Documentation Guide - Row64

This documentation guide is an ongoing development. Its purpose is to assist users and document features that are available inside of Row64. If you are new to Row64 please view the following three videos which detail how to install Row64, the current layout for version 1.3 and a quick guide to how to use Data Science Recipes, a key feature for analyzing data frames in Row64.

Installation Guide / Intro Video-
https://drive.google.com/file/d/1EU5Iu85VoPUQ0QR9P2Yjc7LYoO-po3Ed/view?usp=sharing

User Interface Guide / Intro Video-
https://drive.google.com/file/d/1zBnvR2EEyAZnG1sd1mb0FuTpQAACJOsl/view?usp=sharing

Data Science Recipes Guide / Intro Video -
https://drive.google.com/file/d/1nbKEFvRFdTahTFgEV3wRJBQBA5xzrvMV/view?usp=sharing

Cell Formulas

AUTOCOMPLETE

Description: Autocomplete is a feature in Row64 which uses previously entered information and assists in the completion of other cells on the spreadsheet by anticipating what users want to have entered. By using AutoComplete, users reported a reduction in input errors and an increase in productivity.

Example: When a user begins inputting in a spreadsheet formulas such as “=SU”, a prompt appears suggesting SUM, SUMIF, SUMPRODUCT, etc. Users are then able to select the desired formula without additional input.

SUMIF Function

Function: The **SUMIF** function is used to sum the values in a range that meets a certain specified criteria. For example, suppose that in a column that contains numbers, you want to sum only the values that are larger than 5. You can use the following formula: =SUMIF(B2:B25,">5")

Syntax: SUMIF(range, criteria, [sum_range])

- **range**. Required, the range of cells that you want evaluated by criteria. Cells in each range must be numbers or names, arrays, or references that contain numbers. Blank and text values are ignored.
- **criteria**. Required. The criteria in the form of a number, expression, a cell reference, text, or a function that defines which cells will be added. Wildcard characters can be included - a question mark (?) to match any single character, an asterisk (*) to match any sequence of characters. If you want to find an actual question mark or asterisk, type a tilde (**~**) preceding the character. For example, criteria can be expressed as 32, ">32", B5, "3?", "apple*", "*~?", or TODAY().
- **sum_range** Optional. The actual cells to add, if you want to add cells other than those specified in the ***range*** argument. If the ***sum_range*** argument is omitted, the cells that are specified in the ***range*** argument are used as the ***sum_range***.

  Sum_range should be the same size and shape as ***range***. If it isn't, performance may suffer, and the formula will sum a range of cells that starts with the first cell in ***sum_range*** but has the same dimensions as ***range***. For example:

Video:

- TBD

List of Cell Formulas

abs
acos
acosh
asin
asinh
atan
atanh
average
averagea
averageif
averageifs
ceiling
concat
cosh
cos
csc
cot
clean
char
code
count
countif
countblank
counta
countifs
choose
degrees
dollar
even
exp
exact
floor
fact
fixed
find
gcd
int
if
isnumber
iseven
isodd
isblank
istext
isnontext
index
iserror
iferror
odd
offset

lower
left
len
lookup
log
log10
ln
lcm
mod
mround
mid
max
maxa
min
mina
maxifs
minifs
median
match
neg
pi
product
power
proper
quotient
radians
rand
rounddown
roundup
round
right
rept
replace
rank.avg
sign
sum
sumproduct
sumifs
sumif
sqrt
sinh
sin
search
substitute
trunc
tan
tanh
trim
textjoin
t
upper
value
vlookup

Sheet formulas

Group by Time Series

Sum By Month

Function: The **MONTHLYSUM** sheet formula is used to sum by month a dataframe column. For example, suppose that a dataframe contains two columns of transaction details that include a column with date&time information as well as another column with numbers. The MONTHLYSUM formula will create a new dataframe that sums the transaction values on a month by month basis.

Syntax: MONTHLYSUM(*datetime_column, sum_column*)

- ***datetime_column*.** Required. ***datetime_column*** is the dataframe column which contains the dates to be summarized on a monthly basis.
- ***sum_column***. Required. ***sum_column*** is the dataframe column which contains the range to be summed.

Video:
https://drive.google.com/file/d/1zHsFzh_I7fweIaHB_mvzAU77rQ3lI7Je/view?usp=sharing

Python Code:

```
import pandas as pd
import row64

def MonthlySum(idf, inDateC, inSumC):
       mName = idf.columns[inDateC]
       idf[mName] = pd.to_datetime(idf[mName])
       idf = idf.groupby(pd.Grouper(key=mName, freq='M')).sum().reset_index()
       idf[mName] = idf[mName].dt.strftime('%m-%Y')
       return idf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\GpuSales.csv")
df=dfIn.copy(deep=True)
df=MonthlySum(df,0,1)
```

Average By Month

Function: The **MONTHLYAVG** sheet formula is used to average by month a dataframe column. For example, suppose that a dataframe contains 2 columns of transaction details that include a column with date&time information and another column with numbers. The MONTHLYAVG formula will create a new dataframe that averages the transaction values on a month by month basis.

Syntax: MONTHLYAVG(*datetime_column, avg_column*)

- ***datetime_column*****.** Required. ***datetime_column*** is the dataframe column which contains the dates to be summarized on a monthly basis.
- ***avg_column***. Required. ***avg_column*** is the dataframe column which contains the range to be averaged.

Video:
https://drive.google.com/file/d/19E2YUDv6eaATC6Mbf5z4jdpwMBsr-gEr/view?usp=sharing

Python Code:

```
import pandas as pd
import row64

def MonthlyAve(idf, inDateC, inSumC):
        mName = idf.columns[inDateC]
        idf[mName] = pd.to_datetime(idf[mName])
        idf = idf.groupby(pd.Grouper(key=mName, freq='M')).mean().reset_index()
        idf[mName] = idf[mName].dt.strftime('%m-%Y')
        idf = idf.fillna(0).round(2)
        return idf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\GpuSales.csv")
df=dfIn.copy(deep=True)
df=MonthlyAve(df,0,1)
```

Group & Sum By Month

Function: The **MONTHLYGROUPSUM** sheet formula is used to sum by month a dataframe column by a particular group. For example, suppose that a dataframe contains 3 columns of transaction details that include a column with date&time information, another column with numbers, and another column with categorical information. The **MONTHLYGROUPSUM** formula will create a new dataframe that sums the transaction values on a month by month basis by group (i.e. revenues per month by customer)

Syntax: MONTHLYGROUPSUM(*datetime_column, group_column, sum_column*)

- ***datetime_column.*** Required. ***datetime_column*** is the dataframe column which contains the dates to be summarized on a monthly basis.
- ***group_column***. Required. ***group_column*** is the dataframe column which contains the grouping categories.
- ***sum_column***. Required. ***sum_column*** is the dataframe column which contains the range to be summed.

Video:
https://drive.google.com/file/d/1fz0UimG8iSppFhQUbHyt7jgS_IaHbgM_/view?usp=sharing

Python Code:

```
import pandas as pd

def MonthlyGroupSum(idf, inDateC, inGroupC, inSumC):
       mName, gName, cName = idf.columns[[inDateC, inGroupC, inSumC]]
       idf[mName] = pd.to_datetime(idf[mName])
       idf = idf.groupby([pd.Grouper(key=mName, freq='M'),
       gName]).sum()[cName].reset_index()
       idf[mName] = idf[mName].dt.strftime('%m-%Y')
       idf = pd.pivot_table(idf, index=gName, columns=mName,
       values=cName).reset_index()
       idf = idf.fillna(0)
       return idf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\GpuSales.csv")
df=dfIn.copy(deep=True)
df=MonthlyGroupSum(df,0,1,2)
```

Group & Average By Month

Function: The **MONTHLYGROUPAVE** sheet formula is used to average by month a dataframe column by a particular group. For example, suppose that a dataframe contains 3 columns of transaction details that include a column with date&time information and another column with numbers, and another column with categorical information. The **MONTHLYGROUPAVE** formula will create a new dataframe that averages the transaction values on a month by month basis and by group (i.e. average transaction value per month by customer).

Syntax: MONTHLYGROUPAVE(*datetime_column, group_column, avg_column*)

- ***datetime_column.*** Required. ***datetime_column*** is the dataframe column which contains the dates to be summarized on a monthly basis.
- ***group_column***. Required. ***group_column*** is the dataframe column which contains the grouping categories.
- ***avg_column***. Required. ***avg_column*** is the dataframe column which contains the range to be averaged.

Video:
TBD

Python Code:

```
import pandas as pd

def MonthlyGroupAve(idf, inDateC, inGroupC, inSumC):
        mName = idf.columns[inDateC]
        idf[mName] = pd.to_datetime(idf[mName])
        idf = idf.groupby([pd.Grouper(key=mName, freq='M'),
        gName]).mean()[cName].reset_index()
        idf = pd.pivot_table(idf, index=gName, columns=mName,
        values=cName).reset_index()
        cList = [idf.columns[0]]
        for i,cName in enumerate(idf.columns):
                if(i>0):
                        cList.append(cName.strftime('%m-%Y'))
        idf.columns = cList
        idf = idf.fillna(0).round(2)
        return idf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\GpuSales.csv")
df=dfIn.copy(deep=True)
df=MonthlyGroupAve(df,0,1,2)
```

Group & Sum, With Monthly Change

Function:The MONTHLYGROUPSUMCHANGE sheet formula measures the change between each monthly periods' sum by group. The SUMBYMONTHBYGROUPDELTA formula will create a new dataframe that measures the differences between each month and the previous month's sum value by group.

Syntax: MONTHLYGROUPSUMCHANGE(*datetime_column, group_column, sum_column*)

- ***datetime_column*.** Required. ***datetime_column*** is the dataframe column which contains the dates to be summarized on a monthly basis.
- ***group_column***. Required. ***group_column*** is the dataframe column which contains the grouping categories.
- ***sum_column***. Required. ***sum_column*** is the dataframe column which contains the range to be summed.

Video:
TBD

Python Code:

```
import pandas as pd

def MonthlyGroupSumChange(idf, inDateC, inGroupC, inSumC):
       mName, gName, cName = idf.columns[[inDateC, inGroupC, inSumC]]
       idf[mName] = pd.to_datetime(idf[mName])
       idf = idf.groupby([pd.Grouper(key=mName, freq='M'),
       gName]).sum().reset_index()
       idf = pd.pivot_table(idf, index=gName, columns=mName,
       values=cName).reset_index()
       cList = [idf.columns[0]]
       for i,cName in enumerate(idf.columns):
              if(i>0):
                     cList.append(cName.strftime('%m-%Y'))
       idf.columns = cList
       idf = idf.fillna(0)
       firstCol =idf[gName]
       idf = idf.drop([gName], axis=1)
       idf = idf.diff(axis=1)
       firstName = idf.columns[0]
       idf[firstName] = idf[firstName].fillna('')
       idf.insert(loc=0, column=gName, value=firstCol)
       return idf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\GpuSales.csv")
df=dfIn.copy(deep=True)
df=MonthlyGroupSumChange(df,0,1,2)
```

Sum By Quarter

Function: The **QUARTERLYSUM** sheet formula is used to sum by quarter (3-month period) a dataframe column. The QUARTERLYSUM formula will create a new dataframe that sums the transaction values on a quarter by quarter basis. By default, quarters end in March, June, September and December. Support for other fiscal year periods is available by using the yearEndMonth parameter.

Syntax: QUARTERLYSUM (*datetime_column, sum_column, yearEndMonth*)

- ***datetime_column*.** Required. ***datetime_column*** is the dataframe column which contains the dates to be summarized on a monthly basis.
- ***sum_column***. Required. ***sum_column*** is the dataframe column which contains the range to be summed.
- ***yearEndMonth*.** Optional. yearEndMonth is the month number that the fiscal year ends (i.e., February = 2, March = 3, etc.). Default is 12.

Video:
TBD

Python:

```
import pandas as pd

def QuarterlySum(idf, inDateC, inSumC, yearEndMonth = 12):
       group = ['','Q-JAN','Q-FEB','Q-MAR','Q-APR','Q-MAY','Q-JUN',
       'Q-JUL','Q-AUG','Q-SEP','Q-OCT','Q-NOV','Q-DEC'][yearEndMonth]
       mName = idf.columns[inDateC]
       idf[mName] = pd.to_datetime(idf[mName])
       idf = idf.groupby(pd.Grouper(key=mName, freq=group)).sum().reset_index()
       idf[mName] = idf[mName].dt.strftime('%m-%Y')
       return idf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\GpuSales.csv")
df=dfIn.copy(deep=True)
df=QuarterlySum(df,0,2,7)
```

Average By Quarter

Function: The **QUARTERLYAVE** sheet formula is used to average by quarter a dataframe column. For example, suppose that a dataframe contains 2 columns of transaction details that include a column with date&time information and another column with numbers. The QUARTERLYAVE formula will create a new dataframe that averages the transaction values on a quarter by quarter basis. By default, quarters end in March, June, September and December. Support for other fiscal year periods is available by using the yearEndMonth parameter.

Syntax: QUARTERLYAVE(*datetime_column, avg_column, yearEndMonth*)

- ***datetime_column*.** Required. ***datetime_column*** is the dataframe column which contains the dates to be summarized on a monthly basis.
- ***avg_column***. Required. ***avg_column*** is the dataframe column which contains the range to be averaged.
- ***yearEndMonth*.** Optional. yearEndMonth is the month number that the fiscal year ends (i.e., February = 2, March = 3, etc.). Default is 12.

Video:
TBD

Python:

```
import pandas as pd

def QuarterlyAve(idf, inDateC, inSumC, yearEndMonth = 12):
       group =['','Q-JAN','Q-FEB','Q-MAR','Q-APR','Q-MAY','Q-JUN',
       'Q-JUL','Q-AUG','Q-SEP','Q-OCT','Q-NOV','Q-DEC'][yearEndMonth]
       mName = idf.columns[inDateC]
       idf[mName] = pd.to_datetime(idf[mName])
       idf = idf.groupby(pd.Grouper(key=mName, freq=group)).mean().reset_index()
       idf[mName] = idf[mName].dt.strftime('%m-%Y')
       idf = idf.fillna(0).round(2)
       return idf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\GpuSales.csv")
df=dfIn.copy(deep=True)
df=QuarterlyAve(df,0,2,7)
```

Group & Sum By Quarter

Function: The **QUARTERLYGROUPSUM** sheet formula is used to sum by quarterly a dataframe column by a particular group. For example, suppose that a dataframe contains 3 columns of transaction details that include a column with date&time information and another column with numbers, and another column with categorical inforamtion. The **QUARTERLYGROUPSUM** formula will create a new dataframe that sums the transaction values on a quarter by quarter basis by group (i.e. revenues per quarter by customer). By default, quarters end in March, June, September and December. Support for other fiscal year periods is available by using the yearEndMonth parameter.

Syntax: QUARTERLYGROUPSUM(*datetime_column, group_column, sum_column, yearEndMonth*)

- ***datetime_column*.** Required. ***datetime_column*** is the dataframe column which contains the dates to be summarized on a quarterly basis.
- ***group_column***. Required. ***group_column*** is the dataframe column which contains the grouping categories.
- ***sum_column***. Required. ***sum_column*** is the dataframe column which contains the range to be summed.
- ***yearEndMonth*.** Optional. yearEndMonth is the month number that the fiscal year ends (i.e., February = 2, March = 3, etc.). Default is 12.

Video:
TBD

Python Code:

```
import pandas as pd

def QuarterlyGroupSum(idf, inDateC, inGroupC, inSumC, yearEndMonth = 12):
        group =['','Q-JAN','Q-FEB','Q-MAR','Q-APR','Q-MAY','Q-JUN',
        'Q-JUL','Q-AUG','Q-SEP','Q-OCT','Q-NOV','Q-DEC'][yearEndMonth]
        mName, gName, cName = idf.columns[[inDateC, inGroupC, inSumC]]
        idf[mName] = pd.to_datetime(idf[mName])
        idf = idf.groupby([pd.Grouper(key=mName, freq=group),
        gName]).sum().reset_index()
        idf = pd.pivot_table(idf, index=gName, columns=mName,
        values=cName).reset_index()
        cList = [idf.columns[0]]
        for i,cName in enumerate(idf.columns):
                if(i>0):
                        cList.append(cName.strftime('%m-%Y'))
        idf.columns = cList
        idf = idf.fillna(0)
        return idf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\GpuSales.csv")
df=dfIn.copy(deep=True)
df=QuarterlyGroupSum(df,0,1,2,4)
```

Group & Average By Quarter

Function: The **QUARTERLYGROUPAVG** sheet formula is used to average by quarter a dataframe column by a particular group. For example, suppose that a dataframe contains 3 columns of transaction details that include a column with date & time information and another column with numbers, and another column with categorical information. The **QUARTERLYGROUPAVG** formula will create a new dataframe that averages the transaction values on a quarter by quarter basis and by group (i.e. average transaction value per month by customer). By default, quarters end in March, June, September and December. Support for other fiscal year periods is available by using the yearEndMonth parameter.

Syntax: QUARTERLYGROUPAVG(*datetime_column, group_column, avg_column, yearEndMonth*)

- ***datetime_column*.** Required. ***datetime_column*** is the dataframe column which contains the dates to be summarized on a quarterly basis.
- ***group_column***. Required. ***group_column*** is the dataframe column which contains the grouping categories.
- ***avg_column***. Required. ***avg_column*** is the dataframe column which contains the range to be averaged.
- ***yearEndMonth*.** Optional. yearEndMonth is the month number that the fiscal year ends (i.e., February = 2, March = 3, etc.). Default is 12.

Video:
TBD

Python Code:

```
import pandas as pd

def QuarterlyGroupAvg(idf, inDateC, inGroupC, inSumC, yearEndMonth = 12):
       group=['','Q-JAN','Q-FEB','Q-MAR','Q-APR','Q-MAY','Q-JUN',
       'Q-JUL','Q-AUG','Q-SEP','Q-OCT','Q-NOV','Q-DEC'][yearEndMonth]
       mName, gName, cName = idf.columns[[inDateC, inGroupC, inSumC]]
       idf[mName] = pd.to_datetime(idf[mName])
       idf = idf.groupby([pd.Grouper(key=mName, freq=group),
       gName]).mean()[cName].reset_index()
       idf = pd.pivot_table(idf, index=gName, columns=mName,
       values=cName).reset_index()
       cList = [idf.columns[0]]
       for i,cName in enumerate(idf.columns):
              if(i>0):
                     cList.append(cName.strftime('%m-%Y'))
       idf.columns = cList
       idf = idf.fillna(0).round(2)
       return idf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\GpuSales.csv")
df=dfIn.copy(deep=True)
df=QuarterlyGroupAvg(df,0,1,2,7)
```

Group & Sum, With Quarterly Change

Function:The QUARTERLYGROUPSUMCHANGE sheet formula measures the change between each quarterly periods' sum by group. The QUARTERLYGROUPSUMCHANGE formula will create a new dataframe that measures the differences between each quarter and the previous quarter's sum value by group. By default, quarters end in March, June, September and December. Support for other fiscal year periods is available by using the yearEndMonth parameter.

Syntax: QUARTERLYGROUPSUMCHANGE (*datetime_column, group_column, sum_column, yearEndMonth*)

- ***datetime_column.*** Required. ***datetime_column*** is the dataframe column which contains the dates to be summarized on a monthly basis.
- ***group_column***. Required. ***group_column*** is the dataframe column which contains the grouping categories.
- ***sum_column***. Required. ***sum_column*** is the dataframe column which contains the range to be summed.
- ***yearEndMonth.*** Optional. yearEndMonth is the month number that the fiscal year ends (i.e., February = 2, March = 3, etc.). Default is 12.

Video:
TBD

Python Code:

```
import pandas as pd

def QuarterlyGroupSumChange(idf, inDateC, inGroupC, inSumC, yearEndMonth = 12):
	group = ['','Q-JAN','Q-FEB','Q-MAR','Q-APR','Q-MAY','Q-JUN',
	'Q-JUL','Q-AUG','Q-SEP','Q-OCT','Q-NOV','Q-DEC'][yearEndMonth]
	mName, gName, cName = idf.columns[[inDateC, inGroupC, inSumC]]
	idf[mName] = pd.to_datetime(idf[mName])
	idf = idf.groupby([pd.Grouper(key=mName, freq=group),
	gName]).sum()[cName].reset_index()
	idf = pd.pivot_table(idf, index=gName, columns=mName,
	values=cName).reset_index()
	cList = [idf.columns[0]]
	for i,cName in enumerate(idf.columns):
		if(i>0):
			cList.append(cName.strftime('%m-%Y'))
	idf.columns = cList
	idf = idf.fillna(0)
	firstCol =idf[gName]
	idf = idf.drop([gName], axis=1)
	idf = idf.diff(axis=1)
	firstName = idf.columns[0]
	idf[firstName] = idf[firstName].fillna('')
	idf.insert(loc=0, column=gName, value=firstCol)
	return idf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\GpuSales.csv")
df=dfIn.copy(deep=True)
df=QuarterlyGroupSumChange(df,0,1,2,2)
```

Sum By Year

Function: The **YEARLYSUM** sheet formula is used to sum by year a dataframe column. For example, suppose that a dataframe contains two columns of transaction details that include a column with date&time information as well as another column with numbers. The YEARLYSUM formula will create a new dataframe that sums the transaction values on an annual basis. Support for other fiscal year periods is available by using the yearEndMonth parameter. Default fiscal year end is December 31.

Syntax: YEARLYSUM(*datetime_column*, *sum_column*, *yearEndMonth*)

- ***datetime_column*.** Required. ***datetime_column*** is the dataframe column which contains the dates to be summarized on a monthly basis.
- ***sum_column***. Required. ***sum_column*** is the dataframe column which contains the range to be summed.
- ***yearEndMonth*.** Optional. yearEndMonth is the month number that the fiscal year ends (i.e., February = 2, March = 3, etc.). Default is 12.

Video:
TBD

Python Code:

```
import pandas as pd

def YearlySum(idf, inDateC, inSumC, yearEndMonth = 12):
       group = ['','A-JAN','A-FEB','A-MAR','A-APR','A-MAY','A-JUN',
       'A-JUL','A-AUG','A-SEP','A-OCT','A-NOV','A-DEC'][yearEndMonth]
       mName, cName = idf.columns[[inDateC,inSumC]]
       idf[mName] = pd.to_datetime(idf[mName])
       idf = idf.groupby(pd.Grouper(key=mName,
       freq=group)).sum()[cName].reset_index()
       idf[mName] = idf[mName].dt.strftime('%m-%Y')
       return idf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\GpuSales.csv")
df=dfIn.copy(deep=True)
df=YearlySum(df,0,2,2)
```

Average By Year

Function: The **YEARLYAVG** sheet formula is used to average a dataframe column by year. For example, suppose that a dataframe contains two columns of transaction details that include a column with date & time information as well as another column with invoice numbers. The YEARLYAVG formula will create a new dataframe that averages the invoice numbers on a yearly basis. Support for other fiscal year periods is available by using the yearEndMonth parameter. Default fiscal year end is December 31.

Syntax: YEARLYAVG(*datetime_column*, avg_column)

- ***datetime_column*.** Required. ***datetime_column*** is the dataframe column which contains the dates to be summarized on a yearly basis.
- ***avg_column***. Required. ***sum_column*** is the dataframe column which contains the values to be averaged.
- ***yearEndMonth*.** Optional. yearEndMonth is the month number that the fiscal year ends (i.e., February = 2, March = 3, etc.). Default is 12.

Video:

Python Code:

```
import pandas as pd
def YearlyAvg(idf, inDateC, inSumC, yearEndMonth = 12):
       group = ['','A-JAN','A-FEB','A-MAR','A-APR','A-MAY','A-JUN',
       'A-JUL','A-AUG','A-SEP','A-OCT','A-NOV','A-DEC'][yearEndMonth]
       mName = idf.columns[inDateC]
       idf[mName] = pd.to_datetime(idf[mName])
       idf = idf.groupby(pd.Grouper(key=mName, freq=group)).mean().reset_index()
       idf[mName] = idf[mName].dt.strftime('%m-%Y')
       idf = idf.fillna(0).round(2)
       return idf

dfIn=pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\GpuSales.csv")
df=dfIn.copy(deep=True)
df=YearlyAvg(df,0,2,3)
```

Group & Average By Year

Function: The **YEARLYGROUPAVE** sheet formula is used to average by year a dataframe column by a particular group. For example, suppose that a dataframe contains 3 columns of transaction details that include a column with date&time information and another column with numbers, and another column with categorical information. The **YEARLYGROUPAVE** formula will create a new dataframe that averages the transaction values on an annual basis and by group (i.e. average transaction value per year by customer). Support for other fiscal year periods is available by using the yearEndMonth parameter. Default fiscal year end is December 31.

Syntax: YEARLYGROUPAVE(*datetime_column*, *group_column*, *avg_column*, *yearEndMonth*)

- ***datetime_column*.** Required. ***datetime_column*** is the dataframe column which contains the dates to be summarized on a quarterly basis.
- ***group_column***. Required. ***group_column*** is the dataframe column which contains the grouping categories.
- ***avg_column***. Required. ***avg_column*** is the dataframe column which contains the range to be averaged.
- ***yearEndMonth*.** Optional. yearEndMonth is the month number that the fiscal year ends (i.e., February = 2, March = 3, etc.). Default is 12.

Video:
TBD

Python Code:

```
import pandas as pd

def YearlyGroupAve(idf, inDateC, inGroupC, inSumC, yearEndMonth = 12):
       group = ['','A-JAN','A-FEB','A-MAR','A-APR','A-MAY','A-JUN',
       'A-JUL','A-AUG','A-SEP','A-OCT','A-NOV','A-DEC'][yearEndMonth]
       mName, gName, cName = idf.columns[[inDateC, inGroupC, inSumC]]
       idf[mName] = pd.to_datetime(idf[mName])
       idf = idf.groupby([pd.Grouper(key=mName, freq=group),
       gName]).mean()[cName].reset_index()
       idf = pd.pivot_table(idf, index=gName, columns=mName,
       values=cName).reset_index()
       cList = [idf.columns[0]]
       for i,cName in enumerate(idf.columns):
              if(i>0):
                     cList.append(cName.strftime('%m-%Y'))
       idf.columns = cList
       idf = idf.fillna(0).round(2)
       return idf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\GpuSales.csv")
df=dfIn.copy(deep=True)
df=YearlyGroupAve(df,0,1,2,2)
```

Group & Sum By Year

Function: The **YEARLYGROUPSUM** sheet formula is used to sum by year a dataframe column by a particular group. For example, suppose that a dataframe contains 3 columns of transaction details that include a column with date&time information and another column with numbers, and another column with categorical information. The **YEARLYGROUPSUM** formula will create a new dataframe that sums the transaction values on an annual basis by group (i.e. revenues per quarter by customer). Support for other fiscal year periods is available by using the yearEndMonth parameter. Default fiscal year end is December 31.

Syntax: YEARLYGROUPSUM(*datetime_column*, *group_column*, *sum_column*, *yearEndMonth*)

- ***datetime_column.*** Required. ***datetime_column*** is the dataframe column which contains the dates to be summarized on a quarterly basis.
- ***group_column***. Required. ***group_column*** is the dataframe column which contains the grouping categories.
- ***sum_column***. Required. ***sum_column*** is the dataframe column which contains the range to be summed.
- ***yearEndMonth.*** Optional. yearEndMonth is the month number that the fiscal year ends (i.e., February = 2, March = 3, etc.). Default is 12.

Video:
TBD

Python Code:

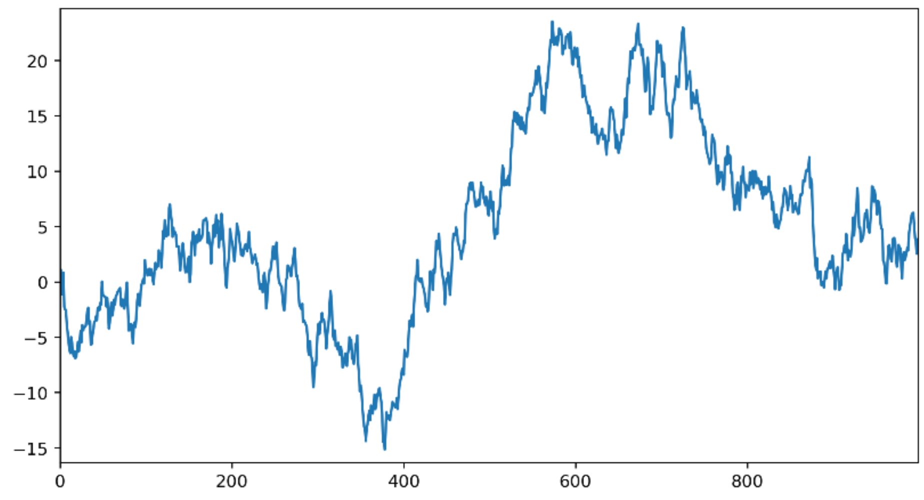
```
import pandas as pd

def YearlyGroupSum(idf, inDateC, inGroupC, inSumC, yearEndMonth = 12):
       group = ['','A-JAN','A-FEB','A-MAR','A-APR','A-MAY','A-JUN',
       'A-JUL','A-AUG','A-SEP','A-OCT','A-NOV','A-DEC'][yearEndMonth]
       mName, gName, cName = idf.columns[[inDateC, inGroupC, inSumC]]
       idf[mName] = pd.to_datetime(idf[mName])
       idf = idf.groupby([pd.Grouper(key=mName, freq=group),
       gName]).sum()[cName].reset_index()
       idf = pd.pivot_table(idf, index=gName, columns=mName,
       values=cName).reset_index()
       cList = [idf.columns[0]]
       for i,cName in enumerate(idf.columns):
              if(i>0):
                     cList.append(cName.strftime('%m-%Y'))
       idf.columns = cList
       idf = idf.fillna(0)
       return idf


dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\GpuSales.csv")
df=dfIn.copy(deep=True)
df=YearlyGroupSum(df,0,1,2,3)
```

Charting Formulas

Line Plot

Function: The **LINEPLOT** sheet formula displays data points as a series of connected line segments. For example, if a dataframe contains a column of numeric data. The **LINEPLOT** function will plot the numeric data sequentially with line segments connecting each data point.

Example:



Syntax: LINEPLOT(*value_column*)

- ***value_column*.** Required. **value_column** is the column which contains the values that are to be plotted on the line chart.

Video:
TBD

Python Code:

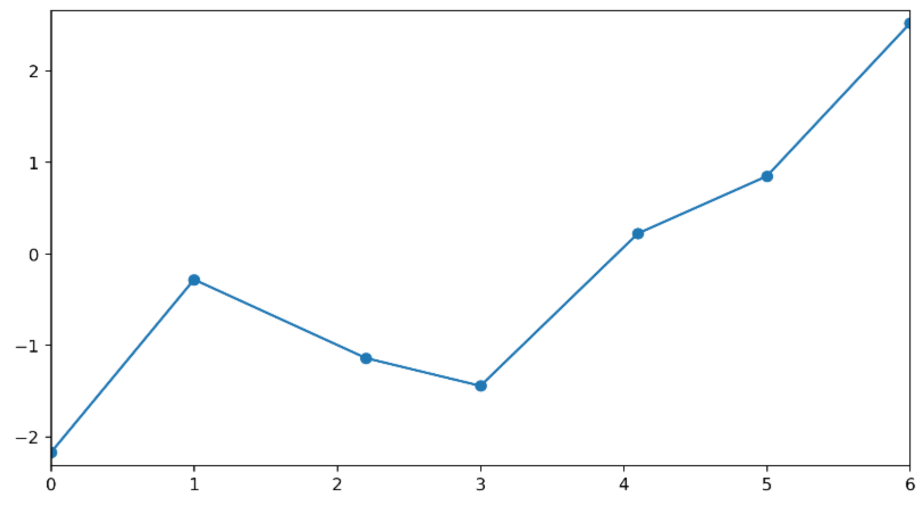
```
import matplotlib.pyplot as plt
import pandas as pd

def LinePlot(inDf, inCol1):
	cName = inDf.columns[inCol1]
	plt.figure(figsize=(9,5))
	plt.margins(0, 0.03)
	plt.plot(inDf[cName])
```

```
dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\traffic.csv")
df=dfIn.copy(deep=True)
LinePlot(df,0)
```

Connected Scatter Plot

Function: The **LINESCATTER** sheet formula displays data as a series of points connected by line segments. For example, if a dataframe contains a column of numeric data. The **LINESCATTER** function will plot the numeric data points sequentially and connect the data points with line segments.

Example:



Syntax: LINESCATTER(*x_values_column, y_values_column*)

- ***x_value_column.*** Required. **x_value_column** is the column which contains the x coordinates of each data point.
- ***y_value_column.*** Required. **y_value_column** is the column which contains the y coordinates of each data point.

Video:
TBD

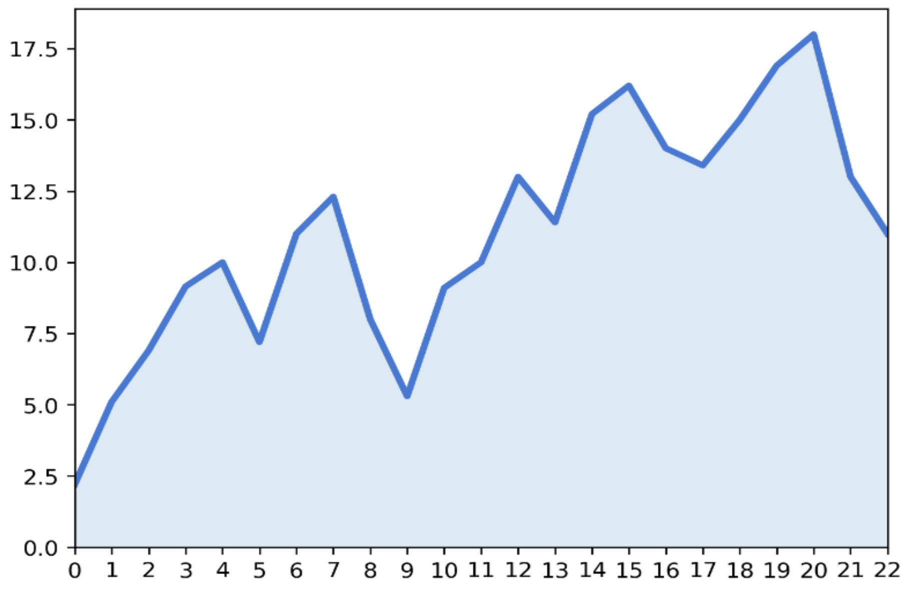
Python Code:

```
import matplotlib.pyplot as plt
import pandas as pd

def LineScatter(inDf, inCol1, inCol2):
       cName1 = inDf.columns[inCol1]
       cName2 = inDf.columns[inCol2]
       plt.figure(figsize=(9,5))
       plt.margins(0, 0.03)
       plt.plot(inDf[cName1], inDf[cName2],marker='o')

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\datapoints.csv")
df=dfIn.copy(deep=True)
LineScatter(df,0,1)
```

Area Plot

Function: The **AREAPLOT** sheet formula takes in a numeric series that serves as labels and a series of numeric values that serve as the quantities measured. The function then plots the numeric/label series (on the x axis) against the corresponding numeric values (on the y axis) and colors in the area under the line created by the plot. For example, if a dataframe contains a column of integers in sequential order, and a column with numeric values. The **AREAPLOT** function will plot the integers on the horizontal axis and the numeric values on the vertical axis at the corresponding value on the horizontal axis. The area under the line created will then be filled in with a color.

Example:



Syntax: AREAPLOT(*label_column*, *value_column*)

- ***label_column.*** Required. **label_column** is the column which contains the numeric labels that are to be placed on the x axis
- ***value_column.*** Required. **value_column** is the column which contains the values that are to be plotted against the labels.

Video:
TBD

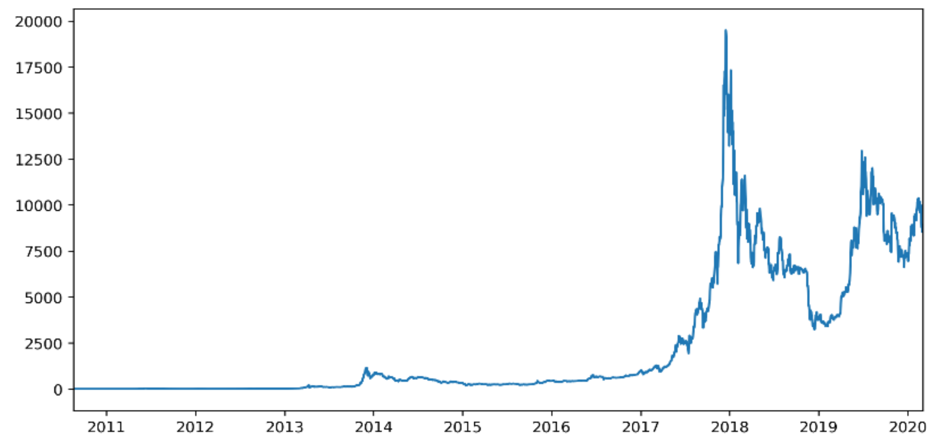
Python Code:

```
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns

def AreaPlot(inDf, inCol1, inCol2):
        x = inDf[inDf.columns[inCol1]]
        y = inDf[inDf.columns[inCol2]]
        blue, = sns.color_palette("muted", 1)
        fig, ax = plt.subplots()
        ax.plot(x, y, color=blue, lw=3)
        ax.fill_between(x, 0, y, alpha=.15)
        ax.set(xlim=(0, len(x) - 1), ylim=(0, None), xticks=x)

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\areadata.csv")
df=dfIn.copy(deep=True)
AreaPlot(df,0,1)
```

Time Series

Function: The **TIMESERIES** sheet formula creates a plot where some measure of time is plotted on the x axis and the y axis contains data that is being measured. For example, if a dataframe contains a column with datetime data and another column with numeric data, The **TIMESERIES** function will plot the numeric data on the y axis respective to its corresponding datetime on the x axis.

Example:



Syntax: TIMESERIES(*datetime_column*, *value_column*)

- ***datetime_column*.** Required. ***datetime_column*** is the dataframe column which contains the dates to be plotted on the x-axis.
- ***value_column*.** Required. **value_column is** the column which contains the values that are to be plotted on the y axis corresponding to their respective dates.

Video:
TBD

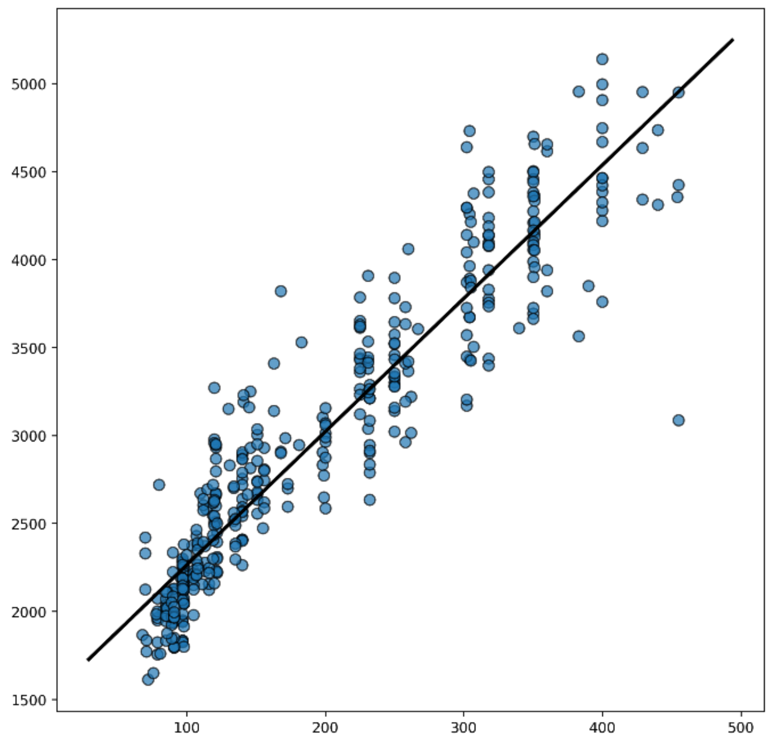
Python Code:

```
import matplotlib.dates as md
import matplotlib.pyplot as plt
import pandas as pd

def TimeSeries(inDf, inDateI, inValI):
       values = inDf[inDf.columns[inValI]]
       dates = inDf[inDf.columns[inDateI]]
       converted_dates = md.datestr2num(dates)
       x_axis = (converted_dates)
       plt.figure(figsize=(10,5))
       plt.margins(0, 0.06)
       plt.plot_date(x_axis, values, '-',color='#1F77B4')

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\bitcoin.csv")
df=dfIn.copy(deep=True)
TimeSeries(df,0,1)
```

Linear Regression

Function: The **LINREG** sheet formula is used to run a simple (single predictor OLS) regression. For example, suppose that a dataframe contains 2 columns of information about a car. One column contains the miles per gallon of the car and the other column contains information about the engine displacement. The **LINREG** function will solve for the slope (m) and intercept (b) of the formula: mpg = m*displacement + b, by minimizing the sum of squared errors between the data points and the line produced by the formula above. This formula answers the question of what kind of effect displacement has on mpg. If the slope (coefficient) is negative, increasing displacement will (on average) decrease mpg. If the coefficient is positive, increasing displacement will (on average) increase mpg. The intercept is interpreted as the mpg value for a car with 0 displacement.

Example:



Syntax: LINREG(in*dependent_variable, dependent_variable*)

- ***independent_variable.*** Required. The column which contains the variable that will be used to make predictions. (Displacement in the example above)
- ***dependent_variable.*** Required. The column which contains the variable that is the outcome. (Mpg in the example above)

Video:
TBD

Python Code:

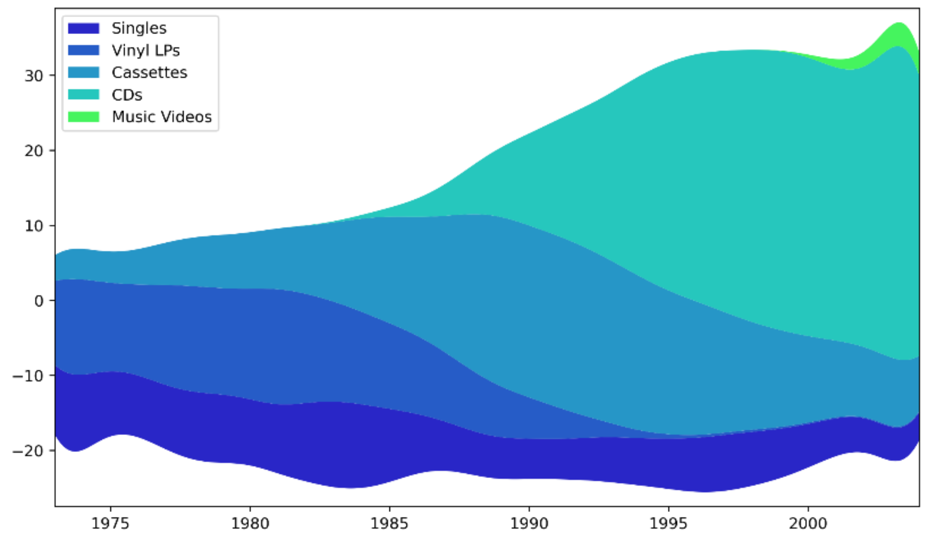
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

```
def LinReg(inDf, inInd1, inInd2):
	values1 = inDf[inDf.columns[inInd1]]
	values2 = inDf[inDf.columns[inInd2]]
	fig, ax = plt.subplots(figsize = (9, 9))
	ax.scatter(values1, values2, s=60, alpha=0.7, edgecolors="k")
	b, a = np.polyfit(values1, values2, deg=1)
	print("intercept: ", a)
	print("coefficient: ", b)
	xmin = min(values1)
	xmax = max(values1)
	xrng = xmax - xmin
	xseq = np.linspace(xmin-.1*xrng, xmax+.1*xrng, num=10)
	ax.plot(xseq, a + b * xseq, color="k", lw=2.5)

dfIn = pd.read_csv("C:\\Program Files\\Row64\\Row64_V1_3\\Data\\Example\\mpg.csv")
df=dfIn.copy(deep=True)
LinReg(df,2,0)
```

Stream Graph

Function: The **STREAMGRAPH** sheet formula is a type of stacked area chart. The function plots the evolution of a list of numeric columns (plotted on the y axis) in relation to another numeric column (plotted on the x axis). Each column has a distinct colored area associated with it and edges are rounded for appearance. For example, if a dataframe contains five columns of numeric data and another column of years. The **STREAMGRAPH** function will plot the years on the horizontal axis and the numeric values on the vertical axis (stacked on top of one another) at the corresponding year on the horizontal axis. Changes in the five numeric columns shown are relative to the whole and are not meant to be interpreted as the actual value change.

Example:



Syntax: STREAMGRAPH(*label_column*, *value_column_list*)

- ***label_column.*** Required. **label_column** is the column which contains the numeric labels that are to be placed on the x axis
- ***value_column_list.*** Required. **value_column_list** is the list of columns which contain the values that are to be plotted evolutionarily against the labels.

Video:
TBD

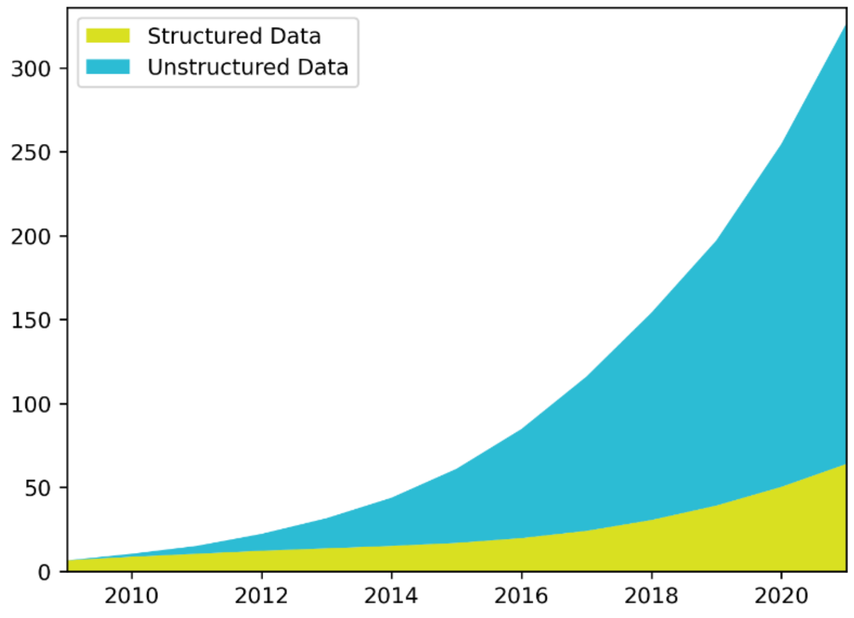
Python Code:

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from scipy import stats

def GaussianSmooth(x, y, grid, sd):
    weights = np.transpose([stats.norm.pdf(grid, m, sd) for m in x])
    weights = weights / weights.sum(0)
    return (weights * y).sum(1)

def Streamgraph(inDf1,inXI,inYIList):
	cList = ["#2926C7","#265CC7","#2696C7","#26C7BD","#45F45E"]
	xData = inDf1[inDf1.columns[inXI]]
	yList = []
	lNames = []
	for ind in inYIList:
		yList.append(inDf1[inDf1.columns[ind]].values);
		lNames.append(inDf1.columns[ind])
	grid = np.linspace(xData.iloc[0], xData.iloc[-1], num=2000)
	y_smoothed = [GaussianSmooth(xData.values, y_, grid, 1) for y_ in yList]
	plt.figure(figsize=(10,6))
	plt.stackplot(grid, y_smoothed,baseline="wiggle",colors=cList,labels=lNames)
	plt.margins(0, 0.03)
	plt.legend(loc='upper left')

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\music.csv")
df=dfIn.copy(deep=True)
Streamgraph(df,0,[1,2,3,4,5])
```

Stack Plot

Function: The **STACKPLOT** sheet formula creates a stacked area plot. The function takes in a numeric/datetime series and a list of numeric series'. The function then plots the numeric/datetime series (on the x axis) against all corresponding numeric values (stacked on the y axis) and colors in the area under each stacked numeric value. For example, if a dataframe contains five columns of numeric data and another column of years. The **STACKPLOT** function will plot the years on the horizontal axis and the stacked numeric values on the vertical axis at the corresponding value on the horizontal axis. The area under each value will be filled with a color and each value will be connected to the next respective value with a line segment.

Example:



Syntax: STACKPLOT(label_column, *value_column*)

- ***label_column.*** Required. **label_column** is the column which contains the numeric/datetime labels that are to be placed on the x axis
- ***value_column_list.*** Required. **value_column_list** is the list of columns which contain the values that are to be plotted in a stacked fashion against their respective labels.

Video:
TBD

Python Code:

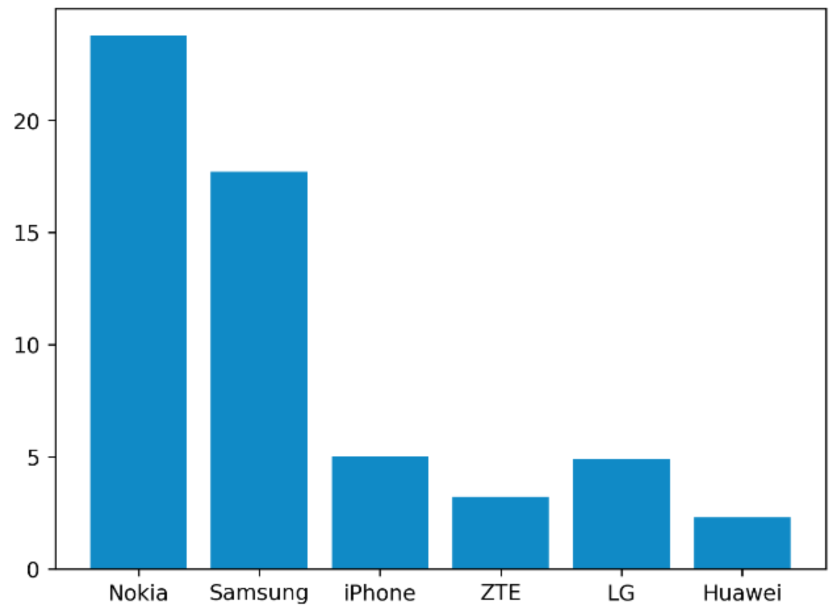
```
import matplotlib.pyplot as plt
import pandas as pd

def StackPlot(inDf1,inI1, inYIList):
        color_map = ["#D9E021", "#2CBCD4","#1679D3"]
        xVals = inDf1[inDf1.columns[inI1]]
```

```
       yList = []
       lNames = []
       for ind in inYIList:
              yList.append(inDf1[inDf1.columns[ind]].values);
              lNames.append(inDf1.columns[ind])
       plt.stackplot(xVals,yList,colors=color_map,labels=lNames)
       plt.legend(loc='upper left')
       plt.margins(0, 0.03)

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\unstructured.csv")
df=dfIn.copy(deep=True)
StackPlot(df,0,[1,2])
```

Basic Bar

Function: The **BASICBAR** sheet formula creates a basic bar chart by taking in one column of categorical data and one column of numeric data. For example, if a dataframe contains a column of cellular carriers and a column of numeric data. The **BASICBAR** function will create a basic bar chart with the categorical data on the x axis and the numeric data determining the height of the bars for each category.

Example:



Syntax: BASICBAR(*categorical_column*, *value_column*)

- ***categorical_column*.** Required. **categorical_column** is the column which contains the categorical data that are to be plotted on the x axis
- ***value_column*.** Required. **value_column** is the column which contains the values that determine the height of each bar.

Video:
TBD

Python Code:

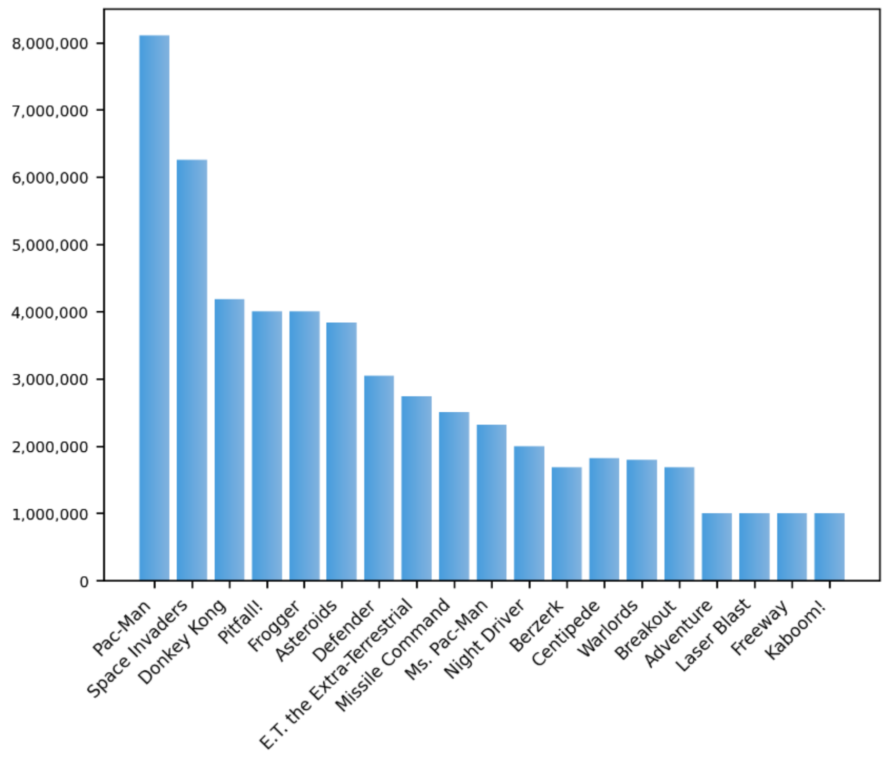
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

def BasicBar(inDf, inNameI, inValueI):
       values = inDf[inDf.columns[inValueI]]
       names = inDf[inDf.columns[inNameI]]
       y_pos = np.arange(len(names))
       plt.bar(y_pos, values, color='#108AC6')
       plt.xticks(y_pos, names)
```

```
dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\phones.csv")
df=dfIn.copy(deep=True)
BasicBar(df,0,1)
```

Gradient Bar

Function: The **GRADIENTBAR** sheet formula creates a bar chart by taking in one column of categorical data and one column of numeric data. For example, if a dataframe contains a column of cellular carriers and a column of numeric data. The **GRADIENTBAR** function will create a basic bar chart with the categorical data on the x axis and the numeric data determining the height of the bars for each category.

Example:



Syntax: GRADIENTBAR(*value_column, categorical_column*)

- ***value_column.*** Required. **value_column** is the column which contains the values that determine the height of each bar.
- ***categorical_column.*** Required. **categorical_column** is the column which contains the categorical data that are to be plotted on the x axis

Video:
TBD

Python Code:

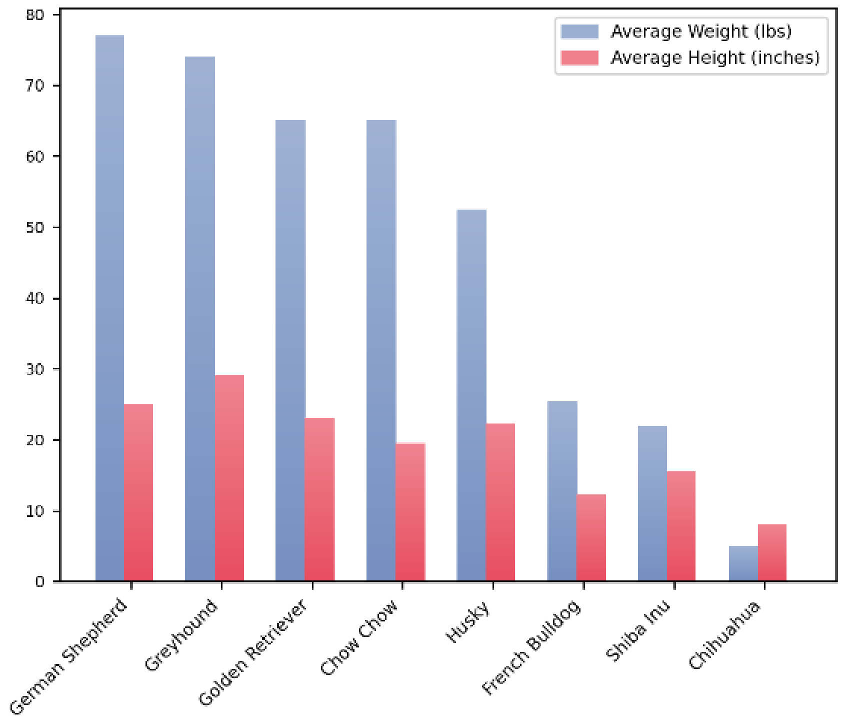
```
import matplotlib
import matplotlib.colors as cl
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

```
def GradientBars(inDf, inValueI, inNameI):
	cList = ['#479DDD', '#83B3DF']
	values = inDf[inDf.columns[inValueI]]
	names = inDf[inDf.columns[inNameI]]
	plt.rc('xtick', labelsize=8)
	plt.rc('ytick', labelsize=7)
	x_pos = np.arange(len(names))
	plt.xticks(x_pos, names,horizontalalignment='right',rotation=45)
	cRGB = [cl.to_rgb(col) for col in cList]
	bars = plt.bar(x_pos, values, color=cList[0])
	cmap = cl.LinearSegmentedColormap.from_list("", cRGB)
	ax = bars[0].axes
	lim = ax.get_xlim()+ax.get_ylim()
	for bar in bars:
		bar.set_facecolor("none")
		x,y = bar.get_xy()
		w, h = bar.get_width(), bar.get_height()
		grad = np.atleast_2d(np.linspace(0,1*w/max(values),256))
		ax.imshow(grad, extent=[x,x+w,y,y+h], aspect="auto", zorder=0, 
cmap=cmap)
	ax.axis(lim)
	ax.get_yaxis().set_major_formatter(matplotlib.ticker.FuncFormatter(lambda x, 
p: format(int(x), ',')))

dfIn = pd.read_csv("C:\\Program 
Files\\Row64\\Row64_V1_3\\Data\\Example\\atari.csv")
df=dfIn.copy(deep=True)
GradientBars(df,4,0)
```

Multi Bar (needs to be changed)

Function: The **MULTIBAR** sheet formula creates a multiple bar chart by taking in two columns of categorical data and one column of numeric data. For example, if a dataframe contains a column of cellular carriers and a column of numeric data. The **MULTIBAR** function will create a basic bar chart with the categorical data on the x axis and the numeric data determining the height of the bars for each category.

Example:



Syntax: MULTIBAR(*value_column, categorical_column*)

- ***value_column.*** Required. **value_column** is the column which contains the values that determine the height of each bar.
- ***categorical_column.*** Required. **categorical_column** is the column which contains the categorical data that are to be plotted on the x axis

Video:
TBD

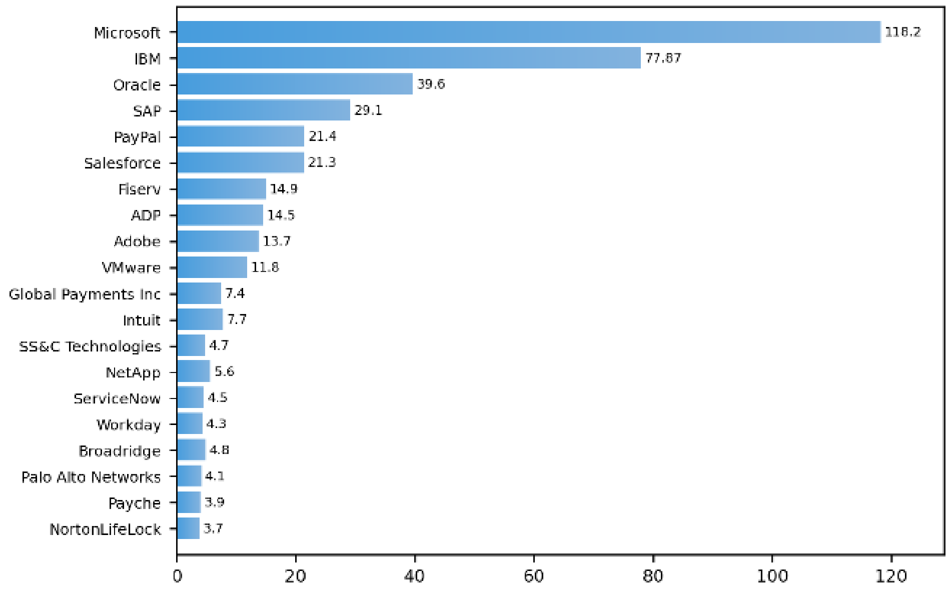
Python Code:

```
import matplotlib
import matplotlib.colors as cl
import matplotlib.patches as mpatches
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

def MultiBars(inDf, inNameI, inValI1, inValI2, inLabel1, inLabel2 ):
	widthFactor = 0.4
	cList = [['#9FB2D4','#7590C1'],['#EF8490','#E84E60']]
	labels = [inLabel1, inLabel2]
	vals = [inDf[inDf.columns[inValI1]], inDf[inDf.columns[inValI2]] ]
	names = inDf[inDf.columns[inNameI]]
	plt.rc('xtick', labelsize=8)
	plt.rc('ytick', labelsize=7)
	hList = []
	x_pos = np.arange(len(names))
	plt.xticks(x_pos, names, horizontalalignment='right',rotation=45)
	for i in range(2):
		cRGB = [cl.to_rgb(col) for col in cList[i]]
		cmap = matplotlib.colors.LinearSegmentedColormap.from_list("", cRGB)
		bars = plt.bar(x_pos, vals[i])
		ax = bars[0].axes
		lim = ax.get_xlim()+ax.get_ylim()
		for bar in bars:
			bar.set_facecolor("none")
			w, h = bar.get_width()* widthFactor, bar.get_height()
			x,y = bar.get_xy()
			if i==1:x+=w;
			grad = np.linspace(y, y + h, 256).reshape(256, 1)
			ax.imshow(grad, extent=[x,x+w,y,y+h], aspect="auto", zorder=0,
cmap=cmap)
		ax.axis(lim)
		hList.append(mpatches.Patch(color=cmap(.1), label=labels[i]))
	plt.legend(handles=hList, prop={'size': 8})

dfIn = pd.read_csv("C:\\Program Files\\Row64\\Row64_V1_3\\Data\\Example\\dogs.csv")
df=dfIn.copy(deep=True)
MultiBars(df,0,1,2,"Average Weight (lbs)","Average Height (inches)")
```

Horizontal Gradient Bar

Function: The **GRADIENTBARSH** sheet formula creates a basic bar chart by taking in one column of categorical data and one column of numeric data. For example, if a dataframe contains a column of software companies and a column of numeric data. The **GRADIENTBARSH** function will create a basic bar chart with the categorical data on the y axis and the numeric data determining the length of the bars for each category.

Example:



Syntax: GRADIENTBARSH(*value_column, categorical_column*)

- ***value_column.*** Required. **value_column** is the column which contains the values that determine the length of each bar.
- ***categorical_column.*** Required. **categorical_column** is the column which contains the categorical data that are to be plotted on the y axis

Video:
TBD

Python Code:

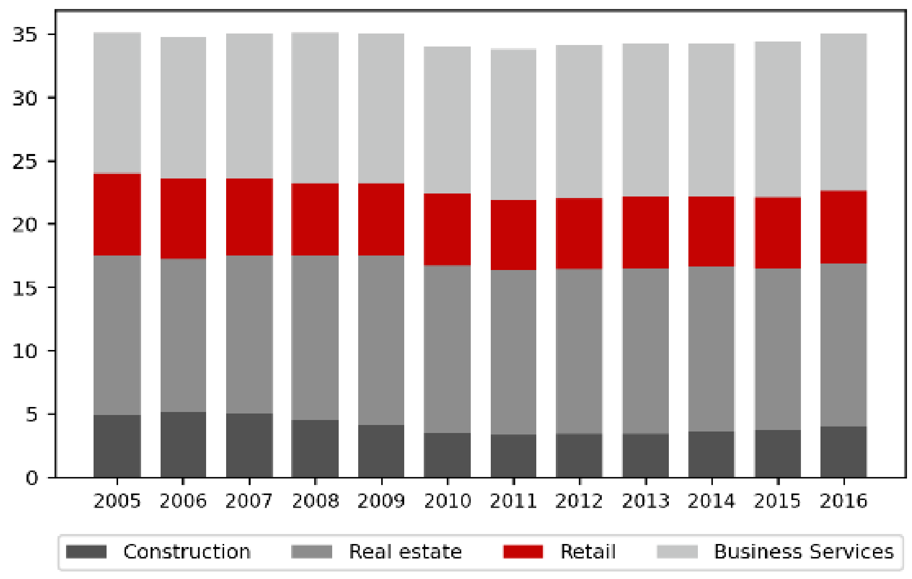
```
import matplotlib
import matplotlib.colors as cl
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

def GradientBarsH(inDf, inValueI, inNameI):
       cList = ['#479DDD', '#83B3DF']
       values = inDf[inDf.columns[inValueI]]
       names = inDf[inDf.columns[inNameI]]
       plt.rc('xtick', labelsize=8)
```

```
        plt.rc('ytick', labelsize=7)
        y_pos = np.arange(len(names))*-1
        plt.yticks(y_pos, names)
        plt.margins(0.09, 0.03)
        cRGB = [cl.to_rgb(col) for col in cList]
        bars = plt.barh(y_pos, values, color=cList[0])
        cmap = matplotlib.colors.LinearSegmentedColormap.from_list("", cRGB)
        ax = bars[0].axes
        lim = ax.get_xlim()+ax.get_ylim()
        for bar in bars:
                bar.set_facecolor("none")
                x,y = bar.get_xy()
                w, h = bar.get_width(), bar.get_height()
                grad = np.atleast_2d(np.linspace(0,1*w/max(values),256))
                ax.imshow(grad, extent=[x,x+w,y,y+h], aspect="auto", zorder=0,
cmap=cmap)
        ax.axis(lim)
        for i, v in enumerate(values):
                ax.text(v+.7, y_pos[i]-.15, str(v), fontdict={'fontsize':6.1})

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\software.csv")
df=dfIn.copy(deep=True)
GradientBarsH(df,2,1)
```

Stacked Bars

Function: The **STACKBAR** sheet formula creates a stacked bar chart by taking in one column of categorical or numeric data and multiple other columns of numeric data. For example, if a dataframe contains a column of years and four numeric columns with income data. The **STACKBAR** function will create a stacked bar chart with the categorical data on the x axis and the numeric data of each column stacked on top of one another for each respective year.

Example:



Syntax: STACKBAR(*categorical_column, value_column_list*)

- ***categorical_column.*** Required. **categorical_column** is the column which contains the categorical data that will be plotted on the x axis.
- ***value_column_list.*** Required. **value_column_list** is the list of columns which contains the numeric data that are to be stacked as bars.

Video:
TBD

Python Code:

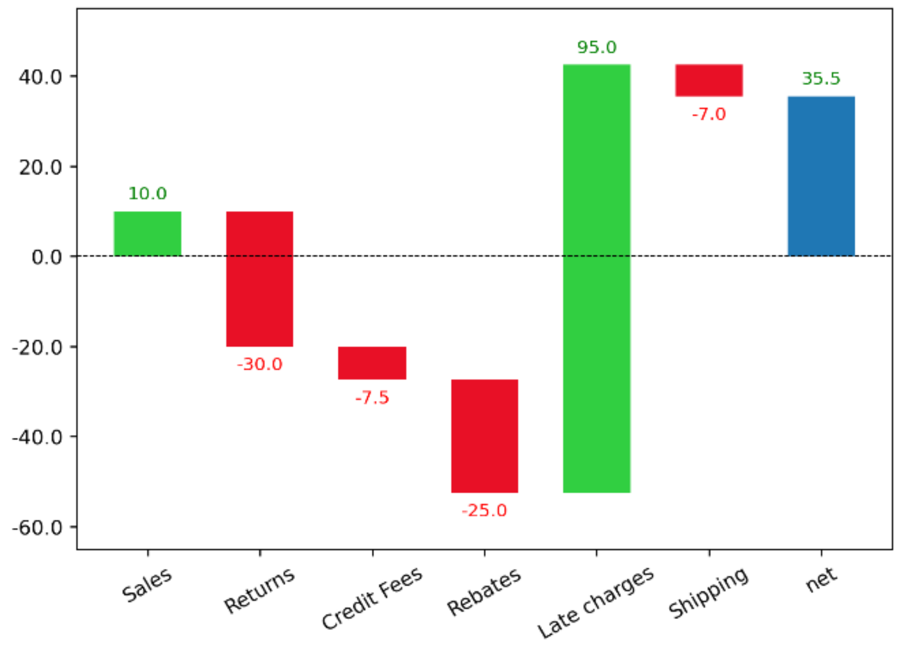
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

def StackedBars(inDf, inI, inIList):
       cList = ['#525252','#8D8D8D','#C50302','#C4C5C5']
       names = inDf[inDf.columns[inI]]
       x_pos = np.arange(len(names))
       y_pos = np.zeros(len(names))
       lables = [];
       for i,ind in enumerate(inIList):
```

```
                lables.append(inDf.columns[ind])
                if i==0:plt.bar(x_pos, inDf[inDf.columns[ind]],
color=cList[i],width=0.7)
                else:plt.bar(x_pos, inDf[inDf.columns[ind]], bottom=y_pos,
color=cList[i],width=0.7)
                y_pos += inDf[inDf.columns[ind]];
        plt.xticks(x_pos, names, fontsize=8)
        plt.yticks(fontsize=9)
        plt.gca().set_aspect(.2)
        plt.legend(lables, fontsize=8.5, loc='lower
center',bbox_to_anchor=(.5,-.22), ncol=len(inIList))

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\industry.csv")
df=dfIn.copy(deep=True)
StackedBars(df,0,[1,2,3,4])
```

Waterfall

Function: The **WATERFALL** sheet formula creates a waterfall chart by taking in one column of categorical data and one column of numeric data. A waterfall chart shows a running total as values are added or subtracted. For example, if a dataframe contains a column of transaction types and a column with numeric data on the transactions. The **WATERFALL** function will create a chart that has bars representing the direction and magnitude of each transaction along the x axis.

Example:



Syntax: WATERFALL(*label_column, data_column*)

- ***label_column*****.** Required. **label_column** is the column which contains the categorical data that will be plotted on the x axis.
- ***data_column*****.** Required. **data_column** is the list of columns which contains the numeric data that are to be stacked as bars.

Video:
TBD

Python Code:

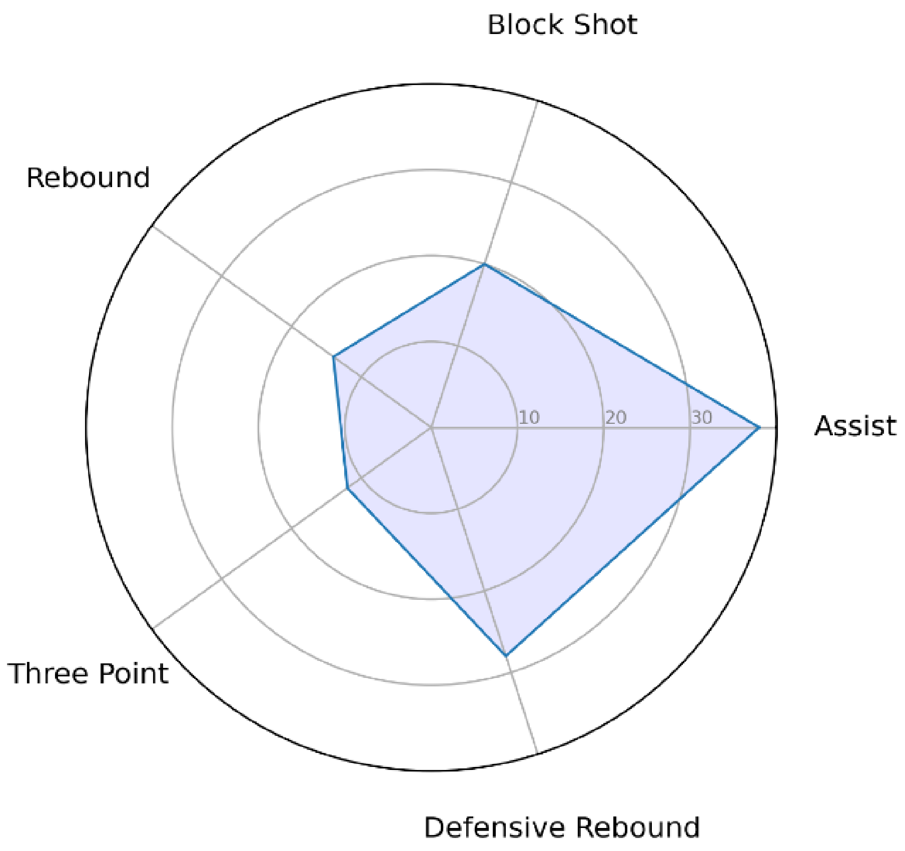
```
import pandas as pd
import waterfall_chart

def Waterfall(inDf, inLabelI, inDataI):
       labelCol = inDf.columns[inLabelI]
       dataCol = inDf.columns[inDataI]
       waterfall_chart.plot(inDf[labelCol], inDf[dataCol],
       red_color='#E81026',green_color='#31CF41', blue_color='#1F77B4')
```

```
dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\salesProcess.csv")
df=dfIn.copy(deep=True)
Waterfall(df,0,1)
```

Radar

Function: The **RADAR** sheet formula creates a radar chart by taking in one column of categorical data and multiple columns of numeric data. A radar chart is used to compare multiple. For example, if a dataframe contains a column of transaction types and a column with numeric data on the transactions. The **RADAR** function will create a chart that has bars representing the direction and magnitude of each transaction along the x axis.

Example:



Syntax: RADAR(*categorical_column, value_column_list*)

- ***categorical_column.*** Required. **categorical_column** is the column which contains the categorical data that will create the axes for the numeric data to lie on.
- ***value_column_list.*** Required. **value_column_list** is the list of columns which contains the numeric data that determine how far out the data lies on each respective axis.

Video:
TBD

Python Code:

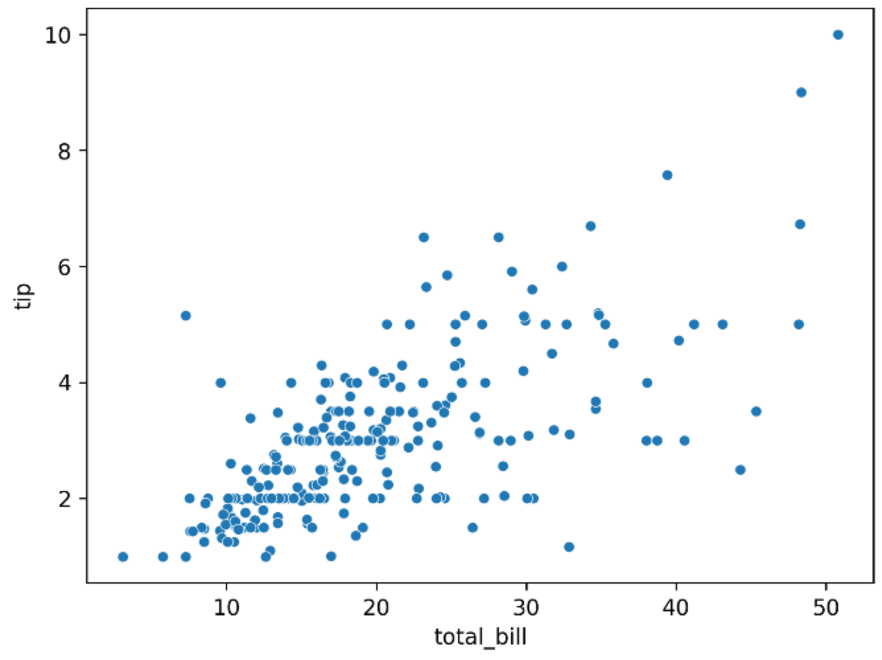
```
import matplotlib.pyplot as plt
import pandas as pd
```

```
from math import pi

def Radar(inDf, inCatI, inRowI):
	col = inDf.columns[inCatI]
	valDf = inDf.drop(columns=col)
	categories=valDf.columns
	N = len(categories)
	values = valDf.loc[inRowI].values.flatten().tolist()
	values += values[:1]
	angles = [n / float(N) * 2 * pi for n in range(N)]
	angles += angles[:1]
	ax = plt.subplot(111, polar=True)
	plt.xticks(angles[:-1], categories, size=11)
	ax.set_rlabel_position(0)
	plt.yticks([10,20,30], ["10","20","30"], color="grey", size=7)
	plt.ylim(0,40)
	ax.xaxis.set_tick_params(pad=20)
	ax.plot(angles, values, linewidth=1, linestyle='solid')
	ax.fill(angles, values, 'b', alpha=0.1)

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\bball.csv")
df=dfIn.copy(deep=True)
Radar(df,0,0)
```

Scatter Plot

Function: The **SCATTERPLOT** sheet formula displays data as a series of points on the x-y axis. For example, if a dataframe contains two columns of numeric data. The **SCATTERPLOT** function will plot the data in the first column on the x axis and the data in the second column on the y axis.

Example:



Syntax: SCATTERPLOT(*x_values_column, y_values_column*)

- ***x_value_column.*** Required. **x_value_column** is the column which contains the x coordinates of each data point.
- ***y_value_column.*** Required. **y_value_column** is the column which contains the y coordinates of each data point.

Video:
TBD

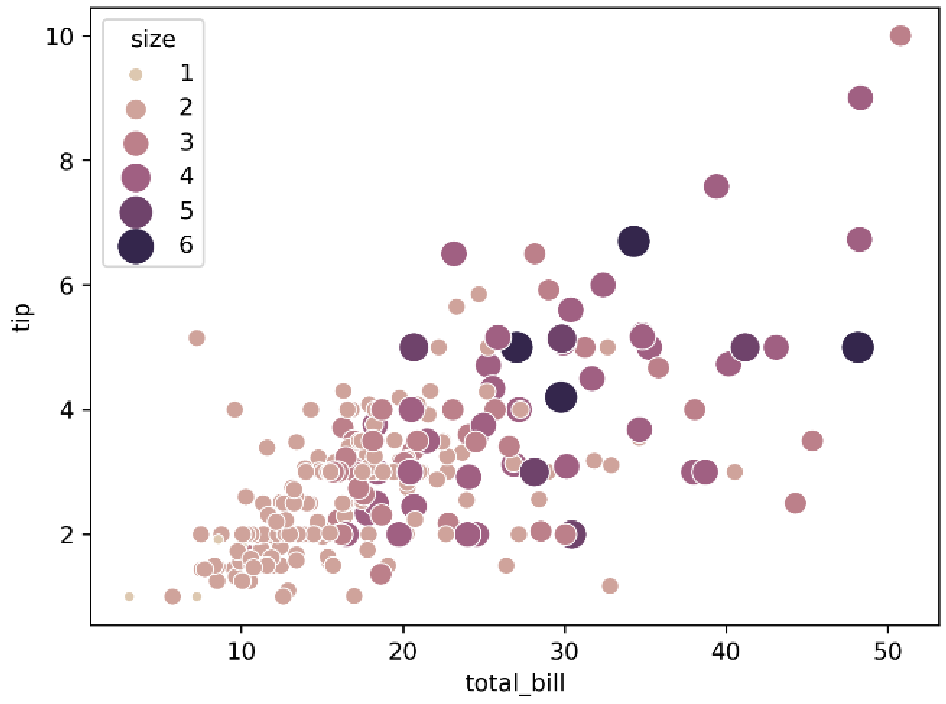
Python Code:

```
import matplotlib.colors as cl
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns

def ScatterPlotBasic(inDf, inCol1, inCol2):
       col1 = inDf.columns[inCol1]
       col2 = inDf.columns[inCol2]
       sns.scatterplot(color="#1F77B4",data=inDf,x=col1,y=col2,s=25)

dfIn = pd.read_csv("C:\\Program Files\\Row64\\Row64_V1_3\\Data\\Example\\tips.csv")
df=dfIn.copy(deep=True)
```

```
ScatterPlotBasic(df,0,1)
```

Bubble Plot

Function: The **BUBBLEPLOT** sheet formula displays data as a series of points on the x-y axis with different sizes. For example, if a dataframe contains three columns of numeric data. The **BUBBLEPLOT** function will plot the data in the first column on the x axis, the data in the second column on the y axis, and the data in the third column will determine the size of each data point.
Example:



Syntax: BUBBLEPLOT(*x_values_column, y_values_column, size_values_column*)

- ***x_value_column.*** Required. **x_value_column** is the column which contains the x coordinates of each data point.
- ***y_value_column.*** Required. **y_value_column** is the column which contains the y coordinates of each data point.
- ***size_value_column.*** Required. **size_valuse_column** is the column which contains the values that change the size for each data point.

Video:
TBD

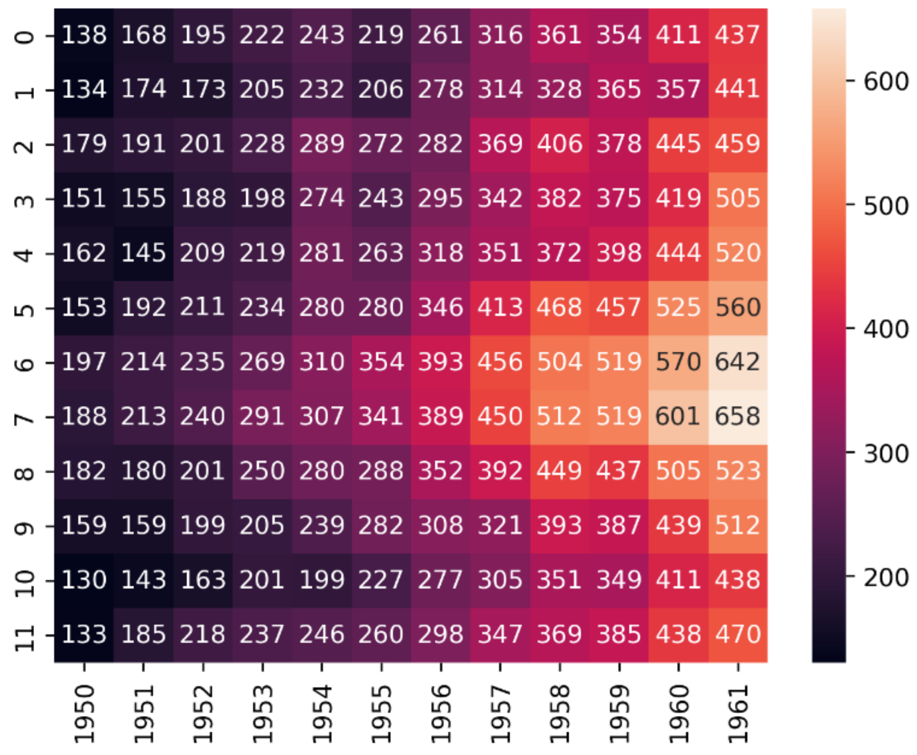
Python Code:

```
import matplotlib.colors as cl
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns

def BubblePlot(inDf, inCol1, inCol2, inCol3):
        col1 = inDf.columns[inCol1]
        col2 = inDf.columns[inCol2]
        col3 = inDf.columns[inCol3]
        cList = ['#DDC8B0', '#C68A8D', '#96567F' , '#34264C']
        cRGB = [cl.to_rgb(col) for col in cList]
        cmap = cl.LinearSegmentedColormap.from_list("", cRGB)

sns.scatterplot(palette=cmap,data=inDf,x=col1,y=col2,hue=col3,size=col3,sizes=(20,2
00),legend="full")

dfIn = pd.read_csv("C:\\Program Files\\Row64\\Row64_V1_3\\Data\\Example\\tips.csv")
df=dfIn.copy(deep=True)
BubblePlot(df,0,1,6)
```

Heatmap

Function: The **HEATMAP** sheet formula creates a heatmap from a dataframe. A heatmap takes the values in each row-column position and colors them according to the magnitude of the value in each position.

Example:



Syntax: HEATMAP(*dataframe*)

- ***dataframe.*** Required. **dataframe** is the dataframe that will be passed in to make a heatmap.

Video:
TBD

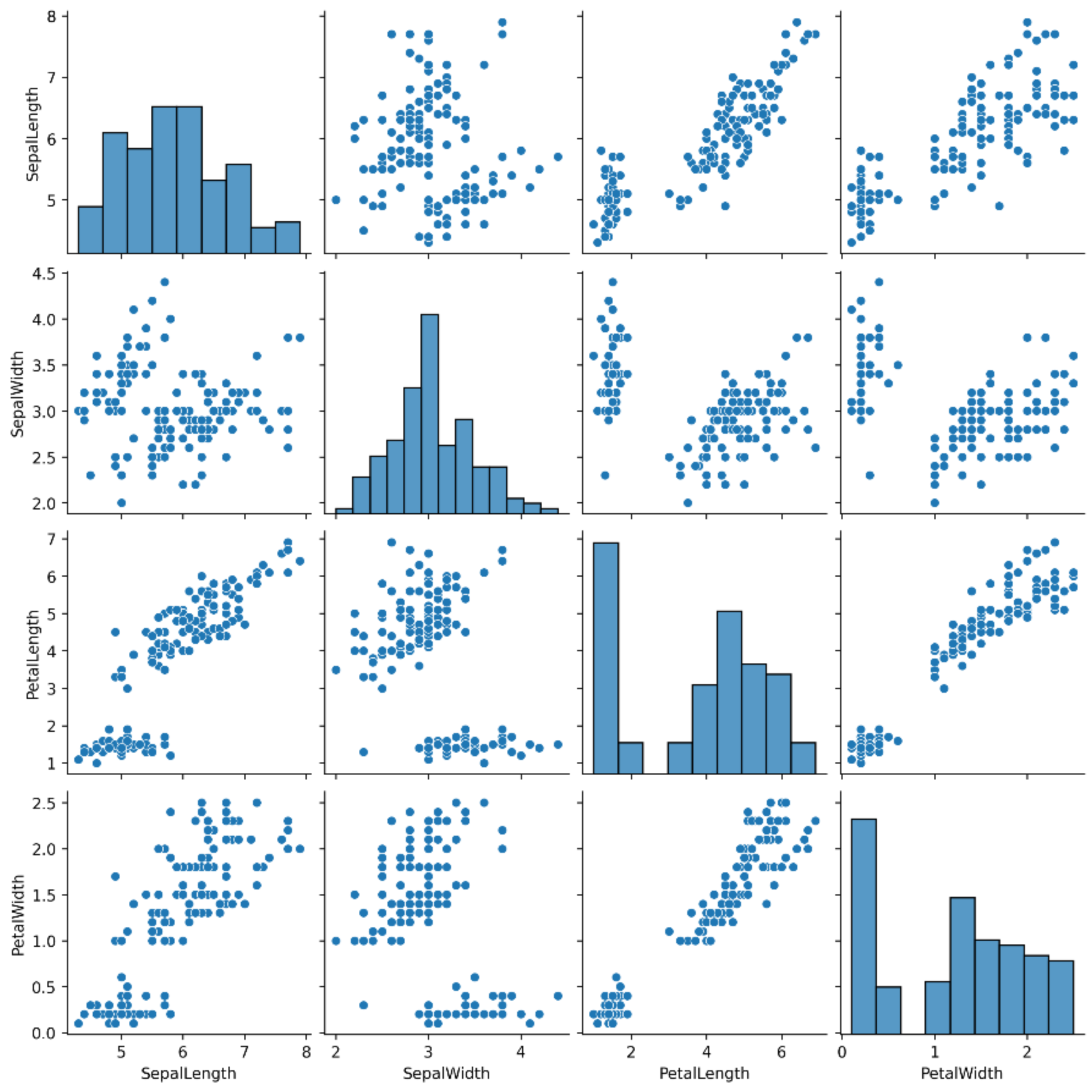
Python Code:

```
import pandas as pd
import seaborn as sns

def HeatMap(inDf):
       sns.heatmap(inDf, annot=True, fmt="d")

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\flights.csv")
df=dfIn.copy(deep=True)
HeatMap(df)
```

Pairplot

Function: The **PAIRPLOT** sheet formula creates a grid of charts from every row and column pair in the dataframe. A heatmap takes the values in each row-column position and colors them according to the magnitude of the value in each position.

Example:



Syntax: PAIRPLOT()

- The pairplot function will take in whichever dataframe is on the same tab being worked in.

Video:
TBD

Python Code:

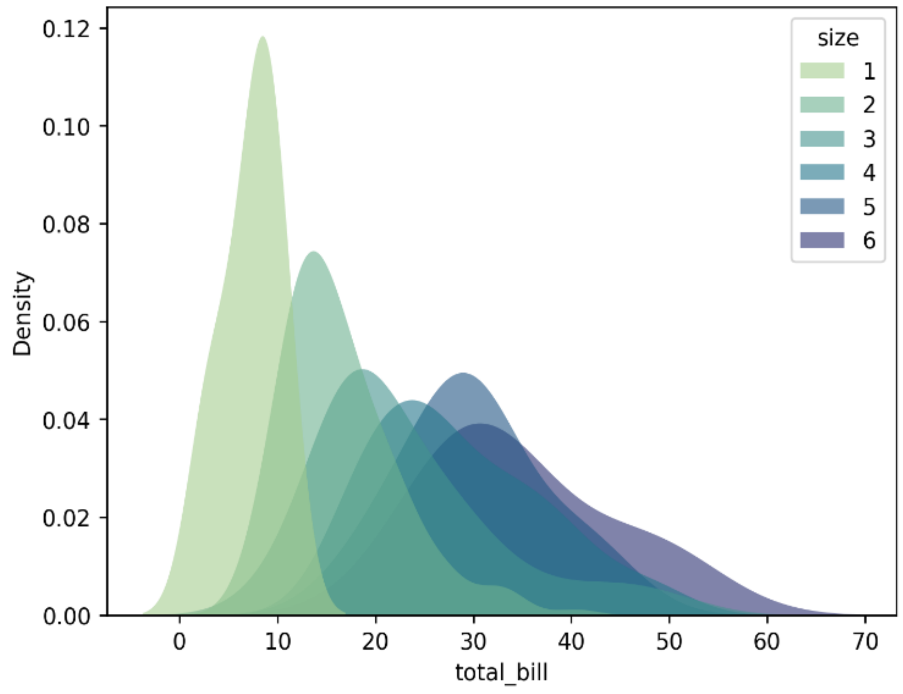
```
import pandas as pd
import seaborn as sns
```

```
dfIn = pd.read_csv("C:\\Program Files\\Row64\\Row64_V1_3\\Data\\Example\\iris.csv")
df=dfIn.copy(deep=True)
sns.pairplot(df)
```

Density Plot

Function: The **DENSITY** sheet formula creates a visualization of the distribution of data over a continuous interval. The **DENSITY** function also allows for multiple groups to be visualized. For example, if a dataframe contains one column of data on the total bill for a dinner and another column on the size of the party, the **DENSITY** function will create a plot that shows the distribution of the total bill for groups of each size.

Example:



Syntax: DENSITY(*numeric_column, group_column*)

- ***value_column*.** Required. **value_column** is the column which contains the values that determine the shape of the distribution.
- ***categorical_column*.** Required. **categorical_column** is the column which contains the categorical data that the distributions are split across.

Video:
TBD

Python Code:

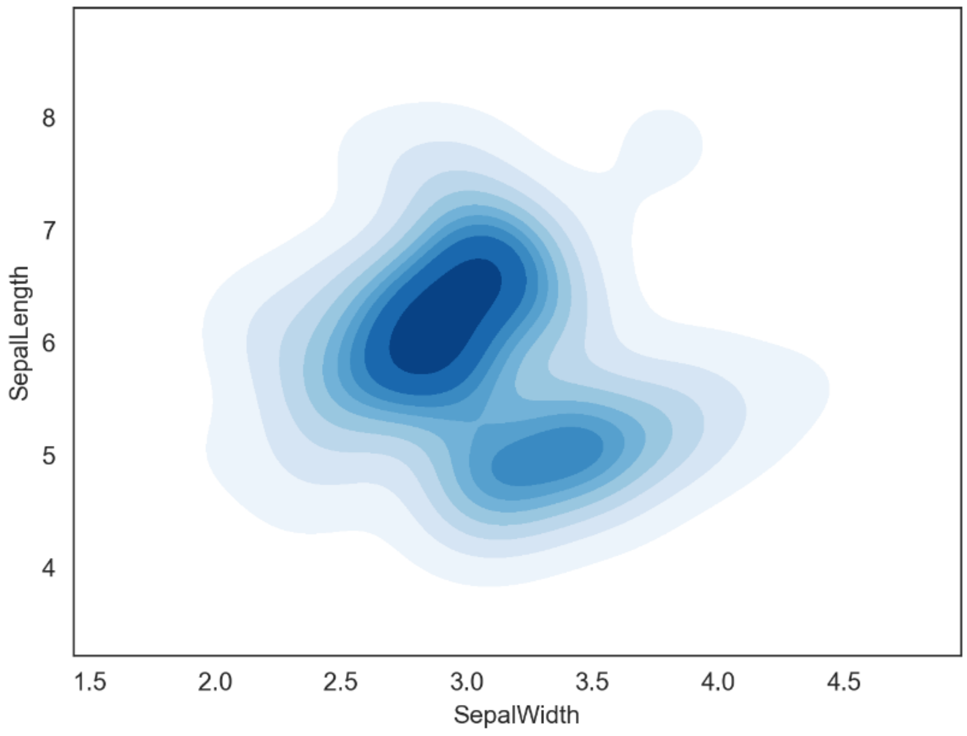
```
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns

def DensityPlot(inDf, inCol1, inCol2):
       cList = ['#A5CD90','#6DB290','#44948F','#24768B','#215584','#2C3172']
       col1 = inDf.columns[inCol1]
       col2 = inDf.columns[inCol2]

sns.kdeplot(data=inDf,x=col1,hue=col2,fill=True,common_norm=False,palette=cList,alp
ha=.6,linewidth=0,)
```

```
dfIn = pd.read_csv("C:\\Program Files\\Row64\\Row64_V1_3\\Data\\Example\\tips.csv")
df=dfIn.copy(deep=True)
DensityPlot(df,0,6)
```

Contour

Function: The **CONTOUR** sheet formula creates a visualization of the distribution of values across two variables. A contour plot is a technique to represent a 3 dimensional surface in 2 dimensions. For example, if a dataframe contains one column of data on the sepal width of a flower and another column on the sepal length of a flower, the **CONTOUR** function will create a plot that shows the distribution of these two variables across all observations in the dataset.

Example:



Syntax: CONTOUR(*x_values_column, y_values_column*)

- ***x_value_column.*** Required. **x_value_column** is the column which contains the x values of each data point.
- ***y_value_column.*** Required. **y_value_column** is the column which contains the y values of each data point.

Video:
TBD

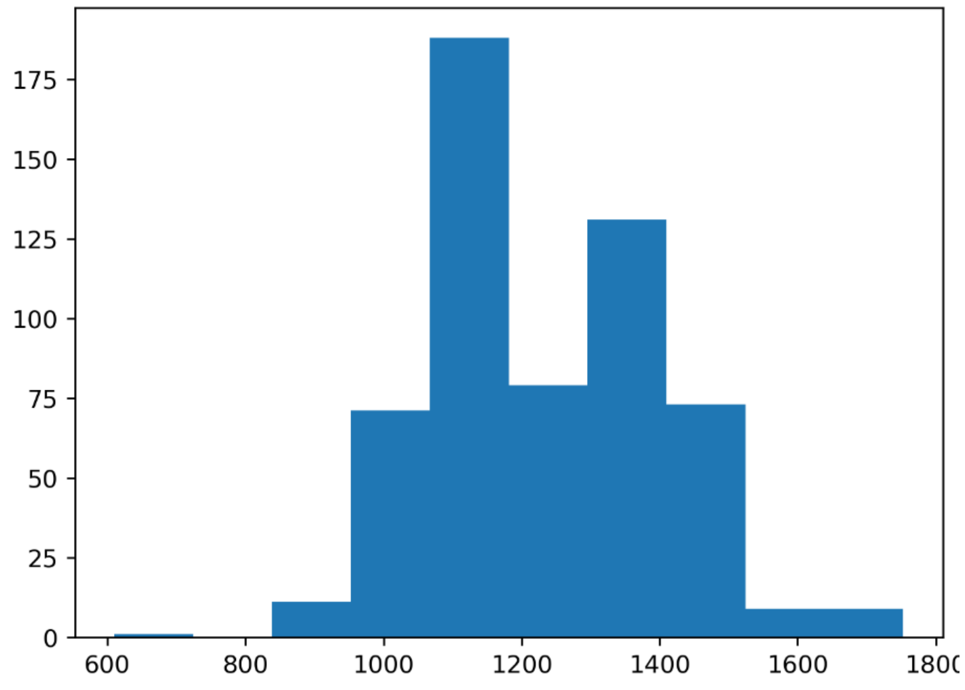
Python Code:

```
import pandas as pd
import seaborn as sns

def ContourPlot(inDf, inCol1, inCol2):
       col1 = inDf.columns[inCol1]
       col2 = inDf.columns[inCol2]
       sns.set_style("white")
sns.kdeplot(data=inDf,x=col1,y=col2,fill=True,common_norm=False,cmap="Blues")

dfIn = pd.read_csv("C:\\Program Files\\Row64\\Row64_V1_3\\Data\\Example\\iris.csv")
df=dfIn.copy(deep=True)
ContourPlot(df,1,0)
```

Histogram

Function: The **HISTOGRAM** sheet formula creates a histogram of a dataframe column. For example, if a dataframe contains a column of the heights of buildings, the **HISTOGRAM** function will create a histogram of the building heights.

Example:



Syntax: HISTOGRAM(*value_column*)

- ***value_column*.** Required. **value_column** is the column which contains the values – the count of which determine the height of the bar in each bin.

Video:
TBD

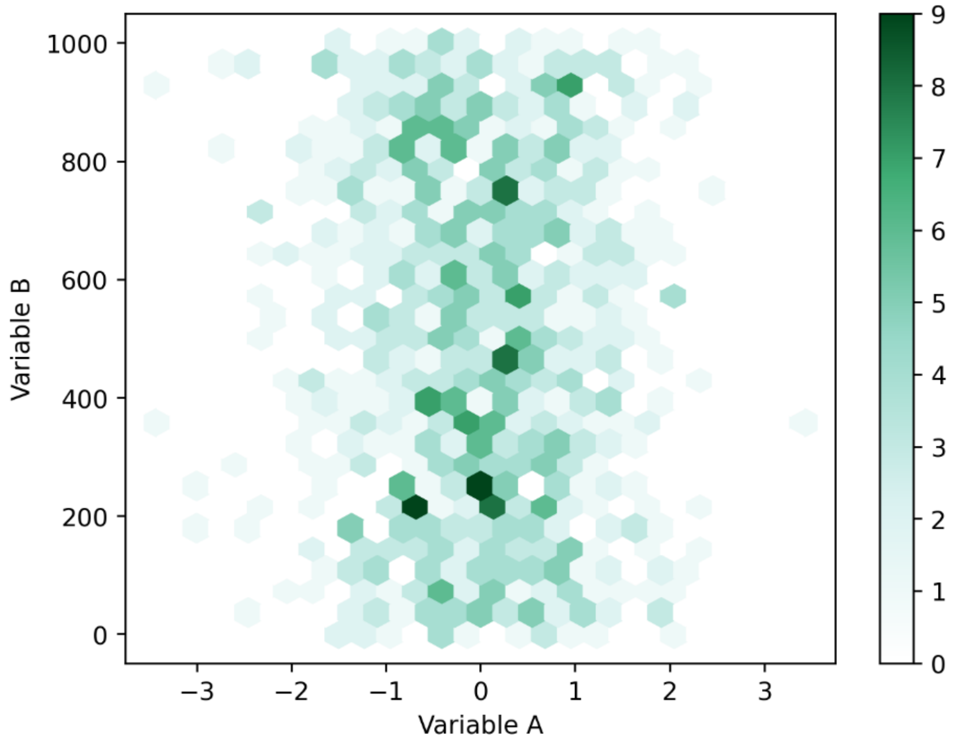
Python Code:

```
import matplotlib.pyplot as plt
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\olive.csv")
df=dfIn.copy(deep=True)
plt.hist( df.iloc[:,3] )
df=df.transpose().reset_index()
```

Hexplot

Function: The **HEXPLOT** sheet formula creates a visualization of the distribution of values across two variables. For example, if a dataframe contains two columns with numeric data, the **HEXPLOT** function will create a plot that shows the distribution of these two variables across all observations in the dataset by creating hexes which differ in shading according to the number of observations that fall in each hex.

Example:



Syntax: HEXPLOT(*x_values_column, y_values_column*)

- ***x_value_column*.** Required. **x_value_column** is the column which contains the x values of each data point.
- ***y_value_column*.** Required. **y_value_column** is the column which contains the y values of each data point.

Video:
TBD

Python Code:

```
import matplotlib.colors as cl
import pandas as pd
import seaborn as sns

def HexPlot(inDf, inI1, inI2):
        cList = ['#FFFFFF','#D9F1F0','#98D8C9','#40AD75','#00441B']
        cRGB = [cl.to_rgb(col) for col in cList]
        clmap = cl.LinearSegmentedColormap.from_list("", cRGB)
        col1 = inDf.columns[inI1]
        col2 = inDf.columns[inI2]
        inDf.plot(kind='hexbin', x=col1, y=col2, gridsize=25, cmap=clmap)

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\sciencedata.csv")
df=dfIn.copy(deep=True)
HexPlot(df,0,1)
```

Word Cloud (White-Look)

Function: The **WORDCLOUD** sheet formula creates a word cloud based on text data. A word cloud is a collection of words in different sizes in which large words are words that appear more often in the text.

Example:

Syntax: CONTOUR(*text_column, TRUE*)

- ***text_column.*** Required. **text_column** is the column which contains the text that the word cloud is to be made on

Video:
TBD

Python Code:

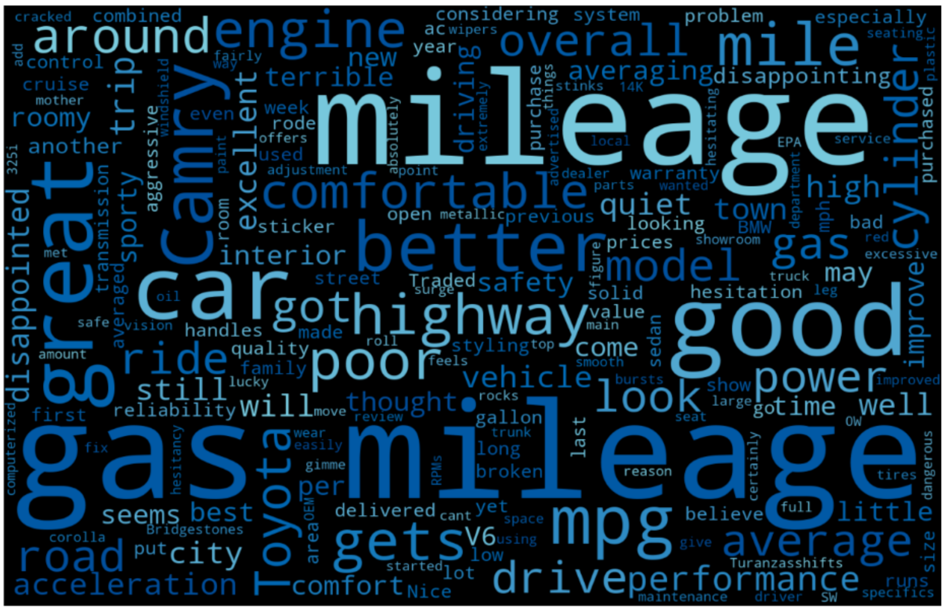
```
import matplotlib.colors as cl
import matplotlib.pyplot as plt
import pandas as pd

from wordcloud import WordCloud

def MakeWordCloud(inDf, inInd, inWhite):
	colors = ['#014D9A','#016BB5','#4AA0CE','#7ACBDD']
	text = ' '.join( inDf[inDf.columns[inInd]].tolist() )
	cRGB = [cl.to_rgb(col) for col in colors]
	cmap = cl.LinearSegmentedColormap.from_list("", cRGB)
	if(inWhite):
		wCloud = 
WordCloud(background_color='white',width=1500,height=1000,colormap=cmap).generate(t
ext)
	else:
		wCloud = 
WordCloud(width=1500,height=1000,colormap=cmap).generate(text)
```

```
        plt.imshow(wCloud, interpolation='bilinear')
        plt.axis("off")

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\review.csv")
df=dfIn.copy(deep=True)
MakeWordCloud(df,0,True)
```

Word Cloud (Dark-Look)

Function: The **WORDCLOUD** sheet formula creates a word cloud based on text data. A word cloud is a collection of words in different sizes in which large words are words that appear more often in the text.

Example:



Syntax: CONTOUR(*text_column, FALSE*)

- **text_column.** Required. **text_column** is the column which contains the text that the word cloud is to be made on

Video:
TBD

Python Code:

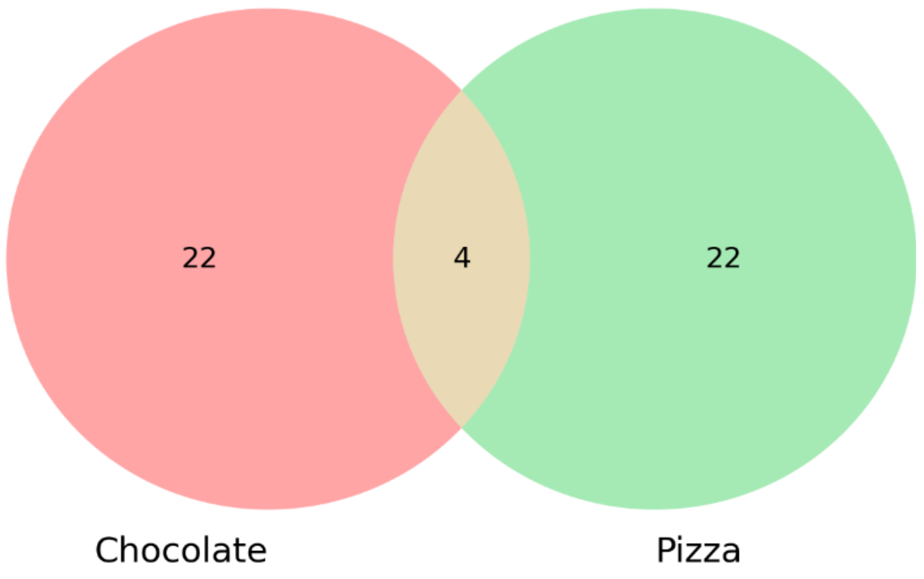
```
import matplotlib.colors as cl
import matplotlib.pyplot as plt
import pandas as pd

from wordcloud import WordCloud

def MakeWordCloud(inDf, inInd, inWhite):
	colors = ['#014D9A','#016BB5','#4AA0CE','#7ACBDD']
	text = ' '.join( inDf[inDf.columns[inInd]].tolist() )
	cRGB = [cl.to_rgb(col) for col in colors]
	cmap = cl.LinearSegmentedColormap.from_list("", cRGB)
	if(inWhite):
		wCloud =
WordCloud(background_color='white',width=1500,height=1000,colormap=cmap).generate(t
ext)
	else:
		wCloud =
WordCloud(width=1500,height=1000,colormap=cmap).generate(text)
	plt.imshow(wCloud, interpolation='bilinear')
```

```
       plt.axis("off")

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\review.csv")
df=dfIn.copy(deep=True)
MakeWordCloud(df,0,False)
```

Venn Diagram (2 Circles)

Function: The **VENN** sheet formula creates a venn diagram of two numeric columns of data.

Example:



Syntax: VENN(*numeric_column1, numeric_column2*)

- ***numeric_column1*.** Required. **numeric_column1** is the column which contains the numeric values to be contained in the circle on the left of the venn diagram.
- ***numeric_column2*.** Required. **numeric_column2** is the column which contains the numeric values to be contained in the circle on the right of the venn diagram.

Video:
TBD

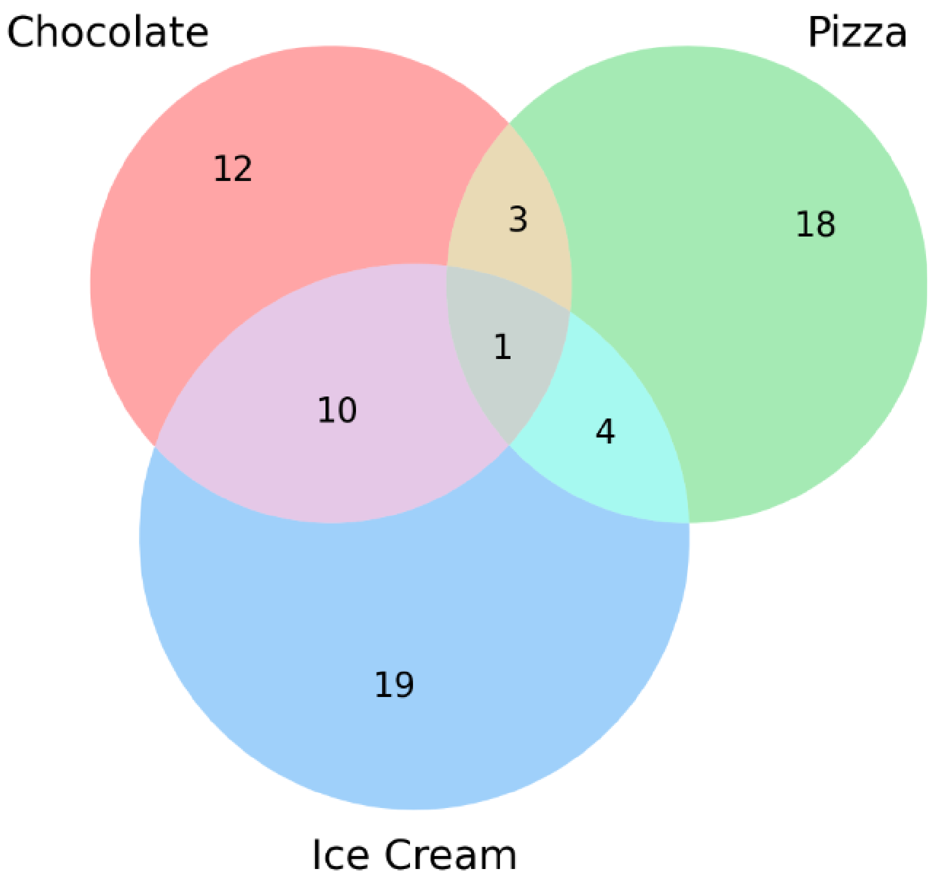
Python Code:

```
import matplotlib.pyplot as plt
import pandas as pd

from matplotlib_venn import venn2

def Venn2Circle(inDf, inI1, inI2):
        col1 = inDf.columns[inI1]
        col2 = inDf.columns[inI2]
        list1 = []
        list2 = []
        for index, row in df.iterrows():
                if ( row[col1] == 1 ) : list1.append(index)
                if ( row[col2] == 1 ) : list2.append(index)
        venn2( [set(list1), set(list2)], (col1,col2), set_colors=['#FF1E22', 
'#1ECC45'])
        plt.show()

dfIn = pd.read_csv("C:\\Program Files\\Row64\\Row64_V1_3\\Data\\Example\\food.csv")
df=dfIn.copy(deep=True)
Venn2Circle(df,4,5)
```

Venn Diagram (3 Circles)

Function: The **VENN** sheet formula creates a venn diagram of three numeric columns of data.

Example:



Syntax: VENN(*numeric_column1, numeric_column2, numeric_column3*)

- ***numeric_column1*****.** Required. **numeric_column1** is the column which contains the numeric values to be contained in the circle on the left of the venn diagram.
- ***numeric_column2*****.** Required. **numeric_column2** is the column which contains the numeric values to be contained in the circle on the right of the venn diagram.
- ***numeric_column3*****.** Required. **numeric_column3** is the column which contains the numeric values to be contained in the circle on the bottom of the venn diagram.

Video:
TBD

Python Code:

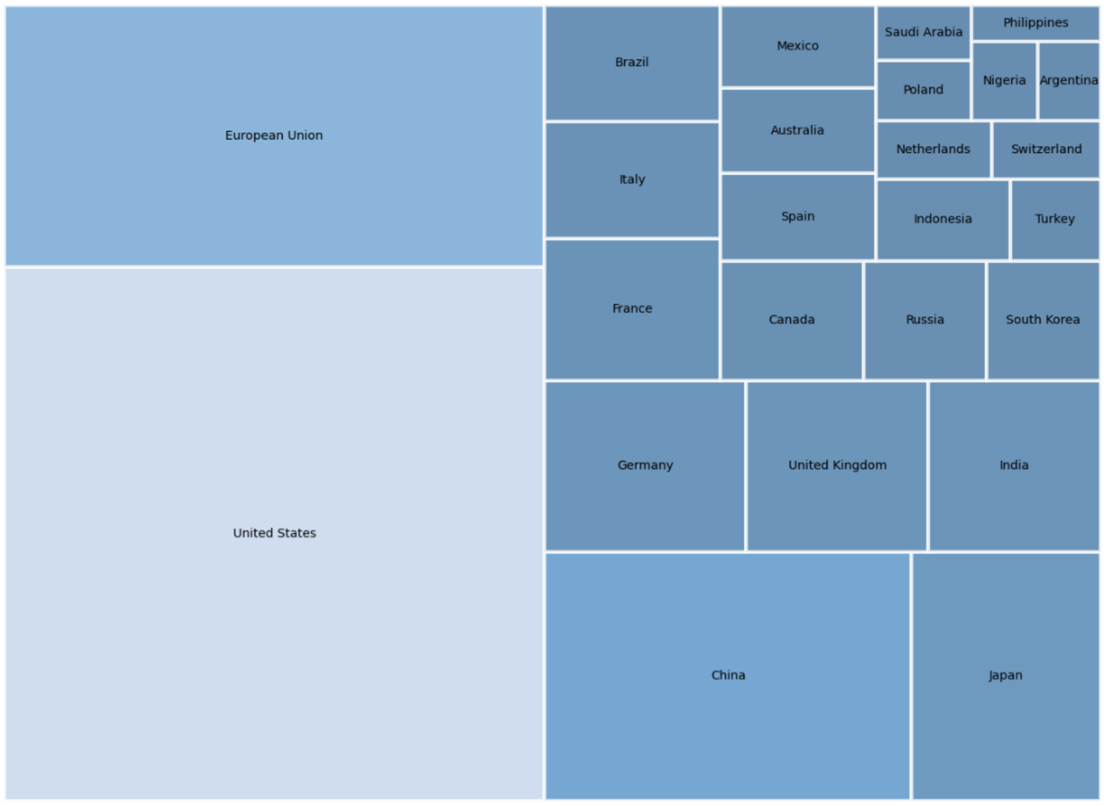
```
import matplotlib.pyplot as plt
import pandas as pd

from matplotlib_venn import venn3
```

```
def Venn3Circle(inDf, inI1, inI2, inI3):
	col1 = inDf.columns[inI1]
	col2 = inDf.columns[inI2]
	col3 = inDf.columns[inI3]
	list1 = []
	list2 = []
	list3 = []
	for index, row in df.iterrows():
		if ( row[col1] == 1 ) : list1.append(index)
		if ( row[col2] == 1 ) : list2.append(index)
		if ( row[col3] == 1 ) : list3.append(index)
	venn3( [set(list1), set(list2), set(list3)], (col1,col2,col3),
set_colors=['#FF1E22', '#1ECC45', '#118BF4'])
	plt.show()

dfIn = pd.read_csv("C:\\Program Files\\Row64\\Row64_V1_3\\Data\\Example\\food.csv")
df=dfIn.copy(deep=True)
Venn3Circle(df,4,5,6)
```

Tree Map

Function: The **TREEMAP** sheet formula creates a visualization depicting the size of different categories. For example, if a dataframe contained a column with countries/unions and another column with household financial consumption expenditure data, the **TREEMAP** function would construct a chart depicting squares of varying sizes for each country/union according to its HFCE value.

Example:



Syntax: TREEMAP(categorical_column, value column)

- ***categorical_column*.** Required. **categorical_column** is the column which contains the categorical data with which the data is split
- ***value_column*.** Required. **value_column** is the column which contains the values that determine the size of each portion in the tree map.

Video:
TBD

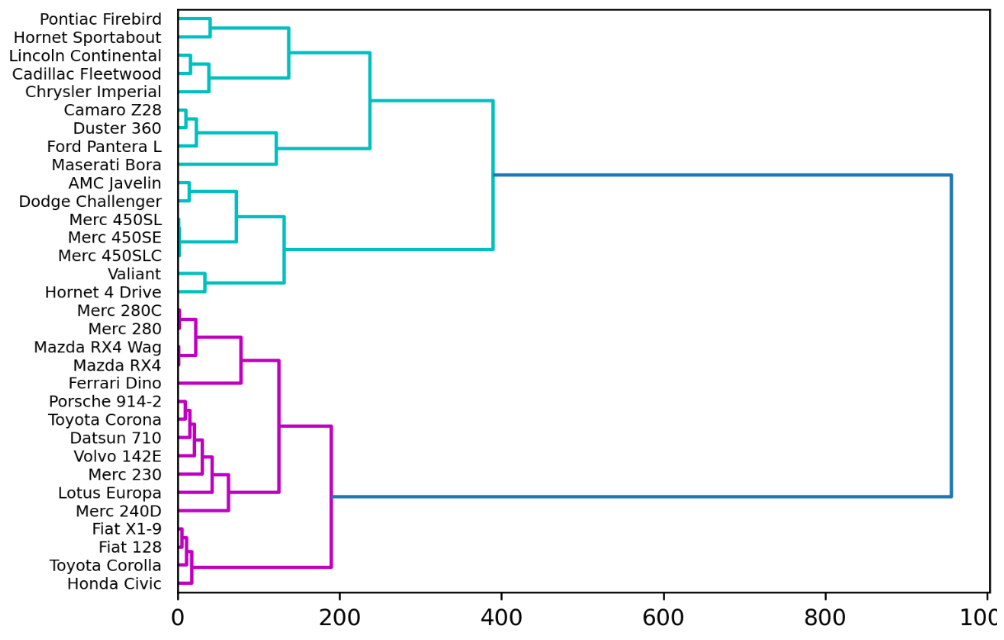
Python Code:

```
import matplotlib.colors as cl
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import squarify

def TreeMap(inDf,inLabelI, inSizeI):
        lables = inDf[inDf.columns[inLabelI]];
        sizeList = inDf[inDf.columns[inSizeI]];
        cList = ["#41719C","#5694CB","#8DB5DB","#C4D5EB"]
        cRGB = [cl.to_rgb(col) for col in cList]
        cmap = cl.LinearSegmentedColormap.from_list("", cRGB)
        mini, maxi = sizeList.min(), sizeList.max()
        norm = cl.Normalize(vmin=mini, vmax=maxi)
        colors = [cmap(norm(value)) for value in sizeList]
        squarify.plot(sizes=sizeList, label=lables, alpha=.8,
        text_kwargs={'fontsize':4}, color=colors, edgecolor="white", linewidth=1.2)
        plt.axis('off')
        plt.show()

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\consumerMarkets.csv")
df=dfIn.copy(deep=True)
TreeMap(df,0,1)
```

Dendrogram

Function: The **DENDROGRAM** sheet formula creates a dendrogram based on a categorical data column. A dendrogram is a type of tree diagram that shows the relationship between similar sets of data.

Example:



Syntax: DENDROGRAM(categorical_column)

- ***categorical_column.*** Required. **categorical_column** is the column which contains the categorical data with which the data is split

Video:
TBD

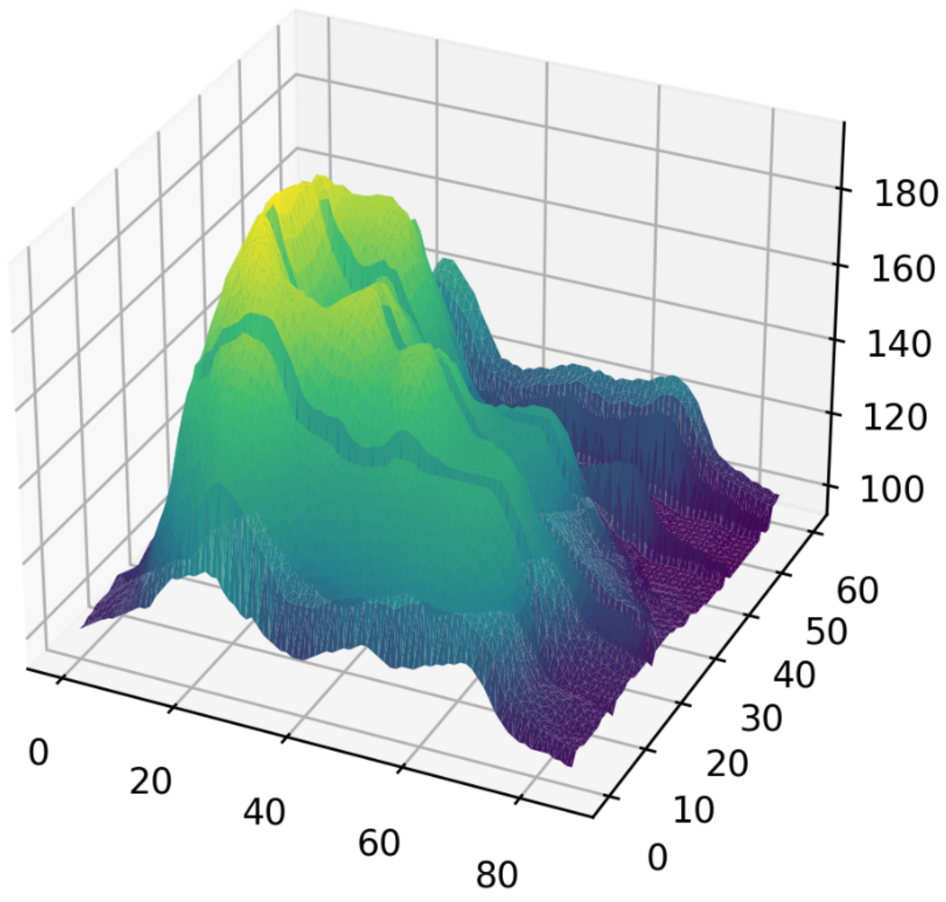
Python Code:

```
import pandas as pd

from scipy.cluster import hierarchy
from scipy.cluster.hierarchy import dendrogram,linkage

def Dendrogram(inDf,inI):
        inDf = inDf.set_index(inDf.columns[inI])
        Z = linkage(inDf, 'ward')
        hierarchy.set_link_color_palette(['m', 'c', 'b', 'k'])
        dendrogram(Z, orientation="right", leaf_font_size=7, labels=inDf.index)

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\mtcars.csv")
df=dfIn.copy(deep=True)
Dendrogram(df,0)
```

Surface Plot

Function: The **SURFACE** sheet formula displays three dimensions of numeric data as a surface in three dimensions. For example, if a dataframe contains three columns of numeric data. The **SURFACE** function will plot the data in the first column on the x axis, the data in the second column on the y axis, the data in the third column on the z axis, and create a surface out of the points plotted.

Example:



Syntax: SURFACE(*x_values_column, y_values_column, z_values_column, angle*)

- ***x_values_column.*** Required. **x_values_column** is the column which contains the x coordinates of each data point.
- ***y_values_column.*** Required. **y_values_column** is the column which contains the y coordinates of each data point.
- ***z_values_column.*** Required. **y_values_column** is the column which contains the y coordinates of each data point.
- ***angle.*** Required. **angle** is the angle at which the plot will be viewed.

Video:
TBD

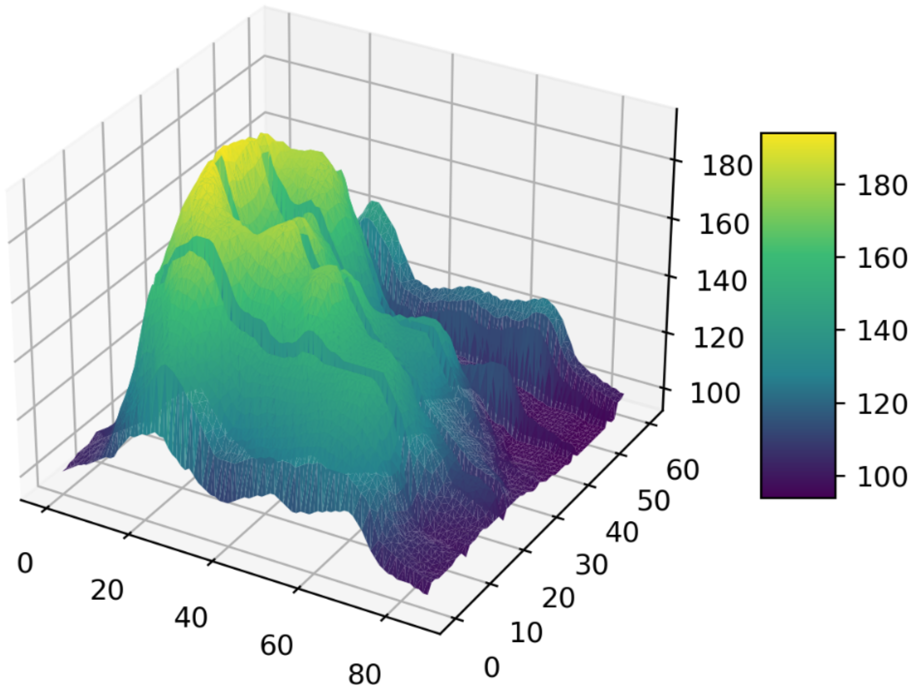
Python Code:

```
import matplotlib.colors as cl
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

def SurfacePlot(inDf, inXI, inYI, inZI, inViewRot):
	cList = ['#440255','#26828E','#3BBB75','#F8E621']
	xv = inDf[inDf.columns[inXI]]
	yv = inDf[inDf.columns[inYI]]
	zv = inDf[inDf.columns[inZI]]
	cRGB = [cl.to_rgb(col) for col in cList]
	cmap = cl.LinearSegmentedColormap.from_list("", cRGB)
	fig = plt.figure()
	ax = fig.add_subplot(projection='3d')
	surf = ax.plot_trisurf(yv, xv, zv, cmap=cmap, linewidth=0.2)
	ax.view_init(30, inViewRot)

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\volcano.csv")
df=dfIn.copy(deep=True)
SurfacePlot(df,0,1,2,-65)
```

Surface Plot (With Legend)

Function: The **SURFACE** sheet formula displays three dimensions of numeric data as a surface in three dimensions with a legend. For example, if a dataframe contains three columns of numeric data. The **SURFACE** function will plot the data in the first column on the x axis, the data in the second column on the y axis, the data in the third column on the z axis, and create a surface out of the points plotted.

Example:



Syntax: SURFACE(*x_values_column, y_values_column, z_values_column, angle*)

- ***x_values_column.*** Required. **x_values_column** is the column which contains the x coordinates of each data point.
- ***y_values_column.*** Required. **y_values_column** is the column which contains the y coordinates of each data point.
- ***z_values_column.*** Required. **y_values_column** is the column which contains the y coordinates of each data point.
- ***angle.*** Required. **angle** is the angle at which the plot will be viewed.

Video:
TBD

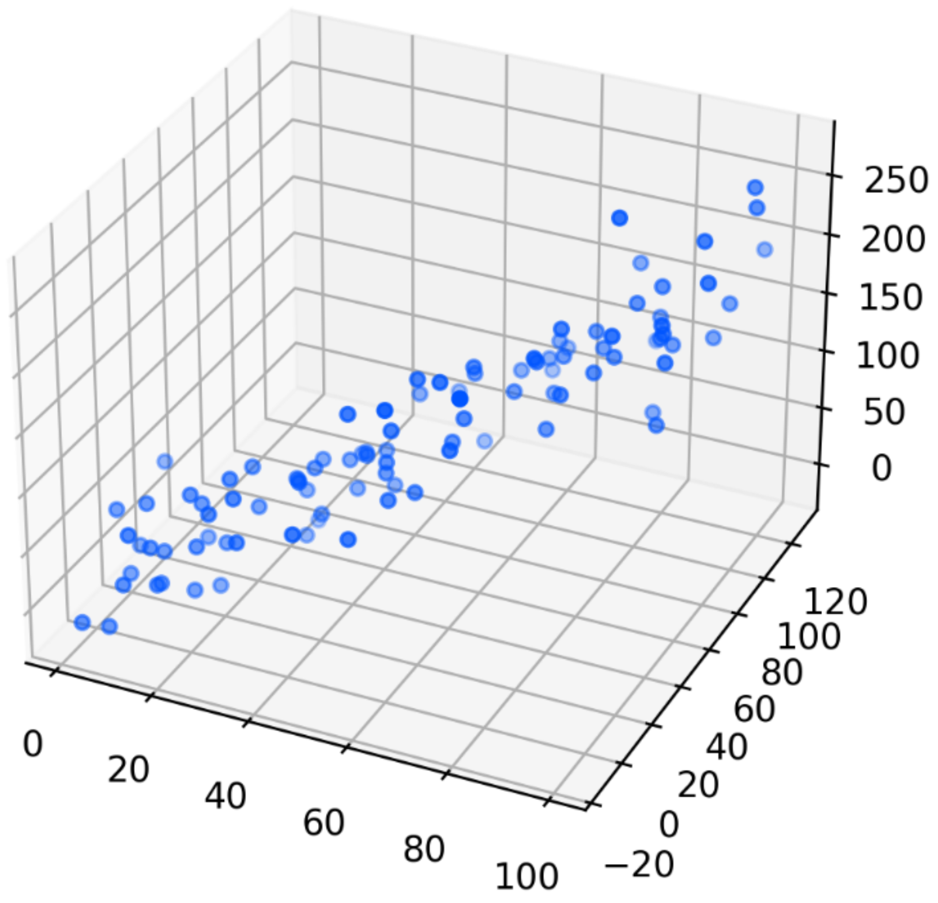
Python Code:

```
import matplotlib.colors as cl
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

def SurfacePlotL(inDf, inXI, inYI, inZI, inViewRot, inLegend):
        cList = ['#440255','#26828E','#3BBB75','#F8E621']
        xv = inDf[inDf.columns[inXI]]
        yv = inDf[inDf.columns[inYI]]
        zv = inDf[inDf.columns[inZI]]
        cRGB = [cl.to_rgb(col) for col in cList]
        cmap = cl.LinearSegmentedColormap.from_list("", cRGB)
        fig = plt.figure()
        ax = fig.add_subplot(projection='3d')
        surf = ax.plot_trisurf(yv, xv, zv, cmap=cmap, linewidth=0.2)
        ax.view_init(30, inViewRot)
        if inLegend:
                fig.colorbar( surf, shrink=0.5, aspect=5)

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\volcano.csv")
df=dfIn.copy(deep=True)
SurfacePlotL(df,0,1,2,-65,True)
```

3D Scatterplot

Function: The **SCATTER3D** sheet formula displays data as a series of points in three dimensions. For example, if a dataframe contains three columns of numeric data. The **SCATTER3D** function will plot the data in the first column on the x axis, the data in the second column on the y axis, and the data in the third column on the z axis.

Example:



Syntax: SCATTERPLOT(*x_values_column, y_values_column, z_values_column*)

- ***x_values_column.*** Required. **x_values_column** is the column which contains the x coordinates of each data point.
- ***y_values_column.*** Required. **y_values_column** is the column which contains the y coordinates of each data point.
- ***z_values_column.*** Required. **y_values_column** is the column which contains the y coordinates of each data point.

Video:
TBD

Python Code:

```
import matplotlib.colors as cl
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

def Scatter3D(inDf, inXI, inYI, inZI, inViewRot):
       xv = inDf[inDf.columns[inXI]]
       yv = inDf[inDf.columns[inYI]]
       zv = inDf[inDf.columns[inZI]]
       fig = plt.figure()
       ax = fig.add_subplot(111, projection='3d')
       ax.scatter(xv,yv,zv, c='#0055FF', s=15)
       ax.view_init(30, inViewRot)

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\scatter.csv")
df=dfIn.copy(deep=True)
Scatter3D(df,0,1,2,-65)
```

Investigation Formulas

View Random Sample

Function: The **RANDOMSAMPLE** sheet formula generates a random sample of the dataframe.

Syntax:RANDOMSAMPLE(percent)

- ***percent***. Required. Percent of the dataframe that is to be returned in the sample

Video:

- TBD

Python Code:

```
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\Vehicles.csv")
df=dfIn.copy(deep=True)
df=df.sample(frac=.2)
```

View Value Range

Function: The **VIEWRANGE** sheet formulareturns all rows of a dataframe for which values in a given column lie in a specified range.

Syntax:VIEWRANGE(columnID, minimum, maximum)

- ***columnID***. Required. The column which contains the data to be compared against the specified range.
- ***minimum***. Required. The minimum value of the desired range.
- ***maximum***. Required. The maximum value of the desired range.

Video:

- TBD

Python Code:

```
import pandas as pd

def ViewRange(inDf, inColI, inMin, inMax):
       cName = inDf.columns[inColI]
       inDf = inDf[(inDf[cName] > inMin) & (inDf[cName] < inMax)]
       return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\Vehicles.csv")
df=dfIn.copy(deep=True)
df=ViewRange(df,6,15,19)
```

View Outside Value Range

Function: The **NOTVIEWRANGE** sheet formulareturns all rows of a dataframe for which values in a given column lie outside a specified range.

Syntax:NOTVIEWRANGE(columnID, minimum, maximum)

- ***columnID***. Required. The column which contains the data to be compared against the specified range.
- ***minimum***. Required. The minimum value of the desired range.
- ***maximum***. Required. The maximum value of the desired range.

Video:

- TBD

Python Code:

```
import pandas as pd

def NotViewRange(inDf, inColI, inMin, inMax):
       cName = inDf.columns[inColI]
       newDf = df[~df[cName].between(inMin, inMax)]
       return newDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\Vehicles.csv")
df=dfIn.copy(deep=True)
df=NotViewRange(df,6,15,19)
```

View Percentage Range

Function: The **VIEWPERC** sheet formulareturns all rows of a dataframe for which values in a given column lie in a specified percentile range. For example, if a column in a dataframe contains MPG data for cars and the user wishes to see cars in 60th to 100th percentile ranges, the VIEWPERC function will return all rows for which MPG values lie in that range.

Syntax:VIEWPERC(columnID, minimumPercentage, maximumPercentage)

- ***columnID***. Required. The column which contains the data to be compared against the specified range.
- ***minimum***. Required. The minimum percentile value of the desired range.
- ***maximum***. Required. The maximum percentile value of the desired range.

Video:

- TBD

Python Code:

```
import pandas as pd

def ViewPerc(inDf, inColI, inMinPerc, inMaxPerc):
       cName = inDf.columns[inColI]
       nbRows = len(df.index)
       midPerc = inMaxPerc - inMinPerc
       nbHead = round((inMinPerc + midPerc)*nbRows)
       nbTail = round(midPerc*nbRows)
       inDf = inDf.sort_values(cName,ascending = True)
       inDf = inDf.sort_values(cName,ascending = True).head(nbHead).tail(nbTail)
       inDf = inDf.iloc[::-1]
       return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\Vehicles.csv")
df=dfIn.copy(deep=True)
df=ViewPerc(df,6,.8,1)
```

Head Rows (First N Rows)

Function: The **VIEWFIRST** sheet formulashows the first n rows of a dataframe.

Syntax:VIEWFIRST(N)

- ***N***. Required. The number of rows at the head of the dataframe to display.

Video:

- TBD

Python Code:

```
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\software.csv")
df=dfIn.copy(deep=True)
df=df.head(10)
```

Tail Rows (Last N Rows)

Function: The **VIEWLAST** function shows the last n rows of a dataframe.

Syntax:VIEWLAST(N)

- ***N***. Required. The number of rows at the tail of the dataframe to display.

Video:

- TBD

Python Code:

```
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\software.csv")
df=dfIn.copy(deep=True)
df=df.tail(8)
```

Largest

Function: The **VIEWLARGE** sheet formula returns the rows of a dataframe that contain the N largest values in a column.

Syntax:VIEWLARGE(columnID, N)

- ***columnID***. Required. The column which contains the data that contains the values of which the N largest determine inclusion in the returned dataframe.
- ***N***. Required. The number of rows to display.

Video:

- TBD

Python Code:

```
import pandas as pd

def LargestRows(inDf, inColI, inNum):
       cName = inDf.columns[inColI]
       newDf = inDf.nlargest(inNum,[cName])
       return newDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\GameData.csv")
df=dfIn.copy(deep=True)
df=LargestRows(df,5,10)
```

Smallest

Function: The **VIEWSMALL** sheet formula returns the rows of a dataframe that contain the N smallest values in a column.

Syntax:VIEWSMALL(columnID, N)

- ***columnID***. Required. The column which contains the data that contains the values of which the N smallest determine inclusion in the returned dataframe.
- ***N***. Required. The number of rows to display.

Video:

- TBD

Python Code:

```
import pandas as pd

def SmallestRows(inDf, inColI, inNum):
       cName = inDf.columns[inColI]
       newDf = inDf.nsmallest(inNum,[cName])
       return newDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\GameData.csv")
df=dfIn.copy(deep=True)
df=SmallestRows(df,5,10)
```

Get Rows By Number

Function: The **VIEWROWS** sheet formula returns any rows specified by the users.

Syntax:VIEWSMALL(rows)

- ***rows***. Required. The indices of the rows that are to be included in the returned dataframe.

Example:
VIEWROWS(1,3,4,7)

Video:

- TBD

Python Code:

```
import pandas as pd

def ViewRows(inDf, indexList):
       newDf =df.loc[df.index[indexList]]
       return newDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\atari.csv")
df=dfIn.copy(deep=True)
df=ViewRows(df,[0,4,5,6])
```

Matches Text List

Function: The **VIEWTEXT** sheet formula returns any rows for which the specified column contains the substrings entered by the user.

Syntax:VIEWTEXT(columnID, textList)

- ***columnID***. Required. The column which contains the strings which are to be parsed for the substrings.
- ***textList***. Required. The list of substrings to be checked against the specified column.

Example:
VIEWTEXT(B:B, “atari”, “Coleco”)

Video:

- TBD

Python Code:

```
import pandas as pd

def ViewText(inDf, inRowI, inList):
       cName = inDf.columns[inRowI]
       newDf = inDf[inDf[cName].str.contains('|'.join(inList), case=False)]
       return newDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\atari.csv")
df=dfIn.copy(deep=True)
df=ViewText(df,1,["atari","Coleco"])
```

Doesn’t Match Text List

Function: The **NOTVIEWTEXT** sheet formula returns any rows for which the specified column does not contain the substrings entered by the user.

Syntax:NOTVIEWTEXT(columnID, textList)

- ***columnID***. Required. The column which contains the strings which are to be parsed for the substrings.
- ***textList***. Required. The list of substrings to be checked against the specified column.

Example:
NOTVIEWTEXT(B:B, “atari”, “Coleco”)

Video:

- TBD

Python Code:

```
import pandas as pd

def NotViewText(inDf, inRowI, inList):
       cName = inDf.columns[inRowI]
       newDf = inDf[~inDf[cName].str.contains('|'.join(inList), case=False)]
       return newDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\atari.csv")
df=dfIn.copy(deep=True)
df=NotViewText(df,1,["atari","Coleco"])
```

Summary Report

Function: The **SUMMARY** sheet formula generates a summary report of the dataframe.

Syntax:SUMMARY(“report”)

Video:

- TBD

Python Code:

```
import pandas as pd
import webbrowser

from pandas_profiling import ProfileReport

def SummaryReport(inDf):
       design_report = ProfileReport(inDf, explorative=True)
       design_report.to_file(output_file='report.html')
       webbrowser.get('windows-default').open('report.html')

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\atari.csv")
df=dfIn.copy(deep=True)
SummaryReport(df)
```

Data Count & Type

Function: The **SUMMARY** sheet formula generates a summary report including information about data type for each column and the number of non-null values.

Syntax:SUMMARY(“type”)

Video:

- TBD

Python Code:

```
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\atari.csv")
df=dfIn.copy(deep=True)
df.info()
```

Column Summary

Function: The **SUMMARY** sheet formula generates a summary report for each column including the following information: Number of values, number of unique values, most frequently occurring value and its frequency, mean, standard deviation, minimum and maximum values, 25th, 50th, and 75th percentiles.

Syntax:SUMMARY(“columns”)

Video:

- TBD

Python Code:

```
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\atari.csv")
df=dfIn.copy(deep=True)
df=df.describe(include='all').reset_index()
```

Correlation Matrix

Function: The **CORR** sheet formula creates a correlation matrix between all of the columns in the dataframe. A correlation matrix includes the correlation coefficient for each column in relation to the others.

Syntax:CORR()

Video:

- TBD

Python Code:

```
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\Vehicles.csv")
df=dfIn.copy(deep=True)
df = df.corr().reset_index()
```

Frequency - Column Values

Function: The **FREQ** sheet formula returns a dataframe including the frequency of each value in a certain column.

Syntax:FREQ(columnID)

- ***columnID***. Required. The column which contains the strings which are to be parsed for the substrings.

Video:

- TBD

Python Code:

```
import pandas as pd
import sidetable

def FreqReport(inDf,inColI):
       cName = inDf.columns[inColI]
       inDf = inDf.stb.freq([cName])
       return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\nameparts.csv")
df=dfIn.copy(deep=True)
df=FreqReport(df,1)
```

Frequency - Words

Function: The **FREQ** sheet formula returns a dataframe including following information for a column of text data: the number of occurrences of each word, the percent of occurrences each word is accountable for, the cumulative occurrences, and the cumulative percentage.

Syntax:FREQ(columnID)

- ***columnID***. Required. The column which contains the text data.

Video:

- TBD

Python Code:

```
import pandas as pd
import sidetable

def FreqWords(inDf,inColI):
        cName = inDf.columns[inColI]
        wList = inDf[cName].str.split(expand=True).stack()
        dfGen = pd.DataFrame({'Words':wList})
        inDf = dfGen.stb.freq(['Words'])
        return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\review.csv")
df=dfIn.copy(deep=True)
df=FreqWords(df,0)
```

Search and Replace

Function: The **FINDREPLACE** sheet formula replaces every instance of a substring with a new replacement substring for every row in a dataframe column.

Syntax:FINDREPLACE(columnID, “originalString”, “replacementString”)

- ***columnID***. Required. The column which contains the text data.
- ***originalString***. Required. The string that is to be replaced.
- ***replacementString***. Required. The string that will replace the original substring.

Video:

- TBD

Python Code:

```
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\Vehicles.csv")
df=dfIn.copy(deep=True)
df=df.replace({df.columns[3]:{"Compact Cars":"====REPLACEMENT====","Midsize
Cars":"====REPLACEMENT===="}})
```

Search and Replace - Multi

Function: The **FINDREPLACE** sheet formula replaces every instance of multiple specified substrings with a new replacement substring for every row in a dataframe column.

Syntax:FINDREPLACE(columnID, originalStringList, replacementString)

- ***columnID***. Required. The column which contains the text data.
- **originalStringList**. Required. The list of strings that are to be replaced
- **replacementString**. Required. The string that will replace the original substring.

Video:

- TBD

Python Code:

```
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\Vehicles.csv")
df=dfIn.copy(deep=True)
df=df.replace({df.columns[3]:{"Compact Cars":"====REPLACEMENT====","Midsize
Cars":"====REPLACEMENT===="}})
```

Search and Replace - With Wildcards

Function: The **FINDREPLACE** sheet formula replaces every instance of a substring (with wildcard characters) with a new replacement substring for every row in a dataframe column.

Syntax:FINDREPLACE(columnID, originalString, replacementString)

- ***columnID***. Required. The column which contains the text data.
- **originalString**. Required. The string (with wildcards) that are to be replaced.
- **replacementString**. Required. The string that will replace the original substring.

Video:

- TBD

Python Code:

```
import pandas as pd
import re

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\Vehicles.csv")
df=dfIn.copy(deep=True)
df.iloc[:,3].replace(to_replace=r'(?i)^.*.Cars$',value='====REPLACEMENT====',regex=
True,inplace=True)
```

Transformation Formulas

Delete Rows

Function: The **DELETEROWS** sheet formula removes selected rows from the dataframe. This formula can be used on multiple rows. Each row is separated by a comma.

Syntax: DELETEROWS(*row_num1, row_num2*, *row_num3*…)

- Each row_num# must be separated by a comma.

Video:

- TBD

Python Code: (DELETEROWS(1,2,3))

```
import pandas as pd

def DeleteRows(inDf, inIList):
       newDf = inDf.drop(inIList)
       return newDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\atari.csv")
df=dfIn.copy(deep=True)
df=DeleteRows(df,[0,1,2])
```

Delete Columns

Function: The **DELETECOL** sheet formula removes selected columns from the dataframe. This formula can be used on multiple rows. Each row is separated by a comma.

Syntax: DELETECOL(*ColumnID1, ColumnID2*, *ColumnID3…*)

- Each **ColumnID** must be separated by a comma.

Video:

- TBD

Python Code: (DELETECOL(C:C,D:D))

```
import pandas as pd

def DeleteColumns(inDf, inIList):
       cNames = []
       for ind in inIList:
              cNames.append(inDf.columns[ind])
       newDf = inDf.drop(columns=cNames)
       return newDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\atari.csv")
df=dfIn.copy(deep=True)
df=DeleteColumns(df,[2,3])
```

Change Specific Cell of Dataframe

Function: The **SETCELL** sheet formula changes the value of a specific cell in a dataframe to a specified string or numeric value. By default the SETCELL formula will set the desired cell with the specified string. In order to insert a numeric value, the quotations surrounding the desired string in the Python IDE must be removed.

Syntax: SETCELL(*cell_reference, input_string*)

- cell_reference: the cell upon which you wish to change
- input_string: user inputted string to change cell to. For numeric values, remove quotations from input_string in the Python IDE.

Video:

- TBD

Python Code: SETCELL(A2,"*specified string or numeric values*")

```
import pandas as pd
import sidetable

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\atari.csv")
df=dfIn.copy(deep=True)
df.iat[1,0]="specified string or numeric values"
```

Transpose DataFrame

Function: The **TRANSPOSE** sheet formula changes orientation of the current dataframe and transposes the dataframe from one orientation to another orientation. The original column headers will be changed into the first column whereas the original first column of the dataframe will become the first row.

Syntax: TRANSPOSE()

- cell_reference: the cell upon which you wish to change

Video:

- TBD

Python Code:
```
df=df.transpose().reset_index()
```

Rename Column

Function: The **RENAMECOL** sheet formula changes the name of a column of the Dataframe.

Syntax: RENAMECOL(*ColumnID*, “*newColumnName”*)

- cell_reference: the cell upon which you wish to change

Video:

- TBD

Python Code:

```
df=RenameColumn(df,0,"newColumnName")
```

Concatenate Columns

Function: The **CONCATCOL** sheet formula combines the specified columns into one unique column.

Syntax: CONCATCOL(*"string between each concatentation"*, "*newColumnName",* *ColumnID1, ColumnID2, …*)

- String between each concatenation: String that is to be placed between each column.
- ColumnID(s): The columns to be combined.

Video:

- TBD

Python Code:
```
import numpy as np
import pandas as pd

def ConcatColumns(inDf, inDivisor, inCName, inCList):
	cNames = []
	merged = []
	for ci in inCList: cNames.append(inDf.columns[ci])
	for index, row in inDf.iterrows():
		nameP = [];
		for cn in cNames:
			if pd.notnull(row[cn]): nameP.append(row[cn])
		mergeN = inDivisor.join(nameP)
		merged.append(mergeN)
	inDf[inCName] = merged
	return inDf

df=dfIn.copy(deep=True)
df=ConcatColumns(df," ","Col Name",[0,1,2,3])
```

Split Column

Function: The **SPLITCOL** sheet formula splits the specified columns into multiple columns based upon a specified delimiter. The number of new columns created must be uniform and equal.

Syntax: SPLITCOL(*ColumnID, "delimiter*", "*newColumnName1", "newColumnName2", …*)

- **ColumnID**. Required. The specified column to be split.
- **delimiter**. Required. The delimiter that signals where each split is to occur.
- **newColumnName**(s). Required. The name of the columns that are being added to the dataframe..

Video:

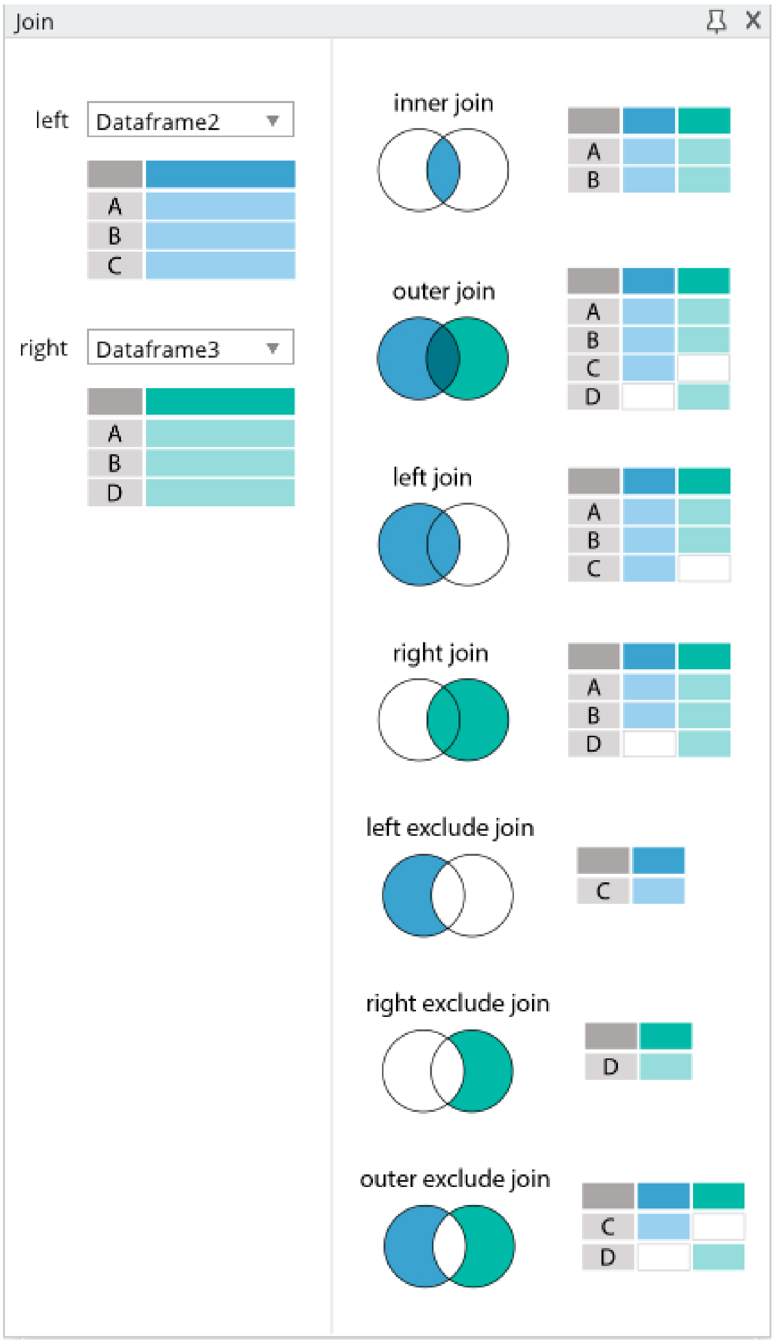
- TBD

Python Code:

```
import numpy as np
import pandas as pd

def SplitColumns(inDf,inColI,inDelimiter, inNames):
       cName = inDf.columns[inColI]
       inDf[inNames] = inDf[cName].str.split(',', expand=True)
       inDf[inNames] = inDf[inNames].apply(lambda x: x.str.strip())
       return inDf

df=dfIn.copy(deep=True)
df=SplitColumns(df,0,",",["newColumnName1","newColumnName2",...)
```

Joining Dataframes

Function: The **JOIN** sheet formula joins two dataframes together off a shared key among both dataframes. Users have the option of performing inner, outer, left, right, left exclude, right exclude, and outer exclude joins. A visual aid is provided to signify what type of join the user is looking to employ.



left: Dataframe2

A	🟦
B	🟦
C	🟦

right: Dataframe3

A	🟩
B	🟩
D	🟩

inner join

A	🟦	🟩
B	🟦	🟩

outer join

A	🟦	🟩
B	🟦	🟩
C	🟦	
D		🟩

left join

A	🟦	🟩
B	🟦	🟩
C	🟦	

right join

A	🟦	🟩
B	🟦	🟩
D		🟩

left exclude join

C	🟦

right exclude join

D	🟩

outer exclude join

C	🟦	
D		🟩

Syntax: JOIN(*Dataframe1KeyColID, Dataframe2KeyColID,* "*JoinType*")

- *Dataframe1KeyColID.* Required. References the first dataframe's column key that is shared among dataframes that is used in order to unify dataframes.
- *Dataframe2KeyColID.* Required. References the second dataframe's column key that is shared among dataframes that is used in order to unify dataframes

- *JoinType*. Required. User must specify the type of join they are seeking to perform. Options include: inner, outer, left, right, left exclude, right exclude, outer exclude.

Python:
```
Varies depending upon the type of join that the user has selected.
```

Fuzzy Merging

Function: The **FUZZYMERGE** sheet formula is a smart data preparation feature used to apply fuzzy matching algorithms when comparing columns. This is done in order to attempt to identify matches across the tables which are being merged.

Syntax: FUZZYMERGE(*DataframeKeyColumnID1, DataframeKeyColumnID2*)

- **Other.** Required. Prior to using this function, two dataframes must be loaded into the IDE. (i.e. GET or LOADCSV must be used twice in order for Function to operate correctly.).
- ***DataframeKeyColumnID1***. Required. Primary dataframe.
- ***DataframeKeyColumnID2***. Required. Secondary dataframe that is added/merged to the right. Similar to what a vlookup / left join would perform.

Video:

- TBD

Python Code: (Note: 2 LOADCSV()’s have been performed here)

```
import pandas as pd

from rapidfuzz import process

def FuzzyMerge(baseFrame, compareFrame, inCol1, inCol2):
        baseCol = baseFrame.columns[inCol1]
        compareCol = compareFrame.columns[inCol2]
        mergeData = []
        for name_left in baseFrame[baseCol]:
                choice, score, index = process.extractOne(name_left,
compareFrame[compareCol], processor=None)
                mRow = [round(score,2)]
                for val in compareFrame.iloc[index]:mRow.append(val)
                mergeData.append(mRow)
        colNames = compareFrame.columns.tolist();
        colNames.insert(0,"Fuzzy Match Score")
        mergeFrame = pd.DataFrame(mergeData, columns=colNames)
        return pd.concat([baseFrame, mergeFrame], axis=1);

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\FuzzyMerge\\Misspelling.csv")
dfIn2 = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\FuzzyMerge\\Spelling.csv")
df=FuzzyMerge(dfIn,dfIn2,0,0)
```

Differences between Two Dataframes

Function: The **DIF** sheet formula is a smart data preparation feature used to identify differences between two similar Dataframes. This is done in order to attempt to identify uniqueness between both DFs.

Syntax: DIF(*Dataframe1, Dataframe2*)

- **Other.** Required. Prior to using this function, two dataframes must be loaded into the IDE.
- ***Dataframe1***. Required. Primary dataframe that is being compared to secondary dataframe..
- ***Dataframe2***. Required. Secondary dataframe that is being compared to primary dataframe.

Video:

- TBD

Python Code:

```
import pandas as pd
import row64

def DifferenceDF(inDName1,inDName2):
       inDf1 = row64.get_dataframe(inDName1)
       inDf2 = row64.get_dataframe(inDName2)
       dfr = pd.concat([inDf1, inDf2])
       dfr = dfr.reset_index(drop=True)
       df_gpby = dfr.groupby(list(dfr.columns))
       idx = [x[0] for x in df_gpby.groups.values() if len(x) == 1]
       dfr = dfr.reindex(idx)
       return dfr

df=DifferenceDF("Dataframe3","Dataframe8")
```

DeDuplicate

Function: The **DEDUP** sheet formula is a technique for eliminating duplicate copies of repeating data. The DEDUP function keeps the first identified row of repeated data and then eliminates all subsequent rows of data keeping only one unique row for the deduplicated column for the new dataframe.

Syntax: DEDUP(*ColumnID*)

- ***ColumnID***. Required. Primary dataframe column to look for duplication.

Video:

- TBD

Python Code:

```
import pandas as pd

df=dfIn.copy(deep=True)
df=df.drop_duplicates(subset=[df.columns[2]])
```

Deduplicate Last

Function: The **DEDUPLAST** sheet formula is a technique for eliminating duplicate copies of repeating data. The DEDUPLAST function keeps the last identified row of repeated data and then eliminates all rows of data keeping only one unique row for the deduplicated column for the new dataframe.

Syntax: DEDUPLAST(*ColumnID*)

- ***ColumnID***. Required. Primary dataframe column to look for duplication.

Video:

- TBD

Python Code:

```
import pandas as pd

df=dfIn.copy(deep=True)
df=df.drop_duplicates(subset=[df.columns[2]],keep='last')
```

Fuzzy Deduplication

Function: The **FUZZYDEDUP** sheet formula is a technique for eliminating near duplicate copies of repeating data. The FUZZYDEDUP function keeps the first identified row of repeated data and then eliminates all rows of data keeping only one unique row for the deduplicated column for the new dataframe. In addition, a new column is added with a list of deduplicated terms. This new column can serve as a quality control check as each fuzzy deduplication is based upon

Syntax: FUZZYDEDUP(*ColumnID, FuzzyDeDupStrength*)

- ***ColumnID***. Required. Primary dataframe column to look for duplication.
- ***FuzzyDeDupStrength****.* Required. This is the score of how strong the relationship must be in order to perform a deduplication.

Video:

- TBD

Python Code:

```
import pandas as pd

from rapidfuzz import process

def FuzzyDedup(df,cIndex,threshold):
    values = df[df.columns[cIndex]].str.lower().tolist()
    ucValues = df[df.columns[cIndex]].tolist()
    colNames = df.columns.tolist()
    colNames.append("Fuzzy Dedup")
    indList = [*range(df.shape[0])]
    results = []
    while values:
        val = values.pop(0)
        ind = indList.pop(0)
        ucValues.pop(0)
        dat =  process.extract(val, values, processor=None, score_cutoff=threshold,
limit=None)
        merged = []
        for dl in dat:
            matchInd = dl[2]
            merged.append(ucValues[matchInd])
            del values[matchInd]
            del indList[matchInd]
            del ucValues[matchInd]
        newRow=[];
        for dfVal in df.iloc[ind]:
            newRow.append(dfVal)
        newRow.append("|".join(merged))
        results.append(newRow)
    return pd.DataFrame(results, columns=colNames)

df=dfIn.copy(deep=True)
df=FuzzyDedup(df,0,60)
```

Group Concatenation

Function: The **GROUPCONCAT** sheet formula is a technique for grouping and then concatenating a specified column into one column. All other data is then deduplicated based upon the first row of data. The GROUPCONCAT function keeps the first identified row of repeated data and then eliminates all rows of data keeping only one unique row for the deduplicated column for the new dataframe with a column containing all values of a specified row that is delimited based upon an user identified delimiter.

Syntax: GROUPCONCAT(*ColumnID1*, *"delimiter"*, *ColumnID2*)

- ***ColumnID1***. Required. Dataframe column to the containing group.
- ***delimiter****.* Required. Value to delimit each value.
- ***ColumnID1***. Required. Dataframe column containing delimited values..

Video:

- TBD

Python Code:

```
import pandas as pd

def GroupAndConcat(idf, mergeColI, delim, joinColumns):
	mName =df.columns[mergeColI]
	for ind in joinColumns:
		cName =df.columns[ind]
		df[cName] = df[cName].astype(str)
		df[cName] = df[[mName,cName]].groupby([mName])[cName].transform(lambda
x:delim.join(x))
	return idf.drop_duplicates(subset=[df.columns[mergeColI]])

df=dfIn.copy(deep=True)
df=GroupAndConcat(df,0,"|",[1])
```

Group Concatenation with Unique Values

Function: The **GROUPCONCAT** sheet formula is a technique for grouping and then concatenating a specified column into one column. All other data is then deduplicated based upon the first row of data. The GROUPCONCAT function keeps the first identified row of repeated data and then eliminates all rows of data keeping only one unique row for the deduplicated column for the new dataframe with a column containing ONLY UNIQUE values of a specified row that is delimited based upon an user identified delimiter.

Syntax: GROUPCONCAT(*ColumnID1*, *"delimiter"*, *ColumnID2,* TRUE)

- ***ColumnID1***. Required. Dataframe column to the containing group.
- ***delimiter**.* Required. Value to delimit each value.
- ***ColumnID1***. Required. Dataframe column containing delimited values..

Video:

- TBD

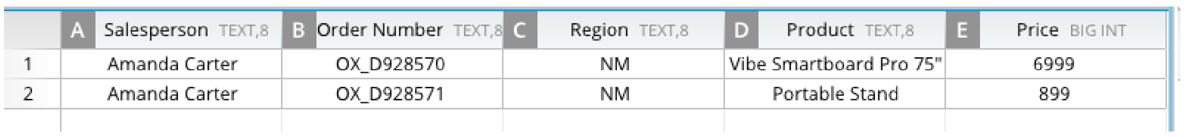
Python Code:

```
import numpy as np
import pandas as pd

def GroupAndConcatUnique(idf, mergeColI, delim, joinColumns):
       mName =df.columns[mergeColI]
       for ind in joinColumns:
              cName =df.columns[ind]
              df[cName] = df[cName].astype(str)
              df[cName] = df[[mName,cName]].groupby([mName])[cName].transform(lambda
x:delim.join(np.unique(x)))
       return idf.drop_duplicates(subset=[df.columns[mergeColI]])

df=dfIn.copy(deep=True)
df=GroupAndConcatUnique(df,0,"|",[1])
```

Reshape Long to Wide

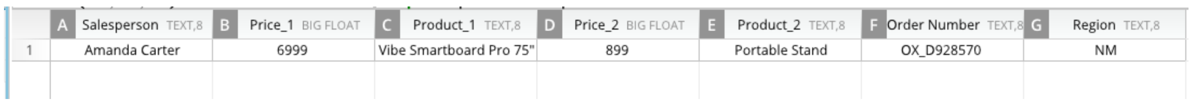
Function: The **WIDEN** sheet formula is a technique for expanding a dataframe by adding additional columns to the dataframe based upon grouping unique cells from one column. For example a dataframe containing two rows (previously long) will be widened to contain additional data. See below for an example.

This data will transform into one row:



	A Salesperson TEXT,8	B Order Number TEXT,8	C Region TEXT,8	D Product TEXT,8	E Price BIG INT
1	Amanda Carter	OX_D928570	NM	Vibe Smartboard Pro 75"	6999
2	Amanda Carter	OX_D928571	NM	Portable Stand	899

As seen here:



	A Salesperson TEXT,8	B Price_1 BIG FLOAT	C Product_1 TEXT,8	D Price_2 BIG FLOAT	E Product_2 TEXT,8	F Order Number TEXT,8	G Region TEXT,8
1	Amanda Carter	6999	Vibe Smartboard Pro 75"	899	Portable Stand	OX_D928570	NM

Syntax: WIDEN(*ColumnID1, ColumnID2, ColumnID3*)

- ***ColumnID1***. Required. Column where values are to be merged into one unique value which will produce one row.
- ***ColumnID2***. Required. 1st Value Col to be expanded on.
- ***ColumnID3***. Required. 2nd Value Col to be expanded on.

Video:

- TBD

Python Code:

```
import numpy as np
import pandas as pd

def WidenShape(idf, mergeColI, widenColums):
       mName=idf.columns[mergeColI]
       wide = [idf.columns[ind] for ind in widenColums]
       allCol = idf.columns.values.tolist()
       notWide = np.setdiff1d(allCol, wide).tolist()
       idf['idx'] = idf.groupby(mName).cumcount()+1
       mergeDF = idf.loc[ idf['idx'] == 1, notWide+['idx'] ].drop(columns='idx')
       idf = idf.pivot_table(index=mName, columns='idx', values=wide,
aggfunc='first')
       idf = idf.sort_index(axis=1, level=1)
       idf.columns = [f'{x}_{y}' for x,y in idf.columns]
       idf = idf.reset_index()
       idf = idf.merge( mergeDF, how='left', on=mName)
       return idf

df=dfIn.copy(deep=True)
df=WidenShape(df,0,[3,4])
```

Lengthen Dataframe based on a Delimiter

Function: The **SPLITLONG** sheet formula is a technique for lengthening a dataframe by making a duplicate row in the dataframe by splitting a cell into another row based upon a user identified delimiter.

Syntax: SPLITLONG(*ColumnID, “delimiter”*)

- ***ColumnID***. Required. Dataframe column that contains items to be split into multiple rows.
- ***delimiter****.* Required. delimiter.

Video:

- TBD

Python Code:

```
import pandas as pd

def SplitLong(idf, textI, delim):
       cName = idf.columns[textI]
       idf[cName]= idf[cName].str.split(delim)
       idf = idf.explode(cName)
       return idf

df=dfIn.copy(deep=True)
df=SplitLong(df,2,"|")
```

Pivot

Function: The **PIVOT** sheet formula takes in two categorical columns and calculates the average of every other numeric column for each unique pairing of levels in the two categorical columns.

Syntax: PIVOT(column1, column2)

- ***column1***. Required. The first dataframe column that is to be pivoted on.
- ***column2***. Required. The second dataframe column that is to be pivoted on.

Video:

Python Code:

```
import pandas as pd

def BasicPivot(inDf,inInd1,inInd2):
       cName1 = inDf.columns[inInd1]
       cName2 = inDf.columns[inInd2]
       inDf = pd.pivot_table(df,index=[cName1,cName2])
       inDf=inDf.reset_index()
       return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\GameData.csv")
df=dfIn.copy(deep=True)
df=BasicPivot(df,4,6)
```

Stack

Function: The **STACK** sheet formula takes in a dataframe and splits each row into multiple by having each column value assigned to a new row with a unique ID.

Syntax: STACK()

Video:

Python Code:

```
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\mtcars.csv")
df=dfIn.copy(deep=True)
df=df.stack().reset_index()
```

Melt

Function: The **MELT** sheet formula takes one or more columns and uses them as identifier variables to unpivot a dataframe from wide to long format.

Syntax: MELT(columnList)

- ***columnList***. Required. The list of columns to be used as identifier variables.

Video:

Python Code:

```
import pandas as pd

def MeltDf(inDf,inIList):
       cNames = [inDf.columns[ind] for ind in inIList]
       inDf =inDf.melt(id_vars=cNames)
       inDf=inDf.reset_index()
       return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\software.csv")
df=dfIn.copy(deep=True)
df=MeltDf(df,[1,3])
```

Cross Tab

Function: The **CROSSTAB** sheet formula takes two categorical columns and creates a cross tabulation with the frequency of each pairing.

Syntax: CROSSTAB(column1, column2)

- ***column1***. Required. The column that will be on the vertical axis of the cross tabulation.
- ***column2***. Required. The column that will be on the horizontal axis of the cross tabulation.

Video:

Python Code:

```
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns

def CrossTab(inDf,inCol1,inCol2):
       sns.set(font_scale=.8)
       cName1 = inDf.columns[inCol1]
       cName2 = inDf.columns[inCol2]
       inDf = pd.crosstab(inDf[cName1], inDf[cName2])
       fig, ax = plt.subplots(figsize=(6,10))
       sns.heatmap(inDf,cmap="PuRd", annot=True, cbar=False, ax=ax)
       inDf=inDf.reset_index()
       return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\Vehicles.csv")
df=dfIn.copy(deep=True)
df=CrossTab(df,0,3)
```

Cleaning Formulas

Bad Data Summary

Function: The **NULLSUMMARY** sheet formula prints out the number of null values in each column for a dataframe.

Syntax: NULLSUMMARY()

Video:

Python Code:

```
import pandas as pd

def NullSummary(inDf):
       print("------------ NULL Value Summary ------------")
```

```
        print(inDf.isnull().sum())

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\atari.csv")
df=dfIn.copy(deep=True)
NullSummary(df)
```

Trim Whitespace

Function: The **COLTRIM** sheet formulastrips the leading and trailing whitespace on all text in a column.

Syntax:COLTRIM(ColumnID)

- ***ColumnID***. Required. Primary dataframe column to look for trimming.

Video:

Python Code:

```
import pandas as pd

def ColTrim(inDf, inColI):
        cName = inDf.columns[inColI]
        inDf[cName] = inDf[cName].str.strip()
        return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\spacingProblems.csv")
df=dfIn.copy(deep=True)
df=ColTrim(df,0)
```

Lower Case

Function: The **COLLOWER** sheet formulachanges all text in a column to lowercase.

Syntax:COLLOWER(ColumnID)

- ***ColumnID***. Required. Primary dataframe column to look for to cast to lowercase.

Video:

Python Code:

```
import pandas as pd

def ColToLower(inDf, inColI):
        cName = inDf.columns[inColI]
        inDf[cName] = inDf[cName].str.lower()
        return inDf
```

```
dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\atari.csv")
df=dfIn.copy(deep=True)
df=ColToLower(df,0)
```

Upper Case

Function: The **COLUPPER** sheet formula changes all text in a column to lowercase.

Syntax:COLUPPER(ColumnID)

- ***ColumnID***. Required. Primary dataframe column to look for to cast to uppercase.

Video:

Python Code:

```
import pandas as pd

def ColToUpper(inDf, inColI):
       cName = inDf.columns[inColI]
       inDf[cName] = inDf[cName].str.upper()
       return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\atari.csv")
df=dfIn.copy(deep=True)
df=ColToUpper(df,0)
```

Proper Case

Function: The **COLPROPER** sheet formula changes all text at the beginning of a word in a column to upper case while lowercasing all other text.

Syntax:COLPROPER(ColumnID)

- ***ColumnID***. Required. Primary dataframe column to look for to cast to proper case.

Video:

Python Code:

```
import pandas as pd

def ColToProper(inDf, inColI):
       cName = inDf.columns[inColI]
       inDf[cName] = inDf[cName].str.title()
       return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\atari.csv")
```

```
df=dfIn.copy(deep=True)
df=ColToProper(df,0)
```

View Null Rows

Function: The **VIEWNULL** sheet formula displays a dataframe including rows that contain a null value in a certain column.

Syntax:VIEWNULL(ColumnID)

- ***ColumnID***. Required. Primary dataframe column to look for rows containing null values.

Video:

Python Code:

```
import pandas as pd

def ViewNull(inDf, inColI):
       cName = inDf.columns[inColI]
       newDf = inDf[ inDf[cName].isnull() ]
       return newDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\atari.csv")
df=dfIn.copy(deep=True)
df=ViewNull(df,2)
```

View Non-Null Rows

Function: The **NOTVIEWNULL** sheet formula displays a dataframe including rows that don't contain a null value in a certain column.

Syntax:NOTVIEWNULL(ColumnID)

- ***ColumnID***. Required. Primary dataframe column to look for rows not containing null values.

Video:

Python Code:

```
import pandas as pd

def NotViewNull(inDf, inColI):
       cName = inDf.columns[inColI]
       newDf = inDf[ inDf[cName].notnull() ]
       return newDf
```

```
dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\atari.csv")
df=dfIn.copy(deep=True)
df=NotViewNull(df,2)
```

Replace Null With 0

Function: The **REPLACENULL** sheet formula replaces all null values in a column with zero.

Syntax:REPLACENULL(ColumnID, “Zero”)

- ***ColumnID***. Required. Primary dataframe column to look for rows containing null so as to replace the nulls with zeros.

Video:

Python Code:

```
import pandas as pd

def ReplaceNullZero(inDf, inColI):
       cName = inDf.columns[inColI]
       inDf[cName] = inDf[cName].fillna(0)
       return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\problemdata.csv")
df=dfIn.copy(deep=True)
df=ReplaceNullZero(df,0)
```

Replace Null With Mean

Function: The **REPLACENULL** sheet formula replaces all null values in a column with the mean value of the column.

Syntax:REPLACENULL(ColumnID, “Mean”)

- ***ColumnID***. Required. Primary dataframe column to look for rows containing null so as to replace the nulls with the mean.

Video:

Python Code:

```
import pandas as pd

def ReplaceNullMean(inDf, inColI):
       cName = inDf.columns[inColI]
       mean_value=inDf[cName].mean()
```

```
        inDf[cName].fillna(value=mean_value, inplace=True)
        return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\problemdata.csv")
df=dfIn.copy(deep=True)
df=ReplaceNullMean(df,0)
```

Replace Null With Previous

Function: The **REPLACENULL** sheet formula replaces all null values in a column with the previous non-null value in the column.

Syntax:REPLACENULL(ColumnID, “Prior”)

- ***ColumnID***. Required. Primary dataframe column to look for rows containing null so as to replace the nulls with the previous non-null value.

Video:

Python Code:

```
import pandas as pd

def ReplaceNullPrior(inDf, inColI):
        cName = inDf.columns[inColI]
        inDf[cName].fillna(method='ffill', inplace=True)
        return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\problemdata.csv")
df=dfIn.copy(deep=True)
df=ReplaceNullPrior(df,0)
```

Replace Null With Blank Text

Function: The **REPLACENULL** sheet formula replaces all null values in a column with blank text.

Syntax:REPLACENULL(ColumnID, “”)

- ***ColumnID***. Required. Primary dataframe column to look for rows containing null so as to replace the nulls with blank text.

Video:

Python Code:

```
import pandas as pd
```

```
def ReplaceNullBlank(inDf, inColI):
       cName = inDf.columns[inColI]
       inDf[cName] = inDf[cName].fillna('')
       return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\problemdata.csv")
df=dfIn.copy(deep=True)
df=ReplaceNullBlank(df,0)
```

View Blank Strings

Function: The **VIEWBLANK** sheet formula displays a dataframe including rows that contain a null value or blank text in a certain column.

Syntax:VIEWBLANK(ColumnID)

- ***ColumnID***. Required. Primary dataframe column to look for rows containing null or blank values.

Video:

Python Code:

```
import pandas as pd

def ViewBlank(inDf, inColI):
       cName = inDf.columns[inColI]
       newDf = inDf[ ( (inDf[cName].isnull()) | (inDf[cName]==u'') )]
       return newDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\BlankText.csv")
df=dfIn.copy(deep=True)
df=ViewBlank(df,4)
```

View Non-Blank Strings

Function: The **NOTVIEWBLANK** sheet formula displays a dataframe including rows that don’t contain a null value or blank text in a certain column.

Syntax:NOTVIEWBLANK(ColumnID)

- ***ColumnID***. Required. Primary dataframe column to look for rows not containing null or blank values.

Video:

Python Code:

```
import pandas as pd

def NotViewBlank(inDf, inColI):
       cName = inDf.columns[inColI]
       newDf = inDf[ (inDf[cName].notnull()) & (inDf[cName]!=u'') ]
       return newDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\BlankText.csv")
df=dfIn.copy(deep=True)
df=NotViewBlank(df,4)
```

Clean Address

Function: The **CLEANADDRESS** sheet formula cleans addresses by executing the following: capitalizing the beginning of words, stripping leading and trailing whitespace, removing periods, replacing the words “Street”, “Apartment”, and “Av” with “St”, “Apt”, and “Ave”, respectively.

Syntax:CLEANADDRESS(ColumnID)

- ***ColumnID***. Required. Primary dataframe column to look for to clean addresses.

Video:

Python Code:

```
import pandas as pd

def CleanAddress(inDf, inColI):
       cName = inDf.columns[inColI]
       newCol = []
       for index, row in inDf.iterrows():
              aParts = row[cName].split()
              wList =[]
              for wd in aParts:
                     if len(wd) > 2:wd = wd.title()
                     wList.append(wd)
              addr = " ".join(wList)
              addr = addr.strip()
              addr = addr.replace('.','')
              addr = addr.replace('Street', 'St')
              addr = addr.replace('Apartment', 'Apt')
              addr = addr.replace('Av', 'Ave')
              newCol.append(addr)
       inDf['Minimize Address'] = newCol
       return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\addressList.csv")
df=dfIn.copy(deep=True)
df=CleanAddress(df,0)
```

Clean Name

Function: The **CLEANNAME** sheet formula parses full names and creates columns that contain cleaned parts of each name: Title, First, Middle, Last.

Syntax:CLEANNAME(ColumnID)

- ***ColumnID***. Required. Primary dataframe column to look for to parse names.

Video:

Python Code:

```
import pandas as pd

from nameparser import HumanName

def CleanNames(inDf, inColI):
       inDf['Title'] = None
       inDf['First'] = None
       inDf['Middle'] = None
       inDf['Last'] = None
       cName = inDf.columns[inColI]
       for index, row in inDf.iterrows():
              name = HumanName(row[cName])
              name.capitalize()
              row['Title'] = name.title
              row['First'] = name.first
              row['Middle'] = name.middle
              row['Last'] = name.last
       return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\fullnames.csv")
df=dfIn.copy(deep=True)
df=CleanNames(df,0)
```

CLEANEMAIL(A:A) (Not sure how this function works)

Function: The **CLEANEMAIL** sheet formula parses emails.

Syntax:CLEANEMAIL(ColumnID)

- ***ColumnID***. Required. Primary dataframe column to look for to parse emails.

Video:

Python Code:

```
import itertools
import pandas as pd
import re

def CleanEmail(idf, textI):
	cName = idf.columns[textI]
	colList = []
	for txt in idf[cName]:
		emails =
re.findall(r"[A-Za-z0-9\.\-+_]+@[A-Za-z0-9\.\-+_]+\.[A-Za-z]+", txt)
		for em in emails:
			colList.append(em)
	newDf = pd.DataFrame(data={'Emails':colList})
	return newDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\GovRss.csv")
df=dfIn.copy(deep=True)
df=CleanEmail(df,0)
```

Clean Phone Numbers

Function: The **CLEANPHONE** sheet formula parses phone numbers, formats them to the region which is specified and gives the location and time zone of each number..

Syntax:CLEANPHONE(ColumnID, “Region”)

- ***ColumnID***. Required. Primary dataframe column to look for to parse phone numbers.
- ***Region.*** Required. The region whose format the phone numbers should conform to.

Video:

Python Code:

```
import pandas as pd
import phonenumbers

from phonenumbers import carrier, timezone, geocoder

def CleanPhone(inDf, inColI, inRegion):
	# inRegion Codes:
https://en.wikipedia.org/wiki/List_of_ISO_3166_country_codes
	inDf['Valid'] = None
	inDf['Formated Number'] = None
	inDf['Time Zone'] = None
	inDf['Location'] = None
	cName = inDf.columns[inColI]
	for index, row in inDf.iterrows():
		matcher = phonenumbers.PhoneNumberMatcher(row[cName], inRegion)
		if matcher.has_next():
			row['Valid'] = 1
			mNum = matcher.next().raw_string
			parsed = phonenumbers.parse(mNum, inRegion)
			row['Formated Number']=phonenumbers.format_number(parsed,
phonenumbers.PhoneNumberFormat.NATIONAL)
```

```
                        row['Time Zone'] = timezone.time_zones_for_number(parsed)[0]
                        row['Location'] = geocoder.description_for_number(parsed, "en")
                else:
                        row['Valid','Formated Number','Time Zone','Location'] =
[0,"","",""]
        return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\PhoneNumbers.csv")
df=dfIn.copy(deep=True)
df=CleanPhone(df,0,"US")
```

Clip Numbers

Function: The **CLIPNUM** sheet formula takes a column of numbers and sets them to a specified minimum value if they are lower than the minimum, to the specified maximum value if they are greater than the maximum, and doesn't change numbers that are in between the minimum and maximum.

Syntax:CLIPNUM(ColumnID, minimum, maximum)

- ***ColumnID***. Required. Primary dataframe column to look for to parse emails.
- ***Minimum***. Required. Lower bound cutoff for numbers to be compared against.
- ***Maximum***. Required. Upper bound cutoff for numbers to be compared against.

Video:

Python Code:

```
import pandas as pd

def ClipNum(inDf, inColI, inMin, inMax):
        cName = inDf.columns[inColI]
        inDf[cName] = inDf[cName].clip(inMin, inMax)
        return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\sciencedata.csv")
df=dfIn.copy(deep=True)
df=ClipNum(df,0,.1,1.9)
```

Only Alpha Numeric

Function: The **RESTRICT** sheet formula displays strips any non alphanumeric characters from strings in a dataframe column.

Syntax:RESTRICT(ColumnID, "alphanum")

- ***ColumnID***. Required. Primary dataframe column in which values will be formatted according to the restriction.

Video:

Python Code:

```
import codecs
import pandas as pd

def RestrictAlphaNum(inDf, inColI):
       cName = inDf.columns[inColI]
       newVals=[]
       for item in inDf[cName]:
              output=""
              for char in item:
                     if char.isalnum() or char==" ":
                            output += char
              newVals.append(output)
       inDf[cName] = newVals
       return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\unicodeproblems.csv")
df=dfIn.copy(deep=True)
df=RestrictAlphaNum(df,0)
```

Only ASCII Characters

Function: The **RESTRICT** sheet formula displays strips any non ASCII characters from strings in a dataframe column.

Syntax:RESTRICT(ColumnID, “ascii”)

- ***ColumnID***. Required. Primary dataframe column in which values will be formatted according to the restriction.

Video:

Python Code:

```
import codecs
import pandas as pd
import string

def RestrictAscii(inDf, inColI):
       cName = inDf.columns[inColI]
       newVals=[]
       printable = set(string.printable)
       for item in inDf[cName]:
              output= ''.join(filter(lambda x: x in printable, item))
              newVals.append(output)
       inDf[cName] = newVals
       return inDf
```

```
dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\unicodeproblems.csv")
df=dfIn.copy(deep=True)
df=RestrictAscii(df,0)
```

Categorical

Function: The **CATEGORICAL** sheet formula appends a column to a dataframe containing 1’s and 0’s based on whether or not a column contains a specified substring.

Syntax:CATEGORICAL(ColumnID, substring)

- ***ColumnID***. Required. Primary dataframe column to look for rows containing substrings.
- ***Substring.*** Required. The substring that is to be checked against every row in the specified column.

Video:

Python Code:

```
import numpy as np
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\Vehicles.csv")
df=dfIn.copy(deep=True)
df["Manual 5-spd"]=np.where(df.iloc[:,5].str.contains("Manual 5-spd"),1,0)
```

Categorical - Multi

Function: The **CATEGORICAL** sheet formula appends two columns to a dataframe containing 1’s and 0’s based on whether or not a column contains two specified substrings.

Syntax:CATEGORICAL(ColumnID, (substring1, substring2),(newColName1, newColName2))

- ***ColumnID***. Required. Primary dataframe column to look for rows containing substrings.
- ***Substring1.*** Required. The first substring that is to be checked against every row in the specified column.
- ***Substring1.*** Required. The second substring that is to be checked against every row in the specified column.
- ***newColName1***. Required. The name of the column created based on the presence of the first substring.
- ***newColName2***. Required. The name of the column created based on the presence of the second substring.

Video:

Python Code:

```
import numpy as np
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\Vehicles.csv")
df=dfIn.copy(deep=True)
df["Manual5"] = np.where(df.iloc[:,5].str.contains("Manual 5-spd"),1,0)
df["Automatic3"] = np.where(df.iloc[:,5].str.contains("Automatic 3-spd"),1,0)
```

Column Formulas

Random Integer

Function: The **GENRANDINT** sheet formula generates a user-specified number of random integers in a user-specified range.

Syntax:GENRANDINT(minValue, maxValue, N)

- ***minValue***. Required. The lower bound on the range of integers that can be generated.
- ***minValue***. Required. The upper bound on the range of integers that can be generated.
- ***N***. Required. The number of integers to be generated.

Video:

Python Code:

```
import pandas as pd

from random import randint

def GenRandomInt(inStart, inEnd, inNumber):
       rc = [randint(inStart, inEnd) for p in range(0, inNumber)]
       dfGen = pd.DataFrame({'Integer':rc})
       return dfGen

dfIn=GenRandomInt(1,28,200)
```

Random Number

Function: The **GENRANDNUM** sheet formula generates a user-specified number of random real-valued numbers in a user-specified range.

Syntax:GENRANDNUM(minValue, maxValue, N)

- ***minValue***. Required. The lower bound on the range of real-valued numbers that can be generated.
- ***minValue***. Required. The upper bound on the range of real-valued numbers that can be generated.
- ***N***. Required. The number of values to be generated.

Video:

Python Code:

```
import pandas as pd

from random import uniform

def GenRandomNum(inStart, inEnd, inNumber):
       rc = [uniform(inStart, inEnd) for p in range(0, inNumber)]
       dfGen = pd.DataFrame({'Integer':rc})
       return dfGen

dfIn=GenRandomNum(1,28,200)
```

Increasing Numbers

Function: The **GENINCREMENT** sheet formula generates a dataframe of all integers starting from a certain integer (inclusive) and ending at a certain value (non inclusive).

Syntax:GENINCREMENT(startValue, endValue)

- ***startValue***. Required. The lower bound on the range of integers (inclusive)
- ***endValue***. Required. The upper bound on the range of integers (non inclusive).

Video:

Python Code:

```
import pandas as pd

from random import uniform

def GenIncrement(inStart, inEnd):
       rc = list(range(inStart, inEnd))
       dfGen = pd.DataFrame({'Values':rc})
       return dfGen

dfIn=GenIncrement(28,200)
```

Random Integer Column

Function: The **RANDINT** sheet formula appends a column of random integers that belong to a user-specified range.

Syntax:RANDINT(minValue, maxValue)

- ***minValue***. Required. The lower bound on the range of integers that can be generated.
- ***maxValue***. Required. The upper bound on the range of integers that can be generated.

Video:

Python Code:

```
import pandas as pd

from random import randint

def RandomInt(inDf, inMin, inMax):
       rc = [randint(inMin, inMax) for p in range(0, inDf.shape[0])]
       inDf['Random Int'] = rc
       return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\traffic.csv")
df=dfIn.copy(deep=True)
df=RandomInt(df,28,200)
```

Random Number Column

Function: The **GENRANDNUM** sheet formula appends a column of random real-valued numbers that belong to a user-specified range.

Syntax:GENRANDNUM(minValue, maxValue)

- ***minValue***. Required. The lower bound on the range of real-valued numbers that can be generated.
- ***maxValue***. Required. The upper bound on the range of real-valued numbers that can be generated.

Video:

Python Code:

```
import pandas as pd

from random import uniform

def RandomNum(inDf, inMin, inMax):
       rc = [uniform(inMin, inMax) for p in range(0, inDf.shape[0])]
       inDf['Random Num'] = rc
       return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\traffic.csv")
df=dfIn.copy(deep=True)
df=RandomNum(df,28,200)
```

Increasing Number Column

Function: The **INCREMENT** sheet formula appends a column of integers increasing by one from a user specified starting value for every row in the dataframe.

Syntax:INCREMENT(startValue)

- ***startValue***. Required. The lower bound on the range of integers (inclusive)

Video:

Python Code:

```
import pandas as pd

def IncrementInt(inDf, inStartNum):
       inDf['Random Num'] =  df.index + inStartNum
       return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\traffic.csv")
df=dfIn.copy(deep=True)
df=IncrementInt(df,945)
```

Random String

Function: The **RANDTEXT** sheet formula appends a column of string with random text – the length of each string being determined by user input.

Syntax:RANDTEXT(stringLength, charType)

- ***stringLength***. Required. The length of the strings in the appended column.
- ***charType***. Required. The type of characters that the string will consist of. Options for charType are: password (consists of alphabet characters, digits, and punctuation), alphanumeric (consist of alphabet characters and digits), uppercase (uppercase alphabet characters), digits (digits 0-9), and lowercase (lowercase alphabet characters).

Video:

Python Code:

```
import pandas as pd
import random
import string

def GenRandomText(inDf, inStringSize, inType):
       rList=[]
       letters = string.ascii_letters
       if inType=="password":letters = string.ascii_letters + string.digits +
string.punctuation
       if inType=="alphanumeric":letters = string.ascii_letters + string.digits
       if inType=="uppercase":letters = string.ascii_uppercase
       if inType=="digits":letters = string.digits
       if inType=="lowercase":letters = string.ascii_lowercase
       for i in range(0,inDf.shape[0]):
              rStr = ''.join(random.choice(letters) for i in range(inStringSize))
              rList.append(rStr)
       inDf['Random String'] =  rList
       return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\traffic.csv")
df=dfIn.copy(deep=True)
df=GenRandomText(df,40,"alphanumeric")
```

ADDCOLUMN

IF

Function: The **IF** function– when fed into the ADDCOLUMN sheet formula– appends a new column with a specified name containing specified values according to boolean if operations. The function allows for nesting.

Syntax:IF(boolean, valueIfTrue, valueIfFalse)
- ***boolean***. Required. The boolean expression that is to be evaluated.
- valueIfTrue. Required. The value that is to be returned if the boolean expression evaluates to TRUE. (Allows for nesting)
- valueIfFalse. Required. The value that is to be returned if the boolean expression evaluates to false. (Allows for nesting)

ADDCOLUMN Syntax: ADDCOLUMN(columnName, IfStatement)
- ***columnName***. Required. The name of the column to be appended.
- ***IfStatement***. Required. The if statement specified by the user.

Video:

Python Code:

```
import numpy as np
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\TestIf.csv")
df=dfIn.copy(deep=True)
df["New Column"] =
np.where(df.iloc[:,0]==df.iloc[:,1],0,np.where(df.iloc[:,0]>df.iloc[:,1],1,-1))
```

ABS

Function: The **ABS** function – when fed into the ADDCOLUMN sheet formula– appends a new column with a specified name containing the absolute value of another chosen dataframe column.

Syntax:ABS(columnID)

- ***columnID***. Required. The column that will have its absolute value appended. Mathematical expression may be added inside the formula (columnID * 2 will return the absolute value multiplied by 2).

ADDCOLUMN Syntax: ADDCOLUMN(columnName, ABSformula)

- ***columnName***. Required. The name of the column to be appended.
- ***ABSformula***. Required. The ABS formula specified by the user.

Video:

Python Code:

```
import numpy as np
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\traffic.csv")
df=dfIn.copy(deep=True)
df["ABS Value"] = np.abs((pd.to_numeric(df.iloc[:,0])*2).to_numpy())
```

CEILING

Function: The **CEILING** function – when fed into the ADDCOLUMN sheet formula– appends a new column with a specified name containing the least common ceiling of two values specified by the users. Similar to the concept of LCM but with a ceiling instead of multiple.

Example: CEILING(2,7) = 8

Syntax:CEILING(columnID/value1, columnID/value1)

- ***columnID/value1***. Required. The first column/value that will be used to calculate the least common ceiling.
- ***columnID/value2***. Required. The second column/value that will be used to calculate the least common ceiling.

ADDCOLUMN Syntax: ADDCOLUMN(columnName, CeilingFormula)

- ***columnName***. Required. The name of the column to be appended.
- ***CeilingFormula***. Required. The ceiling formula specified by the user.

Video:

Python Code:

```
import numpy as np
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\areadata.csv")
df=dfIn.copy(deep=True)
df["Test Ceiling"] = 1*np.ceil(pd.to_numeric(df.iloc[:,1])/1)
```

ADDCOLUMN("Test Even", EVEN(B:B)) (Not sure what this function is supposed to do

FLOOR

Function: The **FLOOR** function – when fed into the ADDCOLUMN sheet formula– appends a new column with a specified name containing the greatest common floor of two values specified by the users. Similar to the concept of GCF but with floor instead of factor.

Example: FLOOR(5.1,2) = 4

Syntax:FLOOR(columnID/value1, columnID/value1)

- ***columnID/value1***. Required. The first column/value that will be used to calculate the greatest common floor.
- ***columnID/value2***. Required. The second column/value that will be used to calculate the greatest common floor.

ADDCOLUMN Syntax: ADDCOLUMN(columnName, FloorFormula)

- ***columnName***. Required. The name of the column to be appended.
- ***FloorFormula***. Required. The floor formula specified by the user.

Video:

Python Code:

```
import numpy as np
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\areadata.csv")
df=dfIn.copy(deep=True)
df["Test Floor"] = 2*np.floor(df.iloc[:,1]/2)
```

INT

Function: The **INT** function – when fed into the ADDCOLUMN sheet formula– appends a new column with a specified name containing the integer value of another column. The INT function strips away any values after the decimal place and leaves only the integer portion.

Syntax:INT(columnID)

- ***columnID***. Required. The column that will have its integer value appended.

ADDCOLUMN Syntax: ADDCOLUMN(columnName, IntFormula)

- ***columnName***. Required. The name of the column to be appended.
- ***IntFormula***. Required. The integer formula specified by the user.

Video:

Python Code:

```
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\areadata.csv")
df=dfIn.copy(deep=True)
df["Test Int"] = pd.to_numeric(df.iloc[:,1]).astype('int64')
```

LCM

Function: The **LCM** function – when fed into the ADDCOLUMN sheet formula– appends a new column with a specified name containing the least common multiple of two values (dataframe columns or hard values can be used).

Syntax:LCM(columnID/value1, columnID/value2)

- ***columnID/value1***. Required. The first column/value that will be used to calculate the least common multiple.
- ***columnID/value2***. Required. The second column/value that will be used to calculate the least common multiple.

ADDCOLUMN Syntax: ADDCOLUMN(columnName, LCMformula)

- ***columnName***. Required. The name of the column to be appended.
- ***LCMformula***. Required. The LCM formula specified by the user.

Video:

Python Code:

```
import numpy as np
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\areadata.csv")
df=dfIn.copy(deep=True)
df["Test LCM"] =
np.lcm.reduce(np.concatenate((df.iloc[:,[0]].to_numpy(),np.array([[10]]*df.shape[0]
)),axis=1),axis=1)
```

MOD

Function: The **MOD** function – when fed into the ADDCOLUMN sheet formula– appends a new column with a specified name containing the modulo of two numbers (dataframe columns or hard values can be used). Modulo is the remainder when one number is divided by another.

Syntax:MOD(columnID/value1, columnID/value2)

- ***columnID/value1***. Required. The first column/value that will be used to calculate the modulo.
- ***columnID/value2***. Required. The second column/value that will be used to calculate the modulo.

ADDCOLUMN Syntax: ADDCOLUMN(columnName, ModFormula)

- ***columnName***. Required. The name of the column to be appended.
- ***IntFormula***. Required. The MOD formula specified by the user.

Video:

Python Code:

```
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\areadata.csv")
df=dfIn.copy(deep=True)
df["Test MOD"] = pd.to_numeric(df.iloc[:,0])%2
```

MROUND

Function: The **MROUND** function – when fed into the ADDCOLUMN sheet formula– appends a new column with a specified name containing a number rounded to the nearest multiple of a second user-specified number. (Dataframe columns or hard values can be used).

Syntax:MROUND(columnID/value1, columnID/value2)

- ***columnID/value1***. Required. The first column/value that is to be rounded.
- ***columnID/value2***. Required. The second column/value that will be the number whose nearest multiple the first number will be rounded to.

ADDCOLUMN Syntax: ADDCOLUMN(columnName, MROUNDformula)

- ***columnName***. Required. The name of the column to be appended.
- ***MROUNDformula***. Required. The MROUND formula specified by the user.

Video:

Python Code:

```
import math
import pandas as pd

def MultipleRound(n, mul):
    if mul==0:return 0
    return mul*(math.floor(n/mul+0.5));

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\areadata.csv")
df=dfIn.copy(deep=True)
df["Test MROUND"] = pd.to_numeric(df.iloc[:,1]).apply(lambda x:MultipleRound(x,3))
```

POWER

Function: The **POWER** function – when fed into the ADDCOLUMN sheet formula– appends a new column with a specified name containing a user-specified value raised to the power of another user-specified value. (Dataframe columns or hard values can be used).

Syntax:POWER(columnID/value1, columnID/value2)

- ***columnID/value1***. Required. The first column/value that serves as the base..
- ***columnID/value2***. Required. The second column/value that will be the value in the exponent.

ADDCOLUMN Syntax: ADDCOLUMN(columnName, POWERformula)

- ***columnName***. Required. The name of the column to be appended.
- ***POWERformula***. Required. The POWER formula specified by the user.

Video:

Python Code:

```
import numpy as np
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\areadata.csv")
df=dfIn.copy(deep=True)
df["Test POWER"] = np.power(pd.to_numeric(df.iloc[:,0]),2)
```

PRODUCT

Function: The **PRODUCT** function – when fed into the ADDCOLUMN sheet formula– appends a new column with a specified name containing a user-specified value multiplied by another user-specified value. (Dataframe columns or hard values can be used).

Syntax:PRODUCT(columnID/value1, columnID/value2)

- ***columnID/value1***. Required. The first column/value that will be multiplied.
- ***columnID/value2***. Required. The second column/value that multiplies the first value.

ADDCOLUMN Syntax: ADDCOLUMN(columnName, PRODUCTformula)

- ***columnName***. Required. The name of the column to be appended.
- ***PRODUCTformula***. Required. The PRODUCT formula specified by the user.

Video:

Python Code:

```
import numpy as np
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\areadata.csv")
df=dfIn.copy(deep=True)
df["Test POWER"] = np.power(pd.to_numeric(df.iloc[:,0]),2)
```

QUOTIENT

Function: The **QUOTIENT** function – when fed into the ADDCOLUMN sheet formula– appends a new column with a specified name containing a user-specified value divided by another user-specified value. (Dataframe columns or hard values can be used).

Syntax:QUOTIENT(columnID/value1, columnID/value2)

- ***columnID/value1***. Required. The first column/value serves as the dividend. (The number on top of the fraction)
- ***columnID/value2***. Required. The second column/value serves as the divisor. (The number on the bottom of the fraction)

ADDCOLUMN Syntax: ADDCOLUMN(columnName, QUOTIENTformula)

- ***columnName***. Required. The name of the column to be appended.
- ***QUOTIENTformula***. Required. The QUOTIENT formula specified by the user.

Video:

Python Code:

```
import numpy as np
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\areadata.csv")
df=dfIn.copy(deep=True)
df["Test QUOTIENT"] =
np.divmod(pd.to_numeric(df.iloc[:,0]).to_numpy(),np.array([5]*df.shape[0]))[0]
```

ROUNDDOWN

Function: The **ROUNDDOWN** function – when fed into the ADDCOLUMN sheet formula– appends a new column with a specified name containing a number to the nearest integer less than the number. (Dataframe columns or hard values can be used).

Syntax:ROUNDDOWN(columnID/value1, columnID/value2)

- ***columnID/value1***. Required. The column/value that is to be rounded.

ADDCOLUMN Syntax: ADDCOLUMN(columnName, ROUNDDOWNformula)

- ***columnName***. Required. The name of the column to be appended.
- ***ROUNDDOWNformula***. Required. The ROUNDDOWN formula specified by the user.

Video:

Python Code:

```
import math
import pandas as pd

def HalfRoundUp(n, decimals=0):
    multiplier = 10 ** decimals
    return math.floor(n*multiplier + 0.5) / multiplier

def RoundDown(n, m):
    mul=pow(10, abs(m));
    sign = 1 if n > 0 else -1
    if m>0:return sign*HalfRoundUp((abs(n)-(1/mul*0.5))*mul)/mul
    elif m==0:return sign*HalfRoundUp((abs(n)-0.5))
    return sign*HalfRoundUp((abs(n)-mul*0.5)/mul)*mul

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\areadata.csv")
df=dfIn.copy(deep=True)
df["Test ROUNDDOWN"] = pd.to_numeric(df.iloc[:,1]).apply(lambda x:RoundDown(x,0))
```

ROUNDUP

Function: The **ROUNDUP** function – when fed into the ADDCOLUMN sheet formula– appends a new column with a specified name containing a number to the nearest integer greater than the number. (Dataframe columns or hard values can be used).

Syntax:ROUNDUP(columnID/value1, columnID/value2)

- ***columnID/value1***. Required. The column/value that is to be rounded.

ADDCOLUMN Syntax: ADDCOLUMN(columnName, ROUNDUPformula)

- ***columnName***. Required. The name of the column to be appended.
- ***ROUNDUPformula***. Required. The ROUNDUP formula specified by the user.

Video:

Python Code:

```
import math
import pandas as pd

def HalfRoundUp(n, decimals=0):
    multiplier = 10 ** decimals
    return math.floor(n*multiplier + 0.5) / multiplier

def RoundUp(n, m):
    mul=pow(10, abs(m));
    sign = 1 if n > 0 else -1
    if m>0:return sign*HalfRoundUp((abs(n)+(1/mul*0.5))*mul)/mul
    elif m==0:return sign*HalfRoundUp((abs(n)+0.5))
    return sign*HalfRoundUp((abs(n)+mul*0.5)/mul)*mul

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\areadata.csv")
df=dfIn.copy(deep=True)
df["Test ROUNDUP"] = df.iloc[:,1].apply(lambda x:RoundUp(x,0))
```

SIGN

Function: The **SIGN** function – when fed into the ADDCOLUMN sheet formula– appends a new column with a specified name containing the sign (-1, 1) of the numbers in another chosen dataframe column.

Syntax:SIGN(columnID)

- ***columnID***. Required. The column that will have the sign of the values in each row returned.

ADDCOLUMN Syntax: ADDCOLUMN(columnName, SIGNformula)

- ***columnName***. Required. The name of the column to be appended.
- ***SIGNformula***. Required. The SIGN formula specified by the user.

Video:

Python Code:

```
import numpy as np
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\traffic.csv")
df=dfIn.copy(deep=True)
df["ABS Value"] = np.sign(pd.to_numeric(df.iloc[:,0]).to_numpy())
```

SQRT

Function: The **SQRT** function – when fed into the ADDCOLUMN sheet formula– appends a new column with a specified name containing the square root of the numbers in another chosen dataframe column.

Syntax:SQRT(columnID)

- ***columnID***. Required. The column that will have the square root of the values in each row returned.

ADDCOLUMN Syntax: ADDCOLUMN(columnName, SQRTformula)

- ***columnName***. Required. The name of the column to be appended.
- ***SQRTformula***. Required. The SQRT formula specified by the user.

Video:

Python Code:

```
import numpy as np
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\traffic.csv")
df=dfIn.copy(deep=True)
df["ABS Value"] = np.sign(pd.to_numeric(df.iloc[:,0]).to_numpy())
```

SUM

Function: The **SUM** function – when fed into the ADDCOLUMN sheet formula– appends a new column with a specified name containing a user-specified value added to another user-specified value. (Dataframe columns or hard values can be used).

Syntax:SUM(columnID/value1, columnID/value2)

- ***columnID/value1***. Required. The first column/value that will be added
- ***columnID/value2***. Required. The second column/value that will be added to the first value.

ADDCOLUMN Syntax: ADDCOLUMN(columnName, SUMformula)

- ***columnName***. Required. The name of the column to be appended.
- ***SUMformula***. Required. The SUM formula specified by the user.

Video:

Python Code:

```
import numpy as np
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\areadata.csv")
df=dfIn.copy(deep=True)
df["Test POWER"] = np.power(pd.to_numeric(df.iloc[:,0]),2)
```

TRUNC

Function: The **TRUNC** function – when fed into the ADDCOLUMN sheet formula– appends a new column with a specified name containing the truncation of another chosen dataframe column. (Only returns the integer portion of the number)

Syntax:TRUNC(columnID)

- ***columnID***. Required. The column that will have the values truncated.

ADDCOLUMN Syntax: ADDCOLUMN(columnName, TRUNCformula)

- ***columnName***. Required. The name of the column to be appended.
- ***TRUNCformula***. Required. The TRUNC formula specified by the user.

Video:

Python Code:

```
import math
import pandas as pd

def Truncate(n, decimals=0):
    multiplier = 10 ** decimals
    if decimals <=0:return int(math.trunc(n*multiplier) / multiplier)
    return math.trunc(n*multiplier) / multiplier

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\areadata.csv")
df=dfIn.copy(deep=True)
df["TRUNC Value"] = pd.to_numeric(df.iloc[:,1]).apply(lambda x:Truncate(x))
```

CHAR

Function: The **CHAR** function – when fed into the ADDCOLUMN sheet formula– appends a new column with a specified name containing the character that is equivalent to the character codes of another specified dataframe column.

Syntax:CHAR(columnID)

- ***columnID***. Required. The column which contains the character codes.

ADDCOLUMN Syntax: ADDCOLUMN(columnName, CHARformula)

- ***columnName***. Required. The name of the column to be appended.
- ***CHARformula***. Required. The CHAR formula specified by the user.

Video:

Python Code:

```
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\charexample.csv")
df=dfIn.copy(deep=True)
df["Char Test"] = pd.to_numeric(df.iloc[:,0]).apply(lambda x:chr(x))
```

CLEAN

Function: The **CLEAN** sheet formula cleans text in a dataframe column.

Syntax:CLEAN(ColumnID)

- ***ColumnID***. Required. Primary dataframe column in which text will be pulled for cleaning.

ADDCOLUMN Syntax: ADDCOLUMN(columnName, CLEANformula)

- ***columnName***. Required. The name of the column to be appended.
- ***CLEANformula***. Required. The CLEAN formula specified by the user.
-

Video:

Python Code:

```
import pandas as pd
import string

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\unicodeproblems.csv")
df=dfIn.copy(deep=True)
df["Clean Test"] = df.iloc[:,0].astype(str).apply(lambda x:''.join(filter(lambda
y:y in string.printable[:-5],x)))
```

CODE

Function: The **CODE** function – when fed into the ADDCOLUMN sheet formula– appends a new column with a specified name containing the character code that is equivalent to the characters in another specified dataframe column.

Syntax:CODE(columnID)

- ***columnID***. Required. The column which contains the character codes.

ADDCOLUMN Syntax: ADDCOLUMN(columnName, CODEformula)

- ***columnName***. Required. The name of the column to be appended.
- ***CODEformula***. Required. The CODE formula specified by the user.

Video:

Python Code:

```
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\charCode.csv")
df=dfIn.copy(deep=True)
df["Code Test"] = df.iloc[:,0].apply(lambda x:ord(x[0]))
```

DOLLAR

Function: The **DOLLAR** function – when fed into the ADDCOLUMN function – appends a new column with a specified name containing the data from another column in currency format.

Syntax:DOLLAR(columnID)

- ***columnID***. Required. The column which contains the values to be converted to currency format.

ADDCOLUMN Syntax: ADDCOLUMN(columnName, DOLLARformula)

- ***columnName***. Required. The name of the column to be appended.
- ***DOLLARformula***. Required. The DOLLAR formula specified by the user.

Video:

Python Code:

```
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\Vegetables.csv")
df=dfIn.copy(deep=True)
df["Final Price"] = (pd.to_numeric(df.iloc[:,2])*300).apply(lambda
x:('${:,.'+str(2)+'f}').format(x))
```

EXACT

Function: The **EXACT** function – when fed into the ADDCOLUMN sheet formula – appends a new column with a specified name containing a 1 or 0 depending on if two other chosen columns contain the exact same string.

Syntax:EXACT(*columnID1, columnID2*)

- ***columnID1***. Required. The column which contains the first string that is being checked for equality.
- ***columnID2***. Required. The column which contains the second string that is being checked for equality.

ADDCOLUMN Syntax: ADDCOLUMN(*columnName, EXACTformula*)

- ***columnName***. Required. The name of the column to be appended.
- ***EXACTformula***. Required. The EXACT formula specified by the user.

Video:

Python Code:

```
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\dogNames.csv")
df=dfIn.copy(deep=True)
df["Final Price"] = (df.iloc[:,0].astype(str)==df.iloc[:,1].astype(str))
```

FIND

Function: The **FIND** function – when fed into the ADDCOLUMN sheet formula – appends a new column with a specified name containing the index at which a user-specified substring first appears in each row in a chosen column (if at all).

Syntax:FIND(columnID)

- ***columnID***. Required. The column which contains the character codes.

ADDCOLUMN Syntax: ADDCOLUMN(columnName, FINDformula)

- ***columnName***. Required. The name of the column to be appended.
- ***FINDformula***. Required. The FIND formula specified by the user.

Video:

Python Code:

```
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\findExample.csv")
df=dfIn.copy(deep=True)
df["Result"] = df.iloc[:,0].astype(str).str.find("dog").replace(-1, float('NaN')
).astype('Int64')
```

LEFT

Function: The **LEFT** function – when fed into the ADDCOLUMN sheet formula – appends a new column with a specified name containing the portion to the left of a specified index of a string contained in another chosen column.

Syntax:LEFT(columnID)

- ***columnID***. Required. The column which contains the strings to be sliced.

ADDCOLUMN Syntax: ADDCOLUMN(columnName, LEFTformula)

- ***columnName***. Required. The name of the column to be appended.
- ***LEFTformula***. Required. The LEFT formula specified by the user.

Video:

Python Code:

```
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\dogNames.csv")
df=dfIn.copy(deep=True)
df["Left Output"] = df.iloc[:,0].astype(str).astype(str).str[:4]
```

LEN

Function: The **LEN** function – when fed into the ADDCOLUMN sheet formula – appends a new column with a specified name containing the length of the strings in another chosen column.

Syntax:LEN(columnID)

- ***columnID***. Required. The column which contains the strings whose lengths will be appended.

ADDCOLUMN Syntax: ADDCOLUMN(columnName, LENformula)

- ***columnName***. Required. The name of the column to be appended.
- ***LENformula***. Required. The LEN formula specified by the user.

Video:

Python Code:

```
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\dogNames.csv")
df=dfIn.copy(deep=True)
df["Left Output"] = df.iloc[:,0].astype(str).astype(str).str[:4]
```

LOWER

Function: The **LOWER** function – when fed into the ADDCOLUMN sheet formula – appends a new column with a specified name containing the strings in another column casted to lowercase.

Syntax:LOWER(columnID)

- ***columnID***. Required. The column which contains the strings which will be cast to lowercase.

ADDCOLUMN Syntax: ADDCOLUMN(columnName, LOWERformula)

- ***columnName***. Required. The name of the column to be appended.
- ***LOWERformula***. Required. The LOWER formula specified by the user.

Video:

Python Code:

```
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\dogNames.csv")
df=dfIn.copy(deep=True)
df["Left Output"] = df.iloc[:,0].astype(str).astype(str).str[:4]
```

MID

Function: The **MID** function – when fed into the ADDCOLUMN function – appends a new column with a specified name containing the portion in between two specified string indices of a string contained in another chosen column.

Syntax:MID(columnID)

- ***columnID***. Required. The column which contains the strings to be sliced.

ADDCOLUMN Syntax: ADDCOLUMN(columnName, MIDformula)

- ***columnName***. Required. The name of the column to be appended.
- ***MIDformula***. Required. The MIDformula specified by the user.

Video:

Python Code:

```
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\dogNames.csv")
df=dfIn.copy(deep=True)
df["MID Output"] = df.iloc[:,0].astype(str).str.slice(6,11)
```

PROPER

Function: The **PROPER** function – when fed into the ADDCOLUMN sheet formula – appends a new column with a specified name containing the strings in another column changed such that all text at the beginning of a word in a column to upper case while lowercasing all other text.

Syntax:PROPER(columnID)

- ***columnID***. Required. The column which contains the strings which will be cast to proper case.

ADDCOLUMN Syntax: ADDCOLUMN(columnName, PROPERformula)

- ***columnName***. Required. The name of the column to be appended.
- ***PROPERformula***. Required. The PROPERformula specified by the user.

Video:

Python Code:

```
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\problemdates.csv")
df=dfIn.copy(deep=True)
df["PROPER Output"] = df.iloc[:,0].astype(str).str.title()
```

REPLACE

Function: The **REPLACE** function – when fed into the ADDCOLUMN sheet formula – appends a new column of a specified name, containing a string with a specified substring replacing a specified index range of the strings contained in another chosen column.

Example:REPLACE(“OX_D928570”, 4,9,”CD”) = “OX_CD70”

Syntax:REPLACE(columnID/string,startIndex, endIndex, replacementString)

- ***columnID/string***. Required. The column which contains the strings that will be modified or the string that will be replaced and appended to every row.
- ***startIndex***. Required. The starting index of the substring that will be replaced
- ***endIndex***. Required. The ending index of the substring that will be replaced.
- ***replacementString***. Required. The string that replaces the substring specified by the indices.

ADDCOLUMN Syntax: ADDCOLUMN(columnName, REPLACEformula)

- ***columnName***. Required. The name of the column to be appended.
- ***REPLACEformula***. Required. The REPLACE formula specified by the user.

Video:

Python Code:

```
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\SimpleSale.csv")
df=dfIn.copy(deep=True)
df["REPLACE"] =
df.iloc[:,1].astype(str).str.slice_replace(start=4-1,stop=5-1,repl="CD")
```

RIGHT

Function: The **RIGHT** function – when fed into the ADDCOLUMN sheet formula – appends a new column with a specified name containing the portion to the right of a specified index of a string contained in another chosen column.

Syntax:RIGHT(columnID)

- ***columnID***. Required. The column which contains the strings to be sliced.

ADDCOLUMN Syntax: ADDCOLUMN(columnName, RIGHTformula)

- ***columnName***. Required. The name of the column to be appended.
- ***RIGHTformula***. Required. The RIGHT formula specified by the user.

Video:

Python Code:

```
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\Websites.csv")
df=dfIn.copy(deep=True)
df["RIGHT EXAMPLE"] = df.iloc[:,0].astype(str).str[-4:]
```

SUBSTITUTE

Function: The **SUBSTITUTE** function – when fed into the ADDCOLUMN sheet formula – appends a new column of a specified name, containing a string that has every instance of a specified substring replaced with a user specified substring.

Example: SUBSTITUTE("919-845-5425", "-", "") = "9198455425"

Syntax:SUBSTITUTE(columnID/string,substring, replacementString)

- ***columnID/string***. Required. The column which contains the strings that will be modified or the string that will be replaced and appended to every row.
- ***substring***. Required. The substring that is to be substituted.
- ***replacementString***. Required. The string that substitutes the substring specified by the indices.

ADDCOLUMN Syntax: ADDCOLUMN(columnName, SUBSTITUTEformula)

- ***columnName***. Required. The name of the column to be appended.
- ***SUBSTITUTEformula***. Required. The SUBSTITUTE formula specified by the user.

Video:

Python Code:

```
import pandas as pd

def SubstituteFunc( x, oldT, newT, instNum=0 ):
    # instNum = 0: Changes all instances, 1: Changes only 1st instance, 2: Changes 
only 2nd instance
    if ( instNum == 0 ) : return x.replace( oldT, newT )
    parts = x.split( oldT );
    if (len(parts) == 1 ) : return x;
    outX = parts[0]
    for i in range( len(parts)-1 ):
        outX += ( newT if (i+1==instNum) else oldT ) + parts[i+1]
    return outX

dfIn = pd.read_csv("C:\\Program 
Files\\Row64\\Row64_V1_3\\Data\\Example\\phoneDashes.csv")
df=dfIn.copy(deep=True)
df["Result"] = df.iloc[:,0].astype(str).astype(str).apply(lambda 
x:SubstituteFunc(x,"-","",0))
```

TRIM

Function: The **TRIM** function – when fed into the ADDCOLUMN sheet formula – appends a new column with a specified name containing the strings in another column that have had their heading and trailing whitespace stripped.

Syntax:TRIM(columnID)

- ***columnID***. Required. The column that will have the strings stripped.

ADDCOLUMN Syntax: ADDCOLUMN(columnName, TRIMformula)

- ***columnName***. Required. The name of the column to be appended.
- ***TRIMformula***. Required. The TRIM formula specified by the user.

Video:

Python Code:

```
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\spacingProblems.csv")
df=dfIn.copy(deep=True)
df["TRIM Result"] = df.iloc[:,0].astype(str).str.strip()
```

UPPER

Function: The **UPPER** function – when fed into the ADDCOLUMN sheet formula – appends a new column with a specified name containing the strings in another column casted to uppercase.

Syntax:UPPER(columnID)

- ***columnID***. Required. The column which contains the strings which will be cast to uppercase.

ADDCOLUMN Syntax: ADDCOLUMN(columnName, UPPERformula)

- ***columnName***. Required. The name of the column to be appended.
- ***UPPERformula***. Required. The UPPER formula specified by the user.

Video:

Python Code:

```
import pandas as pd

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\dogNames.csv")
df=dfIn.copy(deep=True)
df["UPPER Output"] = df.iloc[:,0].astype(str).str.upper()
```

VALUE

Function: The **VALUE** function – when fed into the ADDCOLUMN sheet formula – appends a new column with a specified name containing the strings in another column casted to floats.

Syntax:VALUE(columnID)

- ***columnID***. Required. The column which contains the strings which will be cast to floats.

ADDCOLUMN Syntax: ADDCOLUMN(columnName, VALUEformula)

- ***columnName***. Required. The name of the column to be appended.
- ***VALUEformula***. Required. The VALUE formula specified by the user.

Video:

Python Code:

```
import pandas as pd
import re

def StringToNum(x):
    if ( x == '' ) : return 0;
    p1 = re.compile(r"(\$?)(^\d*\.?\d+|\d{1,3}(,\d{3})*(\.\d+)?)")  # dollar or
decimal with optional commas
    if re.fullmatch( p1, x ) : return eval( x.replace(',','').replace('$','') )
    p2 = re.compile(r"(^\d*\.?\d+|\d{1,3}(,\d{3})*(\.\d+)?)(%?)")   # percentage
with optional commas
    if re.fullmatch( p2, x ) : return eval(  x.replace(',','').replace('%','/100')
)
    return float('NaN')

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\Vegetables.csv")
df=dfIn.copy(deep=True)
df["new"] = df.iloc[:,4].astype(str).apply(lambda x:StringToNum(x))
```

FINANCE DATA SCIENCE RECIPES - TBD

Date & Time Formulas

Extract Date Details

Function: The **READDATE** sheet formula takes in a date column specified from the user and appends columns with the following information: datetime equivalent, year, month, and day.

Syntax:READDATE(columnID)

- ***columnID***. Required. The column which contains the dates to be read

Video:

Python Code:

```
import pandas as pd

def ReadDate(inDf, inColI):
       cName = inDf.columns[inColI]
       dtCol = 'Date Time';
       inDf[dtCol] = pd.to_datetime(df[cName])
       inDf['Year'] = df[dtCol].dt.year
       inDf['Month'] = df[dtCol].dt.month
       inDf['Day'] = df[dtCol].dt.day
       return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\dateList.csv")
df=dfIn.copy(deep=True)
df=ReadDate(df,0)
```

Extract Date & Time Details

Function: The **READDATE** sheet formula takes in a date column specified from the user and appends columns with the following information: datetime equivalent, year, month, day, weekday, hour, and minute.

Syntax:READDATE(columnID, “time”)

- ***columnID***. Required. The column which contains the dates to be read

Video:

Python Code:

```
import pandas as pd

def ReadDateTime(inDf, inColI):
       cName = inDf.columns[inColI]
       dtCol = 'Date Time';
       inDf[dtCol] = pd.to_datetime(df[cName])
       inDf['Year'] = df[dtCol].dt.year
       inDf['Month'] = df[dtCol].dt.month
       inDf['Day'] = df[dtCol].dt.day
       inDf['Weekday'] = df[dtCol].dt.strftime("%A")
       inDf['Hour'] = df[dtCol].dt.hour
       inDf['Minute'] = df[dtCol].dt.minute
       return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\dateList.csv")
df=dfIn.copy(deep=True)
df=ReadDateTime(df,0)
```

Extract Quarter

Function: The **READDATE** sheet formula takes in a date column specified from the user and appends columns with datetime equivalent and quarter.

Syntax:READDATE(columnID, “quarter”)

- ***columnID***. Required. The column which contains the dates to be read

Video:

Python Code:

```
import pandas as pd

def ReadDateQuarter(inDf, inColI):
       cName = inDf.columns[inColI]
       dtCol = 'Date Time';
       inDf[dtCol] = pd.to_datetime(df[cName])
       inDf['Quarter'] = df[dtCol].dt.quarter
       return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\dateList.csv")
df=dfIn.copy(deep=True)
df=ReadDateQuarter(df,0)
```

Extract Quarter (With Fiscal Year)

Function: The **FISCALDATE** function takes in a date column and number representing the month in which the fiscal year ends specified by the user and appends columns with the following information: datetime equivalent, quarter, fiscal quarter, fiscal year, and fiscal year range.

Syntax:FISCALDATE(columnID, yearEndMonth)

- ***columnID***. Required. The column which contains the dates to be read
- ***yearEndMonth***. Required. The month (2,3,4…) in which the fiscal year ends.

Video:

Python Code:

```
import pandas as pd

def ReadFiscalDate(inDf, inColI,inMonthEndingQ4):
       cName = inDf.columns[inColI]
       qList =
['','Q-JAN','Q-FEB','Q-MAR','Q-APR','Q-MAY','Q-JUN','Q-JUL','Q-AUG','Q-SEP','Q-OCT'
,'Q-NOV','Q-DEC']
       inFiscalYear = qList[inMonthEndingQ4]
       dtCol = 'Date Time'
       inDf[dtCol] = pd.to_datetime(df[cName])
       inDf['Quarter'] = df[dtCol].dt.quarter
       inDf['Fiscal Quarter'] =
df[dtCol].dt.to_period(inFiscalYear).dt.strftime('Q%q')
       inDf['Fiscal Year'] = df[dtCol].dt.to_period(inFiscalYear).dt.qyear
       inDf['Fiscal Year Range'] =
df[dtCol].dt.to_period(inFiscalYear).dt.qyear.apply(lambda x: str(x-1) + "-" +
str(x))
       return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\dateList.csv")
df=dfIn.copy(deep=True)
df=ReadFiscalDate(df,0,3)
```

Date From Columns

Function: The **DATEFROMCOL** sheet formula takes in three columns containing day, month, and year data in addition to a specified format and concatenates the data to create a new date columns

Syntax:DATEFROMCOL(format, dayColumn, monthColumn, yearColumn)

- ***format***. Required. The date format that is to be used in the column which concatenates the individual columns. (e.g., "%Y-%m-%d")
- ***dayColumn***. Required. The column which contains the data for days.
- ***monthColumn***. Required. The column which contains the data for months.
- ***yearColumn***. Required. The column which contains the data for years.

Video:

Python Code:

```
import pandas as pd
import pendulum

def DateFromCol(inDf,inFormat,inDayI,inMonthI,inYearI):
        # https://www.w3schools.com/python/gloss_python_date_format_codes.asp
        dName = inDf.columns[inDayI]
        mName = inDf.columns[inMonthI]
        yName = inDf.columns[inYearI]
        inDf['Merged Date'] = None
        for index, row in inDf.iterrows():
                dStr = str(row[yName])+"-"+str(row[mName])+"-"+str(row[dName])
                dateObj = pendulum.parse(dStr, strict=False)
                inDf.loc[index, 'Merged Date']= dateObj.strftime(inFormat)
        return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\dateparts.csv")
df=dfIn.copy(deep=True)
df=DateFromCol(df,"%Y-%m-%d",0,1,3)
```

Format Date

Function: The **FORMATDATE** sheet formula takes in a column containing dates and reformats them according to user specification.

Syntax:FORMATDAT(columnID, format)

- ***columnID***. Required. The column which contains the dates to be formatted.
- ***format***. Required. The date format that is to be used in the new column. (e.g., "%Y-%m-%d")

Video:

Python Code:

```
import pandas as pd
import pendulum

def FormatDate(inDf, inColI,inFormat):
       # https://www.w3schools.com/python/gloss_python_date_format_codes.asp
       cName = inDf.columns[inColI]
       inDf[cName] = pd.to_datetime(inDf[cName])
       df[cName] = df[cName].dt.strftime(inFormat)
       return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\dashDates.csv")
df=dfIn.copy(deep=True)
df=FormatDate(df,0,"%Y-%m-%d")
```

Read Datetime with Timezone

Function: The **READDATE** sheet formula takes in a date column with time zone data and appends columns with the following information: Year, Month, Day, Hour, Minutes, and UTC offset.

Syntax:READDATE(columnID, “zone”)

- ***columnID***. Required. The column which contains the dates to be read.

Video:

Python Code:

```
import pandas as pd
import pendulum

def ParseDatesZone(inDf, inColI):
       cName = inDf.columns[inColI]
       newCols = ['Year','Month','Day','Hour','Minute','UTC Offset']
       inDf = inDf.assign(**dict.fromkeys(newCols, None))
       dtCol = 'Date Time'
       for index, row in inDf.iterrows():
              txt = row[cName]
              try:
                     pend = pendulum.parse(txt, strict=False)
              except ValueError as e:pend = None
              if pend != None:
                     row['Year','Month','Day','Hour','Minute'] =
pend.strftime("%Y-%m-%d-%H-%M").split('-')
                     row['UTC Offset'] = pend.offset_hours

       return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\timezone.csv")
df=dfIn.copy(deep=True)
df=ParseDatesZone(df,0)
```

Convert To TimeZone

Function: The **CONVERTZONE** sheet formula takes in a target time zone and a column time zone specific datetime data and appends the following columns: original datetime data, time zone offset, datetime equivalent of target time zone, target time zone date, target time zone time.

Syntax:CONVERTZONE(columnID, timeZone)

- ***columnID***. Required. The column which contains the dates to be read.
- ***timeZone***. Required. The time zone that the datetime data should be converted to. (e.g., “Asia/Tokyo”, “EST”)

Video:

Python Code:

```
import pandas as pd
import pendulum

def ConvertTimeZone(inDf, inColI, inTargetZone):
       # inTargetZone in this format:
https://en.wikipedia.org/wiki/List_of_tz_database_time_zones
       cName = inDf.columns[inColI]
       newCols = ['Source Time','TimeZone',inTargetZone, inTargetZone+" Date",
inTargetZone+" Time"]
       inDf = inDf.assign(**dict.fromkeys(newCols, None))
       dtCol = 'Date Time'
       for index, row in inDf.iterrows():
              txt = row[cName]
              try:
                     pend = pendulum.parse(txt, strict=False)
              except ValueError as e:pend = None
              if pend != None:
                     row['Source Time'] = pend.format('YYYY-MM-DD HH:mm:ss Z')
                     row['TimeZone'] = pend.timezone_name
                     targetP = pend.in_timezone(inTargetZone)
                     row[inTargetZone] = targetP.format('YYYY-MM-DD HH:mm:ss Z')
                     row[inTargetZone+" Date"] = targetP.format('YYYY-MM-DD')
                     row[inTargetZone+" Time"] = targetP.format('HH:mm:ss')
       return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\timezone.csv")
df=dfIn.copy(deep=True)
df=ConvertTimeZone(df,0,"HST")
```

View Date Range

Function: The **DATERANGE** sheet formula returns all rows of a dataframe for which date values in a given column lie in a specified range.

Syntax:DATERANGE(columnID, minimum, maximum)

- ***columnID***. Required. The column which contains the dates to be compared against the specified range.
- ***minimum***. Required. The minimum value of the desired date range.
- ***maximum***. Required. The maximum value of the desired date range.

Video:

Python Code:

```
import pandas as pd

def ViewDateRange(inDf, inColI, inStartDate, inEndDate):
       cName = inDf.columns[inColI]
       inDf['date detail'] = pd.to_datetime(df[cName])
       startD = pd.to_datetime(inStartDate)
       endD = pd.to_datetime(inEndDate)
       inDf = inDf[inDf['date detail'].between(startD, endD)]
       inDf=inDf.drop(['date detail'], axis=1)
       return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\saledates.csv")
df=dfIn.copy(deep=True)
df=ViewDateRange(df,2,"2019-10-20","2019-10-28")
```

Add Date Column

Function: The **GENDATES** sheet formula appends a column of dates increasing by a user specified frequency from a specified start date.

Syntax:GENDATES(startDate, frequency)

- ***startValue***. Required. The date from which the incrementing begins.
- ***frequency***. Required. The frequency at which the incrementing takes place. Options are (“d”, “B”, ‘W”, “MS”, “YS”) which are (daily, business days, weekly, month start, year start) respectively.

Video:

Python Code:

```
import pandas as pd

def GenDatesCol(inDf, inStartDate, inFreq):
       # options for inFreq:
https://pandas.pydata.org/docs/user_guide/timeseries.html#timeseries-offset-aliases
       inDf['date range'] = (pd.date_range(start=inStartDate,
periods=inDf.shape[0], freq=inFreq))
       return inDf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\trafficmini.csv")
df=dfIn.copy(deep=True)
df=GenDatesCol(df,"2019-10-20","MS")
```

Language Formulas

EXTRACT(A1,"|","URL")

Function: The **EXTRACT** sheet formula parses html code and returns a URL.

Syntax:EXTRACT(columnID)

- ***columnID***. Required. The column which contains the html data.

Video:

Python Code:

```
import itertools
import pandas as pd
import re

def ExtractUrls(idf, textI, delim):
       cName = idf.columns[textI]
       colList = []
       for txt in idf[cName].array:
              urls = re.findall('https?://(?:[-\w.]|(?:%[\da-fA-F]{2}))+', txt)
              urlStr = delim.join(urls);
              if urlStr == "":
                     colList.append("");
              else:
                     colList.append(urlStr);
       idf["Url"] = colList
       return idf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\GovRss.csv")
df=dfIn.copy(deep=True)
df = ExtractUrls(df,0,"|")
```

Sentiment Analysis (Positive or Negative)

Function: The **SENTIMENT** sheet formula is used to perform a sentiment analysis on a column of text data and append the sentiment scores of the text for each row. Sentiment scores are bounded between -1 and 1 which correspond to negative and positive sentiments, respectively.

Syntax: SENTIMENT(*filepath*)

- ***columnID***. Required. The column which contains the text data to be analyzed.

Video:
TBD

Python Code:

```
import pandas as pd

from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer

def Sentiment(idf, textI):
       cName = idf.columns[textI]
       colList = []
       sid_obj = SentimentIntensityAnalyzer()
       for txt in idf[cName].array:
              sentiment_dict = sid_obj.polarity_scores(txt)
              colList.append(sentiment_dict['compound']);
       idf["Sentiment"] = colList
       return idf

dfIn = pd.read_csv("C:\\Program
Files\\Row64\\Row64_V1_3\\Data\\Example\\Tweets.csv")
df=dfIn.copy(deep=True)
df=Sentiment(df,0)
```

Import and Export Functions - TBD