Welcome to SQUALL

Statistics and Quantitative Analysis Library for C++

DATA TYPES

`DataFrame` – Two-dimensional data structure with columns and rows.

`Series` – One-dimensional array capable of holding most primitive data types. A large set statistical / data analysis functions can be applied on a series.

DATA PREPARATION

1) Change all numerical cells to numbers (w/o comma)
   Number such as 1,397,403,523 should be change to 1397403523

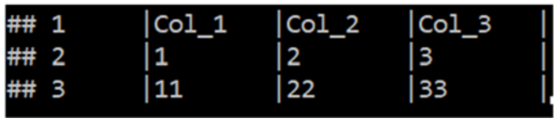
2) Don't use headers or cells that include line break in them (Alt-Enter in excel).
3) It is preferred not to use headers that include space in their names. For example, column name such as "Total 1" should be converted to "Total1" or "Total_1" or "Total-1" etc. Gladly, there is a built-in function that does that for you- please refer to `FormatHeaders` function in the Data Cleaning section.
   - This principal also applies to regular cells (not headers)- it is recommended not to have numeric values in a cell with spaces in them- it will show properly on the screen and saved properly to file but may be interpreted as separate values when applying statistical functions.
4) There is no procedure to "beautify" the console output since this would require setting the console screen buffer using platform-specific resources and limit the portability. This means that datasets with large number of columns, the output won't necessarily be aligned nicely on narrow screens.
5) If you wish to copy the content the content directly from the console to Excel or a text editor, the delimiter character is `|`. However, you can always use `WriteCsv` function to save a `DataFrame` to csv file. See more details in Exporting Data section.

IMPORTING / CREATING DATA

```
std::string filename = "C:\\myfile.csv";
DataFrame df = ioperations::ReadCsv(filename); // default values: has_headers = true, fill_na = false
DataFrame df = ioperations::ReadCsv(filename, bool has_header); // bool has_headers
DataFrame df = ioperations::ReadCsv(filename, bool has_headers, bool fill_na); // bool to fill empty cells with “N/A”.
```

A data frame can be also created by entering the values manually:

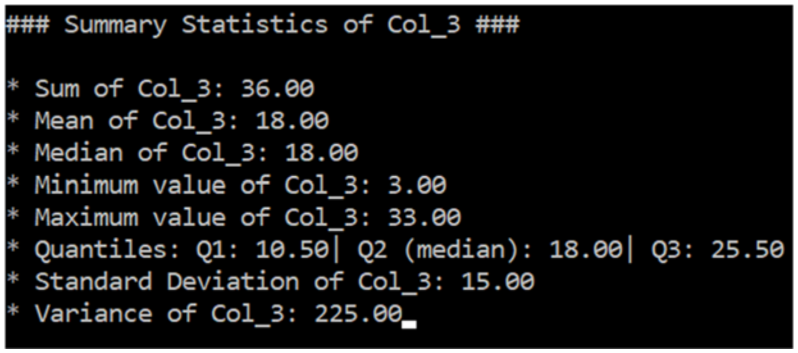
```
DataFrame df ({{"Col_1","Col_2","Col_3"}, {"1","2","3"}, {"11","22","33"}});
```



```
## 1      |Col_1    |Col_2    |Col_3    |
## 2      |1        |2        |3        |
## 3      |11       |22       |33       |
```

Important note: in DataFrame, numerical values are entered as strings, but will be treated as numerical once entered, which means any statistical function / group by / statistical summary can be applied to them. For example:

```
df["Col_3"].Describe();
```

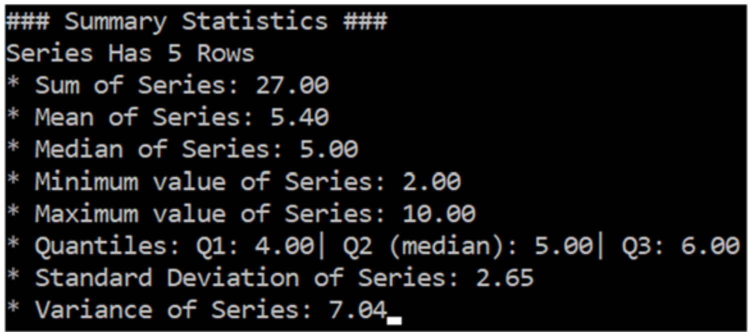


```
### Summary Statistics of Col_3 ###

* Sum of Col_3: 36.00
* Mean of Col_3: 18.00
* Median of Col_3: 18.00
* Minimum value of Col_3: 3.00
* Maximum value of Col_3: 33.00
* Quantiles: Q1: 10.50| Q2 (median): 18.00| Q3: 25.50
* Standard Deviation of Col_3: 15.00
* Variance of Col_3: 225.00
```

in a stand-alone Series assignment, numerical values can be entered as the actual type (see below):

```
Series<float> s1 = {10.5,2,5,4,6}; // Series of floats
Series<std::string> s2 = {"A","B","C","D","E"}; // Series of strings
Series<int> s3 = {10,2,5,4,6}; // Series of integers
```

```
s3.Describe();
```



```
### Summary Statistics ###
Series Has 5 Rows
* Sum of Series: 27.00
* Mean of Series: 5.40
* Median of Series: 5.00
* Minimum value of Series: 2.00
* Maximum value of Series: 10.00
* Quantiles: Q1: 4.00| Q2 (median): 5.00| Q3: 6.00
* Standard Deviation of Series: 2.65
* Variance of Series: 7.04
```

More information about the Series object and the statistical functions that it provides can be found later in this document.

EXPORTING DATA

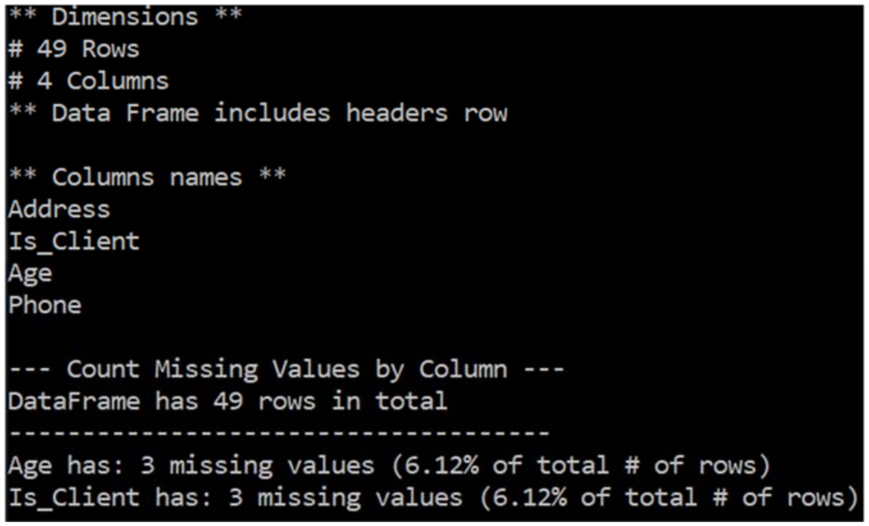
```
ioperations::WriteCsv("C:\\output.csv", df); // to csv file
ioperations::WriteJson("C:\\output.json", df); // to json file
```

VIEWING/INSPECTING DATA

```
df.PrintDataFrame(); // print entire dataframe to console. Default – variable column width
df.PrintDataFrame(bool fixed_width); // print entire dataframe to console. true = column width fixed (based on the widest
column in the table). The option may be useful for datasets with fewer columns, each can have more space across the console window.
df.Head(); // prints the first 5 rows
df.Head(3); // prints the first user defined number of rows
df.Tail(); // prints the last 5 rows
df.Tail(3); // prints the last user defined number of rows
df.PrintColumn("col1"); // print column with label “col1”
df.PrintColumns(1,4); // print columns 1 through 5
df.PrintRow(3); // print row 3
df.PrintRows(1,4); // print rows 1 through 4
df[3].PrintSeries(); // print row 3 as a series (no row numbers / column labels and other DataFrame attributes)
df.Shape(); // number of rows and columns in dataframe
df.DescribeNa(); // prints number of missing values by column and the percentage of missing values in each column out of total # of rows
```

However, the easiest and most comprehensive way to print all the relevant information regarding a dataframe that one may need is Info function:

```
df. Info();
```



```
** Dimensions **
# 49 Rows
# 4 Columns
** Data Frame includes headers row

** Columns names **
Address
Is_Client
Age
Phone

--- Count Missing Values by Column ---
DataFrame has 49 rows in total
-------------------------------------
Age has: 3 missing values (6.12% of total # of rows)
Is_Client has: 3 missing values (6.12% of total # of rows)
```

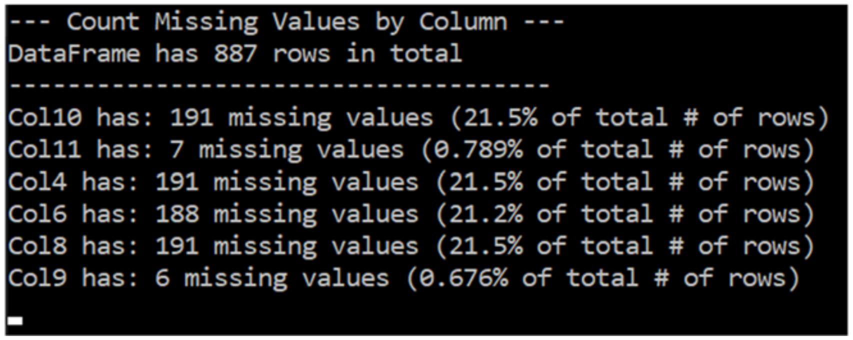
```
df. DescribeDataFrame(); // prints descriptive statistics of each column. See Series Describe function for more details. if a column is numerical – it prints summary statistics of the numerical values. If the column is of datatype string – it prints the count of the unique values.
```

SELECTION

```
df["col1"]; // returns column with label “col1” as a Series. See Data Cleaning and Statistics sections below how statistics / data analysis functions can be chained to a selected row or column, for example:

df["Col1"].Describe(); // If the column is numerical - prints to console summary of statistics of a Series (column/row of a DataFrame or stand-alone Series). If the series is a string – prints count of unique values

df["Col1"].Distinct().PrintSeries(); // Distinct returns a new Series of strings with the distinct values in it (a column in this case). It can also be printed.

DataFrame df2 = df[{2,9}]; // returns rows in a range as a new DataFrame

df[2]; // returns a row index as a Series. See Data Cleaning and Statistics sections below how statistics / data analysis functions can be chained to a selected row or column.
```

DATA CLEANING

```
df.DescribeNa(); // prints number of missing values by column and the % of missing values in each column out of total # of rows
```



```
--- Count Missing Values by Column ---
DataFrame has 887 rows in total
-------------------------------------
Col10 has: 191 missing values (21.5% of total # of rows)
Col11 has: 7 missing values (0.789% of total # of rows)
Col4 has: 191 missing values (21.5% of total # of rows)
Col6 has: 188 missing values (21.2% of total # of rows)
Col8 has: 191 missing values (21.5% of total # of rows)
Col9 has: 6 missing values (0.676% of total # of rows)
```

Squall supports “fluent interface” which means that you can chain functions:

```
DataFrame df2 = df.FillAllNa(); // replaces all empty cells with “N/A” and assign to a new DataFrame

df.FillAllNa().PrintDataFrame(); // replaces all empty cells with “N/A” and chain PrintDataFrame function.
```

The next 5 functions will work with numerical columns only

```
df.NatoMean("Col1"); // replaces all “N/A” or empty cells with the mean of a column

df.NatoSum("Col1"); // replaces all “N/A” or empty cells with the mean of a column

df.NatoSTDEV("Col1"); // replaces all “N/A” or empty cells with standard deviation of a column

df.NatoVariance("Col1"); // replaces all “N/A” or empty cells with the variance of a column

df.NatoZero("Col1"); // replaces all “N/A” or empty cells with zero
```

the missing values can be dropped altogether or under user defined conditions:

```
DataFrame df2 = df.DropNa(); // returns DataFrame, drops all rows with “N/A” or empty
DataFrame df2 = df.DropNa(int axis); // axis 0 = drops all rows with “N/A” or empty. axis 1 = columns
DataFrame df2 = df.DropNa(int axis, int number_of_na); // Drops rows(axis = 0)/columns(axis = 1) with MORE THAN the number of
missing values in them, for example:
DataFrame df2 = df.DropNa(0,4); // Creates a new DataFrame without those rows that contain more than 4 missing values.

DataFrame df2 = df.IsNull(); // returns a DataFrame of Boolean values. True where cells are missing values and false otherwise.
DataFrame df2 = df.IsNull(string is_null, string is_not_null); // returns a DataFrame of user defined values to non-missing
values and missing values, for example:
df.IsNull("1","0").PrintDataFrame();
```

RENAME/CREATE COLUMNS

```
df.RenameColumn(string original_name, string new_name); // renames a column
df.RenameColumns({"A","B","C"}); // renames multiple columns, from left to right. excess names are discarded
df.CreateColumn(string new_column_name, Series new_column_data); // create new column – overload 1. Series as new data
df.CreateColumn (string new_column_name, double new_column_data); // create new column – overload 2. Double as new data
```

examples:

```
df.CreateColumn ("New_Column_Name", df["Col_1"] + df["Col_2"]);
df.CreateColumn ("New_Column_Name", df["Col_1"] * df["Col_2"] / df["Col_3"]);
df.CreateColumn ("New_Column_Name", df["Col_1"].Mean()); // create new column with average of another column
df.CreateColumn ("New_Column_Name", (df["Col_1"] + df["Col_2"]) / 2);
df.CreateColumn ("New_Column_Name",(df["Col_1"] + df["Col_2"] + df["Col_3"]) / 3);
df.CreateColumn ("New_Column_Name", df["Col_1"].Sum() + 1.5);
```

Create a new column with a newly created series:

```
Series<std::string> s2 = {"A","B","C","D","E"};
df.CreateColumn ("New_Column_Name", s2 );
```

Or assigned to a new Series object and added to a new DataFrame:

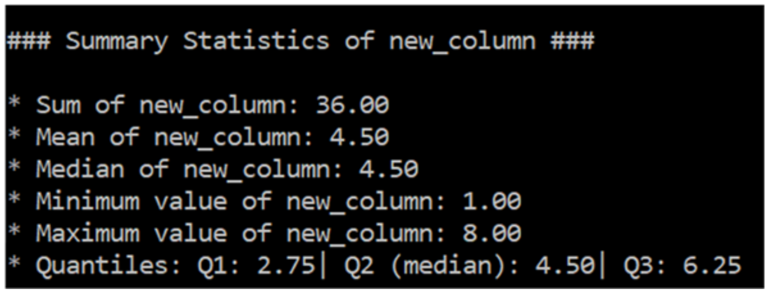
```
 Series<std::string> distinct_series = df["Make"].Distinct();
 DataFrame df2;
 df2.CreateColumn("new_column", distinct_series);
```

the Series can be added to a new data frame directly as well (without assigned it to an object first, rvalue):

```
df2.CreateColumn("new_column", df["Make"].Distinct());
```

important note – if the input series contains numerical values (see examples below), it will be created as series of string but will be treated as a numerical column, which means that any statistical function / group by / statistical summary can be applied to it:

```
Series<std::string> new_series {"1","2","3","4","5","6","7","8"};
df.CreateColumn ("new_column", new_series);
df["new_column"].Describe();
```



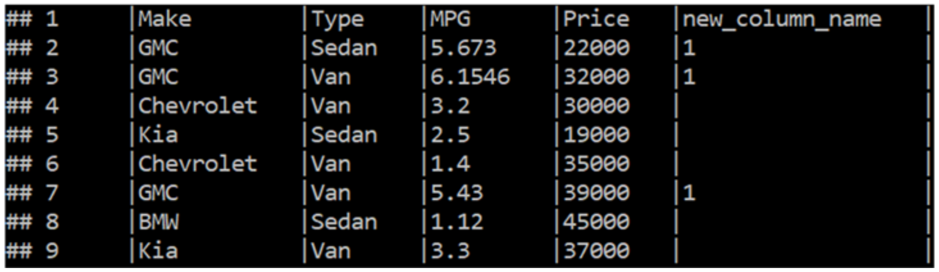
```
### Summary Statistics of new_column ###

* Sum of new_column: 36.00
* Mean of new_column: 4.50
* Median of new_column: 4.50
* Minimum value of new_column: 1.00
* Maximum value of new_column: 8.00
* Quantiles: Q1: 2.75| Q2 (median): 4.50| Q3: 6.25
```

Create a new column and set its values based on other columns values, using Series SetValue function:

```
Series<std::string>.SetValue(condition,"value");
```

Example with one value:

```
 Series<std::string> new_column_values;
  new_column_values.SetValue(df["Make"] == "GMC","1");
  df.CreateColumn ("new_column_name",new_column_values);
```

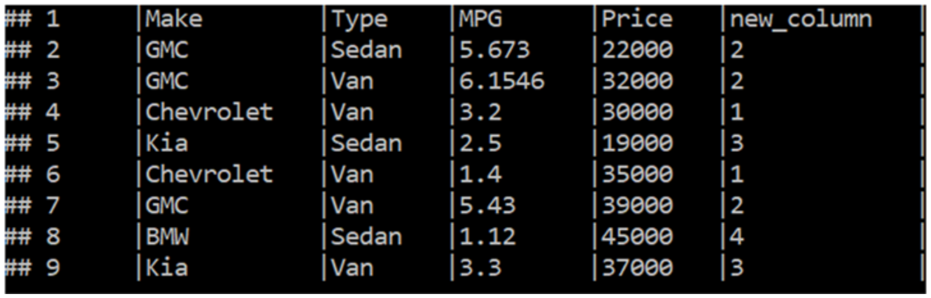


```
## 1     |Make        |Type    |MPG      |Price   |new_column_name  |
## 2     |GMC         |Sedan   |5.673    |22000   |1                |
## 3     |GMC         |Van     |6.1546   |32000   |1                |
## 4     |Chevrolet   |Van     |3.2      |30000   |                 |
## 5     |Kia         |Sedan   |2.5      |19000   |                 |
## 6     |Chevrolet   |Van     |1.4      |35000   |                 |
## 7     |GMC         |Van     |5.43     |39000   |1                |
## 8     |BMW         |Sedan   |1.12     |45000   |                 |
## 9     |Kia         |Van     |3.3      |37000   |                 |
```

The same method can be used for multiple conditions:

```
 auto condition_1 = df["Make"] == "Chevrolet";
 auto condition_2 = df["Make"] == "GMC";
 auto condition_3 = df["Make"] == "kia";
 auto condition_4 = df["Make"] == "BMW";
```

```
Series<std::string> new_column;
new_column.SetValue({condition_1,condition_2,condition_3,condition_4},{"1","2","3","4"});
df.CreateColumn ("new_column", new_column);
```



```
## 1      |Make        |Type    |MPG       |Price    |new_column  |
## 2      |GMC         |Sedan   |5.673     |22000    |2           |
## 3      |GMC         |Van     |6.1546    |32000    |2           |
## 4      |Chevrolet   |Van     |3.2       |30000    |1           |
## 5      |Kia         |Sedan   |2.5       |19000    |3           |
## 6      |Chevrolet   |Van     |1.4       |35000    |1           |
## 7      |GMC         |Van     |5.43      |39000    |2           |
## 8      |BMW         |Sedan   |1.12      |45000    |4           |
## 9      |Kia         |Van     |3.3       |37000    |3           |
```

Another example with the functions AND and OR to set the conditions. More example of AND/OR function can be found below in the Subset section:

```
// set the conditions

auto cond_1 = AND(df["index"] > 0, df["index"] <= 30);

auto cond_2 = df["index"] > 30;

Series<std::string> s1;

s1.SetValue({cond_1, cond_2}, {"less than 30", "more than 30"});

df.CreateColumn ("new_column", s1);
```

Two important notes regarding the `SetValue` function:

1) There are two conditions that must be met:
   - The number of conditions and new values must be identical (4 in the example above). If the number of conditions and values does not match the following message will appear:
     Could not set the values. The number of conditions and new values must be identical
   - The conditions that are provided to the function must encompass a unique set of rows indexes (i.e., no rows that meet more than one condition). If the row indexes of the conditions overlap, a message will appear and no new data will be added to the newly created column.
2) As with most of the functions in Series class- numerical values should be passed as string ("1", "2.5" etc), but once entered, they will be treated as numerical column, which means that any statistical function / group by / statistical summary can be apply to it.

```
df.FormatHeaders(); // formats the headers to be space-less. Whenever a header has a name the include space such as "Col 1" it will become "Col_1". This is highly recommended action to take as it can help prevent errors.
df.FormatHeaders(bool create_new_dataframe); // same as above, but returns a new dataframe.
```

SUBSET

Original DataFrame

Col1	Col2
40	300
55	399
60	500
49	550

Subset Rows Based on Numerical Values

```
DataFrame df2 = df.Subset(df["Col1"] > 50); // returns DataFrame with the subset of rows that meet the logical criterion.
```

Col1	Col2
55	399
60	500

```
DataFrame df2 = df.Subset(df["Col1"] == 55);
```

Col1	Col2
55	399

More subset examples:

```
DataFrame df2 = df.Subset(df["Col1"] <= 55);
DataFrame df2 = df.Subset(df["Col1"] >= 55);
DataFrame df2 = df.Subset(df["Col1"] != 55);
```

AND / OR functions are global function that take 2 conditions:

```
DataFrame df2 = df.Subset(AND(df["Col1"]>50, df["Col2"]<400)); // returns DataFrame with the subset of rows that meet BOTH
logical criteria.
```

Col1	Col2
55	399

```
DataFrame df2 = df.Subset(OR(df["Col1"]>50, df["Col2"]<400)); // returns dataframe with the subset of rows that meet EITHER logical criteria.
```

Col1	Col2
40	300
55	399
60	500

Subset Rows with String Values

Original DataFrame:

FIRST NAME	LAST NAME
ABBY	COLTON
BARB	JOHNSON
CARL	RAYNOLDS
DON	BENSON
MICAROS	DOE

```
DataFrame df2 = df.Subset(df["FIRST NAME"] == "barb"); // case insensitive!
```

FIRST NAME	LAST NAME
BARB	JOHNSON

```
DataFrame df2 = df.Subset(df["FIRST NAME"] != "barb");
```

FIRST NAME	LAST NAME
ABBY	COLTON
CARL	RAYNOLDS
DON	BENSON

```
DataFrame df2 = df.Subset(OR(df["FIRST NAME"] =="barb", df["LAST NAME"] =="BENSON" ));
```

FIRST NAME	LAST NAME
BARB	JOHNSON
DON	BENSON

```
DataFrame df2 = df.Subset(df["FIRST NAME"] < "CARL");
```

FIRST NAME	LAST NAME
ABBY	COLTON
BARB	JOHNSON

```
DataFrame df2 = df.Subset(df["FIRST NAME"] > "CARL");
```

FIRST NAME	LAST NAME
DON	BENSON
MICAROS	DOE

```
DataFrame df2 = df.Subset(df["FIRST NAME"] >= "CARL");
```

FIRST NAME	LAST NAME
CARL	RAYNOLDS
DON	BENSON
MICAROS	DOE

```
DataFrame df2 = df.Subset(df["FIRST NAME"] <= "CARL");
```

FIRST NAME	LAST NAME
ABBY	COLTON
BARB	JOHNSON
CARL	RAYNOLDS

```
DataFrame df2 = df.Subset(df["FIRST NAME"].Contains("ca")); // Subsets a dataframe if a substring is contained in a cell.
```

FIRST NAME	LAST NAME
CARL	RAYNOLDS
MICAROS	DOE

It is also possible to apply such subsets without assigning them to a new DataFrame object, but chaining functions to them instead:

```
df.Subset(df["FIRST NAME"].Contains("ca")).PrintDataFrame();
```

Subset Columns

```
DataFrame df2 = df.Subset({"Col4", "Col7", "Col11"});
```

Subset Rows by Range

```
DataFrame df2 = df.Subset(3,4);
```

FIRST NAME	LAST NAME
CARL	**RAYNOLDS**
MICAROS	**DOE**

The above operation is equivalent to:

```
DataFrame df2 = df[{3,4}]
```

GROUP BY

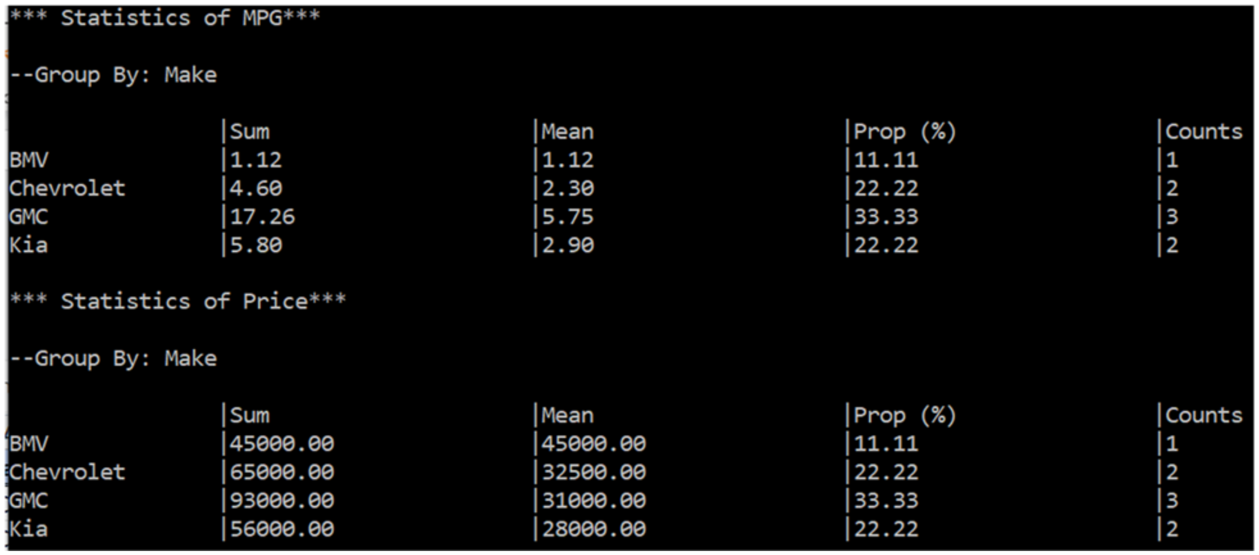
GroupBy is a mechanism which groups rows sharing the same property or intersection of properties, and applies aggregate functions to numerical columns of each group. There are 4 aggregate functions that are currently supported: Sum, Mean, Prop (% of total number of rows) and Count.

```
df.GroupBy({"GroupBy_1", "GroupBy_2"},{"Aggregated_1","Aggregated_2","Aggregated_3” . . .});
```

Original Data:

Make	Type	MPG	Price
GMC	Sedan	5.673	22000
GMC	Van	6.1546	32000
Chevrolet	Van	3.2	30000
Kia	Sedan	2.5	19000
Chevrolet	Van	1.4	35000
GMC	Van	5.43	39000
BMV	Sedan	1.12	45000
Kia	Van	3.3	37000

```
df.GroupBy({"Make"},{"MPG", "Price"}); // example with one Group By column
```



```
*** Statistics of MPG***

--Group By: Make

                |Sum                    |Mean                   |Prop (%)               |Counts
BMV             |1.12                   |1.12                   |11.11                  |1
Chevrolet       |4.60                   |2.30                   |22.22                  |2
GMC             |17.26                  |5.75                   |33.33                  |3
Kia             |5.80                   |2.90                   |22.22                  |2

*** Statistics of Price***

--Group By: Make

                |Sum                    |Mean                   |Prop (%)               |Counts
BMV             |45000.00               |45000.00               |11.11                  |1
Chevrolet       |65000.00               |32500.00               |22.22                  |2
GMC             |93000.00               |31000.00               |33.33                  |3
Kia             |56000.00               |28000.00               |22.22                  |2
```

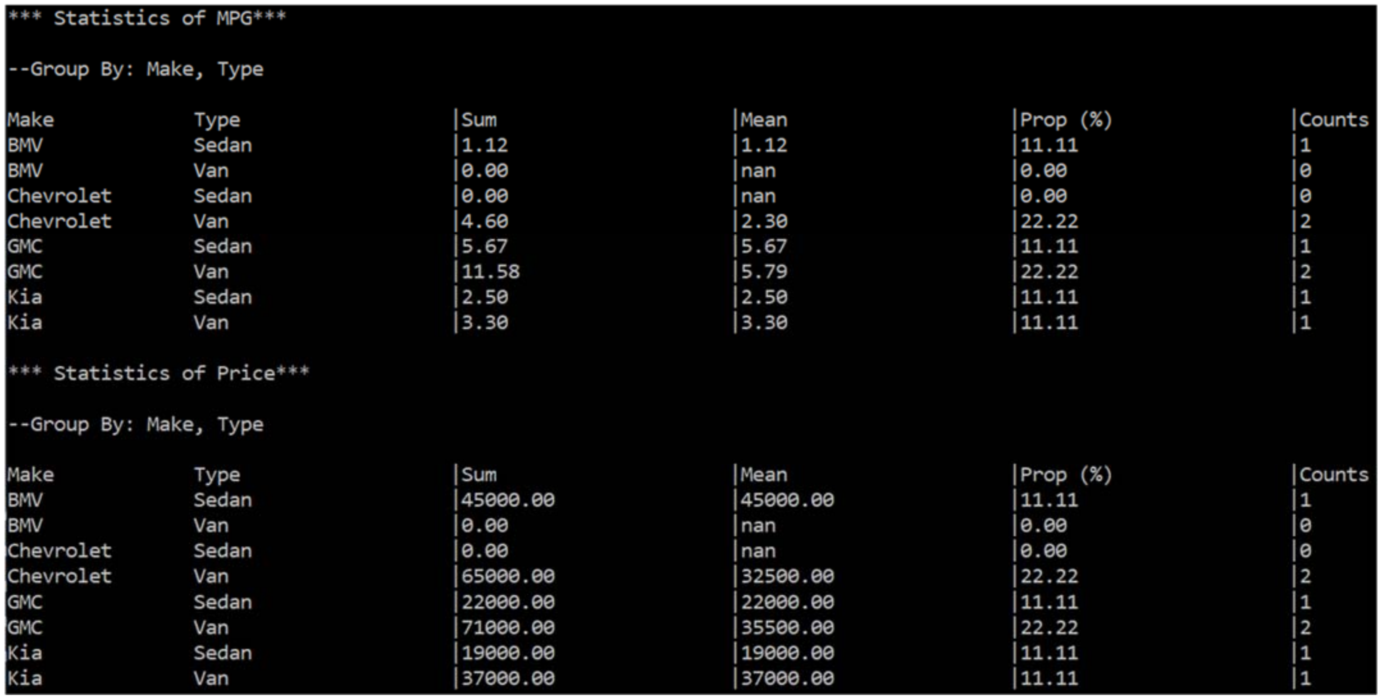
```
df.GroupBy({"Make", "Type"},{"MPG", "Price"}); // example with two Group By columns
```

* Note: as of now, only up to 2 Group By columns are supported (such as “Make” and “Type” in this case). This covers the majority of “real world” needs. However, there is no limit as to the number of aggregated columns.
* The column names are case insensitive.

By default, GroupBy will fill the missing values in the group-by columns with N/A and set to 0 any missing values in the aggregated columns. That means that those rows count, and have impact on Counts, Prop and Mean (but not on Sum)

If you wish to remove all rows with missing values:

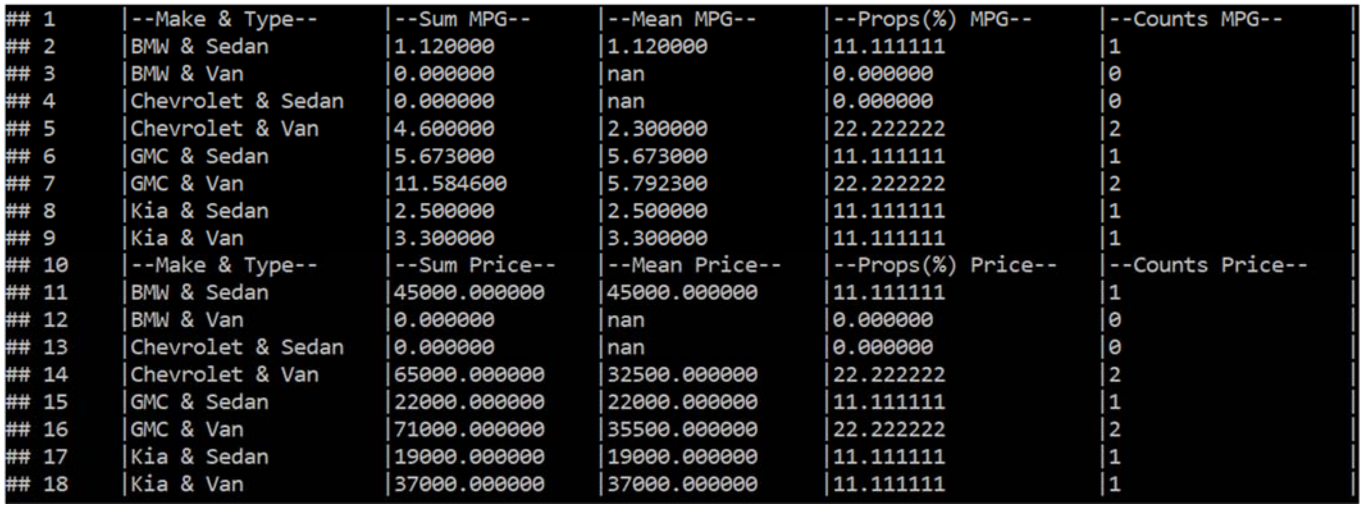
```
df.GroupBy({"Make", "Type"},{"MPG", "Price"}, true);
```



```
*** Statistics of MPG***

--Group By: Make, Type

Make            Type                    |Sum                    |Mean                   |Prop (%)               |Counts
BMV             Sedan                   |1.12                   |1.12                   |11.11                  |1
BMV             Van                     |0.00                   |nan                    |0.00                   |0
Chevrolet       Sedan                   |0.00                   |nan                    |0.00                   |0
Chevrolet       Van                     |4.60                   |2.30                   |22.22                  |2
GMC             Sedan                   |5.67                   |5.67                   |11.11                  |1
GMC             Van                     |11.58                  |5.79                   |22.22                  |2
Kia             Sedan                   |2.50                   |2.50                   |11.11                  |1
Kia             Van                     |3.30                   |3.30                   |11.11                  |1

*** Statistics of Price***

--Group By: Make, Type

Make            Type                    |Sum                    |Mean                   |Prop (%)               |Counts
BMV             Sedan                   |45000.00               |45000.00               |11.11                  |1
BMV             Van                     |0.00                   |nan                    |0.00                   |0
Chevrolet       Sedan                   |0.00                   |nan                    |0.00                   |0
Chevrolet       Van                     |65000.00               |32500.00               |22.22                  |2
GMC             Sedan                   |22000.00               |22000.00               |11.11                  |1
GMC             Van                     |71000.00               |35500.00               |22.22                  |2
Kia             Sedan                   |19000.00               |19000.00               |11.11                  |1
Kia             Van                     |37000.00               |37000.00               |11.11                  |1
```

A GroupBy can also be assigned to a data frame object and therefore be printed, saved to file and any other functionality that a data frame object has to offer:

```
DataFrame df2 = df.GroupBy({"Make","Type"},{"Mpg","Price"});
df2.PrintDataFrame();
```



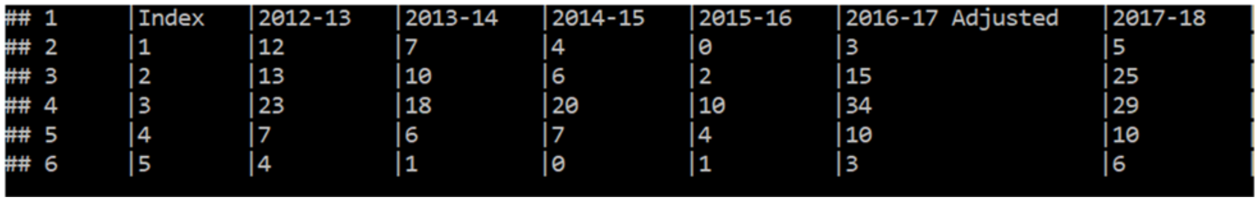
```
## 1      |--Make & Type--     |--Sum MPG--      |--Mean MPG--     |--Props(%) MPG--    |--Counts MPG--    |
## 2      |BMW & Sedan         |1.120000         |1.120000         |11.111111          |1                 |
## 3      |BMW & Van           |0.000000         |nan              |0.000000           |0                 |
## 4      |Chevrolet & Sedan   |0.000000         |nan              |0.000000           |0                 |
## 5      |Chevrolet & Van     |4.600000         |2.300000         |22.222222          |2                 |
## 6      |GMC & Sedan         |5.673000         |5.673000         |11.111111          |1                 |
## 7      |GMC & Van           |11.584600        |5.792300         |22.222222          |2                 |
## 8      |Kia & Sedan         |2.500000         |2.500000         |11.111111          |1                 |
## 9      |Kia & Van           |3.300000         |3.300000         |11.111111          |1                 |
## 10     |--Make & Type--     |--Sum Price--    |--Mean Price--   |--Props(%) Price-- |--Counts Price--  |
## 11     |BMW & Sedan         |45000.000000     |45000.000000     |11.111111          |1                 |
## 12     |BMW & Van           |0.000000         |nan              |0.000000           |0                 |
## 13     |Chevrolet & Sedan   |0.000000         |nan              |0.000000           |0                 |
## 14     |Chevrolet & Van     |65000.000000     |32500.000000     |22.222222          |2                 |
## 15     |GMC & Sedan         |22000.000000     |22000.000000     |11.111111          |1                 |
## 16     |GMC & Van           |71000.000000     |35500.000000     |22.222222          |2                 |
## 17     |Kia & Sedan         |19000.000000     |19000.000000     |11.111111          |1                 |
## 18     |Kia & Van           |37000.000000     |37000.000000     |11.111111          |1                 |
```

Data Manipulation

```
df.Replace("1","100"); // replaces all 1’s with 100’s. Although the function accepts string as arguments, it will treat the
original and modified values as numerical values when appropriate.
```

Transform Table from wide to long

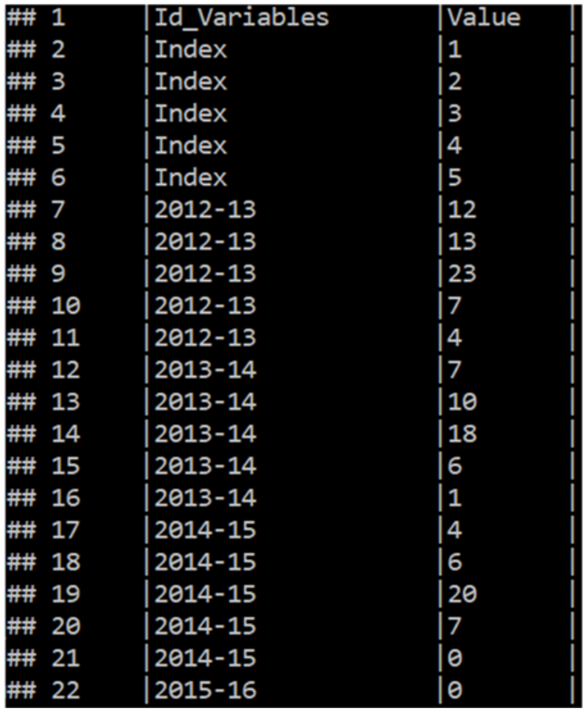
Original data:



```
## 1      |Index   |2012-13  |2013-14  |2014-15  |2015-16  |2016-17 Adjusted  |2017-18  |
## 2      |1       |12       |7        |4        |0        |3                 |5        |
## 3      |2       |13       |10       |6        |2        |15                |25       |
## 4      |3       |23       |18       |20       |10       |34                |29       |
## 5      |4       |7        |6        |7        |4        |10                |10       |
## 6      |5       |4        |1        |0        |1        |3                 |6        |
```

```
DataFrame df2 = df.Melt();
```

Melted data (first few rows):

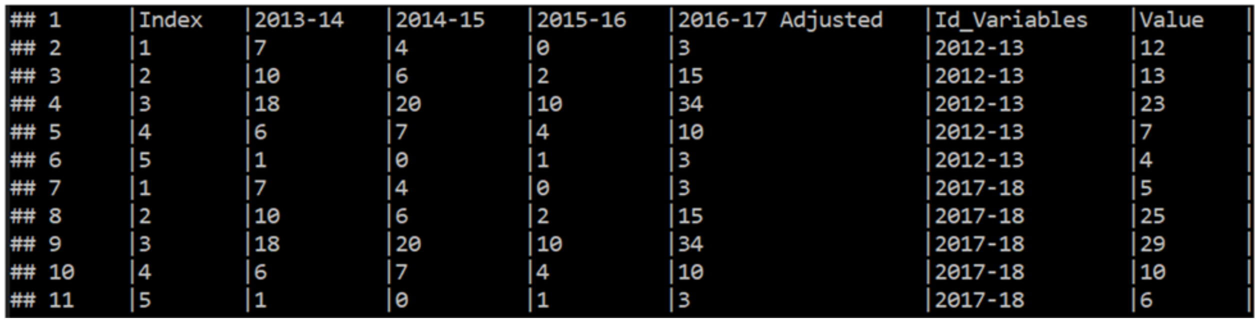


```
## 1      |Id_Variables        |Value   |
## 2      |Index               |1       |
## 3      |Index               |2       |
## 4      |Index               |3       |
## 5      |Index               |4       |
## 6      |Index               |5       |
## 7      |2012-13             |12      |
## 8      |2012-13             |13      |
## 9      |2012-13             |23      |
## 10     |2012-13             |7       |
## 11     |2012-13             |4       |
## 12     |2013-14             |7       |
## 13     |2013-14             |10      |
## 14     |2013-14             |18      |
## 15     |2013-14             |6       |
## 16     |2013-14             |1       |
## 17     |2014-15             |4       |
## 18     |2014-15             |6       |
## 19     |2014-15             |20      |
## 20     |2014-15             |7       |
## 21     |2014-15             |0       |
## 22     |2015-16             |0       |
```

… … …

It is also possible to add id variables. The id variables will be transformed into a long-data format while the rest of the table will remain intact:

```
df.Melt({"2012-13","2017-18"}).PrintDataFrame();
```



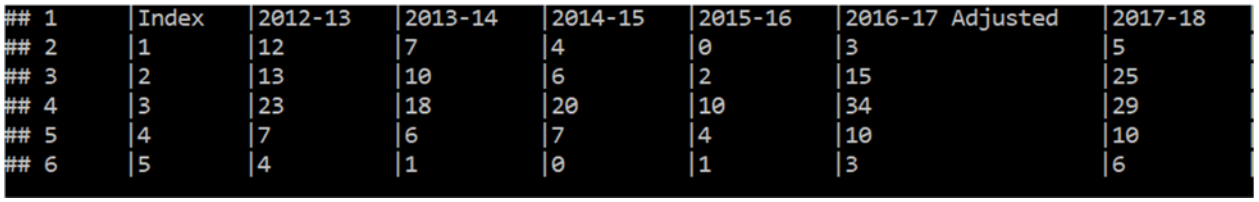
```
## 1     |Index   |2013-14   |2014-15   |2015-16   |2016-17 Adjusted   |Id_Variables   |Value  |
## 2     |1       |7         |4         |0         |3                  |2012-13        |12     |
## 3     |2       |10        |6         |2         |15                 |2012-13        |13     |
## 4     |3       |18        |20        |10        |34                 |2012-13        |23     |
## 5     |4       |6         |7         |4         |10                 |2012-13        |7      |
## 6     |5       |1         |0         |1         |3                  |2012-13        |4      |
## 7     |1       |7         |4         |0         |3                  |2017-18        |5      |
## 8     |2       |10        |6         |2         |15                 |2017-18        |25     |
## 9     |3       |18        |20        |10        |34                 |2017-18        |29     |
## 10    |4       |6         |7         |4         |10                 |2017-18        |10     |
## 11    |5       |1         |0         |1         |3                  |2017-18        |6      |
```

Transposing Data

There are 2 options for transposing data:

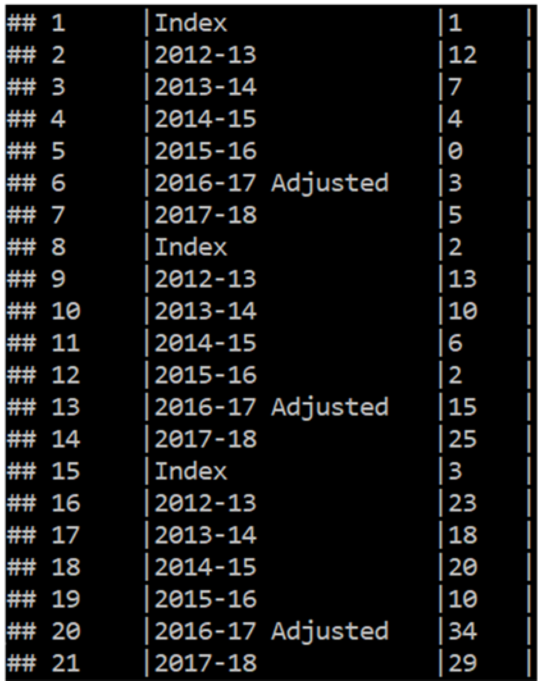
1) Row By Row

Original data:



```
## 1     |Index   |2012-13   |2013-14   |2014-15   |2015-16   |2016-17 Adjusted   |2017-18   |
## 2     |1       |12        |7         |4         |0         |3                  |5         |
## 3     |2       |13        |10        |6         |2         |15                 |25        |
## 4     |3       |23        |18        |20        |10        |34                 |29        |
## 5     |4       |7         |6         |7         |4         |10                 |10        |
## 6     |5       |4         |1         |0         |1         |3                  |6         |
DataFrame df2 = df.Transpose();

df.Transpose().PrintDataFrame();
```

Transposed data (row by row):

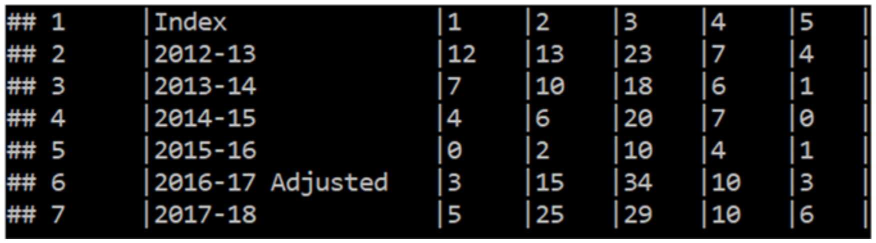


```
## 1      |Index            |1   |
## 2      |2012-13          |12  |
## 3      |2013-14          |7   |
## 4      |2014-15          |4   |
## 5      |2015-16          |0   |
## 6      |2016-17 Adjusted |3   |
## 7      |2017-18          |5   |
## 8      |Index            |2   |
## 9      |2012-13          |13  |
## 10     |2013-14          |10  |
## 11     |2014-15          |6   |
## 12     |2015-16          |2   |
## 13     |2016-17 Adjusted |15  |
## 14     |2017-18          |25  |
## 15     |Index            |3   |
## 16     |2012-13          |23  |
## 17     |2013-14          |18  |
## 18     |2014-15          |20  |
## 19     |2015-16          |10  |
## 20     |2016-17 Adjusted |34  |
## 21     |2017-18          |29  |
```

2) The second Transpose option is “straight” transpose, in which first row (headers) becomes first column:

```
DataFrame df2 = df.Transpose(bool columns_to_rows  = true);
```

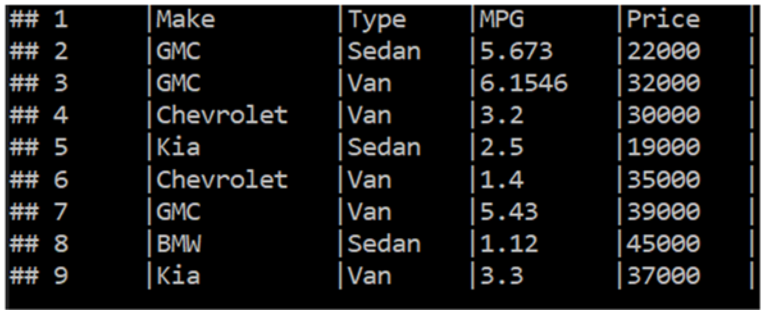
Transposed data (columns to rows):



```
## 1     |Index             |1     |2     |3     |4     |5     |
## 2     |2012-13           |12    |13    |23    |7     |4     |
## 3     |2013-14           |7     |10    |18    |6     |1     |
## 4     |2014-15           |4     |6     |20    |7     |0     |
## 5     |2015-16           |0     |2     |10    |4     |1     |
## 6     |2016-17 Adjusted  |3     |15    |34    |10    |3     |
## 7     |2017-18           |5     |25    |29    |10    |6     |
```

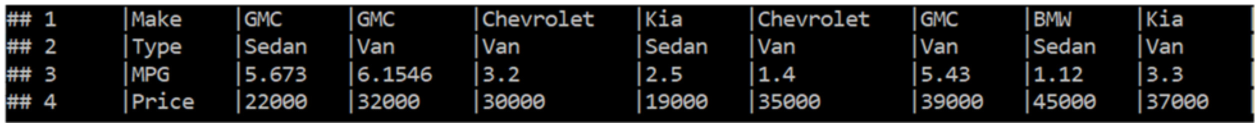
It is possible to de-transpose the data:

Original Data:



```
## 1     |Make        |Type    |MPG       |Price    |
## 2     |GMC         |Sedan   |5.673     |22000    |
## 3     |GMC         |Van     |6.1546    |32000    |
## 4     |Chevrolet   |Van     |3.2       |30000    |
## 5     |Kia         |Sedan   |2.5       |19000    |
## 6     |Chevrolet   |Van     |1.4       |35000    |
## 7     |GMC         |Van     |5.43      |39000    |
## 8     |BMW         |Sedan   |1.12      |45000    |
## 9     |Kia         |Van     |3.3       |37000    |
```

```
DataFrame df2 = df.DeTranspose();
```

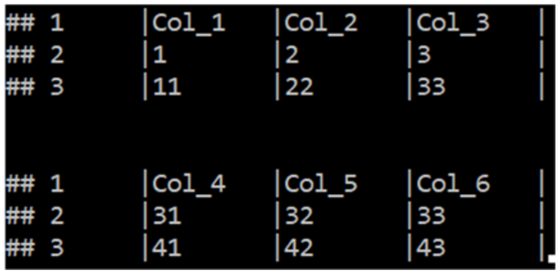


```
## 1     |Make     |GMC      |GMC      |Chevrolet   |Kia      |Chevrolet   |GMC      |BMW      |Kia      |
## 2     |Type     |Sedan    |Van      |Van         |Sedan    |Van         |Van      |Sedan    |Van      |
## 3     |MPG      |5.673    |6.1546   |3.2         |2.5      |1.4         |5.43     |1.12     |3.3      |
## 4     |Price    |22000    |32000    |30000       |19000    |35000       |39000    |45000    |37000    |
```

DATA FRAME CONCATENATION

Manually created data frames:

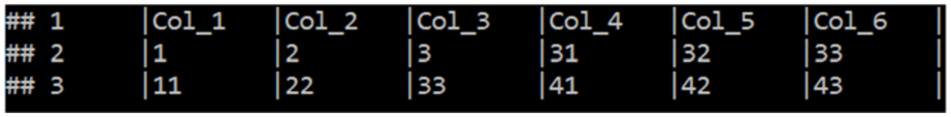
```
DataFrame df ({{"Col_1","Col_2","Col_3"}, {"1","2","3"}, {"11","22","33"}});
DataFrame df2 ({{"Col_4","Col_5","Col_6"}, {"31","32","33"}, {"41","42","43"}});
```



```
## 1       |Col_1   |Col_2   |Col_3   |
## 2       |1       |2       |3       |
## 3       |11      |22      |33      |


## 1       |Col_4   |Col_5   |Col_6   |
## 2       |31      |32      |33      |
## 3       |41      |42      |43      |
```

Concatenating the two data frames into a new data frame (Join):

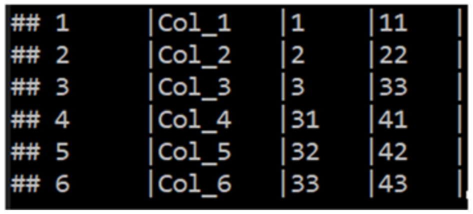
```
DataFrame df3 = df.Concat({df,df2});
```



```
## 1       |Col_1   |Col_2   |Col_3   |Col_4   |Col_5   |Col_6   |
## 2       |1       |2       |3       |31      |32      |33      |
## 3       |11      |22      |33      |41      |42      |43      |
```

It is also possible to concatenate the DataFrames in a transposed shape (Union):

```
DataFrame df3 = df.Concat({df,df2}, bool transposed = true);
```



```
## 1       |Col_1   |1     |11     |
## 2       |Col_2   |2     |22     |
## 3       |Col_3   |3     |33     |
## 4       |Col_4   |31    |41     |
## 5       |Col_5   |32    |42     |
## 6       |Col_6   |33    |43     |
```

** there is no limit as to how many data frames can be concatenated as long as they have the same dimensions (number of rows and columns).

STATISTICS

```
df["col10"].Describe(); // if series is numerical - prints to console summary all the below statistics of a Series (column/row of a DataFrame or stand-alone Series). If the series is a string – prints count of unique values

df2[1].Sum(); // returns a value of type double with sum of a Series (column/row of a DataFrame or stand-alone Series), and print it to console. Returns 0 if Series is non-numerical.

df["col10"].Mean(); // returns a value of type double with mean of a Series (column/row of a DataFrame or a stand-alone Series), and prints it to console. Returns 0 if Series is non-numerical.

df["col10"].Median(); // returns a value of type double with median of a Series (column/row of a DataFrame or a stand-alone Series), and prints it to console. Returns 0 if Series is non-numerical.

df["col10"].FillNaWithMean().Median(); // example of typical chain of functions on a Series. Fill empty cells in column labeled “col10” with mean of the column and find the median.

df["col10"]. Quantiles() // prints 0.25, 0.5 (median), 0.75 quantiles of a numerical Series (column/row of a DataFrame or a stand-alone Series). Returns a vector of 3 doubles.

df[5]. Quantiles(float quantile) // prints the selected quantile, between 0 and 1 of a series. (column/row of a DataFrame or a stand-alone Series). Returns a double.

df["col10"]. STDEV() // returns a value of type double with standard deviation of a Series (column/row of a DataFrame or a stand-alone Series), and prints it to console. Returns 0 if Series is non-numerical.

df["col10"]. Variance() // returns a value of type double with variance of a Series (column/row of a DataFrame or a stand-alone Series), and prints it to console. Returns 0 if Series is non-numerical.
```