PANDAS 101: DATAFRAME CHEATSHEET

Github Repo: [https://github.com/liviaellen/pandas-dataframe-101]

This notebook can be treated as pandas cheatsheet or a beginner-friendly guide to learn from basics.

"Avocado Prices" dataset is used in this notebook :)

In [1]:

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

Creating DataFrames

- From a list of dictionaries (constructed row by row)

In [2]:

```
list_of_dicts = [
     {"name": "Ginger", "breed": "Dachshund", "height_cm": 22,"weight_kg": 10, "date_of_
    {"name": "Scout", "breed": "Dalmatian", "height_cm": 59,"weight_kg": 25, "date_of_b:
]
```

```
new_dogs = pd.DataFrame(list_of_dicts)
new_dogs
```

Out[2]:

	name	breed	height_cm	weight_kg	date_of_birth
0	Ginger	Dachshund	22	10	2019-03-14
1	Scout	Dalmatian	59	25	2019-05-09

- From a dictionary of lists (constructed column by column)

In [3]:

```
dict_of_lists = {
     "name": ["Ginger", "Scout"],
     "breed": ["Dachshund", "Dalmatian"],
     "height_cm": [22, 59],
     "weight_kg": [10, 25],
     "date_of_birth": ["2019-03-14","2019-05-09"]   }
new_dogs = pd.DataFrame(dict_of_lists)
new_dogs
```

Out[3]:

	name	breed	height_cm	weight_kg	date_of_birth
0	Ginger	Dachshund	22	10	2019-03-14
1	Scout	Dalmatian	59	25	2019-05-09

Reading and writing CSVs

- CSV = comma-separated values
- Designed for DataFrame-like data
- Most database and spreadsheet programs can use them or create them

Read CSV and assign index

You can assign columns as index using "index_col" attribute.

Since I want to index Date there is another helpful function called "parse_date" which will parse the date in the rows such that we can perform more complex subsetting(eg monthly, weekly etc).

In [79]:

```
# read CSV from using pandas
avocado = pd.read_csv("https://github.com/liviaellen/pandas-dataframe-101/blob/0112e59ba
# print the first few rows of the dataframe
avocado.head()
```

Out[79]:

	Unnamed: 0	Date	AveragePrice	Total Volume	4046	4225	4770	Total Bags	Small Bags	Large Bags	XLarg Bag
0	0	2015-12-27	1.33	64236.62	1036.74	54454.85	48.16	8696.87	8603.62	93.25	0
1	1	2015-12-20	1.35	54876.98	674.28	44638.81	58.33	9505.56	9408.07	97.49	0
2	2	2015-12-13	0.93	118220.22	794.70	109149.67	130.50	8145.35	8042.21	103.14	0
3	3	2015-12-06	1.08	78992.15	1132.00	71976.41	72.58	5811.16	5677.40	133.76	0
4	4	2015-11-29	1.28	51039.60	941.48	43838.39	75.78	6183.95	5986.26	197.69	0

Remove index from dataframe .reset_index(drop)

To reset the index use this function

In [5]:

```
avocado = avocado.reset_index(drop=True)
avocado.head()
```

Out[5]:

	Unnamed: 0	Date	AveragePrice	Total Volume	4046	4225	4770	Total Bags	Small Bags	Large Bags	XLarg Bag
0	0	2015-12-27	1.33	64236.62	1036.74	54454.85	48.16	8696.87	8603.62	93.25	0
1	1	2015-12-20	1.35	54876.98	674.28	44638.81	58.33	9505.56	9408.07	97.49	0
2	2	2015-12-13	0.93	118220.22	794.70	109149.67	130.50	8145.35	8042.21	103.14	0
3	3	2015-12-06	1.08	78992.15	1132.00	71976.41	72.58	5811.16	5677.40	133.76	0
4	4	2015-11-29	1.28	51039.60	941.48	43838.39	75.78	6183.95	5986.26	197.69	0

To write a CSV file function dataframe.to_csv(FILE_NAME)

In [6]:

```
avocado.to_csv("test_write.csv")
```

Some useful pandas function

- **.head()** or **.head(x)** is used to get the first x rows of the DataFrame (x = 5 by default)

In [7]:

```
avocado.head()
```

Out[7]:

	Unnamed: 0	Date	AveragePrice	Total Volume	4046	4225	4770	Total Bags	Small Bags	Large Bags	XLarg Bag
0	0	2015-12-27	1.33	64236.62	1036.74	54454.85	48.16	8696.87	8603.62	93.25	0
1	1	2015-12-20	1.35	54876.98	674.28	44638.81	58.33	9505.56	9408.07	97.49	0
2	2	2015-12-13	0.93	118220.22	794.70	109149.67	130.50	8145.35	8042.21	103.14	0
3	3	2015-12-06	1.08	78992.15	1132.00	71976.41	72.58	5811.16	5677.40	133.76	0
4	4	2015-11-29	1.28	51039.60	941.48	43838.39	75.78	6183.95	5986.26	197.69	0

- **.sample()** is used to get the random x sample rows of the DataFrame (x = 1 by default)

In [8]:

```
avocado.sample(5)
```

Out[8]:

	Unnamed: 0	Date	AveragePrice	Total Volume	4046	4225	4770	Total Bags	Small Bags	
13509	16	2016-09-04	1.58	13039.44	1.00	231.39	0.00	12807.05	12807.05	
17180	1	2017-12-24	1.57	90080.71	1409.87	16980.39	96.76	71593.69	63498.43	8
8122	15	2017-09-17	1.66	65219.18	16477.84	28242.32	34.47	20464.55	12775.39	7
14725	36	2016-04-17	1.36	37885.90	2655.44	25121.02	177.35	9932.09	5036.29	4
419	3	2015-12-06	1.14	664020.49	53173.18	455048.11	92888.37	62910.83	62473.12	

- **.tail()** or **.tail(x)** is used to get the last x rows of the DataFrame (x = 5 by default)

In [9]:

```
avocado.tail(10)
```

Out[9]:

	Unnamed: 0	Date	AveragePrice	Total Volume	4046	4225	4770	Total Bags	Small Bags	Large Bags
18239	2	2018-03-11	1.56	22128.42	2162.67	3194.25	8.93	16762.57	16510.32	252.25
18240	3	2018-03-04	1.54	17393.30	1832.24	1905.57	0.00	13655.49	13401.93	253.56
18241	4	2018-02-25	1.57	18421.24	1974.26	2482.65	0.00	13964.33	13698.27	266.06
18242	5	2018-02-18	1.56	17597.12	1892.05	1928.36	0.00	13776.71	13553.53	223.18
18243	6	2018-02-11	1.57	15986.17	1924.28	1368.32	0.00	12693.57	12437.35	256.22
18244	7	2018-02-04	1.63	17074.83	2046.96	1529.20	0.00	13498.67	13066.82	431.85
18245	8	2018-01-28	1.71	13888.04	1191.70	3431.50	0.00	9264.84	8940.04	324.80
18246	9	2018-01-21	1.87	13766.76	1191.92	2452.79	727.94	9394.11	9351.80	42.31
18247	10	2018-01-14	1.93	16205.22	1527.63	2981.04	727.01	10969.54	10919.54	50.00
18248	11	2018-01-07	1.62	17489.58	2894.77	2356.13	224.53	12014.15	11988.14	26.01

- **.info()** is used to get a concise summary of the DataFrame

In [10]:

```
avocado.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 18249 entries, 0 to 18248
Data columns (total 14 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   Unnamed: 0    18249 non-null  int64  
 1   Date          18249 non-null  object 
 2   AveragePrice  18249 non-null  float64
 3   Total Volume  18249 non-null  float64
 4   4046          18249 non-null  float64
 5   4225          18249 non-null  float64
 6   4770          18249 non-null  float64
 7   Total Bags    18249 non-null  float64
 8   Small Bags    18249 non-null  float64
 9   Large Bags    18249 non-null  float64
 10  XLarge Bags   18249 non-null  float64
 11  type          18249 non-null  object 
 12  year          18249 non-null  int64  
 13  region        18249 non-null  object 
dtypes: float64(9), int64(2), object(3)
memory usage: 1.9+ MB
```

- **.shape** is used to get the dimensions of the DataFrame

In [11]:

```
print(avocado.shape)
```

```
(18249, 14)
```

- **.describe()** is used to view some basic statistical details like percentile, mean, std etc. of a DataFrame

In [12]:
```
avocado.describe()
```

Out[12]:

	Unnamed: 0	AveragePrice	Total Volume	4046	4225	4770	Total
count	18249.000000	18249.000000	1.824900e+04	1.824900e+04	1.824900e+04	1.824900e+04	1.824900
mean	24.232232	1.405978	8.506440e+05	2.930084e+05	2.951546e+05	2.283974e+04	2.396392
std	15.481045	0.402677	3.453545e+06	1.264989e+06	1.204120e+06	1.074641e+05	9.862424
min	0.000000	0.440000	8.456000e+01	0.000000e+00	0.000000e+00	0.000000e+00	0.000000
25%	10.000000	1.100000	1.083858e+04	8.540700e+02	3.008780e+03	0.000000e+00	5.088640
50%	24.000000	1.370000	1.073768e+05	8.645300e+03	2.906102e+04	1.849900e+02	3.974383
75%	38.000000	1.660000	4.329623e+05	1.110202e+05	1.502069e+05	6.243420e+03	1.107834
max	52.000000	3.250000	6.250565e+07	2.274362e+07	2.047057e+07	2.546439e+06	1.937313

- **.values** this attribute return a Numpy representation of the given DataFrame

In [13]:
```
avocado.values
```

Out[13]:
```
array([[0, '2015-12-27', 1.33, ..., 'conventional', 2015, 'Albany'],
       [1, '2015-12-20', 1.35, ..., 'conventional', 2015, 'Albany'],
       [2, '2015-12-13', 0.93, ..., 'conventional', 2015, 'Albany'],
       ...,
       [9, '2018-01-21', 1.87, ..., 'organic', 2018, 'WestTexNewMexico'],
       [10, '2018-01-14', 1.93, ..., 'organic', 2018, 'WestTexNewMexico'],
       [11, '2018-01-07', 1.62, ..., 'organic', 2018, 'WestTexNewMexico']],
      dtype=object)
```

- **.columns** this attribute return a Numpy representation of columns in the DataFrame

In [14]:
```
print(avocado.columns)
```

```
Index(['Unnamed: 0', 'Date', 'AveragePrice', 'Total Volume', '4046', '4225',
       '4770', 'Total Bags', 'Small Bags', 'Large Bags', 'XLarge Bags', 'type',
       'year', 'region'],
      dtype='object')
```

Appending & Concatenating Series

append(): Series & DataFrame method

- Invocation:
- s1.append(s2)
- Stacks rows of s2 below s1

concat(): pandas module function

- Invocation:
- pd.concat([s1, s2, s3])
- Can stack row-wise or column-wise

In [15]:
```
even = pd.Series([2,4,6,8,10])
odd = pd.Series([1,3,5,7,9])

res = even.append(odd)
res
```

Out[15]:
```
0     2
1     4
2     6
3     8
4    10
0     1
1     3
2     5
3     7
4     9
dtype: int64
```

Observe index got messed up

You can use .reset_index(drop=True) to fix it

Note: if drop = False then previous index will be added as a column

In [16]:
```
res.reset_index(drop=True)
```

Out[16]:
```
0     2
1     4
2     6
3     8
4    10
5     1
6     3
7     5
8     7
9     9
dtype: int64
```

In [17]:
```
res.reset_index(drop=False)
```

Out[17]:

	index	0
0	0	2
1	1	4
2	2	6
3	3	8
4	4	10
5	0	1
6	1	3
7	2	5
8	3	7
9	4	9

Sorting

syntax:

> DataFrame.sort_values(by, axis=0, ascending=True, inplace=False, kind='quicksort', na_position='last')

- by: Single/List of column names to sort Data Frame by.
- axis: 0 or 'index' for rows and 1 or 'columns' for Column.
- ascending: Boolean value which sorts Data frame in ascending order if True.
- inplace: Boolean value. Makes the changes in passed data frame itself if True.
- kind: String which can have three inputs('quicksort', 'mergesort' or 'heapsort') of algorithm used to sort data frame.
- na_position: Takes two string input 'last' or 'first' to set position of Null values. Default is 'last'.

In [18]:

```
# sort values based on "AveragePrice" (ascending) and "year" (descending)
avocado.sort_values(["AveragePrice", "year"], ascending=[True, False])
```

Out[18]:

	Unnamed: 0	Date	AveragePrice	Total Volume	4046	4225	4770	Total Bags	Sma Bag
15261	43	2017-03-05	0.44	64057.04	223.84	4748.88	0.00	59084.32	638.6
7412	47	2017-02-05	0.46	2200550.27	1200632.86	531226.65	18324.93	450365.83	113752.1
15473	43	2017-03-05	0.48	50890.73	717.57	4138.84	0.00	46034.32	1385.0
15262	44	2017-02-26	0.49	44024.03	252.79	4472.68	0.00	39298.56	600.0
1716	0	2015-12-27	0.49	1137707.43	738314.80	286858.37	11642.46	100891.80	70749.0
...	...	...	...	...	...	...	...	...	
16720	18	2017-08-27	3.04	12656.32	419.06	4851.90	145.09	7240.27	6960.9
16055	42	2017-03-12	3.05	2068.26	1043.83	77.36	0.00	947.07	926.6
14124	7	2016-11-06	3.12	19043.80	5898.49	10039.34	0.00	3105.97	3079.3
17428	37	2017-04-16	3.17	3018.56	1255.55	82.31	0.00	1680.70	1542.2
14125	8	2016-10-30	3.25	16700.94	2325.93	11142.85	0.00	3232.16	3232.1

18249 rows × 14 columns

Sorting by index

use df.sort_index(ascending=True/False)

Subsetting

Subsetting is used to get a slice of the original dataframe

In [19]:

```
# Subsetting columns
avocado["AveragePrice"]
```

Out[19]:
```
0        1.33
1        1.35
2        0.93
3        1.08
4        1.28
         ...
18244    1.63
18245    1.71
18246    1.87
18247    1.93
18248    1.62
Name: AveragePrice, Length: 18249, dtype: float64
```

Subsetting multiple columns

In [20]:
```
# Subsetting multiple columns
avocado["AveragePrice"]
```

Out[20]:
```
0        1.33
1        1.35
2        0.93
3        1.08
4        1.28
         ...
18244    1.63
18245    1.71
18246    1.87
18247    1.93
18248    1.62
Name: AveragePrice, Length: 18249, dtype: float64
```

In [21]:
```
avocado.loc[:, ["AveragePrice","Date"]]
```

Out[21]:

	AveragePrice	Date
0	1.33	2015-12-27
1	1.35	2015-12-20
2	0.93	2015-12-13
3	1.08	2015-12-06
4	1.28	2015-11-29
...	...	...
18244	1.63	2018-02-04
18245	1.71	2018-01-28
18246	1.87	2018-01-21
18247	1.93	2018-01-14
18248	1.62	2018-01-07

18249 rows × 2 columns

Subsetting rows

In [22]:
```
# Subsetting rows
avocado["AveragePrice"]<1
```

Out[22]:
```
0        False
1        False
2         True
3        False
4        False
         ...
18244    False
18245    False
18246    False
18247    False
18248    False
Name: AveragePrice, Length: 18249, dtype: bool
```

and then using it for subsetting the original dataframe

In [23]:
```python
# This will print only the rows with price < 1
avocado[avocado["AveragePrice"]<1]
```

Out[23]:

	Unnamed: 0	Date	AveragePrice	Total Volume	4046	4225	4770	Total Bags	Small Bags	
2	2	2015-12-13	0.93	118220.22	794.70	109149.67	130.50	8145.35	8042.21	
6	6	2015-11-15	0.99	83453.76	1368.92	73672.72	93.26	8318.86	8196.81	
7	7	2015-11-08	0.98	109428.33	703.75	101815.36	80.00	6829.22	6266.85	
13	13	2015-09-27	0.99	106803.39	1204.88	99409.21	154.84	6034.46	5888.87	
43	43	2015-03-01	0.99	55595.74	629.46	45633.34	181.49	9151.45	8986.06	
...	...	...	...	...	...	...	...	...	...	
17169	43	2017-03-05	0.99	155011.12	35367.23	5175.81	5.91	114462.17	95379.07	1
17170	44	2017-02-26	0.99	171145.00	34520.03	6936.39	0.00	129688.58	117252.31	1
17536	39	2017-04-02	0.98	402676.23	34093.33	58330.53	207.85	310044.52	155701.41	15
17537	40	2017-03-26	0.90	456645.91	36169.35	51398.72	139.55	368938.29	152159.53	21
17540	43	2017-03-05	0.99	367519.17	61166.48	55123.99	126.80	251101.90	112844.19	13

2796 rows × 14 columns

In [24]:
```python
avocado.query("AveragePrice < 1")
```

Out[24]:

	Unnamed: 0	Date	AveragePrice	Total Volume	4046	4225	4770	Total Bags	Small Bags	
2	2	2015-12-13	0.93	118220.22	794.70	109149.67	130.50	8145.35	8042.21	
6	6	2015-11-15	0.99	83453.76	1368.92	73672.72	93.26	8318.86	8196.81	
7	7	2015-11-08	0.98	109428.33	703.75	101815.36	80.00	6829.22	6266.85	
13	13	2015-09-27	0.99	106803.39	1204.88	99409.21	154.84	6034.46	5888.87	
43	43	2015-03-01	0.99	55595.74	629.46	45633.34	181.49	9151.45	8986.06	
...	...	...	...	...	...	...	...	...	...	
17169	43	2017-03-05	0.99	155011.12	35367.23	5175.81	5.91	114462.17	95379.07	1
17170	44	2017-02-26	0.99	171145.00	34520.03	6936.39	0.00	129688.58	117252.31	1
17536	39	2017-04-02	0.98	402676.23	34093.33	58330.53	207.85	310044.52	155701.41	15
17537	40	2017-03-26	0.90	456645.91	36169.35	51398.72	139.55	368938.29	152159.53	21
17540	43	2017-03-05	0.99	367519.17	61166.48	55123.99	126.80	251101.90	112844.19	13

2796 rows × 14 columns

Subsetting based on text data

In [25]:

```
# it will print all the rows with "type" = "organic"
avocado[avocado["type"]=="organic"]
```

Out[25]:

	Unnamed: 0	Date	AveragePrice	Total Volume	4046	4225	4770	Total Bags	Small Bags	Large Bags
9126	0	2015-12-27	1.83	989.55	8.16	88.59	0.00	892.80	892.80	0.00
9127	1	2015-12-20	1.89	1163.03	30.24	172.14	0.00	960.65	960.65	0.00
9128	2	2015-12-13	1.85	995.96	10.44	178.70	0.00	806.82	806.82	0.00
9129	3	2015-12-06	1.84	1158.42	90.29	104.18	0.00	963.95	948.52	15.43
9130	4	2015-11-29	1.94	831.69	0.00	94.73	0.00	736.96	736.96	0.00
...	...	...	...	...	...	...	...	...	...	...
18244	7	2018-02-04	1.63	17074.83	2046.96	1529.20	0.00	13498.67	13066.82	431.85
18245	8	2018-01-28	1.71	13888.04	1191.70	3431.50	0.00	9264.84	8940.04	324.80
18246	9	2018-01-21	1.87	13766.76	1191.92	2452.79	727.94	9394.11	9351.80	42.31
18247	10	2018-01-14	1.93	16205.22	1527.63	2981.04	727.01	10969.54	10919.54	50.00
18248	11	2018-01-07	1.62	17489.58	2894.77	2356.13	224.53	12014.15	11988.14	26.01

9123 rows × 14 columns

In [26]:

```
#df['column_name'] can be used to select, create, and update
avocado["type"]
```

Out[26]:

```
0         conventional
1         conventional
2         conventional
3         conventional
4         conventional
              ...
18244          organic
18245          organic
18246          organic
18247          organic
18248          organic
Name: type, Length: 18249, dtype: object
```

In [27]:

```
# df.column_name can only be used to select element
avocado.type
```

Out[27]:

```
0        conventional
1        conventional
2        conventional
3        conventional
4        conventional
             ...
18244         organic
18245         organic
18246         organic
18247         organic
18248         organic
Name: type, Length: 18249, dtype: object
```

In [28]:

```
# example , #df['column_name'] can be used to create a new column
avocado["new_col"]=0
avocado
```

Out[28]:

	Unnamed: 0	Date	AveragePrice	Total Volume	4046	4225	4770	Total Bags	Small Bags	Large Bags
0	0	2015-12-27	1.33	64236.62	1036.74	54454.85	48.16	8696.87	8603.62	93.25
1	1	2015-12-20	1.35	54876.98	674.28	44638.81	58.33	9505.56	9408.07	97.49
2	2	2015-12-13	0.93	118220.22	794.70	109149.67	130.50	8145.35	8042.21	103.14
3	3	2015-12-06	1.08	78992.15	1132.00	71976.41	72.58	5811.16	5677.40	133.76
4	4	2015-11-29	1.28	51039.60	941.48	43838.39	75.78	6183.95	5986.26	197.69
...	...	...	...	...	...	...	...	...	...	...
18244	7	2018-02-04	1.63	17074.83	2046.96	1529.20	0.00	13498.67	13066.82	431.85
18245	8	2018-01-28	1.71	13888.04	1191.70	3431.50	0.00	9264.84	8940.04	324.80
18246	9	2018-01-21	1.87	13766.76	1191.92	2452.79	727.94	9394.11	9351.80	42.31
18247	10	2018-01-14	1.93	16205.22	1527.63	2981.04	727.01	10969.54	10919.54	50.00
18248	11	2018-01-07	1.62	17489.58	2894.77	2356.13	224.53	12014.15	11988.14	26.01

18249 rows × 15 columns

In [29]:

```
# example , #df.column_name cannot be used to create a new column, this is attribute cre
avocado.new_colb=0
avocado
```

Out[29]:

	Unnamed: 0	Date	AveragePrice	Total Volume	4046	4225	4770	Total Bags	Small Bags	Large Bag
0	0	2015-12-27	1.33	64236.62	1036.74	54454.85	48.16	8696.87	8603.62	93.2
1	1	2015-12-20	1.35	54876.98	674.28	44638.81	58.33	9505.56	9408.07	97.4
2	2	2015-12-13	0.93	118220.22	794.70	109149.67	130.50	8145.35	8042.21	103.1
3	3	2015-12-06	1.08	78992.15	1132.00	71976.41	72.58	5811.16	5677.40	133.7
4	4	2015-11-29	1.28	51039.60	941.48	43838.39	75.78	6183.95	5986.26	197.6
...	...	...	...	...	...	...	...	...	...	..
18244	7	2018-02-04	1.63	17074.83	2046.96	1529.20	0.00	13498.67	13066.82	431.8
18245	8	2018-01-28	1.71	13888.04	1191.70	3431.50	0.00	9264.84	8940.04	324.8
18246	9	2018-01-21	1.87	13766.76	1191.92	2452.79	727.94	9394.11	9351.80	42.3
18247	10	2018-01-14	1.93	16205.22	1527.63	2981.04	727.01	10969.54	10919.54	50.0
18248	11	2018-01-07	1.62	17489.58	2894.77	2356.13	224.53	12014.15	11988.14	26.0

18249 rows × 15 columns

Subsetting based on dates

In [30]:

```python
# it will print all the rows with "Date" <= 2015-02-04
avocado[avocado["Date"]<="2015-02-04"]
```

Out[30]:

	Unnamed: 0	Date	AveragePrice	Total Volume	4046	4225	4770	Total Bags	Small Bags	Large Bags
47	47	2015-02-01	0.99	70873.60	1353.90	60017.20	179.32	9323.18	9170.82	152.36
48	48	2015-01-25	1.06	45147.50	941.38	33196.16	164.14	10845.82	10103.35	742.47
49	49	2015-01-18	1.17	44511.28	914.14	31540.32	135.77	11921.05	11651.09	269.96
50	50	2015-01-11	1.24	41195.08	1002.85	31640.34	127.12	8424.77	8036.04	388.73
51	51	2015-01-04	1.22	40873.28	2819.50	28287.42	49.90	9716.46	9186.93	529.53
...	...	...	...	...	...	...	...	...	...	...
11928	46	2015-02-01	1.77	7210.19	1634.42	3012.44	0.00	2563.33	2563.33	0.00
11929	47	2015-01-25	1.63	7324.06	1934.46	3032.72	0.00	2356.88	2320.00	36.88
11930	48	2015-01-18	1.71	5508.20	1793.64	2078.72	0.00	1635.84	1620.00	15.84
11931	49	2015-01-11	1.69	6861.73	1822.28	2377.54	0.00	2661.91	2656.66	5.25
11932	50	2015-01-04	1.64	6182.81	1561.30	2958.17	0.00	1663.34	1663.34	0.00

540 rows × 15 columns

Subsetting based on multiple conditions

You can use the logical operators to define a complex condition

- "&" and
- "|" or
- "~" not

> **SEPERATE EACH CONDITION WITH PARENTHESES TO AVOID ERRORS**

In [31]:

```
# it will print all the rows with "Date" before 2015-02-04 and "type" == "organic"
# make sure you have the paranthesis
avocado[(avocado["Date"]<"2015-02-04") & (avocado["type"]=="organic")]
```

Out[31]:

	Unnamed: 0	Date	AveragePrice	Total Volume	4046	4225	4770	Total Bags	Small Bags	Large Bags	XLarge Bags
9173	47	2015-02-01	1.83	1228.51	33.12	99.36	0.0	1096.03	1096.03	0.00	0.0
9174	48	2015-01-25	1.89	1115.89	14.87	148.72	0.0	952.30	952.30	0.00	0.0
9175	49	2015-01-18	1.93	1118.47	8.02	178.78	0.0	931.67	931.67	0.00	0.0
9176	50	2015-01-11	1.77	1182.56	39.00	305.12	0.0	838.44	838.44	0.00	0.0
9177	51	2015-01-04	1.79	1373.95	57.42	153.88	0.0	1162.65	1162.65	0.00	0.0
...	...	...	...	...	...	...	...	...	...	...	...
11928	46	2015-02-01	1.77	7210.19	1634.42	3012.44	0.0	2563.33	2563.33	0.00	0.0
11929	47	2015-01-25	1.63	7324.06	1934.46	3032.72	0.0	2356.88	2320.00	36.88	0.0
11930	48	2015-01-18	1.71	5508.20	1793.64	2078.72	0.0	1635.84	1620.00	15.84	0.0
11931	49	2015-01-11	1.69	6861.73	1822.28	2377.54	0.0	2661.91	2656.66	5.25	0.0
11932	50	2015-01-04	1.64	6182.81	1561.30	2958.17	0.0	1663.34	1663.34	0.00	0.0

270 rows × 15 columns

Subsetting using .isin()

isin() method helps in selecting rows with having a particular(or Multiple) value in a particular column

> Syntax: DataFrame.isin(values)
>
> Parameters: values: iterable, Series, List, Tuple, DataFrame or dictionary to check in the caller Series/Data Frame.
>
> Return Type: DataFrame of Boolean of Dimension.

In [32]:

```
# subset the avocado in the region Boston or SanDiego
regionFilter = avocado["region"].isin(["Boston", "SanDiego"])
avocado[regionFilter]
```

Out[32]:

	Unnamed: 0	Date	AveragePrice	Total Volume	4046	4225	4770	Total Bags	Small Bags	L
208	0	2015-12-27	1.13	450816.39	3886.27	346964.70	13952.56	86012.86	85913.60	9
209	1	2015-12-20	1.07	489802.88	4912.37	390100.99	5887.72	88901.80	88768.47	13
210	2	2015-12-13	1.01	549945.76	4641.02	455362.38	219.40	89722.96	89523.38	19
211	3	2015-12-06	1.02	488679.31	5126.32	407520.22	142.99	75889.78	75666.22	22
212	4	2015-11-29	1.19	350559.81	3609.25	272719.08	105.86	74125.62	73864.52	2
...	...	...	...	...	...	...	...	...	...	
18100	7	2018-02-04	1.81	17454.74	1158.41	7388.27	0.00	8908.06	8908.06	
18101	8	2018-01-28	1.91	17579.47	1145.64	8284.41	0.00	8149.42	8149.42	
18102	9	2018-01-21	1.95	18676.37	1088.49	9282.37	0.00	8305.51	8305.51	
18103	10	2018-01-14	1.81	21770.02	3285.98	14338.52	0.00	4145.52	4145.52	
18104	11	2018-01-07	2.06	16746.82	5150.82	9366.31	0.00	2229.69	2229.69	

676 rows × 15 columns

Multiple parameter Filtering

Use logical operators to combine different filters

In [33]:

```
# subset the avocado in the region Boston or SanDiego in the year 2016 or 2017
regionFilter = avocado["region"].isin(["Boston", "SanDiego"])
yearFilter = avocado["year"].isin(["2016", "2017"])
avocado[regionFilter & yearFilter]
```

Out[33]:

	Unnamed: 0	Date	AveragePrice	Total Volume	4046	4225	4770	Total Bags	Small Bags	l
3016	0	2016-12-25	1.28	447600.75	4349.63	346516.32	4183.69	92551.11	91481.59	10
3017	1	2016-12-18	1.09	579577.33	6123.84	488107.01	7765.43	77581.05	76135.49	14
3018	2	2016-12-11	1.22	510800.58	3711.20	409645.98	5052.84	92390.56	90449.44	16
3019	3	2016-12-04	1.26	473428.36	4371.95	393748.18	3449.16	71859.07	71377.77	3
3020	4	2016-11-27	1.45	391257.01	4243.20	317090.39	3069.37	66854.05	66399.33	
...	...	...	...	...	...	...	...	...	...	
16962	48	2017-01-29	1.21	18191.46	1477.75	8949.53	4.86	7759.32	3304.61	44
16963	49	2017-01-22	1.73	10842.77	2019.23	6869.87	0.00	1953.67	626.78	13
16964	50	2017-01-15	1.82	11578.42	2529.20	7637.66	0.00	1411.56	993.41	4
16965	51	2017-01-08	1.52	16775.97	2363.28	9429.06	0.00	4983.63	3266.31	17
16966	52	2017-01-01	1.45	15752.25	1385.18	8618.28	0.00	5748.79	957.31	47

420 rows × 15 columns

Detecting missing values .isna()

.isna() is a method used to find is there exist any NaN values in the DataFrame

It will give a True bool value if a cell has a NaN value

In [34]:

```
avocado.isna()
```

Out[34]:

	Unnamed: 0	Date	AveragePrice	Total Volume	4046	4225	4770	Total Bags	Small Bags	Large Bags	XLarge Bags	type	y
0	False	False	False	False	False	False	False	False	False	False	False	False	F
1	False	False	False	False	False	False	False	False	False	False	False	False	F
2	False	False	False	False	False	False	False	False	False	False	False	False	F
3	False	False	False	False	False	False	False	False	False	False	False	False	F
4	False	False	False	False	False	False	False	False	False	False	False	False	F
...	...	...	...	...	...	...	...	...	...	...	...	...	
18244	False	False	False	False	False	False	False	False	False	False	False	False	F
18245	False	False	False	False	False	False	False	False	False	False	False	False	F
18246	False	False	False	False	False	False	False	False	False	False	False	False	F
18247	False	False	False	False	False	False	False	False	False	False	False	False	F
18248	False	False	False	False	False	False	False	False	False	False	False	False	F

18249 rows × 15 columns

We can use .any() function to get a consise info

In [35]:

```
avocado.isna().any()
```

Out[35]:

```
Unnamed: 0      False
Date            False
AveragePrice    False
Total Volume    False
4046            False
4225            False
4770            False
Total Bags      False
Small Bags      False
Large Bags      False
XLarge Bags     False
type            False
year            False
region          False
new_col         False
dtype: bool
```

Counting missing values

In [36]:

```
avocado.isna().sum()
```

Out[36]:
```
Unnamed: 0      0
Date            0
AveragePrice    0
Total Volume    0
4046            0
4225            0
4770            0
Total Bags      0
Small Bags      0
Large Bags      0
XLarge Bags     0
type            0
year            0
region          0
new_col         0
dtype: int64
```

Removing missing values

- Drop NaN **.dropna()**
- Fill NaN with value x **.fillna(x)**

In [37]:
```
# Luckily we don't have any NaN but if we have we can use any of the two methods

avocado.dropna()
#avocado.dropna(subset='column_name')

# ****  OR  ****

meanVal = avocado["AveragePrice"].mean()
avocado.fillna(meanVal)
```

Out[37]:

	Unnamed: 0	Date	AveragePrice	Total Volume	4046	4225	4770	Total Bags	Small Bags	Large Bags
0	0	2015-12-27	1.33	64236.62	1036.74	54454.85	48.16	8696.87	8603.62	93.25
1	1	2015-12-20	1.35	54876.98	674.28	44638.81	58.33	9505.56	9408.07	97.49
2	2	2015-12-13	0.93	118220.22	794.70	109149.67	130.50	8145.35	8042.21	103.14
3	3	2015-12-06	1.08	78992.15	1132.00	71976.41	72.58	5811.16	5677.40	133.76
4	4	2015-11-29	1.28	51039.60	941.48	43838.39	75.78	6183.95	5986.26	197.69
...	...	...	...	...	...	...	...	...	...	...
18244	7	2018-02-04	1.63	17074.83	2046.96	1529.20	0.00	13498.67	13066.82	431.85
18245	8	2018-01-28	1.71	13888.04	1191.70	3431.50	0.00	9264.84	8940.04	324.80
18246	9	2018-01-21	1.87	13766.76	1191.92	2452.79	727.94	9394.11	9351.80	42.31
18247	10	2018-01-14	1.93	16205.22	1527.63	2981.04	727.01	10969.54	10919.54	50.00
18248	11	2018-01-07	1.62	17489.58	2894.77	2356.13	224.53	12014.15	11988.14	26.01

18249 rows × 15 columns

Adding a new column

It can easily be done using the [ ] brackets

Lets add a new column to our dataframe called AveragePricePer100

In [38]:

```
avocado["AveragePricePer100"] = avocado["AveragePrice"] * 100
avocado
```

Out[38]:

	Unnamed: 0	Date	AveragePrice	Total Volume	4046	4225	4770	Total Bags	Small Bags	Large Bags
0	0	2015-12-27	1.33	64236.62	1036.74	54454.85	48.16	8696.87	8603.62	93.25
1	1	2015-12-20	1.35	54876.98	674.28	44638.81	58.33	9505.56	9408.07	97.49
2	2	2015-12-13	0.93	118220.22	794.70	109149.67	130.50	8145.35	8042.21	103.14
3	3	2015-12-06	1.08	78992.15	1132.00	71976.41	72.58	5811.16	5677.40	133.76
4	4	2015-11-29	1.28	51039.60	941.48	43838.39	75.78	6183.95	5986.26	197.69
...	...	...	...	...	...	...	...	...	...	...
18244	7	2018-02-04	1.63	17074.83	2046.96	1529.20	0.00	13498.67	13066.82	431.85
18245	8	2018-01-28	1.71	13888.04	1191.70	3431.50	0.00	9264.84	8940.04	324.80
18246	9	2018-01-21	1.87	13766.76	1191.92	2452.79	727.94	9394.11	9351.80	42.31
18247	10	2018-01-14	1.93	16205.22	1527.63	2981.04	727.01	10969.54	10919.54	50.00
18248	11	2018-01-07	1.62	17489.58	2894.77	2356.13	224.53	12014.15	11988.14	26.01

18249 rows × 16 columns

Deleting columns in DataFrame .drop(lst,axis = 1)

> dataFrame.drop(['COLUMN_NAME'], axis = 1)
>
> - the first parameter is a list of columns to be deleted
> - axis = 1 means delete column
> - axis = 0 means delete row

In [40]:

```
avocado.drop(columns=["AveragePricePer100"],inplace=True)
avocado
```

Out[40]:

	Unnamed: 0	Date	AveragePrice	Total Volume	4046	4225	4770	Total Bags	Small Bags	Large Bags
0	0	2015-12-27	1.33	64236.62	1036.74	54454.85	48.16	8696.87	8603.62	93.25
1	1	2015-12-20	1.35	54876.98	674.28	44638.81	58.33	9505.56	9408.07	97.49
2	2	2015-12-13	0.93	118220.22	794.70	109149.67	130.50	8145.35	8042.21	103.14
3	3	2015-12-06	1.08	78992.15	1132.00	71976.41	72.58	5811.16	5677.40	133.76
4	4	2015-11-29	1.28	51039.60	941.48	43838.39	75.78	6183.95	5986.26	197.69
...	...	...	...	...	...	...	...	...	...	..
18244	7	2018-02-04	1.63	17074.83	2046.96	1529.20	0.00	13498.67	13066.82	431.85
18245	8	2018-01-28	1.71	13888.04	1191.70	3431.50	0.00	9264.84	8940.04	324.80
18246	9	2018-01-21	1.87	13766.76	1191.92	2452.79	727.94	9394.11	9351.80	42.31
18247	10	2018-01-14	1.93	16205.22	1527.63	2981.04	727.01	10969.54	10919.54	50.00
18248	11	2018-01-07	1.62	17489.58	2894.77	2356.13	224.53	12014.15	11988.14	26.01

18249 rows × 15 columns

Summary statistics

Some of the functions availabe in pandas are:

> .median() .mode() .min() .max() .var() .std() .sum() .quantile()

In [41]:

```
# mean of the AveragePrice of avocado
avocado["AveragePrice"].mean()
```

Out[41]:

```
1.405978409775878
```

Summarizing dates

To find the min or max date in a dataframe

In [42]:

```
avocado["Date"].max()
```

Out[42]:

```
'2018-03-25'
```

.agg() method

Pandas Series.agg() is used to pass a function or list of function to be applied on a series or even each element of series separately.

> Syntax: Series.agg(func, axis=0)
>
> Parameters: func: Function, list of function or string of function name to be called on Series. axis:0 or 'index' for row wise operation and 1 or 'columns' for column wise operation.
>
> Return Type: The return type depends on return type of function passed as parameter.

In [43]:

```
def pct30(column):
    #return the 0.3 quartile
    return column.quantile(0.3)
def pct50(column):
    #return the 0.5 quartile
    return column.quantile(0.5)

avocado[["AveragePrice","Total Bags"]].agg([pct30,pct50])
```

Out[43]:

	AveragePrice	Total Bags
pct30	1.15	7316.634
pct50	1.37	39743.830

Dropping duplicate names .drop_duplicates(lst)

Delete all the duplicate names from the dataframe

In [44]:

```
temp = avocado.drop_duplicates(subset=["year"])
temp
```

Out[44]:

	Unnamed: 0	Date	AveragePrice	Total Volume	4046	4225	4770	Total Bags	Small Bags	Lar Ba
0	0	2015-12-27	1.33	64236.62	1036.74	54454.85	48.16	8696.87	8603.62	93.
2808	0	2016-12-25	1.52	73341.73	3202.39	58280.33	426.92	11432.09	11017.32	411.
5616	0	2017-12-31	1.47	113514.42	2622.70	101135.53	20.25	9735.94	5556.98	4178.
8478	0	2018-03-25	1.57	149396.50	16361.69	109045.03	65.45	23924.33	19273.80	4270.

Count categorical data .value_counts()

Pandas Series.value_counts() function return a Series containing counts of unique values.

> Syntax: Series.value_counts(normalize=False, sort=True, ascending=False, bins=None, dropna=True)
>
> Parameter : normalize : If True then the object returned will contain the relative frequencies of the unique values. sort : Sort by values. ascending : Sort in ascending order. bins : Rather than count values, group them into half-open bins, a convenience for pd.cut, only works with numeric data. dropna : Don't include counts of NaN.
>
> Returns : counts : Series

In [45]:

```
# count number of avocado in each year in descending order
avocado["year"].value_counts(sort=True, ascending = False)
```

Out[45]:

```
2017    5722
2016    5616
2015    5615
2018    1296
Name: year, dtype: int64
```

Grouped summaries .groupby(col)

This function will group similar categories into one and then we can perform some summary statistics

> Syntax: DataFrame.groupby(by=None, axis=0, level=None, as_index=True, sort=True, group_keys=True, squeeze=False, **kwargs)
>
> Parameters : by : mapping, function, str, or iterable
> axis : int, default 0
> level : If the axis is a MultiIndex (hierarchical), group by a particular level or levels
> as_index : For aggregated output, return object with group labels as the index. Only relevant for DataFrame input. as_index=False is effectively "SQL-style" grouped output
> sort : Sort group keys. Get better performance by turning this off. Note this does not influence the order of observations within each group. groupby preserves the order of rows within each group.
> group_keys : When calling apply, add group keys to index to identify pieces
> squeeze : Reduce the dimensionality of the return type if possible, otherwise return a consistent type
>
> Returns : GroupBy object

In [47]:

```
# group by multiple columns and perform multiple summary statistic operations
avocado.groupby("year").sum()
```

Out[47]:

year	Unnamed: 0	AveragePrice	Total Volume	4046	4225	4770	Total Bags
2015	143157	7723.94	4.385469e+09	1.709450e+09	1.761054e+09	1.427724e+08	7.721922e+08
2016	143208	7517.80	4.820890e+09	1.525123e+09	1.672728e+09	1.598798e+08	1.463159e+09
2017	148721	8669.56	4.934306e+09	1.652038e+09	1.544735e+09	9.121751e+07	1.646289e+09
2018	7128	1746.40	1.382738e+09	4.604997e+08	4.077587e+08	2.293259e+07	4.915359e+08

In [48]:

```
# group by multiple columns and perform multiple summary statistic operations
avocado.groupby(["year","type"])["AveragePrice"].agg([min,max,np.mean,np.median])
```

Out[48]:

year	type	min	max	mean	median
2015	conventional	0.49	1.59	1.077963	1.08
	organic	0.81	2.79	1.673324	1.67
2016	conventional	0.51	2.20	1.105595	1.08
	organic	0.58	3.25	1.571684	1.53
2017	conventional	0.46	2.22	1.294888	1.30
	organic	0.44	3.17	1.735521	1.72
2018	conventional	0.56	1.74	1.127886	1.14
	organic	1.01	2.30	1.567176	1.55

Pivot table

A pivot table is a table of statistics that summarizes the data of a more extensive table.

IMPORRANT parements to remember are
"index": it is the value that appeares on the left most side of the table (it can be a list)
"columns": these are the column you want to add to the pivot table
"aggfunc": it will call the function (it can be a list)
"values": it is the attribute which will be summarized in the table (values inside the table)

> Syntax
> pandas.pivot_table(data, values=None, index=None, columns=None, aggfunc='mean', fill_value=None, margins=False, dropna=True, margins_name='All')
>
> Parameters:
> data : DataFrame
> values : column to aggregate, optional
> index: column, Grouper, array, or list of the previousv columns: column, Grouper, array, or list of the previous
>
> aggfunc: function, list of functions, dict, default numpy.mean
> ....If list of functions passed, the resulting pivot table will have hierarchical columns whose

top level are the function names.
....If dict is passed, the key is column to aggregate and value is function or list of functions
fill_value[scalar, default None] : Value to replace missing values with
margins[boolean, default False] : Add all row / columns (e.g. for subtotal / grand totals)
dropna[boolean, default True] : Do not include columns whose entries are all NaN
margins_name[string, default 'All'] : Name of the row / column that will contain the totals when margins is True.

Returns: DataFrame

In [49]:
```
# this is the same table we build in the previous cell but using pivot table
avocado.pivot_table(index=["year","type"], aggfunc=[min,max,np.mean,np.median], values='
```

Out[49]:

		min	max	mean	median
		AveragePrice	AveragePrice	AveragePrice	AveragePrice
year	**type**				
2015	**conventional**	0.49	1.59	1.077963	1.08
	organic	0.81	2.79	1.673324	1.67
2016	**conventional**	0.51	2.20	1.105595	1.08
	organic	0.58	3.25	1.571684	1.53
2017	**conventional**	0.46	2.22	1.294888	1.30
	organic	0.44	3.17	1.735521	1.72
2018	**conventional**	0.56	1.74	1.127886	1.14
	organic	1.01	2.30	1.567176	1.55

Explicit indexes

Indexes make subsetting simpler using .loc and .iloc

Setting column as the index

In [50]:
```
regionIndex = avocado.set_index(["region"])
regionIndex
```

Out[50]:

region	Unnamed: 0	Date	AveragePrice	Total Volume	4046	4225	4770	Total Bags	:
Albany	0	2015-12-27	1.33	64236.62	1036.74	54454.85	48.16	8696.87	86
Albany	1	2015-12-20	1.35	54876.98	674.28	44638.81	58.33	9505.56	94
Albany	2	2015-12-13	0.93	118220.22	794.70	109149.67	130.50	8145.35	80
Albany	3	2015-12-06	1.08	78992.15	1132.00	71976.41	72.58	5811.16	56
Albany	4	2015-11-29	1.28	51039.60	941.48	43838.39	75.78	6183.95	59
...	...	...	...	...	...	...	...	...	
WestTexNewMexico	7	2018-02-04	1.63	17074.83	2046.96	1529.20	0.00	13498.67	130
WestTexNewMexico	8	2018-01-28	1.71	13888.04	1191.70	3431.50	0.00	9264.84	89
WestTexNewMexico	9	2018-01-21	1.87	13766.76	1191.92	2452.79	727.94	9394.11	93
WestTexNewMexico	10	2018-01-14	1.93	16205.22	1527.63	2981.04	727.01	10969.54	109
WestTexNewMexico	11	2018-01-07	1.62	17489.58	2894.77	2356.13	224.53	12014.15	119

18249 rows × 14 columns

In [51]:

```
# Insted of doing this
avocado[avocado["region"].isin(["Albany", "WestTexNewMexico"])]
```

Out[51]:

	Unnamed: 0	Date	AveragePrice	Total Volume	4046	4225	4770	Total Bags	Small Bags	Large Bags
0	0	2015-12-27	1.33	64236.62	1036.74	54454.85	48.16	8696.87	8603.62	93.25
1	1	2015-12-20	1.35	54876.98	674.28	44638.81	58.33	9505.56	9408.07	97.49
2	2	2015-12-13	0.93	118220.22	794.70	109149.67	130.50	8145.35	8042.21	103.14
3	3	2015-12-06	1.08	78992.15	1132.00	71976.41	72.58	5811.16	5677.40	133.76
4	4	2015-11-29	1.28	51039.60	941.48	43838.39	75.78	6183.95	5986.26	197.69
...	...	...	...	...	...	...	...	...	...	...
18244	7	2018-02-04	1.63	17074.83	2046.96	1529.20	0.00	13498.67	13066.82	431.85
18245	8	2018-01-28	1.71	13888.04	1191.70	3431.50	0.00	9264.84	8940.04	324.80
18246	9	2018-01-21	1.87	13766.76	1191.92	2452.79	727.94	9394.11	9351.80	42.31
18247	10	2018-01-14	1.93	16205.22	1527.63	2981.04	727.01	10969.54	10919.54	50.00
18248	11	2018-01-07	1.62	17489.58	2894.77	2356.13	224.53	12014.15	11988.14	26.01

673 rows × 15 columns

In [52]:

```
# we can simply do
regionIndex.loc[["Albany", "WestTexNewMexico"]]
```

Out[52]:

region	Unnamed: 0	Date	AveragePrice	Total Volume	4046	4225	4770	Total Bags	!
Albany	0	2015-12-27	1.33	64236.62	1036.74	54454.85	48.16	8696.87	86
Albany	1	2015-12-20	1.35	54876.98	674.28	44638.81	58.33	9505.56	94
Albany	2	2015-12-13	0.93	118220.22	794.70	109149.67	130.50	8145.35	80
Albany	3	2015-12-06	1.08	78992.15	1132.00	71976.41	72.58	5811.16	56
Albany	4	2015-11-29	1.28	51039.60	941.48	43838.39	75.78	6183.95	59
...	...	...	...	...	...	...	...	...	
WestTexNewMexico	7	2018-02-04	1.63	17074.83	2046.96	1529.20	0.00	13498.67	130
WestTexNewMexico	8	2018-01-28	1.71	13888.04	1191.70	3431.50	0.00	9264.84	89
WestTexNewMexico	9	2018-01-21	1.87	13766.76	1191.92	2452.79	727.94	9394.11	93
WestTexNewMexico	10	2018-01-14	1.93	16205.22	1527.63	2981.04	727.01	10969.54	109
WestTexNewMexico	11	2018-01-07	1.62	17489.58	2894.77	2356.13	224.53	12014.15	119

673 rows × 14 columns

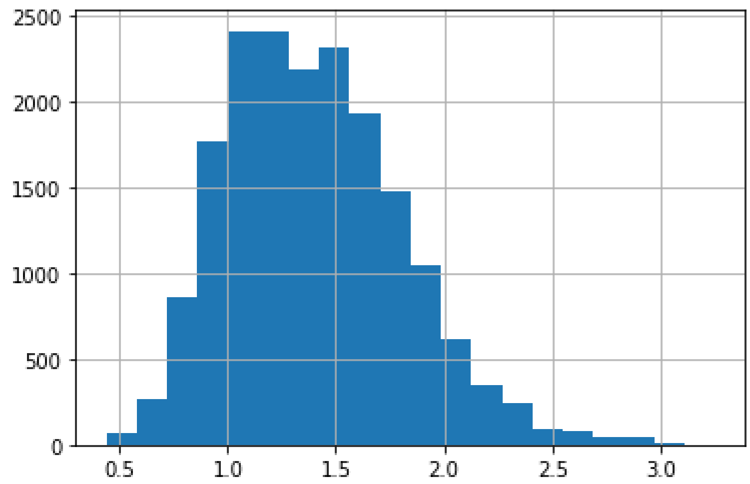
Visualizing your data

Histograms

use the function .hist()

In [53]:

```
avocado["AveragePrice"].hist(bins=20)
plt.show()
```



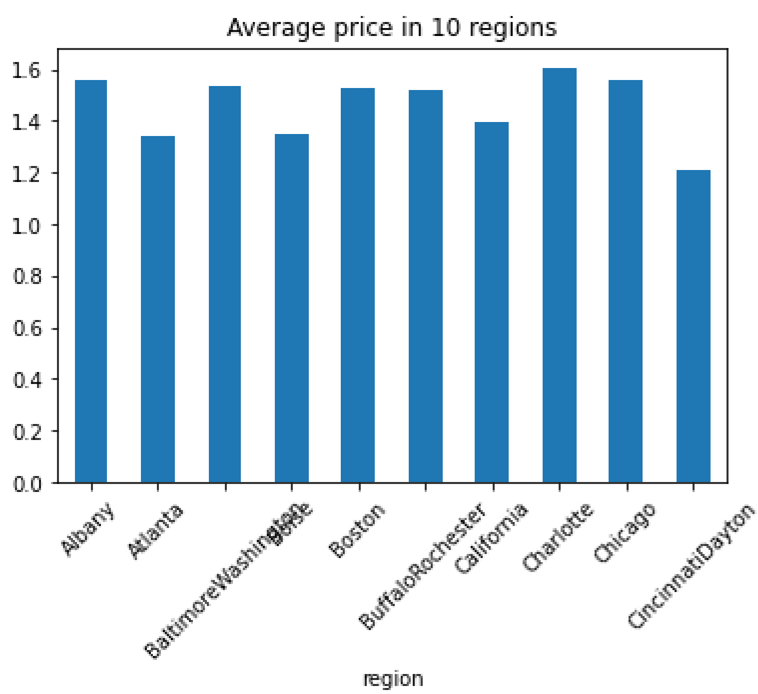
Bar plots

In [54]:
```
regionFilter = avocado.groupby("region")["AveragePrice"].mean().head(10)
regionFilter
```

Out[54]:
```
region
Albany                    1.561036
Atlanta                   1.337959
BaltimoreWashington       1.534231
Boise                     1.348136
Boston                    1.530888
BuffaloRochester          1.516834
California                1.395325
Charlotte                 1.606036
Chicago                   1.556775
CincinnatiDayton          1.209201
Name: AveragePrice, dtype: float64
```

In [55]:
```
regionFilter.plot(kind = "bar",rot=45,title="Average price in 10 regions")
```

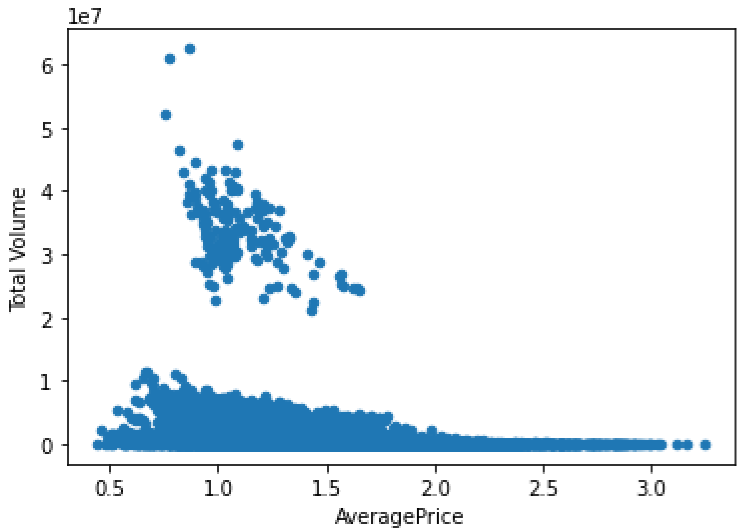
Out[55]:
```
<AxesSubplot:title={'center':'Average price in 10 regions'}, xlabel='region'>
```



Scatter plot

In [56]:
```
avocado.plot(x="AveragePrice", y="Total Volume", kind="scatter")
```

Out[56]: <AxesSubplot:xlabel='AveragePrice', ylabel='Total Volume'>



Arithmetic with Series & DataFrames

You can use arithmetic operators directly on series but sometimes you need more control while performing these operations, here is where these explicit arithmetic functions come into the picture

Add/Subtract function (just replece add with sub)

```
Syntax: Series.add(other, level=None, fill_value=None, axis=0)

Parameters:
other: other series or list type to be added into caller series
fill_value: Value to be replaced by NaN in series/list before adding
level: integer value of level in case of multi index

Return type: Caller series with added values
```

Multiplication function

```
Syntax: Series.mul(other, level=None, fill_value=None, axis=0)

Parameters:
other: other series or list type to be added into caller series
fill_value: Value to be replaced by NaN in series/list before adding
level: integer value of level in case of multi index

Return type: Caller series with added values
```

Division function

```
Syntax: Series.div(other, level=None, fill_value=None, axis=0)

Parameters:
other: other series or list type to be divided by the caller series
fill_value: Value to be replaced by NaN in series/list before division
```

level: integer value of level in case of multi index

Return type: Caller series with divided values

In [58]:
```
# subtract AveragePrice with AveragePrice :P
# Duhh its 0
avocado["AveragePrice"].sub(avocado["AveragePrice"])
```

Out[58]:
```
0        0.0
1        0.0
2        0.0
3        0.0
4        0.0
        ...
18244    0.0
18245    0.0
18246    0.0
18247    0.0
18248    0.0
Name: AveragePrice, Length: 18249, dtype: float64
```

Merge DataFrames

Syntax:

> DataFrame.merge(self, right, how='inner', on=None, left_on=None, right_on=None, left_index=False, right_index=False, sort=False, suffixes=('_x', '_y'), copy=True, indicator=False, validate=None) → 'DataFrame'[source]¶ Merge DataFrame or named Series objects with a database-style join.

The join is done on columns or indexes. If joining columns on columns, the DataFrame indexes will be ignored. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on.

Parameters right: DataFrame or named Series Object to merge with.

how{'left', 'right', 'outer', 'inner'}, default 'inner'

on: label or list Column or index level names to join on. These must be found in both DataFrames. If on is None and not merging on indexes then this defaults to the intersection of the columns in both DataFrames.

left_on: label or list, or array-like Column or index level names to join on in the left DataFrame. Can also be an array or list of arrays of the length of the left DataFrame. These arrays are treated as if they are columns.

right_on: label or list, or array-like Column or index level names to join on in the right DataFrame. Can also be an array or list of arrays of the length of the right DataFrame. These arrays are treated as if they are columns.

left_index: bool, default False Use the index from the left DataFrame as the join key(s). If it is a MultiIndex, the number of keys in the other DataFrame (either the index or a number of columns) must match the number of levels.

right_index: bool, default False Use the index from the right DataFrame as the join key. Same caveats as left_index.

sort: bool, default False Sort the join keys lexicographically in the result DataFrame. If False, the order of the join keys depends on the join type (how keyword).

suffixes: tuple of (str, str), default (‘_x’, ‘_y’) Suffix to apply to overlapping column names in the left and right side, respectively. To raise an exception on overlapping columns use (False, False).

In [60]:

```
df1 = pd.DataFrame({"A":[1,3,4],"B":[4,5,6]})
df2 = pd.DataFrame({"A":[2,3,4],"C":[8,9,9]})
```

In [61]:

```
df1
```

Out[61]:

	A	B
0	1	4
1	3	5
2	4	6

In [62]:

```
df2
```

Out[62]:

	A	C
0	2	8
1	3	9
2	4	9

In [64]:

```
#Outer Join
df1.merge(df2, on='A', how='outer')
#pd.merge(df1,df2,on='A', how='outer')
```

Out[64]:

	A	B	C
0	1	4.0	NaN
1	3	5.0	9.0
2	4	6.0	9.0
3	2	NaN	8.0

In [65]:

```
#inner Join
df1.merge(df2, on='A', how='inner')
#pd.merge(df1,df2,on='A', how='inner')
```

Out[65]:

	A	B	C
0	3	5	9
1	4	6	9

In [66]:

```
#left Join
```

```
df1.merge(df2, on='A', how='left')
#pd.merge(df1,df2,on='A', how='left')
```

Out[66]:

	A	B	C
0	1	4	NaN
1	3	5	9.0
2	4	6	9.0

In [67]:

```
#right Join
df1.merge(df2, on='A', how='right')
#pd.merge(df1,df2,on='A', how='right')
```

Out[67]:

	A	B	C
0	2	NaN	8
1	3	5.0	9
2	4	6.0	9

Join

> DataFrame.merge(self, right, how='inner', on=None, left_on=None, right_on=None, left_index=False, right_index=False, sort=False, suffixes=('_x', '_y'), copy=True, indicator=False, validate=None) → 'DataFrame'[source]¶ Merge DataFrame or named Series objects with a database-style join.

The join is done on columns or indexes. If joining columns on columns, the DataFrame indexes will be ignored. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on.

Parameters rightDataFrame or named Series Object to merge with.

how{'left', 'right', 'outer', 'inner'}, default 'inner' on: label or list Column or index level names to join on. These must be found in both DataFrames. If on is None and not merging on indexes then this defaults to the intersection of the columns in both DataFrames.

left_on: label or list, or array-like Column or index level names to join on in the left DataFrame. Can also be an array or list of arrays of the length of the left DataFrame. These arrays are treated as if they are columns.

right_on: label or list, or array-like Column or index level names to join on in the right DataFrame. Can also be an array or list of arrays of the length of the right DataFrame. These arrays are treated as if they are columns.

left_index: bool, default False Use the index from the left DataFrame as the join key(s). If it is a MultiIndex, the number of keys in the other DataFrame (either the index or a number of columns) must match the number of levels.

right_index: bool, default False Use the index from the right DataFrame as the join key. Same caveats as left_index.

sort: bool, default False Sort the join keys lexicographically in the result DataFrame. If False, the order of the join keys depends on the join type (how keyword).

suffixes: tuple of (str, str), default ('_x', '_y') Suffix to apply to overlapping column names in the left and right side, respectively. To raise an exception on overlapping columns use (False, False).

View or Copy

Main differences

- If object A is a **copy** of object B, then each object will be allocated their own memory block for its data. This means that modifying the copy will not mutate the original data and vice versa.
- If object A is a **view** of object B, then they both share a single memory block for their data. This means that modifying the copy will mutate the original data and vice versa.

In the context of Pandas, when you access values of Series or a DataFrame, what is returned can either be a copy or view. This distinction is important for two reasons:

if you don't know whether the return value is a copy or view, then you will not know what happens when you modify the return value - will it mutate the original data or not?

if you're dealing with large datasets, then you might not want the return value to be a copy since copies take up more memory.

Accessing values of a Series/DataFrame Unfortunately, when you access values from a Series or a DataFrame, the rule that decides whether a copy or view is returned is quite complicated.

Here is a general rule of thumb:

- **if you access a single column, then a view is returned (e.g. df["A"]).**
- **if you access multiple columns, then a copy is returned (e.g. df[["A","B"]])**

Here's a quick demo - consider the following DataFrame:

In [68]:
```
df = pd.DataFrame({"A":[3,4],"B":[5,6]})
df
```

Out[68]:

	A	B
0	3	5
1	4	6

Getting a view

To illustrate that accessing a single column returns a view:

In [69]:
```
col_A = df["A"]    # col_A is a view
col_A[0] = 9
df
```

Out[69]:

	A	B
0	9	5
1	4	6

Notice how modifying col_A mutated the original df. Note that modifying df will also mutate col_A.

Getting a copy

To illustrate that accessing multiple columns returns a copy:

In [70]:

```
cols_A_B = df[["A","B"]]    # cols_A_B is a copy
cols_A_B.iloc[0,0] = 9
df
```

Out[70]:

	A	B
0	9	5
1	4	6

Notice how df did not get mutated.

Other cases

For other cases, whether a copy or view is returned depends on the situation. Whenever in doubt, it is good practise to use the _is_view property to verify:

Copying Pandas object

Pandas has the method copy(~) that makes a copy of a Pandas object:

In [71]:

```
df = pd.DataFrame({"A":[3,4],"B":[6,7]}, index=["a","b"])
df_copy = df.copy()
df_copy.iloc[0,0] = 10
df
```

Out[71]:

	A	B
a	3	6
b	4	7

In [72]:

```
#check dataframe/series object is a view using ._is_view
df._is_view
```

Out[72]:

```
False
```

When in doubt, use copy()