

```
In [1]: %%html
<h2>Chapter 5 - Getting started with pandas</h2>
```

## Chapter 5 - Getting started with pandas

```
In [2]: import pandas as pd
import numpy as np
```

```
In [34]: %%html
<h3>Pandas Data Structures - Series</h3>

Code examples are taken from <a href="https://github.com/wesm/pydata-book/blob/3rd-edition/ch05.ipynb">https://gith
```

### Pandas Data Structures - Series

Code examples are taken from <https://github.com/wesm/pydata-book/blob/3rd-edition/ch05.ipynb>

```
In [4]: # series is a one dimension array like object containing sequence of values
obj = pd.Series([4,7,-5, 3])
obj
# left most column indicates the indices of the series
```

```
Out[4]: 0    4
        1    7
        2   -5
        3    3
        dtype: int64
```

```
In [8]: obj.index
```

```
Out[8]: RangeIndex(start=0, stop=4, step=1)
```

```
In [10]: # we can manually specify the index
obj2 = pd.Series([4,7,-5,3], index=["d","b","a","c"])
obj2
```

```
Out[10]: d    4  
        b    7  
        a   -5  
        c    3  
        dtype: int64
```

```
In [11]: #We can get the value based on the index  
obj2["d"]
```

```
Out[11]: 4
```

```
In [12]: # we can get multiple values by providing a list  
obj2[["d","b"]]
```

```
Out[12]: d    4  
        b    7  
        dtype: int64
```

```
In [13]: # we can filter  
obj2[obj2>0]
```

```
Out[13]: d    4  
        b    7  
        c    3  
        dtype: int64
```

```
In [14]: # We can create a series from python dictionary. In this case key become index and value become series's value.
```

```
In [15]: # a series can be converted to dictionary  
obj2.to_dict()
```

```
Out[15]: {'d': 4, 'b': 7, 'a': -5, 'c': 3}
```

```
In [16]: # we can check got NaN  
pd.isna(obj2)
```

```
Out[16]: d    False  
        b    False  
        a    False  
        c    False  
        dtype: bool
```

```
In [17]: obj2.isna()
```

```
Out[17]: d    False  
        b    False  
        a    False  
        c    False  
        dtype: bool
```

```
In [18]: pd.notna(obj2)
```

```
Out[18]: d     True  
        b     True  
        a     True  
        c     True  
        dtype: bool
```

```
In [19]: obj2.notna()
```

```
Out[19]: d     True  
        b     True  
        a     True  
        c     True  
        dtype: bool
```

```
In [21]: # We can do arithmetic - the index should be same other wise we will get NaN  
obj + obj2
```

```
Out[21]: 0    NaN
          1    NaN
          2    NaN
          3    NaN
          a    NaN
          b    NaN
          c    NaN
          d    NaN
          dtype: float64
```

```
In [22]: obj+obj
```

```
Out[22]: 0      8
          1     14
          2    -10
          3      6
          dtype: int64
```

```
In [26]: # We can add the name to series and index
obj2.index.name="something"
obj2.name = "series_name"
obj2
```

```
Out[26]: something
d      4
b      7
a     -5
c      3
Name: series_name, dtype: int64
```

```
In [27]: # we can change the index by assigning a new value
```

```
In [30]: obj2.index = ["w","x","y","z"]
obj2
```

```
Out[30]: w      4
          x      7
          y     -5
          z      3
          Name: series_name, dtype: int64
```

```
In [32]: %%html
<h3>Pandas Data Structures - DataFrame</h3>
```

## Pandas Data Structures - DataFrame

```
In [35]: # A DF represents a rectangular table of data contains a sorted, named collection of columns, each of which can be a
# It's similar to a table in the 2 dimensional df. It can be used to represent higher dimensional data in a tabular
```

```
In [37]: data = {"state": ["Ohio", "Ohio", "Ohio", "Nevada", "Nevada", "Nevada"],
                 "year": [2000, 2001, 2002, 2001, 2002, 2003],
                 "pop": [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
df = pd.DataFrame(data)
df
```

```
Out[37]:
```

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9
5	Nevada	2003	3.2

```
In [38]: df.head()
```

```
Out[38]:
```

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9

```
In [39]: df.tail()
```

```
Out[39]:
```

	state	year	pop
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9
5	Nevada	2003	3.2

```
In [40]: # We can arrange th order of the columns  
pd.DataFrame(data, columns=["year", "state", "pop"])
```

```
Out[40]:
```

	year	state	pop
0	2000	Ohio	1.5
1	2001	Ohio	1.7
2	2002	Ohio	3.6
3	2001	Nevada	2.4
4	2002	Nevada	2.9
5	2003	Nevada	3.2

```
In [64]: # we can pass a column that is not in the original data  
df = pd.DataFrame(data, columns=["year", "state", "pop", "debt s"])
```

```
In [49]: %%html  
<h4>Data retrieval from column</h4>
```

## Data retrieval from column

```
In [44]: # We can retrieve the data in the column using dictionary type or dot attribute type. Note: Column name should ne a  
df["year"]
```

```
Out[44]: 0    2000
         1    2001
         2    2002
         3    2001
         4    2002
         5    2003
         Name: year, dtype: int64
```

```
In [45]: df.year
```

```
Out[45]: 0    2000
         1    2001
         2    2002
         3    2001
         4    2002
         5    2003
         Name: year, dtype: int64
```

```
In [46]: df["debt s"]
```

```
Out[46]: 0    NaN
         1    NaN
         2    NaN
         3    NaN
         4    NaN
         5    NaN
         Name: debt s, dtype: object
```

```
In [48]: df.debt s
```

```
Cell In[48], line 1
      df.debt s
            ^
SyntaxError: invalid syntax
```

```
In [50]: %%html
         <h4>Data retrieval from row</h4>
```

**Data retrieval from row**

```
In [51]: # we can use loc & iloc(mainly for integer location)
# in the above example index is integer so we can use both loc & iloc

df.loc[1]
```

```
Out[51]: year      2001
state      Ohio
pop        1.7
debt s     NaN
Name: 1, dtype: object
```

```
In [52]: df.iloc[1]
```

```
Out[52]: year      2001
state      Ohio
pop        1.7
debt s     NaN
Name: 1, dtype: object
```

```
In [54]: # we can modify the column values

df["debt s"] = 16
df
```

```
Out[54]:
```

	year	state	pop	debt s
0	2000	Ohio	1.5	16
1	2001	Ohio	1.7	16
2	2002	Ohio	3.6	16
3	2001	Nevada	2.4	16
4	2002	Nevada	2.9	16
5	2003	Nevada	3.2	16

```
In [65]: df["debt s"] = np.arange(6)
df
```



```
Out[65]:
```

	year	state	pop	debt s
0	2000	Ohio	1.5	0
1	2001	Ohio	1.7	1
2	2002	Ohio	3.6	2
3	2001	Nevada	2.4	3
4	2002	Nevada	2.9	4
5	2003	Nevada	3.2	5

```
In [66]: # we can delete the column  
del df["debt s"]  
df
```

```
Out[66]:
```

	year	state	pop
0	2000	Ohio	1.5
1	2001	Ohio	1.7
2	2002	Ohio	3.6
3	2001	Nevada	2.4
4	2002	Nevada	2.9
5	2003	Nevada	3.2

```
In [73]: # we can do transpose which swap rows and columns  
new_df = df.T  
new_df.index
```

```
Out[73]: Index(['year', 'state', 'pop'], dtype='object')
```

```
In [68]:
```

Out[68]:

	year	state	pop
0	2000	Ohio	1.5
1	2001	Ohio	1.7
2	2002	Ohio	3.6
3	2001	Nevada	2.4
4	2002	Nevada	2.9
5	2003	Nevada	3.2

```
In [74]: populations = {"Ohio": {2000: 1.5, 2001: 1.7, 2002: 3.6},
                        "Nevada": {2001: 2.4, 2002: 2.9}}
df3 = pd.DataFrame(populations)
df3
```

Out[74]:

	Ohio	Nevada
2000	1.5	NaN
2001	1.7	2.4
2002	3.6	2.9

```
In [76]: df3.index.name = "year"
df3.columns.name = "state"
df
```

```
Out[76]:
```

	state	year	state	pop
0		2000	Ohio	1.5
1		2001	Ohio	1.7
2		2002	Ohio	3.6
3		2001	Nevada	2.4
4		2002	Nevada	2.9
5		2003	Nevada	3.2

```
In [77]: # We can convert it to numpy  
df.to_numpy()
```

```
Out[77]: array([[2000, 'Ohio', 1.5],  
               [2001, 'Ohio', 1.7],  
               [2002, 'Ohio', 3.6],  
               [2001, 'Nevada', 2.4],  
               [2002, 'Nevada', 2.9],  
               [2003, 'Nevada', 3.2]], dtype=object)
```

```
In [81]: %%html  
<h3>Index objects</h3>  
  
Index objects are responsible for holding the axis labels including a DF's column names
```

## Index objects

Index objects are responsible for holding the axis labels including a DF's column names

```
In [82]: df.columns
```

```
Out[82]: Index(['year', 'state', 'pop'], dtype='object', name='state')
```

```
In [83]: df.index
```

```
Out[83]: RangeIndex(start=0, stop=6, step=1)
```

```
In [84]: # row index is referred by df.index and column index is referred by df.columns
```

```
In [85]: %%html
<h3>Reindexing</h3>
```

## Reindexing

```
In [86]: obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=["d", "b", "a", "c"])
obj
```

```
Out[86]: d    4.5
         b    7.2
         a   -5.3
         c    3.6
         dtype: float64
```

```
In [88]: # For the series, we can refer it by directly passing the new index list
obj2 = obj.reindex(["a", "b", "c", "d", "e"])
obj2
```

```
Out[88]: a   -5.3
         b    7.2
         c    3.6
         d    4.5
         e    NaN
         dtype: float64
```

```
In [92]: #For dataframes, you need to mention index for the row and columns for the columns
df3 = pd.DataFrame(np.arange(9).reshape((3, 3)),
                    index=["a", "c", "d"],
                    columns=["Ohio", "Texas", "California"])
df3
```

```
Out[92]:
```

	Ohio	Texas	California
<b>a</b>	0	1	2
<b>c</b>	3	4	5
<b>d</b>	6	7	8

```
In [95]: df3.reindex(index=["a", "b", "c", "d"])
```

```
Out[95]:
```

	Ohio	Texas	California
<b>a</b>	0.0	1.0	2.0
<b>b</b>	NaN	NaN	NaN
<b>c</b>	3.0	4.0	5.0
<b>d</b>	6.0	7.0	8.0

```
In [96]: df3.reindex(columns=["Texas", "Utah", "California"])
```

```
Out[96]:
```

	Texas	Utah	California
<b>a</b>	1	NaN	2
<b>c</b>	4	NaN	5
<b>d</b>	7	NaN	8

```
In [102... #Another way to reindex is using axis which can be similar to series with only addition of axis=index or columns or
df3 = df3.reindex(["Texas", "Utah", "California"], axis=1)
```

```
In [103... %%html
<h3>Dropping entries from an Axis</h3>
```

## Dropping entries from an Axis

```
In [104... df3.drop(columns="Utah")
```

Out[104... 

	Texas	California
a	1	2
c	4	5
d	7	8

```
In [108... df3 = df3.reindex(columns=["Texas", "Utah", "California"])
df3.drop(columns=["Texas", "Utah"])
```

Out[108... 

	California
a	2
c	5
d	8

```
In [111... df3.drop(index=["d"])
```

Out[111... 

	Texas	Utah	California
a	1	NaN	2
c	4	NaN	5

```
In [112... %%html
<h3>Indexing, Selection, and Filtering</h3>
```

## Indexing, Selection, and Filtering

```
In [113... %%html
Pandas indexing methods
<img src='images/pandas_slicing.png', width=500/>
```

Pandas indexing methods



No description has been provided for this image

```
In [6]: # similar to numpy, we can do the slicing with df as show in the picture above
```

```
data = pd.DataFrame(np.arange(16).reshape((4, 4)),  
                    index=["Ohio", "Colorado", "Utah", "New York"],  
                    columns=["one", "two", "three", "four"])  
data
```

Out[6]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

```
In [117... #df.loc[rows]  
data.loc["Ohio"]
```

```
Out[117... one      0  
two      1  
three    2  
four     3  
Name: Ohio, dtype: int64
```

```
In [119... #df.loc[rows,columns]  
data.loc["Ohio","one"]
```

Out[119... 0

```
In [121... # In pandas end index is inclusive in slicing. if you do just : means all  
#df.loc[rowIndexStart: rowIndexEnd, columnIndexStart:columnIndexEnd]  
  
data.loc["Ohio": "Utah", :]
```

Out[121...

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11

In [122... data.loc["Ohio": "Utah", "one":"two"]

Out[122...

	one	two
Ohio	0	1
Colorado	4	5
Utah	8	9

In [123... data.loc["Ohio": "Utah", "two":]

Out[123...

	two	three	four
Ohio	1	2	3
Colorado	5	6	7
Utah	9	10	11

In [124... data.loc["Ohio": "Utah", "two"]

Out[124... Ohio 1  
Colorado 5  
Utah 9  
Name: two, dtype: int64

In [129... data.loc[data["three"]>2]



```
Out[129...
```

	one	two	three	four
<b>Colorado</b>	4	5	6	7
<b>Utah</b>	8	9	10	11
<b>New York</b>	12	13	14	15

```
In [126... # We can do conditional filling - the below fill all the values in the row with 0 if the conditio matches

data.loc[data["four"]>5] = 3
data
```

```
Out[126...
```

	one	two	three	four
<b>Ohio</b>	0	1	2	3
<b>Colorado</b>	3	3	3	3
<b>Utah</b>	3	3	3	3
<b>New York</b>	3	3	3	3

```
In [131... %%html
<h3>Arithmetic & Data Alignment</h3>
```

## Arithmetic & Data Alignment

```
In [15]:
```

```
/tmp/ipykernel_262/3913871598.py:2: FutureWarning: Series.__getitem__ treating keys as positions is deprecated. In a
future version, integer keys will always be treated as labels (consistent with DataFrame behavior). To access a valu
e by position, use `ser.iloc[pos]`
ser2[-1]
```

```
Out[15]: Ohio      3
Colorado  7
Utah     11
New York  15
Name: four, dtype: int64
```

```
In [21]: df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)),
                           columns=list("abcd"))
df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)),
                   columns=list("abcde"))
df2.loc[1, "b"] = np.nan
```

```
In [22]: df1+df2
```

```
Out[22]:
```

	a	b	c	d	e
0	0.0	2.0	4.0	6.0	NaN
1	9.0	NaN	13.0	15.0	NaN
2	18.0	20.0	22.0	24.0	NaN
3	NaN	NaN	NaN	NaN	NaN

```
In [23]: # The addition only add the values with matching row and column index.
# Alternatively, we can add and the fill value insted of nan in the areas where non matching element present.
df1.add(df2, fill_value=0.0)
```

```
Out[23]:
```

	a	b	c	d	e
0	0.0	2.0	4.0	6.0	4.0
1	9.0	5.0	13.0	15.0	9.0
2	18.0	20.0	22.0	24.0	14.0
3	15.0	16.0	17.0	18.0	19.0

```
In [25]: # We can do arithmetic operation between DataFrame & series. By default, df & series matches the indes of the serie
frame = pd.DataFrame(np.arange(12.).reshape((4, 3)),
                     columns=list("bde"),
                     index=["Utah", "Ohio", "Texas", "Oregon"])
series = frame.iloc[0]
print(frame)
series
```

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

```
Out[25]: b    0.0
         d    1.0
         e    2.0
         Name: Utah, dtype: float64
```

```
In [26]: frame - series # matching on the index of series on the column index of the df.
```

```
Out[26]:
```

	b	d	e
<b>Utah</b>	0.0	0.0	0.0
<b>Ohio</b>	3.0	3.0	3.0
<b>Texas</b>	6.0	6.0	6.0
<b>Oregon</b>	9.0	9.0	9.0

```
In [28]: # we can change this behaviour using arithmetic methods
series3 = frame["d"]
series3
```

```
Out[28]: Utah      1.0
         Ohio      4.0
         Texas     7.0
         Oregon   10.0
         Name: d, dtype: float64
```

```
In [29]: frame.sub(series3, axis="index") # In this case ,atch on the df's row index and broadcast across the column
```

```
Out[29]:
```

	b	d	e
Utah	-1.0	0.0	1.0
Ohio	-1.0	0.0	1.0
Texas	-1.0	0.0	1.0
Oregon	-1.0	0.0	1.0

```
In [30]: %%html
<h3>Function application and Mapping</h3>
```

## Function application and Mapping

```
In [35]: def f1(x):
          return x.max()-x.min()

frame = pd.DataFrame(np.random.standard_normal((4, 3)),
                      columns=list("bde"),
                      index=["Utah", "Ohio", "Texas", "Oregon"])

frame.apply(f1)
# By default, the function f, which computes the difference between the minimum & maximum of a series, is invoked on
# The result is a series having the columns of the frame as its index.
```

```
Out[35]: b    2.453837
         d    1.445227
         e    2.399423
         dtype: float64
```

```
In [36]: # We can change this behaviour by specifying the axis:
frame.apply(f1, axis="columns") # this will invoke once per row instead
```

```
Out[36]: Utah    1.649759
         Ohio    0.315926
         Texas    1.511424
         Oregon    2.982811
         dtype: float64
```

```
In [37]: # Element wise python function can be used
```

```
def my_format(x):  
    return f"{x:.2f}"
```

```
frame.applymap(my_format)
```

```
/tmp/ipykernel_262/3768607378.py:6: FutureWarning: DataFrame.applymap has been deprecated. Use DataFrame.map instead.  
    frame.applymap(my_format)
```

```
Out[37]:
```

	<b>b</b>	<b>d</b>	<b>e</b>
<b>Utah</b>	-0.96	0.66	0.69
<b>Ohio</b>	-0.46	-0.15	-0.20
<b>Texas</b>	-1.26	-0.25	0.25
<b>Oregon</b>	1.20	-0.79	2.20

```
In [39]:
```

```
%%html  
<h3>Sorting and ranking</h3>
```

## Sorting and ranking

```
In [40]: obj = pd.Series(np.arange(4), index=["d", "a", "b", "c"])  
obj
```

```
Out[40]: d    0  
         a    1  
         b    2  
         c    3  
         dtype: int64
```

```
In [41]: obj.sort_index()
```

```
Out[41]: <bound method Series.sort_index of d      0
a      1
b      2
c      3
dtype: int64>
```

```
In [42]: frame = pd.DataFrame(np.arange(8).reshape((2, 4)),
                             index=["three", "one"],
                             columns=["d", "a", "b", "c"])

frame
```

```
Out[42]:
```

	d	a	b	c
three	0	1	2	3
one	4	5	6	7

```
In [43]: frame.sort_index()
```

```
Out[43]:
```

	d	a	b	c
one	4	5	6	7
three	0	1	2	3

```
In [44]: frame.sort_index(axis="columns")
```

```
Out[44]:
```

	a	b	c	d
three	1	2	3	0
one	5	6	7	4

```
In [50]: frame.sort_values(["a", "b"])
```

```
Out[50]:
```

	d	a	b	c
three	0	1	2	3
one	4	5	6	7

```
In [51]: # Ranking

obj = pd.Series([7, -5, 7, 4, 2, 0, 4])
obj
```

```
Out[51]: 0    7
         1   -5
         2    7
         3    4
         4    2
         5    0
         6    4
         dtype: int64
```

```
In [53]: obj.rank(ascending=False)
```

```
Out[53]: 0    1.5
         1    7.0
         2    1.5
         3    3.5
         4    5.0
         5    6.0
         6    3.5
         dtype: float64
```

```
In [56]: frame = pd.DataFrame({"b": [4.3, 7, -3, 2], "a": [0, 1, 0, 1],
                                "c": [-2, 5, 8, -2.5]})
frame
frame.rank(axis="columns")
```

```
Out[56]:
```

	<b>b</b>	<b>a</b>	<b>c</b>
<b>0</b>	3.0	2.0	1.0
<b>1</b>	3.0	1.0	2.0
<b>2</b>	1.0	2.0	3.0
<b>3</b>	3.0	2.0	1.0

```
In [57]: frame["c"]
```

```
Out[57]:
```

0	-2.0
1	5.0
2	8.0
3	-2.5

Name: c, dtype: float64

```
In [62]: frame.loc[:, "b": "c"]
```

```
Out[62]:
```

	<b>b</b>	<b>a</b>	<b>c</b>
<b>0</b>	4.3	0	-2.0
<b>1</b>	7.0	1	5.0
<b>2</b>	-3.0	0	8.0
<b>3</b>	2.0	1	-2.5

```
In [65]: frame = frame.sort_index(axis = "columns")
```

```
In [66]: frame.loc[:, "b": "c"]
```



```
Out[66]:
```

	b	c
0	4.3	-2.0
1	7.0	5.0
2	-3.0	8.0
3	2.0	-2.5

```
In [68]: %%html
<h3>Summarising and computing descriptive statistics</h3>
```

## Summarising and computing descriptive statistics

```
In [69]: # Sum method returns a series containing column sums

frame.sum()

# passing axis="columns" sums across the columns instead.
frame.sum(axis="columns")
```

```
Out[69]: 0      2.3
         1     13.0
         2      5.0
         3      0.5
         dtype: float64
```


```
In [70]: frame.describe()
```

Out[70]:

	a	b	c
count	4.00000	4.000000	4.000000
mean	0.50000	2.575000	2.125000
std	0.57735	4.241364	5.202163
min	0.00000	-3.000000	-2.500000
25%	0.00000	0.750000	-2.125000
50%	0.50000	3.150000	1.500000
75%	1.00000	4.975000	5.750000
max	1.00000	7.000000	8.000000

```
In [71]: %%html
Descriptive & summary methods
<img src='images/stats_method.png', width=500/>
```

Descriptive & summary methods

 No description has been provided for this image

```
In [83]: %%html
<h3>Correlation & covariance</h3>

We can compute the correlation & covriance between two columns/attributes/features.
<br/>
<br/>
<p>
<b>Covariance vs. Correlation</b>
<br/>
Covariance reveals how two variables change together while correlation determines how closely two variables are rel
<br/>
<br/>
Both covariance and correlation measure the relationship and the dependency between two variables.
<br/>
Covariance indicates the direction of the linear relationship between variables
<br/>
Correlation measures both the strength and direction of the linear relationship between two variables.
```

```
<br/>
Correlation values are standardized.
<br/>
Covariance values are not standardized.
</p>
```

ref: <https://builtin.com/data-science/covariance-vs-correlation>

## Correlation & covariance

We can compute the correlation & covariance between two columns/attributes/features.

### Covariance vs. Correlation

Covariance reveals how two variables change together while correlation determines how closely two variables are related to each other.

Both covariance and correlation measure the relationship and the dependency between two variables.

Covariance indicates the direction of the linear relationship between variables

Correlation measures both the strength and direction of the linear relationship between two variables.

Correlation values are standardized.

Covariance values are not standardized.

ref: <https://builtin.com/data-science/covariance-vs-correlation>

```
In [73]: price = pd.read_pickle("examples/yahoo_price.pkl")
         volume = pd.read_pickle("examples/yahoo_volume.pkl")
```

```
In [74]: returns = price.pct_change()
         returns.tail()
```

Out[74]:

	AAPL	GOOG	IBM	MSFT
Date				
2016-10-17	-0.000680	0.001837	0.002072	-0.003483
2016-10-18	-0.000681	0.019616	-0.026168	0.007690
2016-10-19	-0.002979	0.007846	0.003583	-0.002255
2016-10-20	-0.000512	-0.005652	0.001719	-0.004867
2016-10-21	-0.003930	0.003011	-0.012474	0.042096

In [84]: *# This will give the correlation & covariance with two columns*

```
print(returns["AAPL"].corr(returns["GOOG"]))  
print(returns["AAPL"].cov(returns["GOOG"]))
```

0.40791857616797006

0.00010745748920152612

In [86]: *#We can get the full correlation & covaraince*

```
returns.corr()
```

Out[86]:

	AAPL	GOOG	IBM	MSFT
AAPL	1.000000	0.407919	0.386817	0.389695
GOOG	0.407919	1.000000	0.405099	0.465919
IBM	0.386817	0.405099	1.000000	0.499764
MSFT	0.389695	0.465919	0.499764	1.000000

In [87]: `returns.cov()`

```
Out[87]:
```

	AAPL	GOOG	IBM	MSFT
AAPL	0.000277	0.000107	0.000078	0.000095
GOOG	0.000107	0.000251	0.000078	0.000108
IBM	0.000078	0.000078	0.000146	0.000089
MSFT	0.000095	0.000108	0.000089	0.000215

```
In [88]: # To calculate pair wise correlation

returns.corrwith(returns["AAPL"])
```

```
Out[88]: AAPL    1.000000
GOOG     0.407919
IBM       0.386817
MSFT     0.389695
dtype: float64
```

```
In [89]: %%html
<h3>Unique values, Value counts, and Membership</h3>
```

## Unique values, Value counts, and Membership

```
In [91]: obj = pd.Series(["c", "a", "d", "a", "a", "b", "b", "c", "c"])
obj.unique()
```

```
Out[91]: array(['c', 'a', 'd', 'b'], dtype=object)
```

```
In [92]: obj.value_counts()
```

```
Out[92]: c    3
a    3
b    2
d    1
Name: count, dtype: int64
```

```
In [93]: # We can use these method with Numpy array
```

```
pd.value_counts(obj.to_numpy())
```

/tmp/ipykernel\_262/2583183724.py:3: FutureWarning: pandas.value\_counts is deprecated and will be removed in a future version. Use pd.Series(obj).value\_counts() instead.

```
pd.value_counts(obj.to_numpy())
```

```
Out[93]: c    3  
        a    3  
        b    2  
        d    1  
        Name: count, dtype: int64
```

```
In [94]: data = pd.DataFrame({"Qu1": [1, 3, 4, 3, 4],  
                             "Qu2": [2, 3, 1, 2, 3],  
                             "Qu3": [1, 5, 2, 4, 4]})  
data
```

```
Out[94]:
```

	Qu1	Qu2	Qu3
0	1	2	1
1	3	3	5
2	4	1	2
3	3	2	4
4	4	3	4

```
In [96]: data["Qu1"].value_counts()
```

```
Out[96]: Qu1  
3    2  
4    2  
1    1  
        Name: count, dtype: int64
```

```
In [98]: # to compute this for all columns, we can use apply method  
data.apply(pd.value_counts).fillna(0)
```

/tmp/ipykernel\_262/4121580047.py:2: FutureWarning: pandas.value\_counts is deprecated and will be removed in a future version. Use pd.Series(obj).value\_counts() instead.

```
data.apply(pd.value_counts).fillna(0)
```

```
Out[98]:
```

	Qu1	Qu2	Qu3
1	1.0	1.0	1.0
2	0.0	2.0	1.0
3	2.0	2.0	0.0
4	2.0	0.0	2.0
5	0.0	0.0	1.0

```
In [100... %%html
<h2>Chapter 6 - Data Loading, Storage, and File formats</h2>

Code examples are taken from <a href="https://github.com/wesm/pydata-book/blob/3rd-edition/ch06.ipynb">https://gith
```

## Chapter 6 - Data Loading, Storage, and File formats


Code examples are taken from <https://github.com/wesm/pydata-book/blob/3rd-edition/ch06.ipynb>


```
In [102... %%html
<h3>Text & binary data loading function in pandas</h3>
```

### text & binary data loading function in pandas

```
In [105... %%html

<img src='images/data_loading2.png', width=500/>
<img src='images/data_loading1.png', width=500/>
```

 No description has been provided for this image

 No description has been provided for this image

```
In [107... df = pd.read_csv('examples/ex1.csv')
df
```

```
Out[107...
```

	a	b	c	d	message
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

```
In [108... df = pd.read_csv('examples/ex1.csv', header=None)
df
```

```
Out[108...
```

	0	1	2	3	4
0	a	b	c	d	message
1	1	2	3	4	hello
2	5	6	7	8	world
3	9	10	11	12	foo

```
In [114... # if there is no header in csv
df = pd.read_csv('examples/ex2.csv', names = ["a", "b", "v", "u", "message"])
df
```

```
Out[114...
```

	a	b	v	u	message
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

```
In [117... # assigning one column as row index
columns = ["a", "b", "v", "u", "message"]
df = pd.read_csv('examples/ex2.csv', names = columns, index_col="message")
df
```



```
Out[117...
```

	a	b	v	u
message				
hello	1	2	3	4
world	5	6	7	8
foo	9	10	11	12


```
In [119... # If we pass multiple columns to the index_col, it will be treated as hierarchical indexing
df = pd.read_csv('examples/csv_mindex.csv', index_col=["key1","key2"])
df
```


```
Out[119...
```

		value1	value2
key1	key2		
one	a	1	2
	b	3	4
	c	5	6
	d	7	8
two	a	9	10
	b	11	12
	c	13	14
	d	15	16

```
In [121... %%html
Pandas.read_csv function arguments
<img src='images/read_csv1.png', width=500/>
<img src='images/read_csv2.png', width=500/>
```

Pandas.read\_csv function arguments

No description has been provided for this image

No description has been provided for this image

In [122... *# Write data to text format*

```
df = pd.read_csv('examples/ex1.csv')
df
```

Out[122...

	a	b	c	d	message
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

In [123... df.to\_csv('examples/out.csv')

In [127... *# Pandas has got read\_json & dump equivalent of json.load() & json.dump()*  
**import** sys

```
df = pd.read_json('examples/example.json')
df
df.to_csv(sys.stdout)
```

```
,a,b,c
0,1,2,3
1,4,5,6
2,7,8,9
```

In [131... *# HTML can be read by using pd.read\_html(). By default, it attempts to find and parse the data in the tabular data*  
*# XML can be parsed using pd.read\_xml()*  
*# Excel files can be read by xlsx = pd.ExcelFile() then data stored on the sheet1 can be read by xlsx.parse(sheet\_n*

In [133... *# Binary format such as pickles can be handled using pd.read\_pickle() and df.to\_pickle()*  
*# Connecting with DB using sqlalchemy library is much easier.*  
*#import sqlalchemy as sqla*

```
#db = sqlalchemy.create_engine("sqlite:///mydata.sqlite")  
#pd.read_sql("SELECT * FROM test", db)
```

In [ ]: