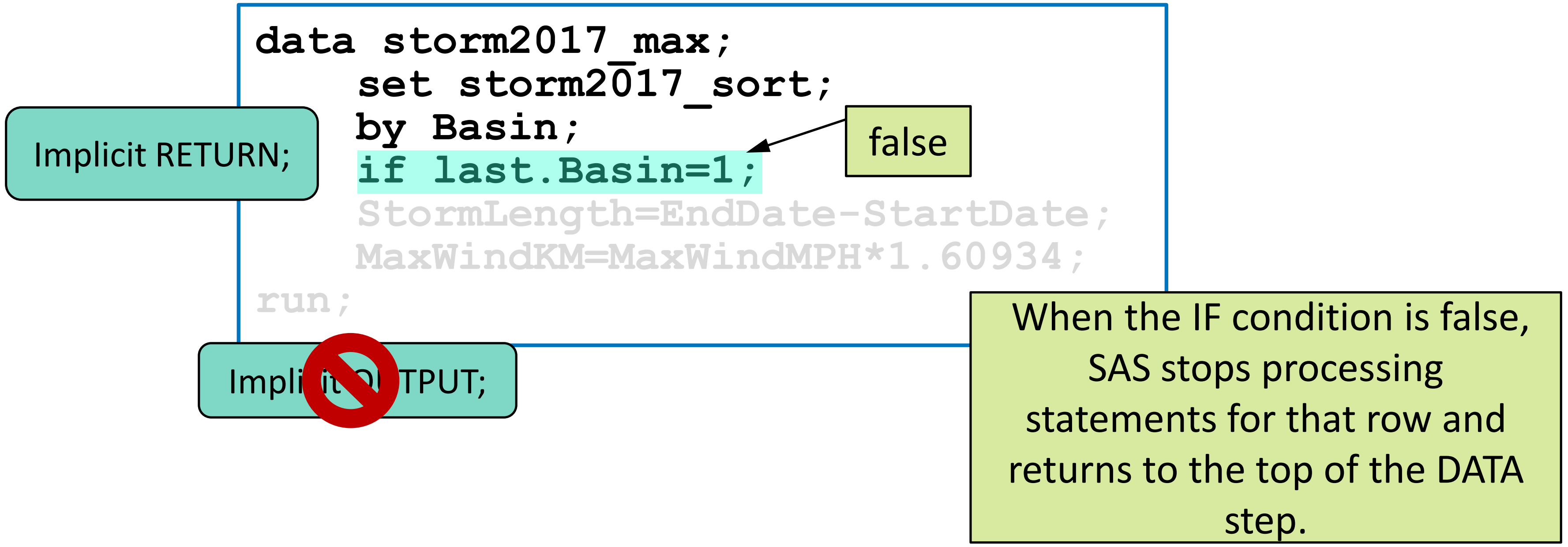
```
data storm2017_max;
    set storm2017_sort;
    by Basin;
    if last.Basin=1;
    StormLength=EndDate-StartDate;
    MaxWindKM=MaxWindMPH*1.60934;
run;
```

When the IF condition is true, SAS continues processing statements for that row.

Implicit OUTPUT;
Implicit RETURN;

Ref. SAS Programming 2: Data Manipulation

# Processing Data in Group



```
data storm2017_max;
    set storm2017_sort;
    by Basin;
    if last.Basin=1;
    StormLength=EndDate-StartDate;
    MaxWindKM=MaxWindMPH*1.60934;
run;
```

When the IF condition is false, SAS stops processing statements for that row and returns to the top of the DATA step.

Ref. SAS Programming 2: Data Manipulation

## Processing Data in Group (hands on)

- Q1: Calculate IBM stock's volumes by year
  - Use **`sashelp.stocks`** data set
  - Filter out IBM sort by date
  - Calculate total volume by year

- Q2: Calculate IBM stock's monthly accumulated volumes by year
  - Use **`sashelp.stocks`** data set
  - Filter out IBM stock and sort by date
  - Calculate monthly accumulated volumes by year

## Combing Datasets

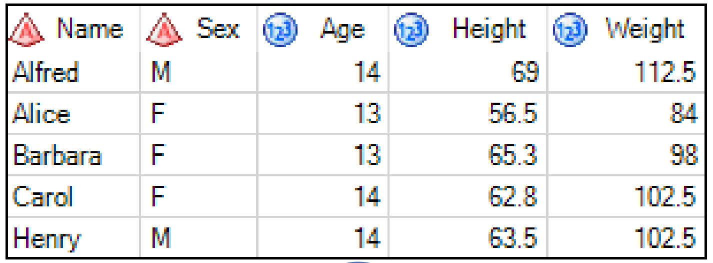
- DATA STEP can be used to join data sets
  - Methods of Combining SAS Data Sets: Concatenating, Merging, etc.
  - Merging SAS Data Sets
  - DATA STEP merge vs PROC SQL join

- You will learn this part on your own
  - The links above and a few slides next can get you started
  - You will learn combing datasets in DATA STEP in Allan's class too

# Concatenating Tables with Matching Columns

```
data class_current;
    set sashelp.class pg2.class_new;
run;
```

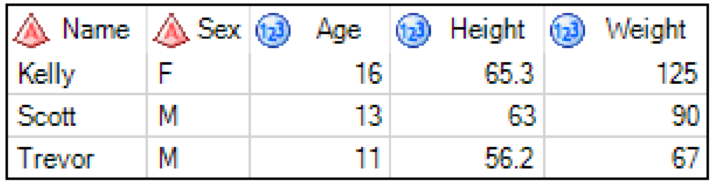
**sashelp.class**



| Name | Sex | Age | Height | Weight |
|---|---|---|---|---|
| Alfred | M | 14 | 69 | 112.5 |
| Alice | F | 13 | 56.5 | 84 |
| Barbara | F | 13 | 65.3 | 98 |
| Carol | F | 14 | 62.8 | 102.5 |
| Henry | M | 14 | 63.5 | 102.5 |

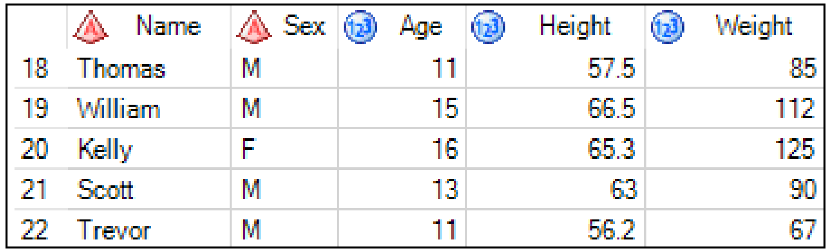
+

**pg2.class_new**



| Name | Sex | Age | Height | Weight |
|---|---|---|---|---|
| Kelly | F | 16 | 65.3 | 125 |
| Scott | M | 13 | 63 | 90 |
| Trevor | M | 11 | 56.2 | 67 |

**class_current**

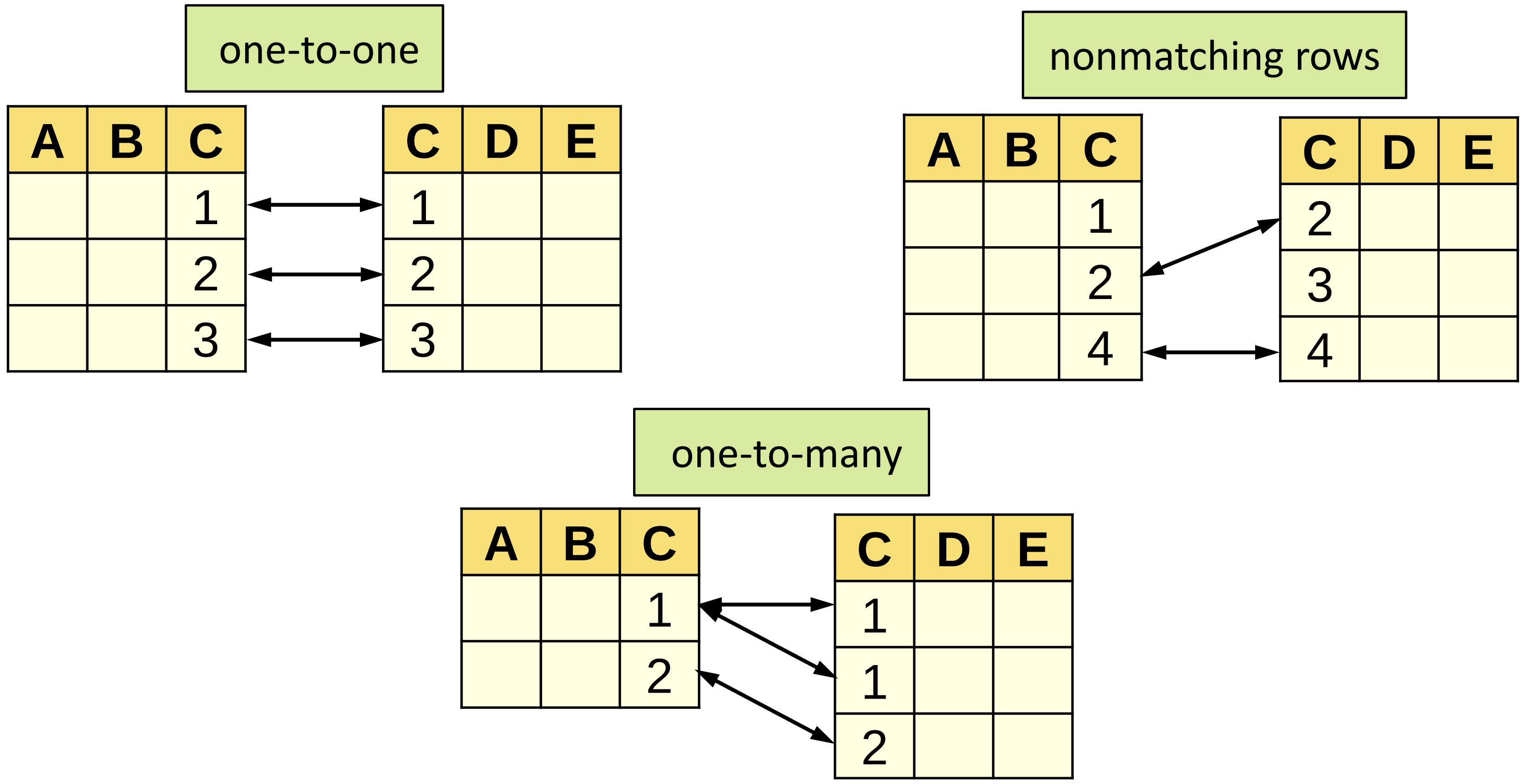


| | Name | Sex | Age | Height | Weight |
|---|---|---|---|---|---|
| 18 | Thomas | M | 11 | 57.5 | 85 |
| 19 | William | M | 15 | 66.5 | 112 |
| 20 | Kelly | F | 16 | 65.3 | 125 |
| 21 | Scott | M | 13 | 63 | 90 |
| 22 | Trevor | M | 11 | 56.2 | 67 |

rows from second table added after rows from the first table

Ref. SAS Programming 2: Data Manipulation

# Merging Tables



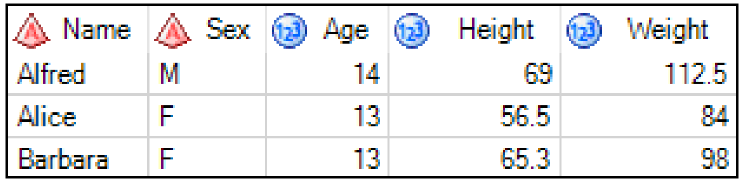
Ref. SAS Programming 2: Data Manipulation

# Merging Tables: One-to-One

```
data class2;
    merge sashelp.class pg2.class_teachers;
    by Name;
run;
```

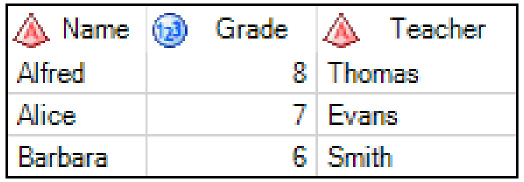
Columns are combined in the new table by matching values of **Name**.

**sashelp.class**



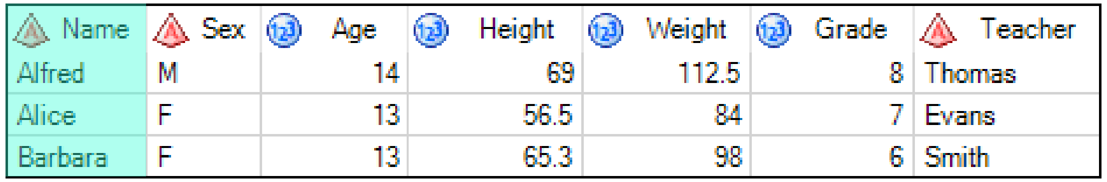
| Name | Sex | Age | Height | Weight |
|---|---|---|---|---|
| Alfred | M | 14 | 69 | 112.5 |
| Alice | F | 13 | 56.5 | 84 |
| Barbara | F | 13 | 65.3 | 98 |

**pg2.class_teachers**



| Name | Grade | Teacher |
|---|---|---|
| Alfred | 8 | Thomas |
| Alice | 7 | Evans |
| Barbara | 6 | Smith |

**class2**



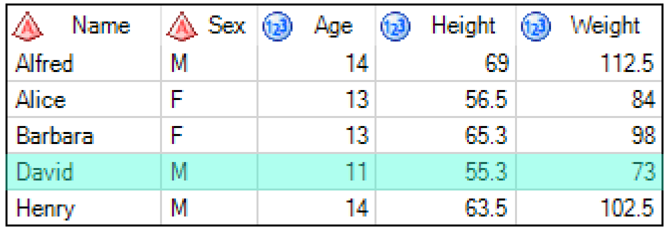
| Name | Sex | Age | Height | Weight | Grade | Teacher |
|---|---|---|---|---|---|---|
| Alfred | M | 14 | 69 | 112.5 | 8 | Thomas |
| Alice | F | 13 | 56.5 | 84 | 7 | Evans |
| Barbara | F | 13 | 65.3 | 98 | 6 | Smith |

Ref. SAS Programming 2: Data Manipulation

## Merging Tables: with Non-matching Rows

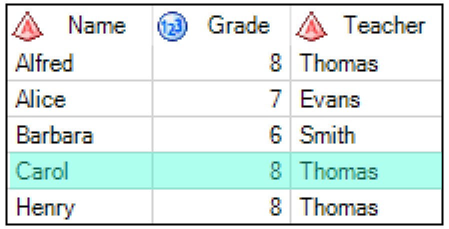
```
data class2;
    merge pg2.class_update pg2.class_teachers;
    by name;
run;
```

**class_update**



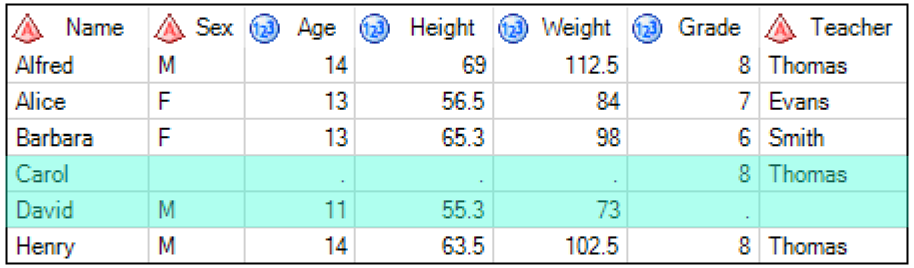
| Name | Sex | Age | Height | Weight |
|---|---|---|---|---|
| Alfred | M | 14 | 69 | 112.5 |
| Alice | F | 13 | 56.5 | 84 |
| Barbara | F | 13 | 65.3 | 98 |
| David | M | 11 | 55.3 | 73 |
| Henry | M | 14 | 63.5 | 102.5 |

**class_teachers**



| Name | Grade | Teacher |
|---|---|---|
| Alfred | 8 | Thomas |
| Alice | 7 | Evans |
| Barbara | 6 | Smith |
| Carol | 8 | Thomas |
| Henry | 8 | Thomas |

**class2**



| Name | Sex | Age | Height | Weight | Grade | Teacher |
|---|---|---|---|---|---|---|
| Alfred | M | 14 | 69 | 112.5 | 8 | Thomas |
| Alice | F | 13 | 56.5 | 84 | 7 | Evans |
| Barbara | F | 13 | 65.3 | 98 | 6 | Smith |
| Carol | | . | . | . | 8 | Thomas |
| David | M | 11 | 55.3 | 73 | . | |
| Henry | M | 14 | 63.5 | 102.5 | 8 | Thomas |

The new table includes matches and nonmatches.

Ref. SAS Programming 2: Data Manipulation

# DATA STEP Merge and PROC SQL Join

DATA step merge

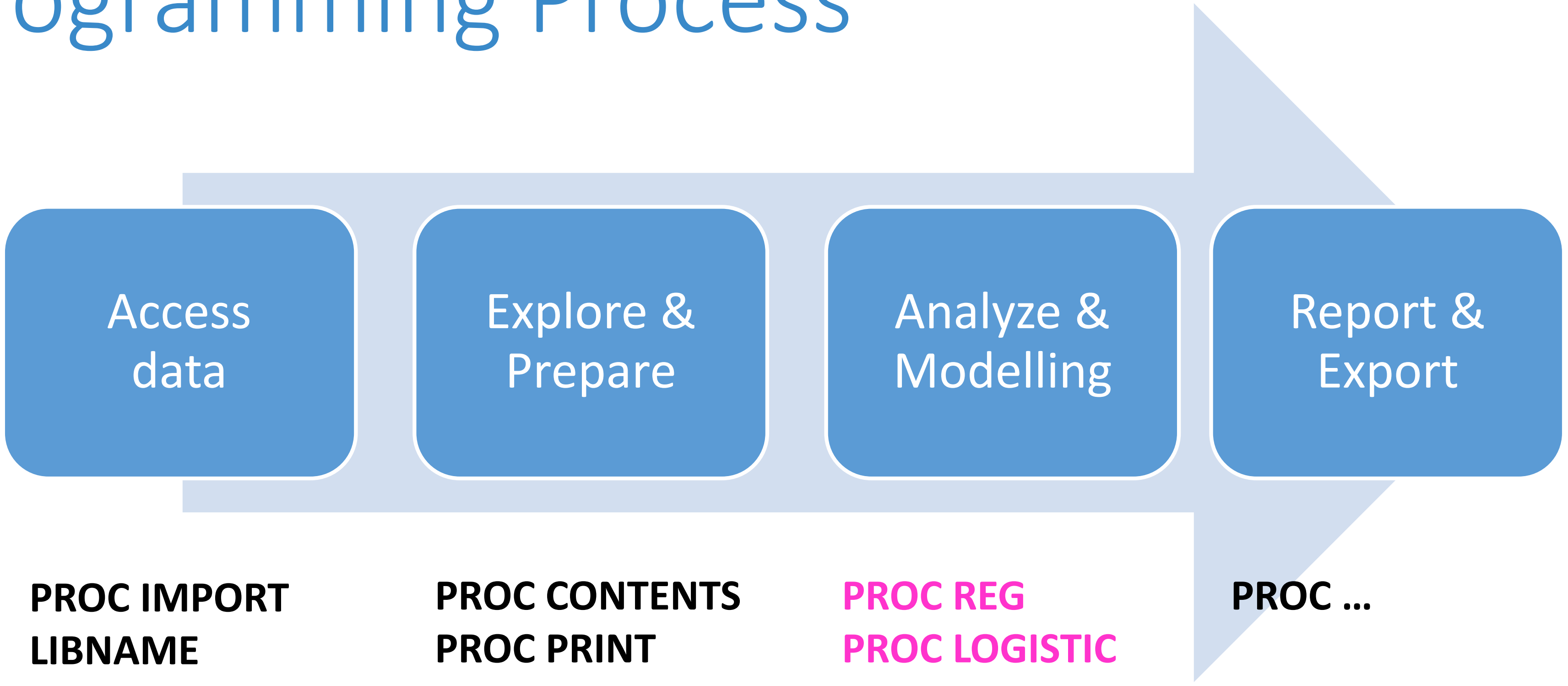
- requires sorted input data
- efficient, sequential processing
- can create multiple tables for matches and nonmatches in one step
- provides additional complex data processing syntax

PROC SQL join

- does not require sorted data
- matching columns do not need the same name
- easy to define complex matching criteria between multiple tables in a single query
- can be used to create a Cartesian product for many-to-many joins

Ref. SAS Programming 2: Data Manipulation

## SAS Programming Process



This workshop

**PROC IMPORT**
**LIBNAME**

**PROC CONTENTS**
**PROC PRINT**
**PROC MEANS**
**PROC UNIVARIATE**
**PROC FREQ**

**DATA STEP**
**PROC SQL**

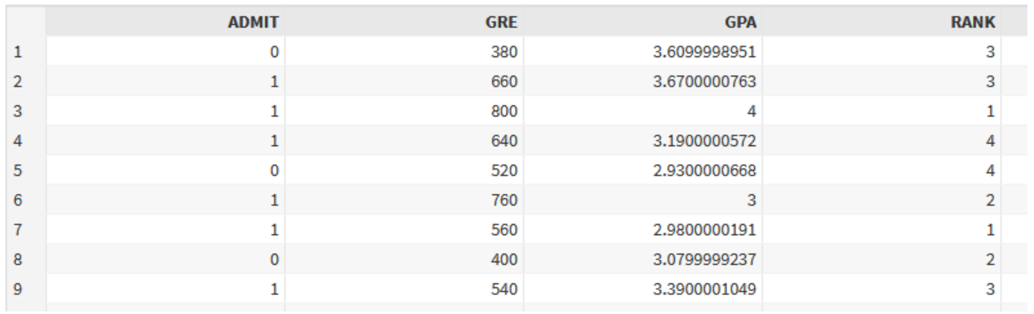
**PROC REG**
**PROC LOGISTIC**

**PROC …**

**PROC EXPORT**

Ref. SAS Programming 1: Essentials

# PROC LOGISTIC



| | ADMIT | GRE | GPA | RANK |
|---|---|---|---|---|
| 1 | 0 | 380 | 3.6099998951 | 3 |
| 2 | 1 | 660 | 3.6700000763 | 3 |
| 3 | 1 | 800 | 4 | 1 |
| 4 | 1 | 640 | 3.1900000572 | 4 |
| 5 | 0 | 520 | 2.9300000668 | 4 |
| 6 | 1 | 760 | 3 | 2 |
| 7 | 1 | 560 | 2.9800000191 | 1 |
| 8 | 0 | 400 | 3.0799999237 | 2 |
| 9 | 1 | 540 | 3.3900001049 | 3 |

PROC SORT DATA=*input-table*
  CLASS *variables* / PARAM=*keyword*;
  MODEL *variables = effects*;
RUN;

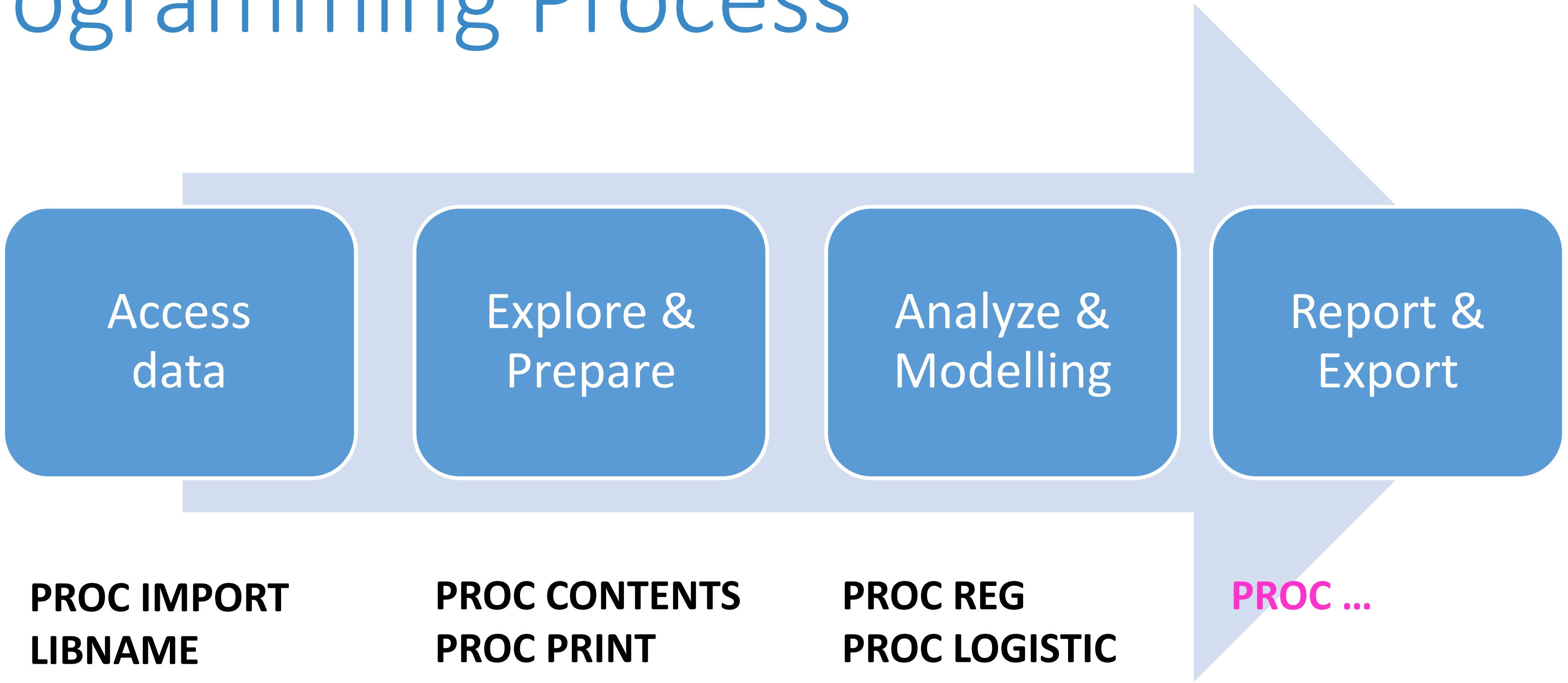
```
proc logistic data="c:\data\college_admission" descending;
  class rank / param=ref ;
  model admit = gre gpa rank;
run;
```

Ref. 1) PROC LOGISTIC document; 2) Source: https://stats.idre.ucla.edu/sas/dae/logit-regression/

## PROC LOGISTIC (hands-on)

- Run a logistic churning model
  - Use the `churn_telecom.sas7bdat` dataset
  - Try use both SAS Studio Point-and-Click and pure coding

# SAS Programming Process



PROC IMPORT
LIBNAME

PROC CONTENTS
PROC PRINT
PROC MEANS
PROC UNIVARIATE
PROC FREQ

DATA STEP
PROC SQL

PROC REG
PROC LOGISTIC

PROC …

PROC EXPORT

**This workshop**

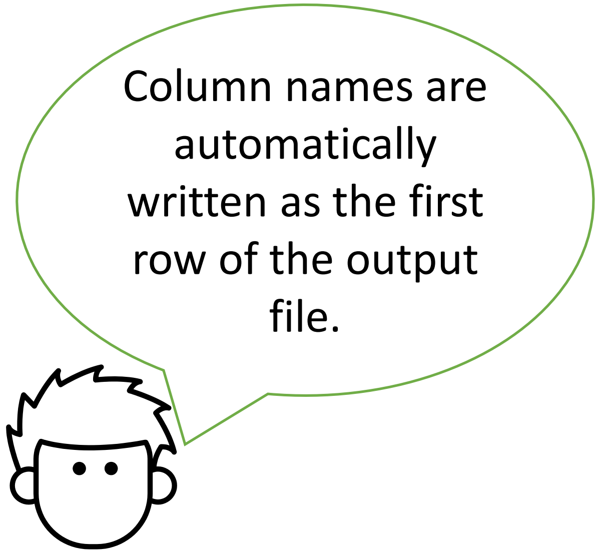
Ref. SAS Programming 1: Essentials

## Exporting Data

PROC EXPORT DATA=*input-table* OUTFILE="*output-file*"
<DBMS=*identifier*> <REPLACE>;
RUN;

tells SAS how to format the output

```
proc export data=sashelp.cars
    outfile="~/Workshop_SAS/data/cars.txt"
    dbms=tab replace;
run;
```


Column names are automatically written as the first row of the output file.

Ref. SAS Programming 1: Essentials

## Exporting Data (hands-on)

- Export the **`sashelp.cars`** dataset as csv format.

Ref. PROC EXPORT document

## Quick Intro to SAS Macro Programming

- SAS Macro programming allows you to
  - avoid repetitive sections of code
    - like function in other languages
  - create macro variables that can take different values for different runs
    - like variables or function parameters in other languages

Ref. SAS Macro Programming for Beginner

## SAS Macro Programming (example)

```
* A quick SAS Macro Intro
* path to the csv files (no "/" at the end);
%let path = /global/home/ut_jcao/Workshop_SAS/data;

* write a macro to load one table and display its content;
%macro load_data(csvfile=);

  PROC IMPORT DATAFILE="&path./&csvfile..csv" OUT=work.&csvfile DBMS=CSV REPLACE;
  RUN;

  PROC CONTENTS DATA=&csvfile.;
  RUN;
%mend;

* load tables;
%load_data(csvfile=Suppliers);
%load_data(csvfile=Categories);
%load_data(csvfile=Products);
```

## SAS Macro Programming (example)

```
* A quick SAS Macro Intro
* path to the csv files (no "/" at the end);
%let path = /global/home/ut_jcao/Workshop_SAS/data;

* write a macro to load one table and display its content;
%macro load_data(csvfile=);

  PROC IMPORT DATAFILE="&path./&csvfile..csv" OUT=work.&csvfile. DBMS=CSV REPLACE;
  RUN;

  PROC CONTENTS DATA=&csvfile.;
  RUN;
%mend;

* load tables;
%load_data(csvfile=Suppliers);
%load_data(csvfile=Categories);
%load_data(csvfile=Products);
```

# SAS Macro Programming (example)

```
* A quick SAS Macro Intro
* path to the csv files (no "/" at the end);
%let path = /global/home/ut_jcao/Workshop_SAS/data;

* write a macro to load one table and display its content;
%macro load_data(csvfile=);

  PROC IMPORT DATAFILE="&path./&csvfile..csv" OUT=work.&csvfile. DBMS=CSV REPLACE;
  RUN;

  PROC CONTENTS DATA=&csvfile.;
  RUN;
%mend;

* load tables;
%load_data(csvfile=Suppliers);
%load_data(csvfile=Categories);
%load_data(csvfile=Products);
```

## Quick Intro to SAS Viya Programming

```
* start a session with the CAS server;
cas;

* create a libref mycas and bind it to the casuser caslib;
libname mycas cas caslib=casuser;

* load sashelp.cars to casuser caslib;
* this DATA STEP runs in SAS, but not in CAS (Cloud Analytics Services);
data mycas.cars;
  set sashelp.cars;
run;

* add an indicator variable efficient;
* this DATA STEP runs in CAS;
data mycas.cars2;
  set mycas.cars;
  if mpg_city > 25 then efficient = 'Y';
    else efficient='N';
run;
```

# Quick Intro to SAS Viya Programming

- DATA STEP can run on CAS

- Many PROCs run on CAS
  - they look just like SAS 9 PROCs,
  - however, those PROCs can run across multiple machines

- CAS has its own scripting language as well
  - it's called CAS Language (CASL)
  - It's written inside PROC CAS

https://documentation.sas.com/api/docsets/pgmdiff/3.4/content/pgmdiff.pdf?locale=en