ML_3

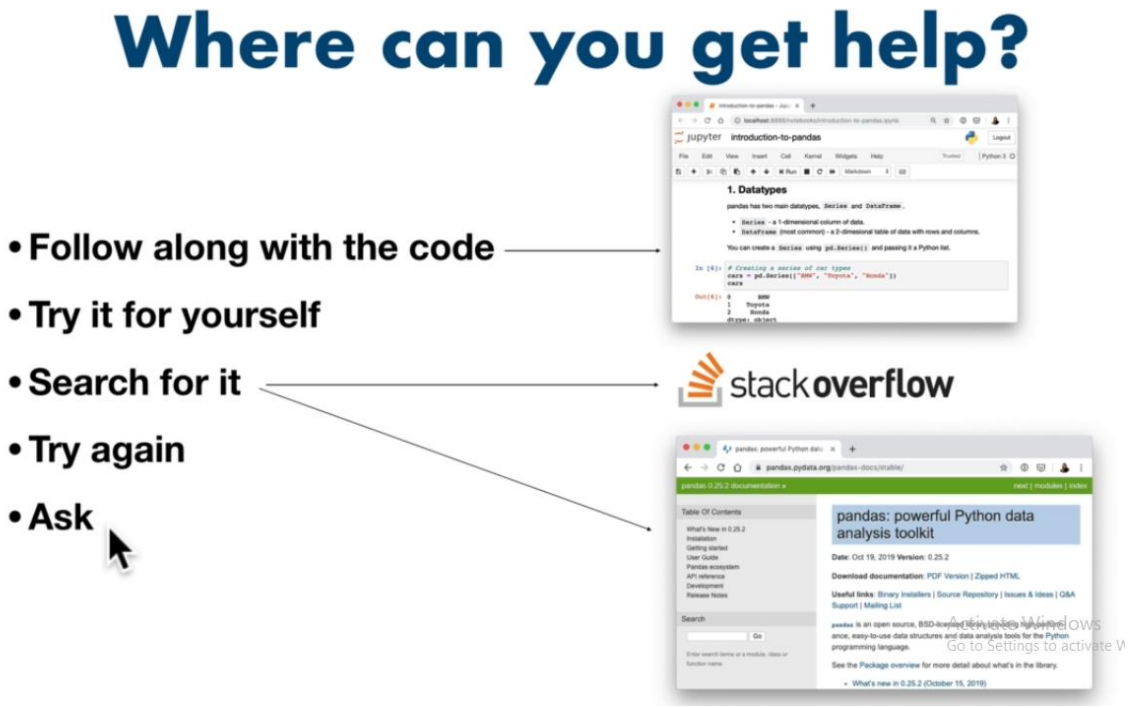
PANDAS

Muhammad Naqeeb | Machine Learning / pandas | August 6, 2021

Contents

What is pandas?

Pandas is data analysis library. Now, if you're doing machine learning or data science in Python, you're going to be using pandas and it's used to explore data, analyze data, manipulate data, get it ready for machine learning.



To begin with pandas the first step is to import pandas as pd

Two main datatypes

Series

Series is a **one-dimensional** labeled array capable of holding data of any type (integer, string, float, python objects, etc.). The axis labels are collectively called index.

```
series = pd.Series(["BMW","Toyota","Honda"])
```

```
color = pd.Series(["Blue","Red","Black"])
```

DataFrame

A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns. Dataframe is little bit different because it takes a Python dictionary.

```
car_data = pd.DataFrame({"Car Maker":series , "color":color})
```

2) DataFrame

```
In [7]: car_data = pd.DataFrame({"Car Maker":series , "color":color})

In [8]: car_data
```

Out[8]:

	Car Maker	color
0	BMW	Blue
1	Toyota	Red
2	Honda	Black

Now rather than creating a data frame from some series you're going to **import data.**

```
car_sales = pd.read_csv("7.1 car-sales.csv")
```

Now the beautiful thing about this is that because it's now in a panda's data frame we can take advantage of all the functions that pandas has to offer and manipulating viewing and changing this data.

import data

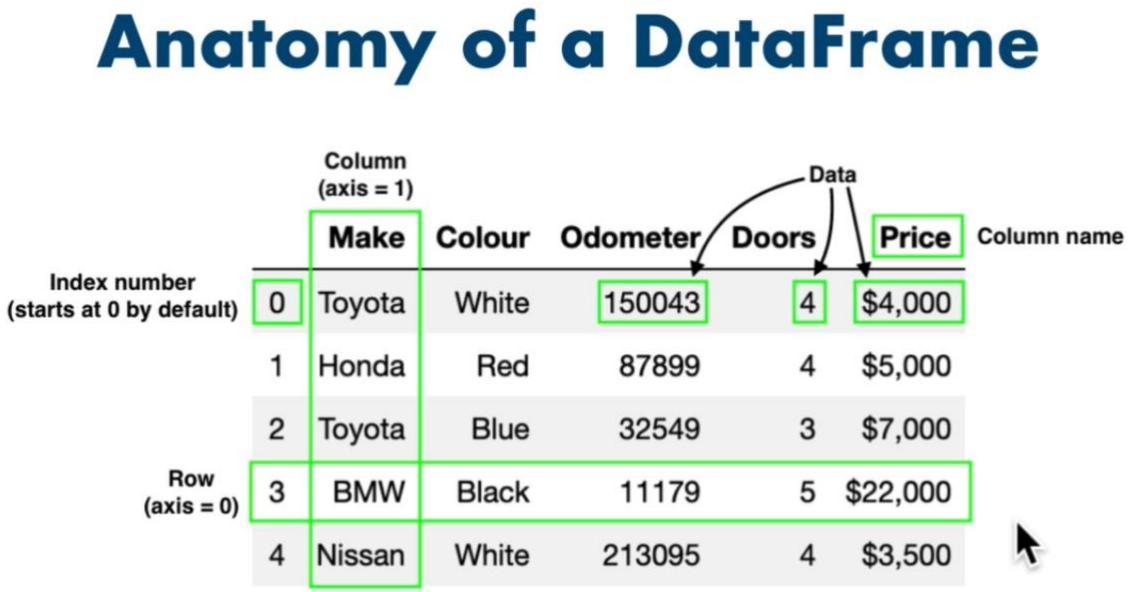
```
In [9]: car_sales = pd.read_csv("7.1 car-sales.csv")
```

```
In [10]: car_sales
```

Out[10]:

	Make	Colour	Odometer (KM)	Doors	Price
0	Toyota	White	150043	4	$4,000.00
1	Honda	Red	87899	4	$5,000.00
2	Toyota	Blue	32549	3	$7,000.00
3	BMW	Black	11179	5	$22,000.00
4	Nissan	White	213095	4	$3,500.00
5	Toyota	Green	99213	4	$4,500.00
6	Honda	Blue	45698	4	$7,500.00
7	Honda	Blue	54738	4	$7,000.00
8	Toyota	White	60000	4	$6,250.00
9	Nissan	White	31600	4	$9,700.00

Anatomy of a DataFrame



Export the dataFrame

```
car_sales.to_csv("Name-we-want-to-call-after-export.csv")
```

```
car_sales.to_csv("Name-we-want-to-call-after-export.csv" ,
index=False)
```

Describing Data

.dtypes

`DataFrame.dtypes` attribute to find out the data type (dtype) of each column in the given dataframe.

It returns a Series with the data type of each column.

```
car_sales.dtypes
```

```
In [16]: car_sales.dtypes

Out[16]: Make             object
         Colour           object
         Odometer (KM)     int64
         Doors             int64
         Price            object
         dtype: object
```

.columns

This is going to tell us our column names & it is going to return it as a list, we can store it into another variable so, we can perform some operations.

```
car_sales.columns
```

```
In [18]: car_sales.columns

Out[18]: Index(['Make', 'Colour', 'Odometer (KM)', 'Doors', 'Price'], d
         type='object')
```

.index

```
car_sales.index
```

```
In [19]: car_sales.index

Out[19]: RangeIndex(start=0, stop=10, step=1)
```

.describe()

describe() gives us some statistical information about our numeric columns. Describe() works on only numeric columns.

```
car_sales.describe()
```

```
In [20]: car_sales.describe()
```

Out[20]:

	Odometer (KM)	Doors
count	10.000000	10.000000
mean	78601.400000	4.000000
std	61983.471735	0.471405
min	11179.000000	3.000000
25%	35836.250000	4.000000
50%	57369.000000	4.000000
75%	96384.500000	4.000000
max	213095.000000	5.000000

.info()

Pandas dataframe.info() function is used to get a concise summary of the dataframe.

The information contains the number of columns, column labels, column data types, memory usage, range index, and the number of cells in each column

Note: the info() method actually prints the info. You do not use the print() method to print the info.

```
car_sales.info()
```

```
In [21]: car_sales.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 5 columns):
 #   Column          Non-Null Count  Dtype 
---  ------          --------------  ----- 
 0   Make            10 non-null     object
 1   Colour          10 non-null     object
 2   Odometer (KM)   10 non-null     int64 
 3   Doors           10 non-null     int64 
 4   Price           10 non-null     object
dtypes: int64(2), object(3)
memory usage: 528.0+ bytes
```

.mean()

mean we'll give you the average of your numerical columns

```
car_sales.mean()
```

```
In [22]: car_sales.mean()

<ipython-input-22-04f02239622f>:1: FutureWarning: Dropping of
nuisance columns in DataFrame reductions (with 'numeric_only=N
one') is deprecated; in a future version this will raise TypeE
rror.  Select only valid columns before calling the reduction.
  car_sales.mean()

Out[22]: Odometer (KM)    78601.4
         Doors                4.0
         dtype: float64
```

mean can be applied in individual series

```
In [23]: car_prices = pd.Series([3000,1202,1859,2589])
         car_prices.mean()

Out[23]: 2162.5
```

.sum()

This will sum up all of the different columns value.

```
car_sales.sum()
```

```
In [24]: car_sales.sum()

Out[24]: Make             ToyotaHondaToyotaBMWNissanToyotaHondaHondaToyo...
         Colour               WhiteRedBlueBlackWhiteGreenBlueBlueWhiteWhite
         Odometer (KM)                                                786014
         Doors                                                            40
         Price            $4,000.00$5,000.00$7,000.00$22,000.00$3,500.00...
         dtype: object
```

To select a single column we use dot & type in the name of the column in square brackets as a string.

```
In [25]: car_sales["Doors"].sum()

Out[25]: 40
```

Len

```
In [26]: len(car_sales)

Out[26]: 10
```

Viewing and selecting data

head()

This is going to return the first or the top five rows of your data frame.

```
car_sales.head()
```

```
In [27]: car_sales.head()
```

Out[27]:

	Make	Colour	Odometer (KM)	Doors	Price
0	Toyota	White	150043	4	$4,000.00
1	Honda	Red	87899	4	$5,000.00
2	Toyota	Blue	32549	3	$7,000.00
3	BMW	Black	11179	5	$22,000.00
4	Nissan	White	213095	4	$3,500.00

You can also specify the number of rows to be displayed.

```
car_sales.head(7)
```

.tail()

If you wanted the bottom of your data frame you can use dot tail and that will return bottom five rows.

```
car_sales.tail()
```

Manually Indexing

```
animals= pd.Series(["dog","Cat","panda","snake","Lion"],
index=[0,8,6,4,9])
```

.loc

Use DataFrame.loc attribute to access a particular cell in the given Dataframe using the index and column labels.

Loc refers to **index**

```
animals.loc[2]
```

```
In [34]: animals= pd.Series(["dog","Cat","panda","snake","Lion"], index=[0,2,5,7,2])
         animals.loc[2]

Out[34]: 2     Cat
         2    Lion
         dtype: object
```

```
In [35]: car_sales.loc[3]

Out[35]: Make                  BMW
         Colour              Black
         Odometer (KM)       11179
         Doors                   5
         Price            $22,000.00
         Name: 3, dtype: object
```

.iloc

iloc refers to **position**.

.iloc I purely integer-location based indexing for selection by position.

```
In [36]: animals.iloc[2]

Out[36]: 'panda'
```

```
In [37]: car_sales.iloc[3]

Out[37]: Make                  BMW
         Colour              Black
         Odometer (KM)       11179
         Doors                   5
         Price            $22,000.00
         Name: 3, dtype: object
```

slicing

With iloc and loc you can use slicing.

```
In [42]: animals.iloc[:5]

Out[42]: 0      dog
         2      Cat
         5    panda
         7    snake
         2     Lion
         dtype: object
```

```
In [41]: animals.loc[:5]

Out[41]: 0      dog
         2      Cat
         5    panda
         dtype: object
```

Select a column

Way to select a column is to type in its name in square brackets next to the name of the data frame.

```
car_sales["Make"]
```

```
car_sales.Make
```

- both work same

```
In [43]: car_sales["Make"]

Out[43]: 0    Toyota
         1     Honda
         2    Toyota
         3       BMW
         4    Nissan
         5    Toyota
         6     Honda
         7     Honda
         8    Toyota
         9    Nissan
         Name: Make, dtype: object
```

Filters

```
car_sales[car_sales["Make"] == "Toyota"]
```

```
In [45]: car_sales[car_sales["Make"] == "Toyota"]
```

Out[45]:

	Make	Colour	Odometer (KM)	Doors	Price
0	Toyota	White	150043	4	$4,000.00
2	Toyota	Blue	32549	3	$7,000.00
5	Toyota	Green	99213	4	$4,500.00
8	Toyota	White	60000	4	$6,250.00

```
car_sales[car_sales["Odometer (KM)"] > 100000]
```

```
In [47]: car_sales[car_sales["Odometer (KM)"] > 100000]
```

Out[47]:

	Make	Colour	Odometer (KM)	Doors	Price
0	Toyota	White	150043	4	$4,000.00
4	Nissan	White	213095	4	$3,500.00

crosstab()

The crosstab() function is used to compute a simple cross tabulation of two (or more) factors

By default computes a frequency table of the factors unless an array of values and an aggregation function are passed.

```
pd.crosstab(car_sales["Make"],car_sales["Doors"])
```

	Make	Colour	Odometer (KM)	Doors	Price
0	Toyota	White	150043	4	$4,000.00
1	Honda	Red	87899	4	$5,000.00
2	Toyota	Blue	32549	3	$7,000.00
3	BMW	Black	11179	5	$22,000.00
4	Nissan	White	213095	4	$3,500.00
5	Toyota	Green	99213	4	$4,500.00
6	Honda	Blue	45698	4	$7,500.00
7	Honda	Blue	54738	4	$7,000.00
8	Toyota	White	60000	4	$6,250.00
9	Nissan	White	31600	4	$9,700.00

```
In [48]: pd.crosstab(car_sales["Make"],car_sales["Doors"])
```

Out[48]:

Doors	3	4	5
Make			
BMW	0	0	1
Honda	0	3	0
Nissan	0	2	0
Toyota	1	3	0

Groupby()

The groupby() function is used to group DataFrame or Series using a mapper or by a Series of columns.

A groupby operation involves some combination of splitting the object, applying a function, and combining the results. This can be used to group large amounts of data and compute operations on these groups.

```
sales.groupby(["Make"]).mean()
```

```
In [51]: car_sales.groupby(["Make"]).mean()
```

Out[51]:

Make	Odometer (KM)	Doors
BMW	11179.000000	5.00
Honda	62778.333333	4.00
Nissan	122347.500000	4.00
Toyota	85451.250000	3.75

.plot()

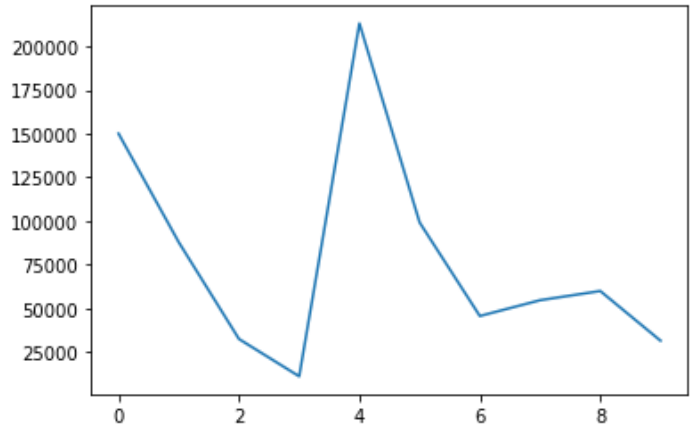
Pandas uses the plot() method to create diagrams.

We can use Pyplot, a submodule of the Matplotlib library to visualize the diagram on the screen.

```
car_sales["Odometer (KM)"].plot()
```

```
In [57]: car_sales["Odometer (KM)"].plot()
```

Out[57]: <AxesSubplot:>



If your plots don't show up make sure you've run these two lines of code, at the top of your notebook

```
%matplotlib inline
import matplotlib.pyplot as plt
```

.hist()

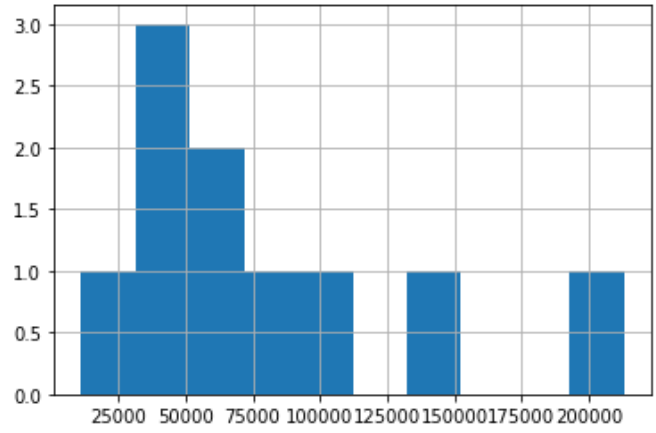
Histograms are the backbone to understanding distribution within your series of data. Pandas Histogram provides an easy way to plot a chart right from your data.

This function splits up the values into the numeric variables. Its main functionality is to make the Histogram of a given Data frame.

The distribution of data is represented by **Histogram**. When Function Pandas DataFrame.hist() is used, it automatically calls the function matplotlib.pyplot.hist() on each series in the DataFrame.

```
In [60]: car_sales["Odometer (KM)"].hist()
```

```
Out[60]: <AxesSubplot:>
```



Manipulating Data

.str.lower()

Make string to lower case.

```
car_sales["Make"].str.lower()
```

Now here's an important concept to remember it's that pandas requires if you want to change a column it requires reassigning that column.

```
car_sales["Make"] = car_sales["Make"].str.lower()
```

.fillna()

The fillna() method replaces the NULL values with a specified value.

The fillna() method returns a new DataFrame object unless the **inplace** parameter is set to **True**, in that case the fillna() method does the replacing in the original DataFrame instead.

Syntax

```
dataframe.fillna(value, method, axis, inplace, limit, downcast)
```

```
car_sales_missing["Odometer"].fillna(car_sales_missing["Odometer"
                    ].mean(),inplace=True )
```

inplace will automatically fill in the data we don't have to manually re-assign them. By default it is False.

.dropna()

Remove all rows wit NULL values from the DataFrame.

The **dropna**() method removes the rows that contains NULL values.

The **dropna**() method returns a new DataFrame object unless the inplace parameter is set to True

Syntax

```
dataframe.dropna(axis, how, thresh, subset, inplace)
```

```
car_sales_missing.dropna(inplace = True)
```

After doing all the dropping or apply the filter we can export the data to csv or in other file format.

Adding new columns

In [99]:
```
## column from series
seats_column = pd.Series([5,5,5,5,5,5])

## new column called seats
car_sales["Seats"] = seats_column
```

In [100]:
```
car_sales
```

Out[100]:

	Make	Colour	Odometer (KM)	Doors	Price	Seats
0	toyota	White	150043	4	400000	5.0
1	honda	Red	87899	4	500000	5.0
2	toyota	Blue	32549	3	700000	5.0
3	bmw	Black	11179	5	2200000	5.0
4	nissan	White	213095	4	350000	5.0
5	toyota	Green	99213	4	450000	5.0
6	honda	Blue	45698	4	750000	NaN
7	honda	Blue	54738	4	700000	NaN
8	toyota	White	60000	4	625000	NaN
9	nissan	White	31600	4	970000	NaN

Adding column from python list

```
In [101]: ## Adding column from python list
          fuel_economy = [1,5,3,8,9,6,7,8,9,2]
          car_sales["Fuel per 100KM"] = fuel_economy
```

```
In [102]: car_sales
```

Out[102]:

	Make	Colour	Odometer (KM)	Doors	Price	Seats	Fuel per 100KM
0	toyota	White	150043	4	400000	5.0	1
1	honda	Red	87899	4	500000	5.0	5
2	toyota	Blue	32549	3	700000	5.0	3
3	bmw	Black	11179	5	2200000	5.0	8
4	nissan	White	213095	4	350000	5.0	9
5	toyota	Green	99213	4	450000	5.0	6
6	honda	Blue	45698	4	750000	NaN	7
7	honda	Blue	54738	4	700000	NaN	8
8	toyota	White	60000	4	625000	NaN	9
9	nissan	White	31600	4	970000	NaN	2

Making columns from data of existing columns

```
In [103]: car_sales["Total fuel used"] = car_sales["Odometer (KM)"]/100 * car_sales["Fuel per 100KM"]
```

```
In [104]: car_sales
```

Out[104]:

	Make	Colour	Odometer (KM)	Doors	Price	Seats	Fuel per 100KM	Total fuel used
0	toyota	White	150043	4	400000	5.0	1	1500.43
1	honda	Red	87899	4	500000	5.0	5	4394.95
2	toyota	Blue	32549	3	700000	5.0	3	976.47
3	bmw	Black	11179	5	2200000	5.0	8	894.32
4	nissan	White	213095	4	350000	5.0	9	19178.55
5	toyota	Green	99213	4	450000	5.0	6	5952.78
6	honda	Blue	45698	4	750000	NaN	7	3198.86
7	honda	Blue	54738	4	700000	NaN	8	4379.04
8	toyota	White	60000	4	625000	NaN	9	5400.00
9	nissan	White	31600	4	970000	NaN	2	632.00

Creating a column from a single value

```
In [105]: ## Creating a column from a single value
          car_sales["Number of wheels"] = 4

In [106]: car_sales
```

Out[106]:

	Make	Colour	Odometer (KM)	Doors	Price	Seats	Fuel per 100KM	Total fuel used	Number of wheels
0	toyota	White	150043	4	400000	5.0	1	1500.43	4
1	honda	Red	87899	4	500000	5.0	5	4394.95	4
2	toyota	Blue	32549	3	700000	5.0	3	976.47	4
3	bmw	Black	11179	5	2200000	5.0	8	894.32	4
4	nissan	White	213095	4	350000	5.0	9	19178.55	4
5	toyota	Green	99213	4	450000	5.0	6	5952.78	4
6	honda	Blue	45698	4	750000	NaN	7	3198.86	4
7	honda	Blue	54738	4	700000	NaN	8	4379.04	4
8	toyota	White	60000	4	625000	NaN	9	5400.00	4
9	nissan	White	31600	4	970000	NaN	2	632.00	4

```
In [ ]:
```

Remove a column / drop()

```
car_sales.drop("Number of wheels", axis=1,inplace=True)
```

```
In [110]: car_sales.drop("Number of wheels", axis=1,inplace=True)

In [111]: car_sales
```

Out[111]:

	Make	Colour	Odometer (KM)	Doors	Price	Seats	Fuel per 100KM	Total fuel used
0	toyota	White	150043	4	400000	5.0	1	1500.43
1	honda	Red	87899	4	500000	5.0	5	4394.95
2	toyota	Blue	32549	3	700000	5.0	3	976.47
3	bmw	Black	11179	5	2200000	5.0	8	894.32
4	nissan	White	213095	4	350000	5.0	9	19178.55
5	toyota	Green	99213	4	450000	5.0	6	5952.78
6	honda	Blue	45698	4	750000	NaN	7	3198.86
7	honda	Blue	54738	4	700000	NaN	8	4379.04
8	toyota	White	60000	4	625000	NaN	9	5400.00
9	nissan	White	31600	4	970000	NaN	2	632.00

sample() / Shuffled Data

```
car_sales_shuffled = car_sales.sample(frac=1)
```

Frac = 0.5 for half of data

Another handy thing about the sample function is that say for example you had like a dataframe with two million rows.

But in practice you're going to be working on data sets with a lot more rows sometimes running functions

In pandes takes a long time on millions of different rows what you might want to do is practice on only 20 percent of the data. Only select 20 percent of data.

And now this number could be arbitrary right. If you had two million rows maybe you want to practice on 1 percent of the data. So that's still 20000 rows.

And so that will allow you to do lots of different experiments a lot quicker than doing it all on two million rows at one time.

Only select 20% of data

```
car_sales_shuffled.sample(frac=0.2)
```

```
In [73]: ## Only select 20% of data
         car_sales_shuffled.sample(frac=0.2)
```

Out[73]:

	Make	Colour	Odometer (KM)	Doors	Price	Seats	Fuel per 100KM	Total fuel used
6	honda	Blue	45698	4	750000	NaN	7	3198.86
7	honda	Blue	54738	4	700000	NaN	8	4379.04

.reset_index()

```
car_sales_shuffled.reset_index(drop = True , inplace=True)
```

```
In [75]: car_sales_shuffled.reset_index(drop = True , inplace=True)
```

```
In [76]: car_sales_shuffled
```

Out[76]:

	Make	Colour	Odometer (KM)	Doors	Price	Seats	Fuel per 100KM	Total fuel used
0	toyota	White	60000	4	625000	NaN	9	5400.00
1	nissan	White	31600	4	970000	NaN	2	632.00
2	honda	Blue	54738	4	700000	NaN	8	4379.04
3	bmw	Black	11179	5	2200000	5.0	8	894.32
4	toyota	Green	99213	4	450000	5.0	6	5952.78
5	toyota	Blue	32549	3	700000	5.0	3	976.47
6	honda	Blue	45698	4	750000	NaN	7	3198.86
7	honda	Red	87899	4	500000	5.0	5	4394.95
8	nissan	White	213095	4	350000	5.0	9	19178.55
9	toyota	White	150043	4	400000	5.0	1	1500.43

.apply()

Pandas.apply allow the users to pass a function and apply it on every single value of the Pandas series.

The apply() function is used to apply a function along an axis of the DataFrame.

Objects passed to the function are Series objects whose index is either the DataFrame's index (axis=0) or the DataFrame's columns (axis=1).

```
car_sales["Odometer (KM)"] = car_sales["Odometer
        (KM)"].apply(lambda x: x / 1.6)
```