All about Data Engineering

Python Pandas The definitive hands on guide

Swipe Right

Pandas from basic to advanced

Dr. Mohamed Marzouk Sobaih, Postdoctoral Researcher

Twitter | GitHub | GoogleScholar | Website | LinkedIn

Table of Content

1. Introduction

1.1 What is pandas? and Why it is important?

- Pandas offers an extensive array of tools designed for handling tabular data, encompassing tasks such as data cleaning, aggregation, and visualization.
- At the heart of Pandas is its core data structure, the DataFrame (DF), which represents a versatile two-dimensional table with flexible data types and operations.
- This structure facilitates efficient data manipulation, allowing for actions like indexing, slicing, and filtering.
- This notebook constitutes a component of my preparation for the application of machine learning (ML) models for drug design. Within this notebook, I aim to

curate a thorough compilation of commands, spanning from basic to advanced levels, for utilizing the Pandas library.

The version of pandas used in this notebook is '2.1.4'

1.2 What is the version of pandas used in this notebook?

> The version of pandas used in this notebook is '2.1.4'

In [ ]:
```
import pandas as pd
pd.__version__
```

Out[ ]:
```
'2.1.4'
```

2. Pandas' fundamental operations

2.1 Series

- Series in pandas is a one-dimensional labeled array, capable of holding data of any type and facilitating efficient data manipulation.
- We use Series to create 1D arrays or vectors, also known as 1D matrices

2.1.1 Create series

In [ ]:
```
data = pd.Series([0.25, 0.5, 0.75, 1.0])
print(data)
```

```
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64
```

2.1.2 Info. about the series

> When seeking information about our data, we can obtain it through:

- **value** --> to retrieve the values within the dataset.
- **index** --> to ascertain the range of indices for all elements in the dataset.

In [ ]:
```
data = pd.Series((0.25, 0.5, 0.75, 1.0))
# print(data.values)
# print(data.index)
print(data.keys)
```

```
<bound method Series.keys of 0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64>
```

- The `describe` function provides statistical insights into our data, we used to use different function in numpy to obtain such inofrmation, but with pandas employing describe alone is sufficient.
- we have the flexibility to tailor the information obtained from describe using the `agg` command based on our specific requirements.

In [ ]:
```
data = pd.Series((3,6,9,8,5,4,2,6,3,5,8))
print(data.describe())
```

```
count    11.000000
mean      5.363636
std       2.292280
min       2.000000
25%       3.500000
50%       5.000000
75%       7.000000
max       9.000000
dtype: float64
```

In [ ]:
```
data = pd.Series((3,6,9,8,5,4,2,6,3,5,8))
print(data.agg(['max','min','sum','mean','std']))
```

```
max      9.000000
min      2.000000
sum     59.000000
mean     5.363636
std      2.292280
dtype: float64
```

2.1.3 How to access and slice data in a Series?

> 📍 Keep in your mind 🧐 --> data[start : end : steps]

In [ ]:
```
data = pd.Series((0.25, 0.5, 0.75, 1.0))
print(data[1])
print(data[1:3])
print(data[1:3:2])
```

```
0.5
1    0.50
2    0.75
dtype: float64
1    0.5
dtype: float64
```

2.1.4 Customize the index

> How to manually manage and customize the index in Pandas for those who prefer not to use the default index provided by the library ?

- Method 1: List of Index - You can specify the index by providing a list with the number of indexes equal to the number of elements.
- Method 2: Dictionary Approach - You can utilize the dictionary format to define the index.

In [ ]:
```
data1 = pd.Series([1,2,3,4], index=['a', 'b', 'c', 'd'])

data2 = pd.Series({'a':1,'b':2,'c':3,'d':4})

print(data1)
print(data2)
```

```
a    1
b    2
c    3
d    4
dtype: int64
a    1
b    2
c    3
d    4
dtype: int64
```

> Then, we can access or slice by new indexes

In [ ]:
```
import pandas as pd
data1 = pd.Series([1,2,3,4], index=['a', 'b', 'c', 'd'])
data2 = pd.Series({'a':1,'b':2,'c':3,'d':4})
print(data1['a'])
print(data2['b'])
```

```
1
2
```

2.2 List and List operations by Pandas

> In Pandas you can create list by use the `Index` class

In [ ]:
```
x = pd.Index([2,3,5,7,11])
print(x)
```

```
Index([2, 3, 5, 7, 11], dtype='int64')
```

> Then it is easy to deal with it as list and do other operations such as logic gates

In [ ]:
```
import pandas as pd
a = pd.Index([1, 3, 5, 7, 9])
b = pd.Index([2, 3, 5, 7, 11])

print(a)
print(b)
print(a & b)
print(a | b)
print(a ^ b)
```

```
Index([1, 3, 5, 7, 9], dtype='int64')
Index([2, 3, 5, 7, 11], dtype='int64')
Index([0, 3, 5, 7, 9], dtype='int64')
Index([3, 3, 5, 7, 11], dtype='int64')
Index([3, 0, 0, 0, 2], dtype='int64')
```

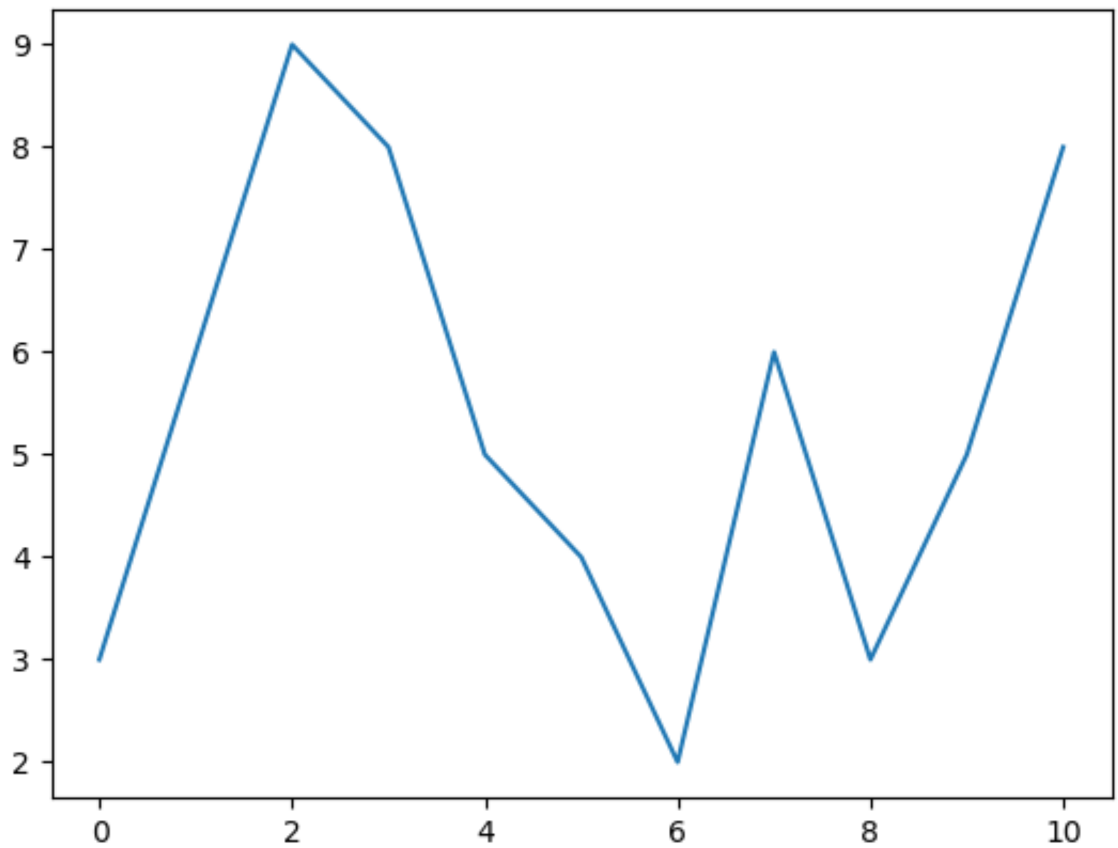
3. Data Visualization with Pandas

Important Notes

- When supplying only the 'y' data, Pandas plot will automatically create the x-axis data as 1, 2, 3, ..., aligning with the quantity of input data on the y-axis.
- The default **kind** parameter in the plot command is set to **line**. Therefore, if you desire a different plot type, you will need to specify it accordingly.

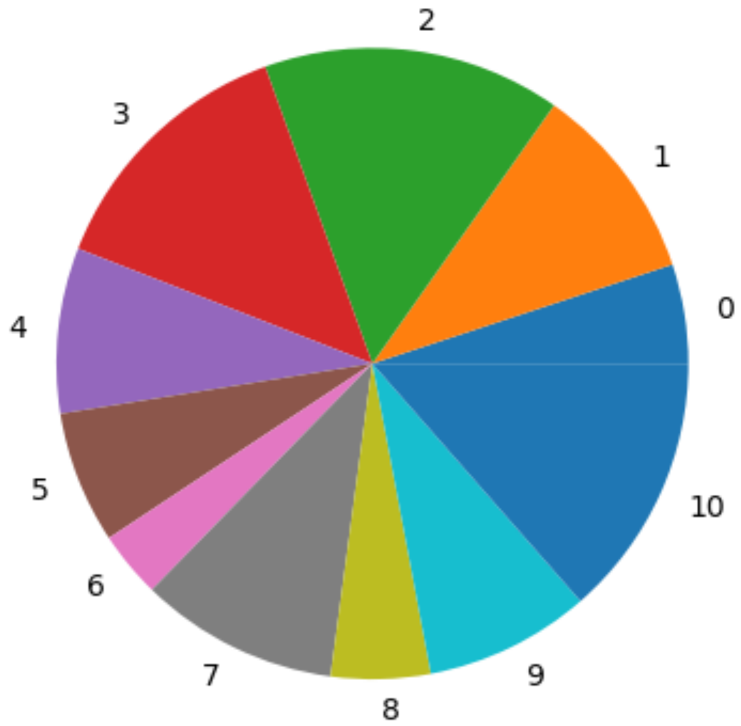
- Plot as line

In [ ]:
```
data = pd.Series((3,6,9,8,5,4,2,6,3,5,8))
# data.plot()
data.plot(kind='line');
```



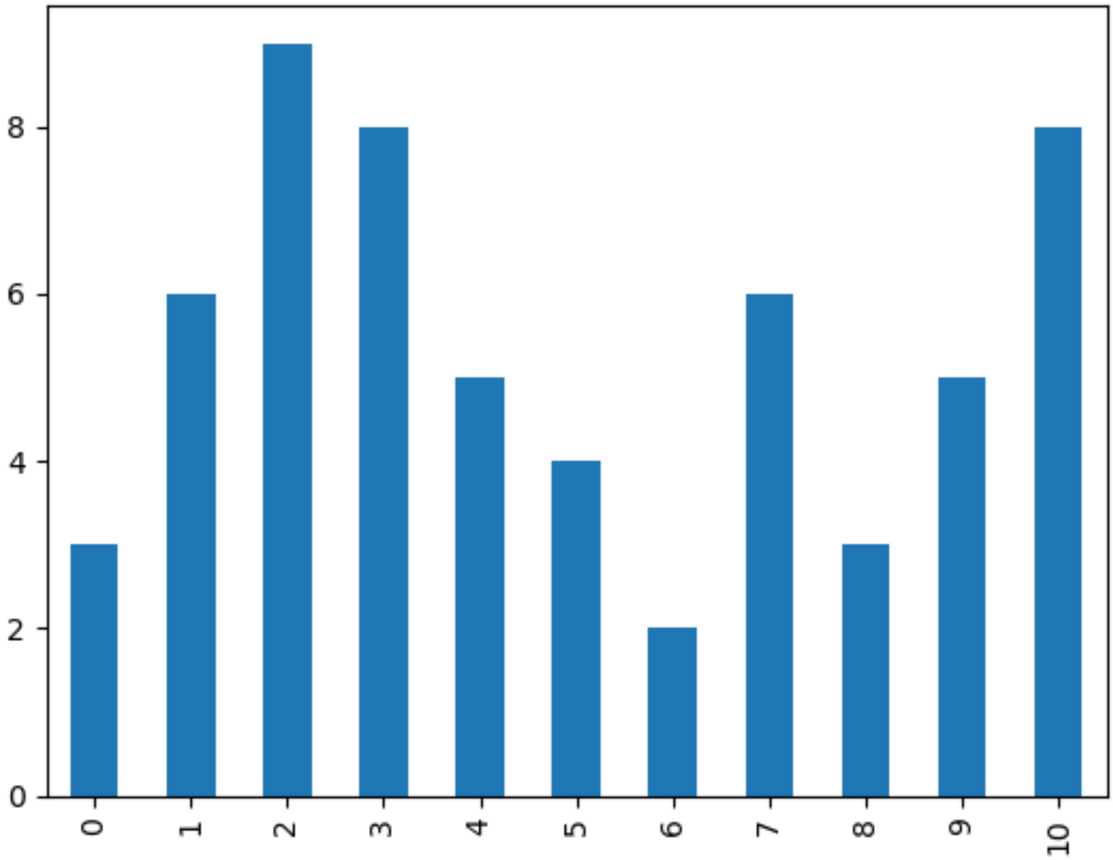
- Plot as Pie

In [ ]:
```
import pandas as pd
data = pd.Series((3,6,9,8,5,4,2,6,3,5,8))
data.plot(kind='pie');
```

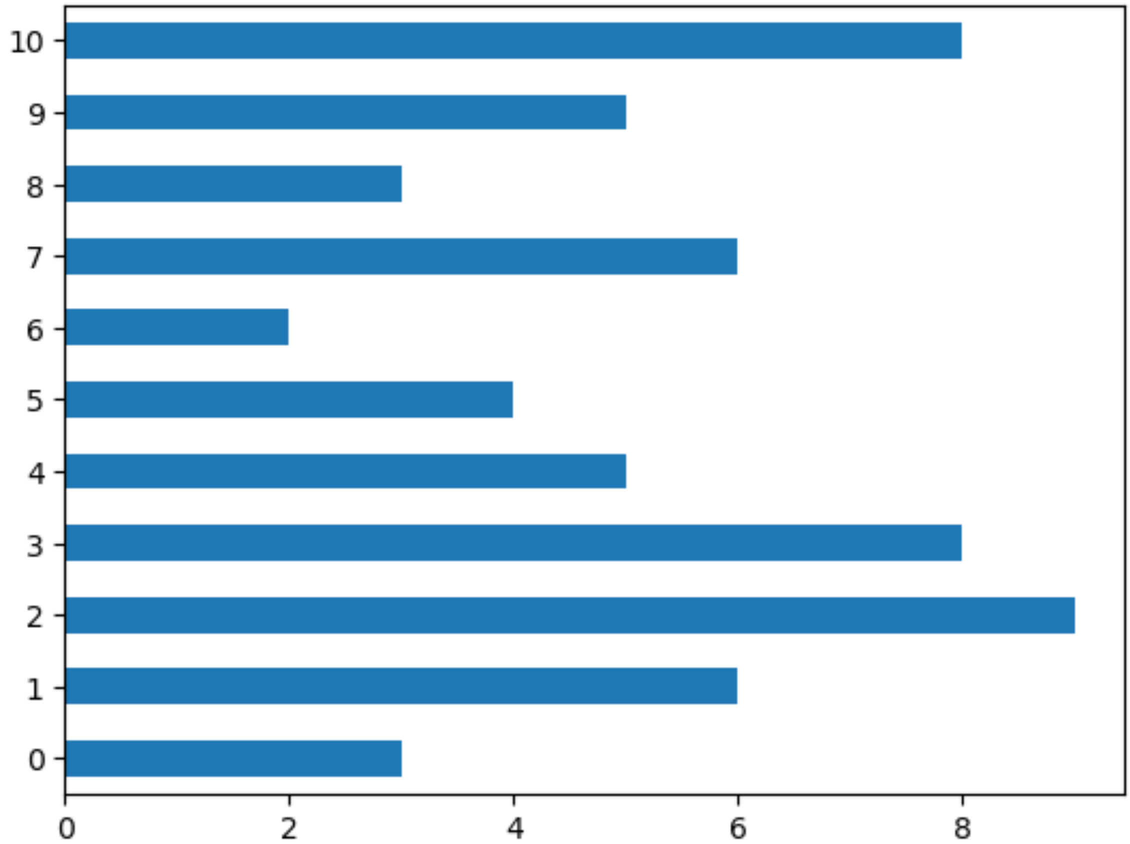


- Plot as Bar

In [ ]:
```
import pandas as pd
data = pd.Series((3,6,9,8,5,4,2,6,3,5,8))
data.plot(kind='bar');
```

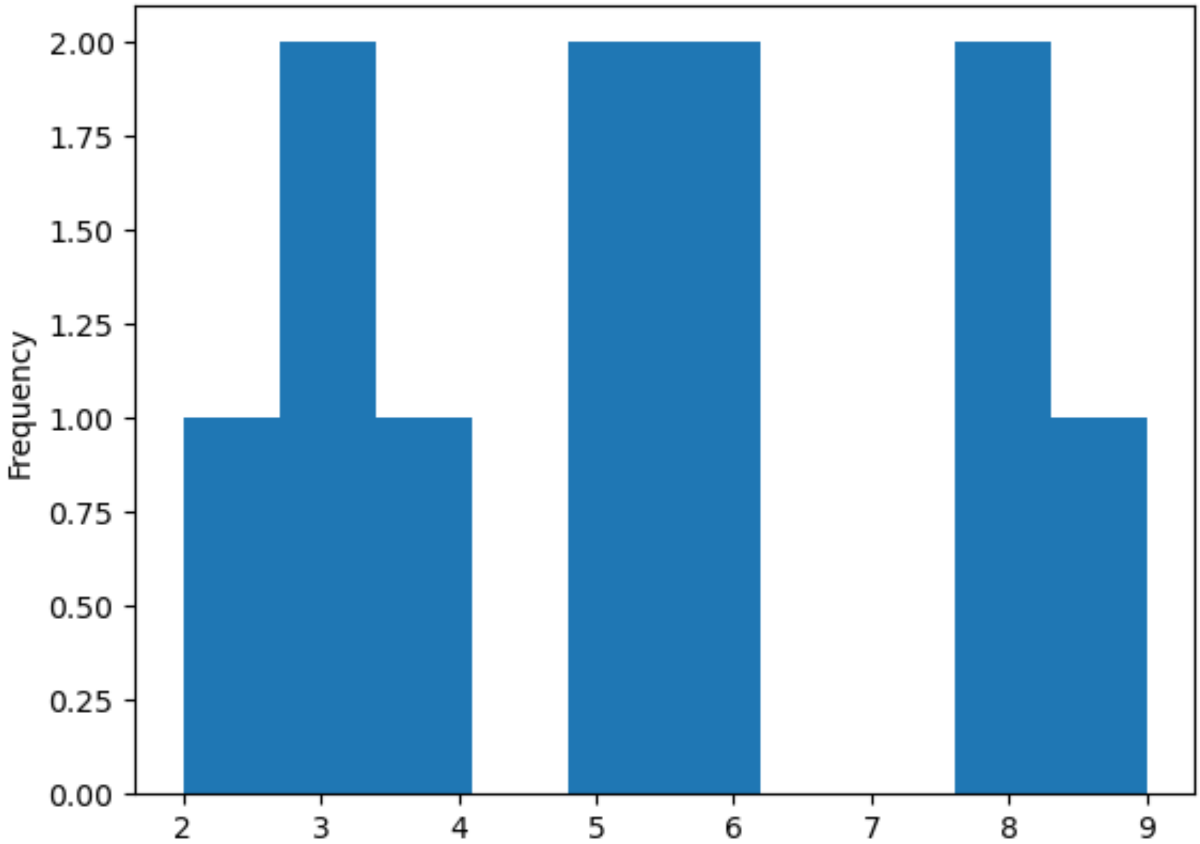


In [ ]:
```
import pandas as pd
data = pd.Series((3,6,9,8,5,4,2,6,3,5,8))
data.plot(kind='barh');
```



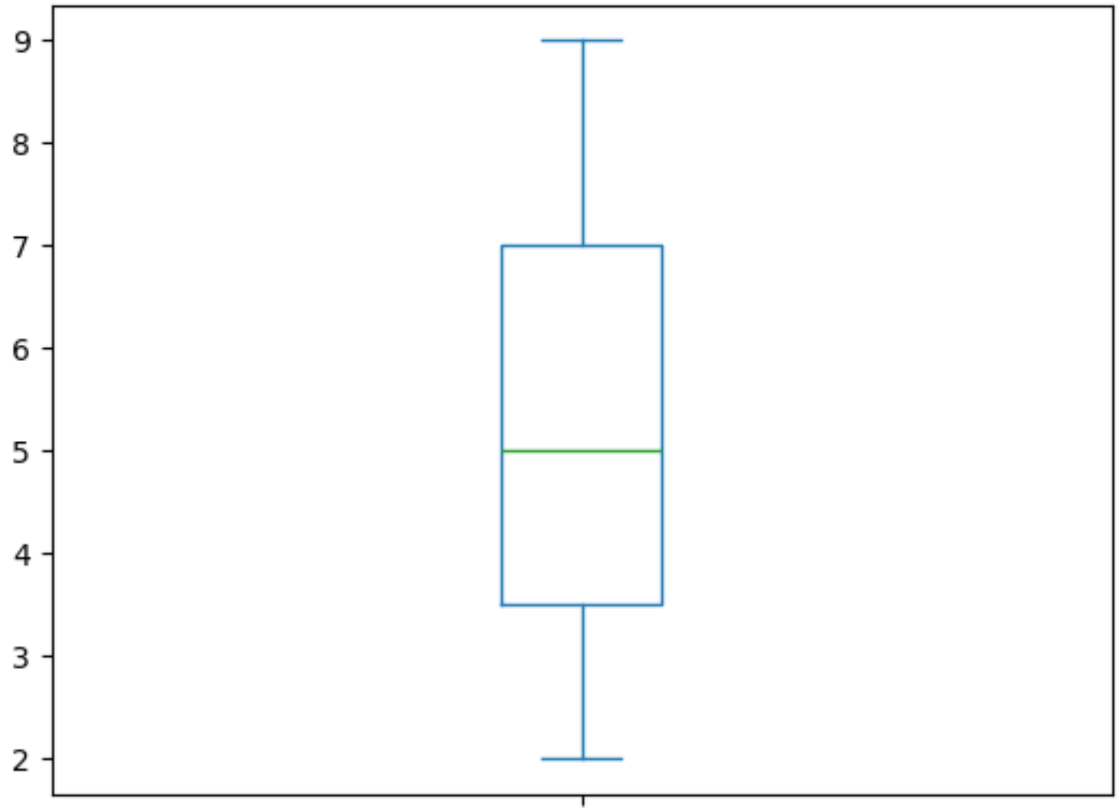
- **Plot as Histogram**
  - The concept behind a histogram is to determine the frequency of each element in the provided list.
  - It requires a single column as input to analyze and display the distribution.

In [ ]:
```
import pandas as pd
data = pd.Series((3,6,9,8,5,4,2,6,3,5,8))
data.plot(kind='hist');
```



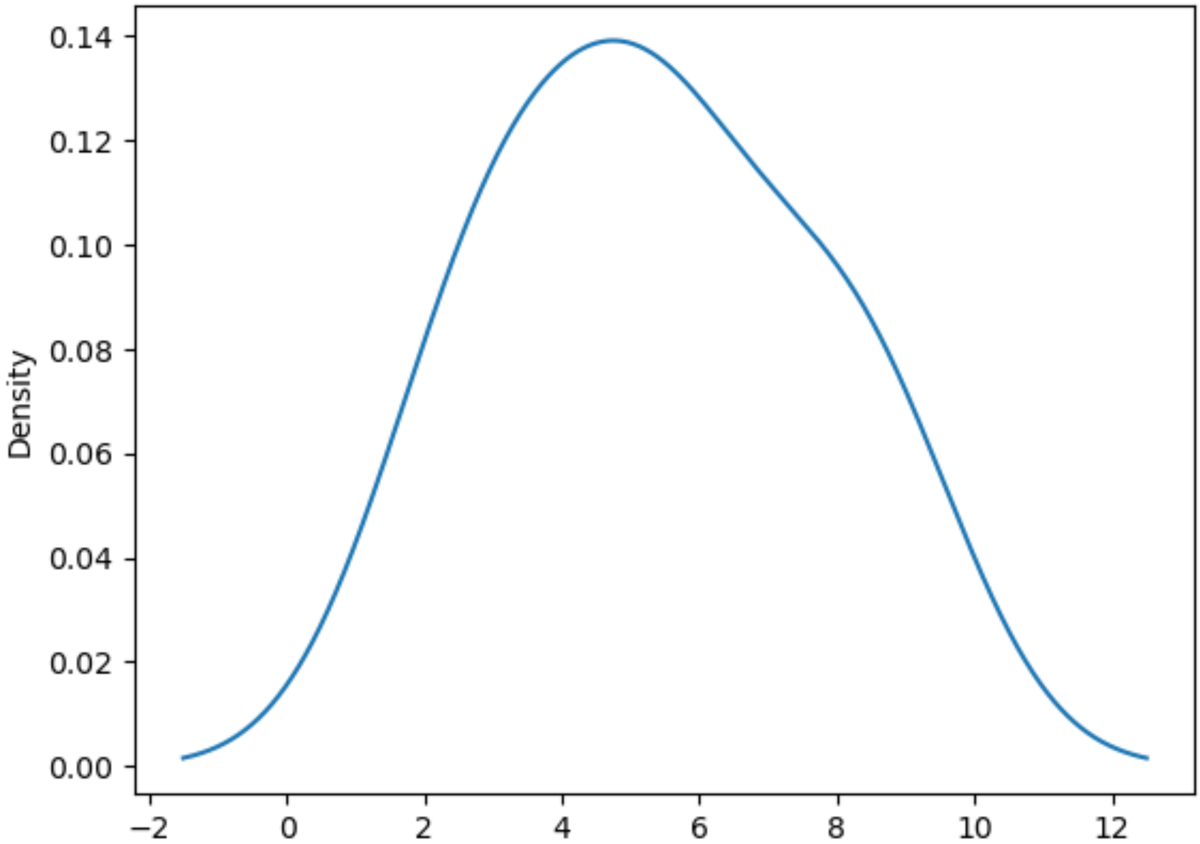
- Plot as box
  - The box plotted based on the maximum, lower, and median for the given data

In [ ]:
```
data = pd.Series((3,6,9,8,5,4,2,6,3,5,8))
data.plot(kind='box');
```



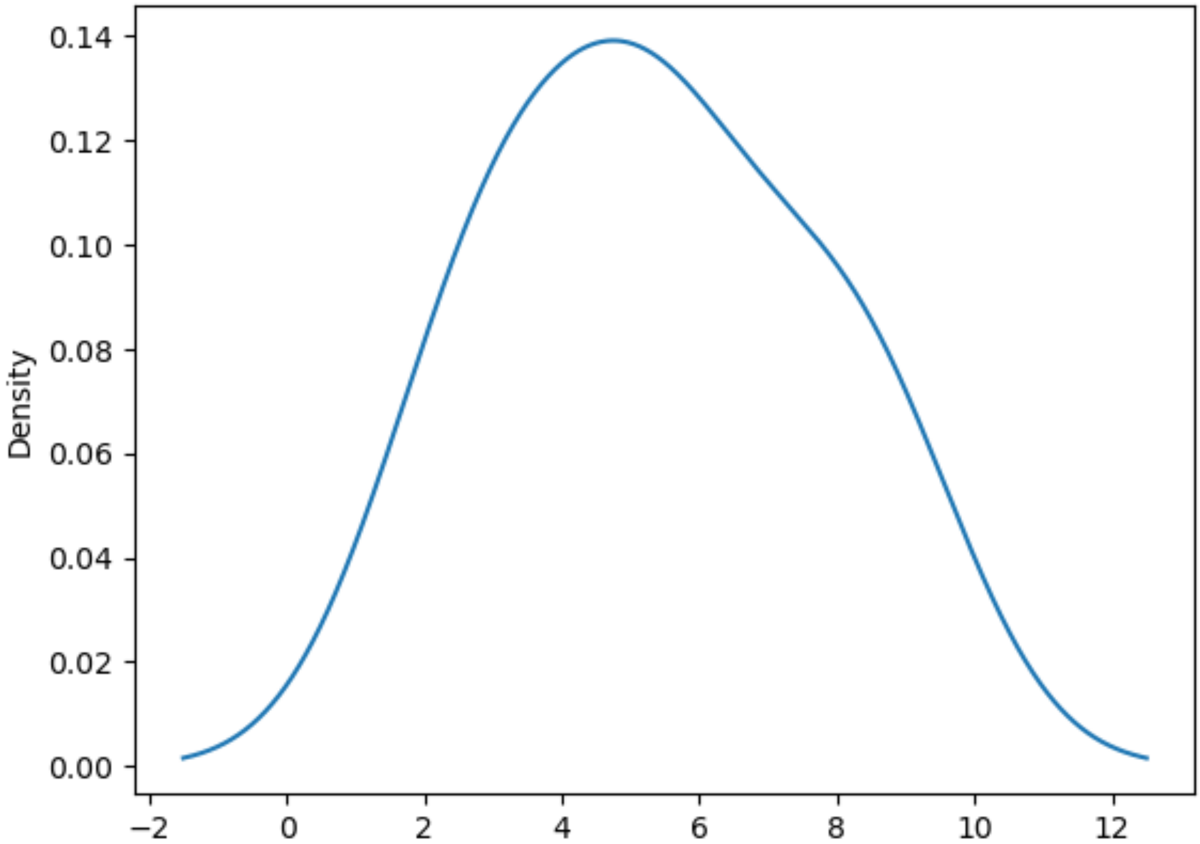
- **Plot by KDE**
  - It is something related to the density distribution from the number of repetition of each value

In [ ]:
```
data = pd.Series((3,6,9,8,5,4,2,6,3,5,8))
data.plot(kind='kde');
```



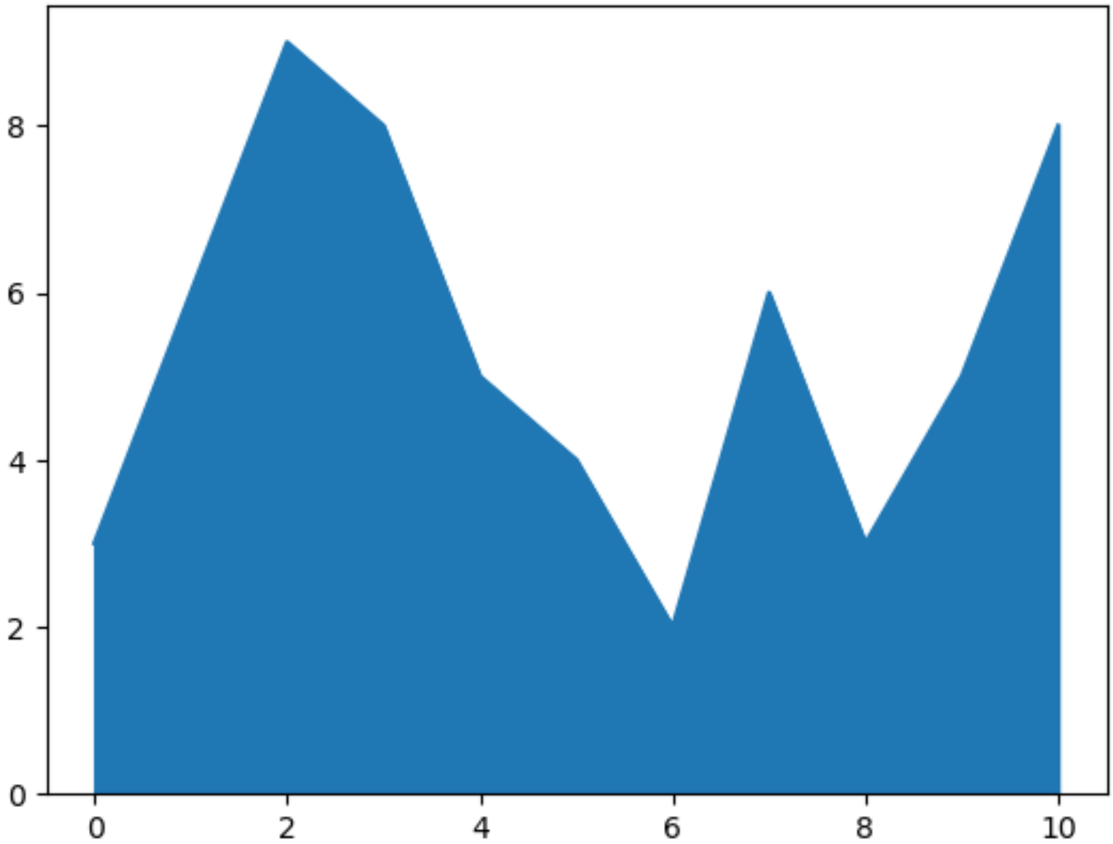
In [ ]:
```
import pandas as pd
data = pd.Series((3,6,9,8,5,4,2,6,3,5,8))

data.plot(kind='density');
```



- Plot by the area under the curve
  - We can plot the area under the curve for our data by the **area** kind in the plot command

In [ ]:
```
data = pd.Series((3,6,9,8,5,4,2,6,3,5,8))
data.plot(kind='area');
```



4. DataFrames

One of the crucial roles played by Pandas is through its DataFrame tool, specifically designed for working with tables and managing **Big Data** in **Data Science**

4.1 Creating DataFrame

- There are various ways to create Data frames (DF) in pandas, including using arrays and series.
- We will explore each method in this notebook, but you will notice that the DF class requires several optional parameters, such as index, column names, and others.
- However, one parameter is mandatory— the data itself (either an array or a series)

4.1.1 Creating DF from array

It's essential to note that the length of the index list should match the number of rows in the data, and similarly, the length of the columns list should correspond to the number of columns in the provided data.

In [ ]:
```
import pandas as pd
import numpy as np
myarray = np.array([[6,9,8,5,4,2],[0,2,5,6,3,9],
                    [8,5,4,1,2,3],[6,9,8,5,4,2],
                    [0,5,3,6,9,8],[8,7,4,5,2,3]])
row_names = ['a', 'b','c','d','e','f']
col_names = ['one', 'two', 'three','four','five','six']
df = pd.DataFrame(myarray, index=row_names, columns=col_names)
print(df)
```

```
   one  two  three  four  five  six
a    6    9      8     5     4    2
b    0    2      5     6     3    9
c    8    5      4     1     2    3
d    6    9      8     5     4    2
e    0    5      3     6     9    8
f    8    7      4     5     2    3
```

4.1.2 Creating DF from list

In [ ]:
```
import pandas as pd

# Example list of lists
data_list = [
    ['Alice', 25, 'Engineer'],
    ['Bob', 30, 'Data Scientist'],
    ['Charlie', 28, 'Designer']
]

# Define column names
columns = ['Name', 'Age', 'Occupation']

# Create DataFrame
df = pd.DataFrame(data_list, columns=columns)

# Display the DataFrame
print(df)
```

```
      Name  Age      Occupation
0    Alice   25        Engineer
1      Bob   30  Data Scientist
2  Charlie   28        Designer
```

4.1.3 Create DF from many Series

As previously mentioned, a Series is essentially a 1D matrix. If you have multiple Series, you can combine them to create a DataFrame.

In [ ]:
```
import pandas as pd
w = pd.Series({'a':1 ,'b':2 ,'c':3 ,'d':4 ,'e':5})
x = pd.Series({'a':6 ,'b':7 ,'c':8 ,'d':9 ,'e':10})
```

```
y = pd.Series({'a':11 ,'b':12 ,'c':13 ,'d':14 ,'e':15})
z = pd.Series({'a':16 ,'b':17 ,'c':18 ,'d':19 ,'e':20})

grades = pd.DataFrame({'Math':w,'Physics':x,'French':y,'Chemistry':z})

print(grades)
```

```
   Math  Physics  French  Chemistry
a     1        6      11         16
b     2        7      12         17
c     3        8      13         18
d     4        9      14         19
e     5       10      15         20
```

> Alternatively, in Pandas, you can use a list to serve as the index for your DataFrame.

In [ ]:

```
import pandas as pd
data1 = pd.Index([3,5,6,0,18,38,48,54,3,5])
data2 = pd.Index([13,15,16,10,118,138,148,154,13,15])
row_names = ["Row no 1 ", "Row no 2 ", "Row no 3 ", "Row no 4 ", "Row no
df= pd.DataFrame({"Column No 1":data1,"Column No 2":data2},index=row_name
df
```

Out[ ]:

	Column No 1	Column No 2
Row no 1	3	13
Row no 2	5	15
Row no 3	6	16
Row no 4	0	10
Row no 5	18	118
Row no 6	38	138
Row no 7	48	148
Row no 8	54	154
Row no 9	3	13
Row no 10	5	15

4.1.4 Creating DF by Dictionary Method

We can create DF by dictionary methods column by column and using the conditions for fill the values

In [ ]:

```
import pandas as pd
```

```
data = [{'square': i**2} for i in range(10)]
df = pd.DataFrame(data)
print(df)
```

```
   square
0       0
1       1
2       4
3       9
4      16
5      25
6      36
7      49
8      64
9      81
```

In [ ]:
```
import pandas as pd
data = [{'square': i**2,'cube': i**3
         ,'root': i**0.5} for i in range(10)]

df = pd.DataFrame(data)

print(df)
```

```
   square  cube      root
0       0     0  0.000000
1       1     1  1.000000
2       4     8  1.414214
3       9    27  1.732051
4      16    64  2.000000
5      25   125  2.236068
6      36   216  2.449490
7      49   343  2.645751
8      64   512  2.828427
9      81   729  3.000000
```

In [ ]:
```
import pandas as pd
d = pd.DataFrame([{'a':1,'b':2},{'a':3,'b':4},{'a':5,'b':6}])

print(d)
```

```
   a  b
0  1  2
1  3  4
2  5  6
```

> Note : if there is value missing for any element will add Nan instead

In [ ]:
```
import pandas as pd

d = pd.DataFrame([{'a':1,'b':2},{'b':3,'c':4},{'d':5,'e':6}])

print(d)
```

```
     a    b    c    d    e
0  1.0  2.0  NaN  NaN  NaN
1  NaN  3.0  4.0  NaN  NaN
2  NaN  NaN  NaN  5.0  6.0
```

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

d =pd.DataFrame(np.random.rand(3, 2),
                columns=['food', 'drink'],index=['a', 'b', 'c'])
print(d)
```

```
       food     drink
a  0.903192  0.006704
b  0.222576  0.001998
c  0.236427  0.224673
```

4.1.5 Creating DF by function (advanced method)

In [ ]:
```
import pandas as pd

def make_df(cols, ind):
    data = {c: [str(c) + str(i) for i in ind] for c in cols}
    return pd.DataFrame(data, ind)

print(make_df('ABC', range(3)))
```

```
    A   B   C
0  A0  B0  C0
1  A1  B1  C1
2  A2  B2  C2
```

4.2 Basic Operations on DataFrame

> You can perform various operations on a DataFrame, including slicing and handling a multitude of tasks.

4.2.1 Slicing or accessing columns in the DF

When you retrieve a single column, it includes the index. If you prefer excluding the index, you can utilize the keys and values options to precisely customize the output based on your requirements.

In [ ]:
```
import pandas as pd
w = pd.Series({'a':1 ,'b':2 ,'c':3 ,'d':4 ,'e':5})
x = pd.Series({'a':6 ,'b':7 ,'c':8 ,'d':9 ,'e':10})
y = pd.Series({'a':11 ,'b':12 ,'c':13 ,'d':14 ,'e':15})
z = pd.Series({'a':16 ,'b':17 ,'c':18 ,'d':19 ,'e':20})
```

```
grades = pd.DataFrame({'Math':w,'Physics':x,'French':y,'Chemistry':z})
print(grades)
print(grades['Chemistry'])
```

```
   Math  Physics  French  Chemistry
a     1        6      11         16
b     2        7      12         17
c     3        8      13         18
d     4        9      14         19
e     5       10      15         20
a    16
b    17
c    18
d    19
e    20
Name: Chemistry, dtype: int64
```

4.2.2 Transpose the DF

In [ ]:
```
import pandas as pd
w = pd.Series({'a':1 ,'b':2 ,'c':3 ,'d':4 ,'e':5})
x = pd.Series({'a':6 ,'b':7 ,'c':8 ,'d':9 ,'e':10})
y = pd.Series({'a':11 ,'b':12 ,'c':13 ,'d':14 ,'e':15})
z = pd.Series({'a':16 ,'b':17 ,'c':18 ,'d':19 ,'e':20})

grades = pd.DataFrame({'Math':w,'Physics':x,'French':y,'Chemistry':z})

print(grades.T)
```

```
            a   b   c   d   e
Math        1   2   3   4   5
Physics     6   7   8   9  10
French     11  12  13  14  15
Chemistry  16  17  18  19  20
```

4.2.3 Accessing only keys or values from the DF

In [ ]:
```
import pandas as pd

w = pd.Series({'a':1 ,'b':2 ,'c':3 ,'d':4 ,'e':5})
x = pd.Series({'a':6 ,'b':7 ,'c':8 ,'d':9 ,'e':10})
y = pd.Series({'a':11 ,'b':12 ,'c':13 ,'d':14 ,'e':15})
z = pd.Series({'a':16 ,'b':17 ,'c':18 ,'d':19 ,'e':20})

grades = pd.DataFrame({'Math':w,'Physics':x,'French':y,'Chemistry':z})
print(grades.keys())
print(grades.values)
```

```
Index(['Math', 'Physics', 'French', 'Chemistry'], dtype='object')
[[ 1  6 11 16]
 [ 2  7 12 17]
 [ 3  8 13 18]
 [ 4  9 14 19]
 [ 5 10 15 20]]
```

4.2.4 Applying conditions to specific keys or values in the DF

> **Note** : Keys --> Columns header

In [ ]:
```
import pandas as pd

w = pd.Series({'a':1 ,'b':2 ,'c':3 ,'d':4 ,'e':5})
x = pd.Series({'a':6 ,'b':7 ,'c':8 ,'d':9 ,'e':10})
y = pd.Series({'a':11 ,'b':12 ,'c':13 ,'d':14 ,'e':15})
z = pd.Series({'a':16 ,'b':17 ,'c':18 ,'d':19 ,'e':20})

grades = pd.DataFrame({'Math':w,'Physics':x,'French':y,'Chemistry':z})

print('Math' in grades.keys())
print('math' in grades.keys())
print(12 in grades.values)
print(55 in grades.values)
```

```
True
False
True
False
```

4.2.5 Vertical representation of elements with keys and values

In [ ]:
```
import pandas as pd

w = pd.Series({'a':1 ,'b':2 ,'c':3 ,'d':4 ,'e':5})
x = pd.Series({'a':6 ,'b':7 ,'c':8 ,'d':9 ,'e':10})
y = pd.Series({'a':11 ,'b':12 ,'c':13 ,'d':14 ,'e':15})
z = pd.Series({'a':16 ,'b':17 ,'c':18 ,'d':19 ,'e':20})

grades = pd.DataFrame({'Math':w,'Physics':x,'French':y,'Chemistry':z})

print(grades.stack())
```

```
a  Math          1
   Physics       6
   French       11
   Chemistry    16
b  Math          2
   Physics       7
   French       12
   Chemistry    17
c  Math          3
   Physics       8
   French       13
   Chemistry    18
d  Math          4
   Physics       9
   French       14
   Chemistry    19
e  Math          5
   Physics      10
   French       15
   Chemistry    20
dtype: int64
```

4.2.6 Locating specific elements for slicing and searching within the DF

We have two primary methods for this task:

Method 1: iloc (i for index) - It locates the position by index, similar to the method we are familiar with in lists.

Example: --> iloc[:3, :2]

Method 2: loc - You need to specify the names of the elements, rows, and columns you are searching for.

Example: --> loc["b":"c", "Math":] This implies selecting rows from 'b' to 'c' and columns from 'Math' to the end.

Note: In this method, you need to reference columns by their names. If the index is numeric, you can use numbers.

Example: df.loc[3:6, : "Square of x"]

In [ ]:
```
import pandas as pd

w = pd.Series({'a':1 ,'b':2 ,'c':3 ,'d':4 ,'e':5})
x = pd.Series({'a':6 ,'b':7 ,'c':8 ,'d':9 ,'e':10})
y = pd.Series({'a':11 ,'b':12 ,'c':13 ,'d':14 ,'e':15})
z = pd.Series({'a':16 ,'b':17 ,'c':18 ,'d':19 ,'e':20})

grades = pd.DataFrame({'Math':w,'Physics':x,'French':y,'Chemistry':z})
```

```
print(grades.iloc[:3, :2])
```

```
   Math  Physics
a     1        6
b     2        7
c     3        8
```

In [ ]:
```
import pandas as pd

w = pd.Series({'a':1 ,'b':2 ,'c':3 ,'d':4 ,'e':5})
x = pd.Series({'a':6 ,'b':7 ,'c':8 ,'d':9 ,'e':10})
y = pd.Series({'a':11 ,'b':12 ,'c':13 ,'d':14 ,'e':15})
z = pd.Series({'a':16 ,'b':17 ,'c':18 ,'d':19 ,'e':20})

grades = pd.DataFrame({'Math':w,'Physics':x,'French':y,'Chemistry':z})

print(grades.loc['b':'c', 'Math':])
```

```
   Math  Physics  French  Chemistry
b     2        7      12         17
c     3        8      13         18
```

> You can apply conditions with **loc** command

In [ ]:
```
import pandas as pd

w = pd.Series({'a':1 ,'b':2 ,'c':3 ,'d':4 ,'e':5})
x = pd.Series({'a':6 ,'b':7 ,'c':8 ,'d':9 ,'e':10})
y = pd.Series({'a':11 ,'b':12 ,'c':13 ,'d':14 ,'e':15})
z = pd.Series({'a':16 ,'b':17 ,'c':18 ,'d':19 ,'e':20})

grades = pd.DataFrame({'Math':w,'Physics':x,'French':y,'Chemistry':z})

print(grades.loc[grades.Math >3])
```

```
   Math  Physics  French  Chemistry
d     4        9      14         19
e     5       10      15         20
```

> Applying conditions to rows and selecting specific columns for display using the loc command.

In [ ]:
```
import pandas as pd
w = pd.Series({'a':1 ,'b':2 ,'c':3 ,'d':4 ,'e':5})
x = pd.Series({'a':6 ,'b':7 ,'c':8 ,'d':9 ,'e':10})
y = pd.Series({'a':11 ,'b':12 ,'c':13 ,'d':14 ,'e':15})
z = pd.Series({'a':16 ,'b':17 ,'c':18 ,'d':19 ,'e':20})

grades = pd.DataFrame({'Math':w,'Physics':x,'French':y,'Chemistry':z})

print(grades.loc[grades.Math >3,['French' ,'Math']])
```

```
   French  Math
d      14     4
e      15     5
```

4.2.7 Access names of all columns and the index in the DF

In [ ]:
```
import pandas as pd

w = pd.Series({'a':1 ,'b':2 ,'c':3 ,'d':4 ,'e':5})
x = pd.Series({'a':6 ,'b':7 ,'c':8 ,'d':9 ,'e':10})
y = pd.Series({'a':11 ,'b':12 ,'c':13 ,'d':14 ,'e':15})
z = pd.Series({'a':16 ,'b':17 ,'c':18 ,'d':19 ,'e':20})

grades = pd.DataFrame({'Math':w,'Physics':x,'French':y,'Chemistry':z})

print(grades.columns)   ## same as df.keys()
print(grades.index)
```

```
Index(['Math', 'Physics', 'French', 'Chemistry'], dtype='object')
Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
```

In [ ]:
```
import pandas as pd

w = pd.Series({'a':1 ,'b':2 ,'c':3 ,'d':4 ,'e':5})
x = pd.Series({'a':6 ,'b':7 ,'c':8 ,'d':9 ,'e':10})
y = pd.Series({'a':11 ,'b':12 ,'c':13 ,'d':14 ,'e':15})
z = pd.Series({'a':16 ,'b':17 ,'c':18 ,'d':19 ,'e':20})

grades = pd.DataFrame({'Math':w,'Physics':x,'French':y,'Chemistry':z})

print(grades['Math'])
```

```
a    1
b    2
c    3
d    4
e    5
Name: Math, dtype: int64
```

> Example: Changing the index to one of the columns in the DF

In [ ]:
```
import pandas as pd
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering',
                              'Engineering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date': [2004, 2008, 2012, 2014]})

df3 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'salary': [70000, 80000, 120000, 90000]})

print(df1)
```

```
df2 = df1.set_index('employee')

print(df2)
```

```
  employee        group
0      Bob   Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue           HR
                group
employee
Bob        Accounting
Jake      Engineering
Lisa      Engineering
Sue                HR
```

4.2.8 Sort the elements in the DF

Sorting the DataFrame based on the values of a specific column in ascending or descending order (using the parameter **ascending=False** if needed).

In [ ]:
```
import pandas as pd

w = pd.Series({'a':1 ,'b':2 ,'c':3 ,'d':4 ,'e':5})
x = pd.Series({'a':6 ,'b':7 ,'c':8 ,'d':9 ,'e':10})
y = pd.Series({'a':11 ,'b':12 ,'c':13 ,'d':14 ,'e':15})
z = pd.Series({'a':16 ,'b':17 ,'c':18 ,'d':19 ,'e':20})

grades = pd.DataFrame({'Math':w,'Physics':x,'French':y,'Chemistry':z})

print(grades.sort_values(['Math'],ascending= False))
print(grades.sort_values(['French'],ascending= True))
```

```
   Math  Physics  French  Chemistry
e     5       10      15         20
d     4        9      14         19
c     3        8      13         18
b     2        7      12         17
a     1        6      11         16
   Math  Physics  French  Chemistry
a     1        6      11         16
b     2        7      12         17
c     3        8      13         18
d     4        9      14         19
e     5       10      15         20
```

4.2.9 Statistics for the entire DF or per column

In [ ]:
```
import pandas as pd
```

```
w = pd.Series({'a':1 ,'b':2 ,'c':3 ,'d':4 ,'e':5})
x = pd.Series({'a':6 ,'b':7 ,'c':8 ,'d':9 ,'e':10})
y = pd.Series({'a':11 ,'b':12 ,'c':13 ,'d':14 ,'e':15})
z = pd.Series({'a':16 ,'b':17 ,'c':18 ,'d':19 ,'e':20})

grades = pd.DataFrame({'Math':w,'Physics':x,'French':y,'Chemistry':z})

print("The max. for DF all columns\n", grades.max())
print("The min. for DF all columns\n ",grades.min())
print("The mean for DF all columns\n ",grades.mean())
print("The std. for DF all columns\n ",grades.std())
```

```
The max. for DF all columns
 Math          5
Physics      10
French       15
Chemistry    20
dtype: int64
The min. for DF all columns
  Math          1
Physics       6
French       11
Chemistry    16
dtype: int64
The mean for DF all columns
  Math          3.0
Physics       8.0
French       13.0
Chemistry    18.0
dtype: float64
The std. for DF all columns
  Math         1.581139
Physics      1.581139
French       1.581139
Chemistry    1.581139
dtype: float64
```

In [ ]:
```
import pandas as pd

w = pd.Series({'a':1 ,'b':2 ,'c':3 ,'d':4 ,'e':5})
x = pd.Series({'a':6 ,'b':7 ,'c':8 ,'d':9 ,'e':10})
y = pd.Series({'a':11 ,'b':12 ,'c':13 ,'d':14 ,'e':15})
z = pd.Series({'a':16 ,'b':17 ,'c':18 ,'d':19 ,'e':20})

grades = pd.DataFrame({'Math':w,'Physics':x,'French':y,'Chemistry':z})

print(grades['Math'].max())
print(grades['French'].min())
print(grades['Physics'].mean())
print(grades['Chemistry'].std())
```

```
5
11
8.0
1.5811388300841898
```

4.2.10 The correlation between elements in the DF

When the numbers are closely aligned, the correlation tends to approach 1, and conversely, when they are distant, the correlation tends to be closer to -1.

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

df = pd.DataFrame(np.random.rand(5, 3), columns=['A', 'B', 'C'])

print(df)
print(df.corr())
```

```
          A         B         C
0  0.124522  0.802319  0.508907
1  0.082320  0.473766  0.173403
2  0.050012  0.651602  0.569051
3  0.926583  0.744118  0.619364
4  0.572466  0.403565  0.449702
          A         B         C
A  1.000000  0.045352  0.455363
B  0.045352  1.000000  0.637697
C  0.455363  0.637697  1.000000
```

4.2.11 The skewness among the elements of each column in the DF

The `df.skew()` function yields a Series, providing one value for each column in the DataFrame. These Series values represent the skewness of each corresponding column.

> Skewness == انحراف

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

df = pd.DataFrame(np.random.rand(5, 3), columns=['A', 'B', 'C'])

print(df)
print(df.skew())
```

```
          A         B         C
0  0.235355  0.389166  0.181239
1  0.726775  0.072747  0.996716
2  0.351588  0.133315  0.599001
3  0.510098  0.833063  0.328395
4  0.219143  0.561235  0.131055
A    0.914599
B    0.460093
C    1.073441
dtype: float64
```

4.2.12 Arithmetic operations between Columns

> We have two options:

- Applying the arithmetic operation directly
- Utilizing the DataFrame method (`eval`).

> However, please note that when using the pandas method, you need to specify the operation as text.

In [ ]:
```python
import pandas as pd

w = pd.Series({'a':1 ,'b':2 ,'c':3 ,'d':4 ,'e':5})
x = pd.Series({'a':6 ,'b':7 ,'c':8 ,'d':9 ,'e':10})
y = pd.Series({'a':11 ,'b':12 ,'c':13 ,'d':14 ,'e':15})
z = pd.Series({'a':16 ,'b':17 ,'c':18 ,'d':19 ,'e':20})

grades = pd.DataFrame({'Math':w,'Physics':x,'French':y,'Chemistry':z})

grades['Total'] = (grades['Math'] + grades['French'] +
      grades['Chemistry']+ grades['Physics']) /100


print(grades)
```

```
   Math  Physics  French  Chemistry  Total
a     1        6      11         16   0.34
b     2        7      12         17   0.38
c     3        8      13         18   0.42
d     4        9      14         19   0.46
e     5       10      15         20   0.50
```

> Performing calculations through direct arithmetic operations

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

df = pd.DataFrame(np.random.rand(5, 3), columns=['A', 'B', 'C'])
```

```python
result = (df['A'] + df['B']) / (df['C'] - 1)

print(df)
print(result)
```

```
          A         B         C
0  0.997893  0.933147  0.630340
1  0.013309  0.722930  0.126046
2  0.372473  0.378301  0.846271
3  0.700943  0.540816  0.077703
4  0.864855  0.524521  0.439235
0   -5.223823
1   -0.842422
2   -4.883745
3   -1.346377
4   -2.477642
dtype: float64
```

> Performing calculations using the evaluation method of Pandas

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

df = pd.DataFrame(np.random.rand(5, 3), columns=['A', 'B', 'C'])

result = pd.eval("(df.A + df.B) / (df.C - 1)")

print(df)
print(result)
```

```
          A         B         C
0  0.192101  0.036617  0.135488
1  0.810343  0.320694  0.475438
2  0.023259  0.276737  0.777996
3  0.399341  0.928936  0.560557
4  0.988831  0.058670  0.399062
0   -0.264563
1   -2.156157
2   -1.351307
3   -3.022635
4   -1.743111
dtype: float64
```

4.2.13 Selecting a Specific Row by Conditions

> Similarly, for selection, you can either do it manually or use the Pandas method (**query**).

- However, when utilizing the Pandas method, remember to express the operation as text.

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

df = pd.DataFrame(np.random.rand(5, 3), columns=['A', 'B', 'C'])

result = df.query('A < 0.5 and B < 0.5')

print(df)
print(result)
```

```
          A         B         C
0  0.869640  0.652818  0.211815
1  0.549392  0.466400  0.347395
2  0.319789  0.239428  0.212668
3  0.052973  0.569942  0.686194
4  0.730785  0.442917  0.276722
          A         B         C
2  0.319789  0.239428  0.212668
```

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

df = pd.DataFrame(np.random.rand(5, 3), columns=['A', 'B', 'C'])

tmp1 = df.A < 0.5
tmp2 = df.B < 0.5
tmp3 = tmp1 & tmp2
result1 = df[tmp3]
result2 = df[(df.A < 0.5) & (df.B < 0.5)]
result3 = df[(df.A < 0.5) | (df.B < 0.5)]

print(df)
print(result1)
print(result2)
print(result3)
```

```
          A         B         C
0  0.891539  0.142778  0.229673
1  0.578983  0.223848  0.746335
2  0.114691  0.885383  0.815843
3  0.628647  0.439630  0.800615
4  0.022767  0.307606  0.918541
          A         B         C
4  0.022767  0.307606  0.918541
          A         B         C
4  0.022767  0.307606  0.918541
          A         B         C
0  0.891539  0.142778  0.229673
1  0.578983  0.223848  0.746335
2  0.114691  0.885383  0.815843
3  0.628647  0.439630  0.800615
4  0.022767  0.307606  0.918541
```

Note here : this sign "**|**" means **"or"**

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

df = pd.DataFrame(np.random.rand(5, 3), columns=['A', 'B', 'C'])

result = df[(df.A < 0.5) | (df.B < 0.5)]

print(df)
print(result)
```

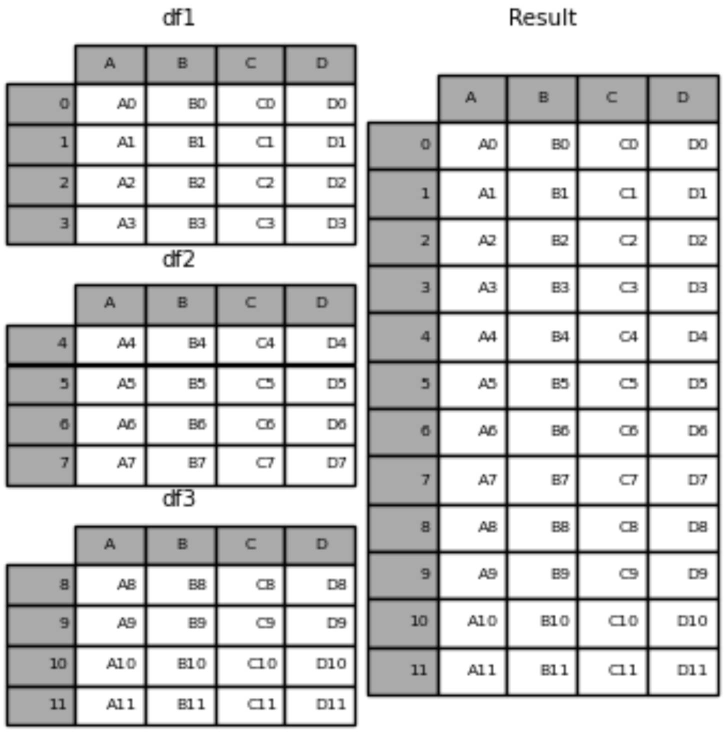
```
          A         B         C
0  0.551104  0.540216  0.482121
1  0.607297  0.441072  0.829768
2  0.360749  0.134419  0.994606
3  0.928518  0.858054  0.772040
4  0.328330  0.648579  0.807195
          A         B         C
1  0.607297  0.441072  0.829768
2  0.360749  0.134419  0.994606
4  0.328330  0.648579  0.807195
```

4.3 Table Concatenation and Merging

4.3.1 Simple Concatenation

The `concat()` function does all of the heavy lifting of performing concatenation operations along an axis while performing optional set logic (union or intersection) of the indexes (if any) on the other axes.

> Concatenation simple example



How Concatenation working

In [ ]:
```python
df1 = pd.DataFrame(
    {
        "A": ["A0", "A1", "A2", "A3"],
        "B": ["B0", "B1", "B2", "B3"],
        "C": ["C0", "C1", "C2", "C3"],
        "D": ["D0", "D1", "D2", "D3"],
    },
    index=[0, 1, 2, 3],
)

df2 = pd.DataFrame(
    {
        "A": ["A4", "A5", "A6", "A7"],
        "B": ["B4", "B5", "B6", "B7"],
        "C": ["C4", "C5", "C6", "C7"],
        "D": ["D4", "D5", "D6", "D7"],
    },
    index=[4, 5, 6, 7],
)

df3 = pd.DataFrame(
    {
        "A": ["A8", "A9", "A10", "A11"],
        "B": ["B8", "B9", "B10", "B11"],
        "C": ["C8", "C9", "C10", "C11"],
        "D": ["D8", "D9", "D10", "D11"],
    },
```

```
    index=[8, 9, 10, 11],
)

frames = [df1, df2, df3]
result = pd.concat(frames)
result
```

Out[ ]:

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

> To add key for each DF in the resultant DF

In [ ]:

```
result = pd.concat(frames, keys=["x", "y", "z"])
result
```

Out[ ]:

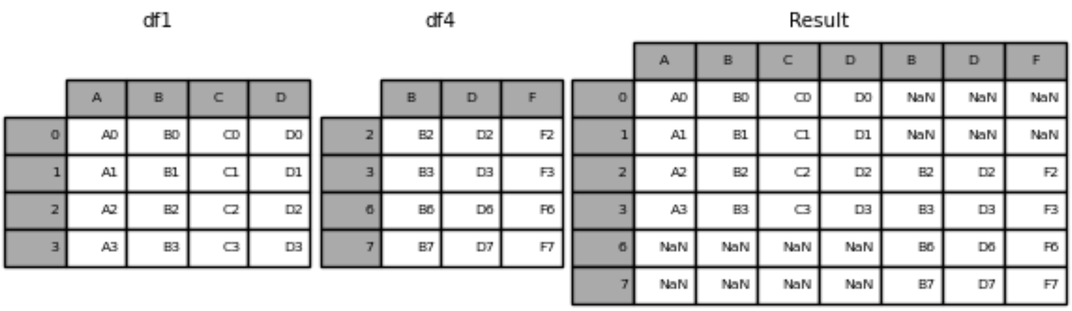
		A	B	C	D
x	0	A0	B0	C0	D0
	1	A1	B1	C1	D1
	2	A2	B2	C2	D2
	3	A3	B3	C3	D3
y	4	A4	B4	C4	D4
	5	A5	B5	C5	D5
	6	A6	B6	C6	D6
	7	A7	B7	C7	D7
z	8	A8	B8	C8	D8
	9	A9	B9	C9	D9
	10	A10	B10	C10	D10
	11	A11	B11	C11	D11

4.3.2 Set logic on the other axes (join outer and inner)

> When gluing together multiple DataFrames, you have a choice of how to handle the other axes (other than the one being concatenated). This can be done in the following two ways:

- Take the union of them all, join='outer'. This is the default option as it results in zero information loss.
- Take the intersection, join='inner'.

> Example for the default case, the outer



df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df4

	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

Result

	A	B	C	D	B	D	F
0	A0	B0	C0	D0	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3
6	NaN	NaN	NaN	NaN	B6	D6	F6
7	NaN	NaN	NaN	NaN	B7	D7	F7

How joining outer working

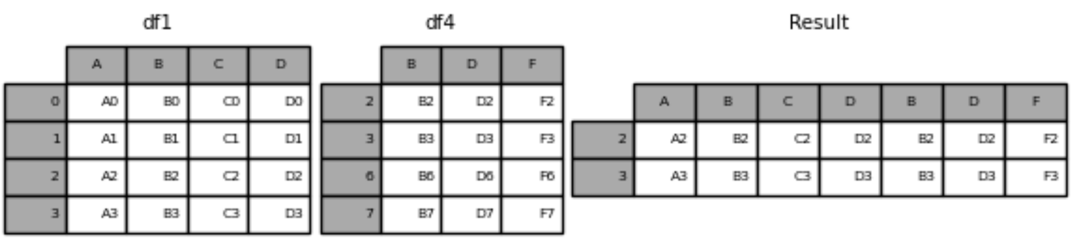
In [ ]:
```
df4 = pd.DataFrame(
    {
        "B": ["B2", "B3", "B6", "B7"],
        "D": ["D2", "D3", "D6", "D7"],
        "F": ["F2", "F3", "F6", "F7"],
    },
    index=[2, 3, 6, 7],
)

result = pd.concat([df1, df4], axis=1)
result
```

Out[ ]:

	A	B	C	D	B	D	F
0	A0	B0	C0	D0	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3
6	NaN	NaN	NaN	NaN	B6	D6	F6
7	NaN	NaN	NaN	NaN	B7	D7	F7

> Example for the inner case



How joining inner working

In [ ]:
```
result = pd.concat([df1, df4], axis=1, join="inner")
result
```

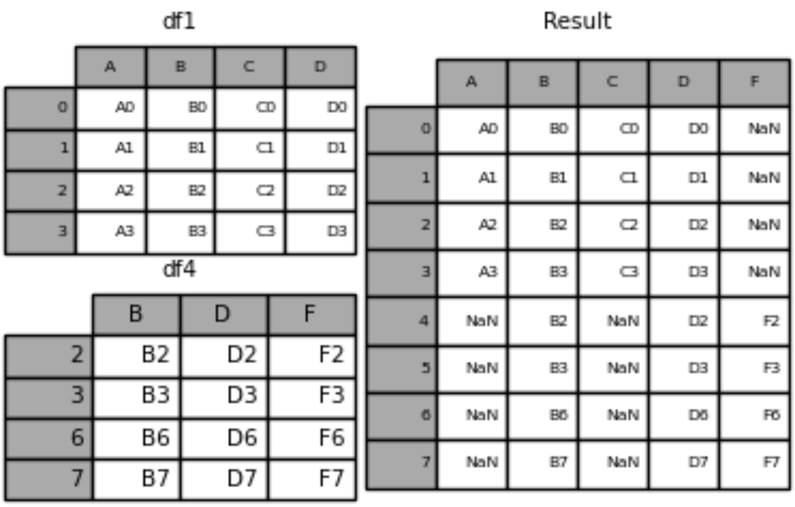
Out[ ]:

	A	B	C	D	B	D	F
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3

4.3.3 Ignoring indexes on the concatenation axis

For DataFrame objects which don't have a meaningful index, you may wish to append them and ignore the fact that they may have overlapping indexes.

> To do this, use the ignore_index argument:



How to ignore index

In [ ]:
```
result = pd.concat([df1, df4], ignore_index=True, sort=False)
result
```

Out[ ]:

	A	B	C	D	F
0	A0	B0	C0	D0	NaN
1	A1	B1	C1	D1	NaN
2	A2	B2	C2	D2	NaN
3	A3	B3	C3	D3	NaN
4	NaN	B2	NaN	D2	F2
5	NaN	B3	NaN	D3	F3
6	NaN	B6	NaN	D6	F6
7	NaN	B7	NaN	D7	F7

4.3.4 Merging DataFrames

When merging two DataFrames, Pandas will search for any common column and use it as the basis for the merge. If no common column is found, Pandas will introduce NaN values in the resulting DataFrame.

In [ ]:
```
import pandas as pd
```

```
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering',
                              'Engineering', 'HR']})

df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date': [2004, 2008, 2012, 2014]})
print(df1)
print(df2)
df3 = pd.merge(df1, df2)
print('--------------------------')
print(df3)
```

```
  employee        group
0      Bob   Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue           HR
  employee  hire_date
0     Lisa       2004
1      Bob       2008
2     Jake       2012
3      Sue       2014
--------------------------
  employee        group  hire_date
0      Bob   Accounting       2008
1     Jake  Engineering       2012
2     Lisa  Engineering       2004
3      Sue           HR       2014
```

> Example: Adding more than two DataFrames

In [ ]:

```
import pandas as pd

df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering',
                              'Engineering', 'HR']})

df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date': [2004, 2008, 2012, 2014]})

df3 = pd.merge(df1, df2)
print(df3)
print('--------------------------')

df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                    'supervisor': ['Carly', 'Guido', 'Steve']})
df5 = pd.merge(df3, df4)
print(df4)
print('--------------------------')
print(df5)
```

```
  employee        group  hire_date
0      Bob   Accounting       2008
1     Jake  Engineering       2012
2     Lisa  Engineering       2004
3      Sue           HR       2014
--------------------------
         group supervisor
0   Accounting      Carly
1  Engineering      Guido
2           HR      Steve
--------------------------
  employee        group  hire_date supervisor
0      Bob   Accounting       2008      Carly
1     Jake  Engineering       2012      Guido
2     Lisa  Engineering       2004      Guido
3      Sue           HR       2014      Steve
```

> Example: Merging two DataFrames and specifying which one is on the left and which one is on the right.

In [ ]:
```
import pandas as pd
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering',
                              'Engineering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date': [2004, 2008, 2012, 2014]})

df3 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'salary': [70000, 80000, 120000, 90000]})

print(df1)
print(df3)
print(pd.merge(df1, df3, left_on="employee", right_on="name"))
```

```
  employee        group
0      Bob   Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue           HR
   name  salary
0   Bob   70000
1  Jake   80000
2  Lisa  120000
3   Sue   90000
  employee        group  name  salary
0      Bob   Accounting   Bob   70000
1     Jake  Engineering  Jake   80000
2     Lisa  Engineering  Lisa  120000
3      Sue           HR   Sue   90000
```

4.3.4 Set logic on the other axes during merging

Important Note:

- In pandas axis = 1 parameter specifies that the operation should be performed on the columns of the DataFrame
- In pandas axis = 0 parameter specifies that the operation should be performed on the rows of the DataFrame

In [ ]:
```
import pandas as pd
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering',
                              'Engineering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date': [2004, 2008, 2012, 2014]})

df3 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'salary': [70000, 80000, 120000, 90000]})

print(df1)
print(df3)
print(pd.merge(df1, df3, left_on="employee",
               right_on="name").drop('name', axis=1))
```

```
  employee        group
0      Bob   Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue           HR
   name  salary
0   Bob   70000
1  Jake   80000
2  Lisa  120000
3   Sue   90000
  employee        group  salary
0      Bob   Accounting   70000
1     Jake  Engineering   80000
2     Lisa  Engineering  120000
3      Sue           HR   90000
```

> Example: When merging, only the intersecting elements in the common column will appear in the resulting DataFrame.

In [ ]:
```
import pandas as pd
df1 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'],
                    'food': ['fish', 'beans', 'bread']},
                    columns=['name', 'food'])

df2 = pd.DataFrame({'name': ['Mary', 'Joseph'],
                    'drink': ['cola', '7 up']},
                    columns=['name', 'drink'])
print(df1)
```

```
print('------------------------------')
print(df2)
print('------------------------------')
df3 = pd.merge(df1, df2)
print(df3)
```

```
    name   food
0  Peter   fish
1   Paul  beans
2   Mary  bread
------------------------------
     name drink
0    Mary  cola
1  Joseph  7 up
------------------------------
   name   food drink
0  Mary  bread  cola
```

- Example for the inner case

> The same intersection between DataFrames can be achieved by using the **"inner"** parameter.
> It's worth noting that "inner" is the default value for the parameter in the Pandas merge method.

In [ ]:
```
import pandas as pd
df1 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'],
                    'food': ['fish', 'beans', 'bread']},
                    columns=['name', 'food'])

df2 = pd.DataFrame({'name': ['Mary', 'Joseph'],
                    'drink': ['cola', '7 up']},
                    columns=['name', 'drink'])
print(df1)
print('------------------------------')
print(df2)
print('------------------------------')
df3 = pd.merge(df1, df2, how='inner')
print(df3)
```

```
    name   food
0  Peter   fish
1   Paul  beans
2   Mary  bread
------------------------------
     name drink
0    Mary  cola
1  Joseph  7 up
------------------------------
   name   food drink
0  Mary  bread  cola
```

- Example for the outer case

> Merging to obtain all elements, with NaN filling in the missing values, using the "outer" parameter.

In [ ]:
```
import pandas as pd
df1 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'],
                    'food': ['fish', 'beans', 'bread']},
                    columns=['name', 'food'])

df2 = pd.DataFrame({'name': ['Mary', 'Joseph'],
                    'drink': ['cola', '7 up']},
                    columns=['name', 'drink'])
print(df1)
print('------------------------------')
print(df2)
print('------------------------------')
df3 = pd.merge(df1, df2, how='outer')
print(df3)
```

```
    name   food
0  Peter   fish
1   Paul  beans
2   Mary  bread
-----------------------------
     name drink
0    Mary  cola
1  Joseph  7 up
-----------------------------
     name   food drink
0   Peter   fish   NaN
1    Paul  beans   NaN
2    Mary  bread  cola
3  Joseph    NaN  7 up
```

- Example for the right case

> Merging to obtain all elements from one DataFrame and filling in missing values in the other DataFrame with NaN, using the "right" parameter.

In [ ]:
```
import pandas as pd
df1 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'],
                    'food': ['fish', 'beans', 'bread']},
                    columns=['name', 'food'])

df2 = pd.DataFrame({'name': ['Mary', 'Joseph'],
                    'drink': ['cola', '7 up']},
                    columns=['name', 'drink'])
print(df1)
```

```
print('------------------------------')
print(df2)
print('------------------------------')
df3 = pd.merge(df1, df2, how='right')
print(df3)
```

```
    name   food
0  Peter   fish
1   Paul  beans
2   Mary  bread
------------------------------
     name drink
0    Mary  cola
1  Joseph  7 up
------------------------------
     name   food drink
0    Mary  bread  cola
1  Joseph    NaN  7 up
```

- Example for the left case

> Merging to obtain all elements from one DataFrame and filling in missing values in the other DataFrame with NaN, using the "left" parameter.

In [ ]:
```
import pandas as pd
df1 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'],
                    'food': ['fish', 'beans', 'bread']},
                    columns=['name', 'food'])

df2 = pd.DataFrame({'name': ['Mary', 'Joseph'],
                    'drink': ['cola', '7 up']},
                    columns=['name', 'drink'])
print(df1)
print('------------------------------')
print(df2)
print('------------------------------')
df3 = pd.merge(df1, df2, how='left')
print(df3)
```

```
    name   food
0  Peter   fish
1   Paul  beans
2   Mary  bread
-----------------------------
     name drink
0    Mary  cola
1  Joseph  7 up
-----------------------------
    name   food drink
0  Peter   fish   NaN
1   Paul  beans   NaN
2   Mary  bread  cola
```

4.4 Advanced operations on DataFrames

4.4.1 Statistics for the entire DF

In [ ]:

```
import pandas as pd
import numpy as np

df = pd.DataFrame({'A': np.random.rand(10),'B': np.random.rand(10)})
print(df)
print('-------------------------------')
print(df.sum())
print('-------------------------------')
print(df.prod())
print('-------------------------------')
print(df.mean())
```

```
          A         B
0  0.005275  0.590053
1  0.319625  0.230584
2  0.937504  0.333223
3  0.545917  0.104463
4  0.501140  0.597576
5  0.998301  0.465496
6  0.017465  0.196450
7  0.793709  0.366809
8  0.879614  0.028697
9  0.858705  0.389454
------------------------------
A    5.857254
B    3.302805
dtype: float64
------------------------------
A    0.000005
B    0.000001
dtype: float64
------------------------------
A    0.585725
B    0.330280
dtype: float64
```

4.4.2 Statistics on one Column of the DF

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

df = pd.DataFrame({'A': np.random.rand(10),'B': np.random.rand(10)})
print(df)
print('------------------------------')
print(df['A'].sum())
print('------------------------------')
print(df['B'].prod())
print('------------------------------')
print(df['A'].mean())
```

```
          A         B
0  0.151915  0.384361
1  0.157454  0.137242
2  0.194169  0.637392
3  0.836226  0.316493
4  0.235182  0.972446
5  0.423518  0.418742
6  0.381023  0.182254
7  0.143662  0.506586
8  0.531885  0.204914
9  0.877983  0.488592
------------------------------
3.9330174514184097
------------------------------
4.005510831011704e-05
------------------------------
0.39330174514184096
```

4.4.3 Statistics on all columns of the DF

> This is the default behavior, and you can specify to compute by rows. For example, computing the average for all rows will return the average of the elements across the entire column.

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

df = pd.DataFrame({'A': np.random.rand(10),'B': np.random.rand(10)})
print(df)
print('------------------------------')
print(df.mean(axis='rows'))
print('------------------------------')
print(df.count())
print('------------------------------')
print(df.min())
print('------------------------------')
print(df.max())
print('------------------------------')
print(df.std())
print('------------------------------')
```

```
          A         B
0  0.620815  0.940360
1  0.865903  0.801033
2  0.374614  0.307738
3  0.970911  0.176068
4  0.735641  0.536717
5  0.369581  0.591873
6  0.413798  0.072219
7  0.098215  0.288579
8  0.144818  0.194250
9  0.305043  0.910270
------------------------------
A    0.489934
B    0.481910
dtype: float64
------------------------------
A    10
B    10
dtype: int64
------------------------------
A    0.098215
B    0.072219
dtype: float64
------------------------------
A    0.970911
B    0.940360
dtype: float64
------------------------------
A    0.296368
B    0.320148
dtype: float64
------------------------------
```

4.4.4 Statistics on all rows of the DF

> In this case, you need to pass the parameter **axis="column"**.
> This allows the process to be applied to all elements by column and return results for each row.

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

df = pd.DataFrame({'A': np.random.rand(10),'B': np.random.rand(10)})
print(df)
print('------------------------------')
print(df.mean(axis='columns'))
```

```
          A         B
0  0.409199  0.838762
1  0.520183  0.080766
2  0.928907  0.882454
3  0.605618  0.086910
4  0.456624  0.702574
5  0.620493  0.309700
6  0.780535  0.672074
7  0.534290  0.996859
8  0.247131  0.935650
9  0.900387  0.839409
------------------------------
0    0.623981
1    0.300474
2    0.905681
3    0.346264
4    0.579599
5    0.465096
6    0.726305
7    0.765575
8    0.591391
9    0.869898
dtype: float64
```

4.4.5 Statistics on the entire DF using the Describe method

In [ ]:
```python
import pandas as pd

df = pd.DataFrame({'key':['A','B','C','A','B','C'],
                   'data': range(6)},columns=['key', 'data'])

print(df)
print(df.describe())
```

```
  key  data
0   A     0
1   B     1
2   C     2
3   A     3
4   B     4
5   C     5
           data
count  6.000000
mean   2.500000
std    1.870829
min    0.000000
25%    1.250000
50%    2.500000
75%    3.750000
max    5.000000
```

4.4.6 Groupby

> Groupby is a crucial method in Pandas that facilitates grouping similar elements and applying various operations on each group.

In [ ]:
```python
import pandas as pd

df = pd.DataFrame({'key':['A','B','C','A','B','C'],
                   'data': range(6)},columns=['key', 'data'])

print(df)
print(df.groupby('key').sum())
```

```
  key  data
0   A     0
1   B     1
2   C     2
3   A     3
4   B     4
5   C     5
     data
key
A       3
B       5
C       7
```

4.4.7 Grouping with the Describe Method

In [ ]:
```python
import pandas as pd

df = pd.DataFrame({'key':['A','B','C','A','B','C'],
                   'data': range(6)},columns=['key', 'data'])

print(df)
print(df.groupby('key').describe())
```

```
  key  data
0   A     0
1   B     1
2   C     2
3   A     3
4   B     4
5   C     5
     data
    count mean      std  min   25%  50%   75%  max
key
A     2.0  1.5  2.12132  0.0  0.75  1.5  2.25  3.0
B     2.0  2.5  2.12132  1.0  1.75  2.5  3.25  4.0
C     2.0  3.5  2.12132  2.0  2.75  3.5  4.25  5.0
```

> Example: **Unstack** the returned data from the describe method.

In [ ]:
```
import pandas as pd

df = pd.DataFrame({'key':['A','B','C','A','B','C'],
                   'data': range(6)},columns=['key', 'data'])

print(df)
print(df.groupby('key').describe().unstack())
```

```
  key  data
0   A     0
1   B     1
2   C     2
3   A     3
4   B     4
5   C     5
              key
data   count  A      2.00000
              B      2.00000
              C      2.00000
       mean   A      1.50000
              B      2.50000
              C      3.50000
       std    A      2.12132
              B      2.12132
              C      2.12132
       min    A      0.00000
              B      1.00000
              C      2.00000
       25%    A      0.75000
              B      1.75000
              C      2.75000
       50%    A      1.50000
              B      2.50000
              C      3.50000
       75%    A      2.25000
              B      3.25000
              C      4.25000
       max    A      3.00000
              B      4.00000
              C      5.00000
dtype: float64
```

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

df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data1': range(6),
                   'data2': np.random.randint(0, 10, 6)},
                    columns = ['key', 'data1', 'data2'])

print(df)
df2 = df.groupby('key').aggregate({'data1': 'min','data2': 'max'})
print(df2)
```

```
  key  data1  data2
0   A      0      0
1   B      1      6
2   C      2      6
3   A      3      9
4   B      4      5
5   C      5      1
     data1  data2
key
A        0      9
B        1      6
C        2      6
```

> Example: **filter** the returned data based on function.

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

df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data1': range(6),
                   'data2': np.random.randint(0, 10, 6)},
                    columns = ['key', 'data1', 'data2'])

print(df)
def filter_func(x):
    return x['data2'].std() > 4

df2 = df.groupby('key').filter(filter_func)
print(df2)
```

```
  key  data1  data2
0   A      0      0
1   B      1      6
2   C      2      5
3   A      3      5
4   B      4      1
5   C      5      5
Empty DataFrame
Columns: [key, data1, data2]
Index: []
```

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

df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data1': range(6),
                   'data2': np.random.randint(0, 10, 6)},
                    columns = ['key', 'data1', 'data2'])

print(df)

df2 = df.groupby('key').transform(lambda x: x**2)
```

```
print(df2)
```

```
  key  data1  data2
0   A      0      9
1   B      1      6
2   C      2      5
3   A      3      1
4   B      4      4
5   C      5      5
   data1  data2
0      0     81
1      1     36
2      4     25
3      9      1
4     16     16
5     25     25
```

4.5 Data Transformation and Handling Missing Data

4.5.1 How to Address Missing Data ?

- Why Handling Missing Data is Essential ?
    - Real-world data often contains missing values for various reasons.
    - Many machine learning models and statistical methods cannot effectively process data with missing values.
    - In such cases, it becomes essential to determine how to handle this missing data.
    - When reading data with missing values, Pandas represents them as NaN values.
- Options for handling missing data include:
    1. Keeping the missing data:
        - Pros: Does not manipulate or alter the true data.
        - Cons: Many methods or models do not support NaN values.
    2. Dropping or Removing the missing data:
        - Pros: Easy and can be based on predefined rules.
        - Cons: May result in the loss of a significant amount of data or valuable information.
    3. Filling the missing data:
        - Pros: Has the potential to retain a substantial amount of data for training a model.
        - Cons: Can be challenging and somewhat arbitrary, with the potential to lead to false conclusions.

4.5.2 Dropping Unnecessary Data

Dropping helps in focusing on what's essential by removing unneeded data.

> Example to drop unneeded columns

In [ ]:
```
# Sample DataFrame with unnecessary data
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 22],
        'Unneeded_Column': ['A', 'B', 'C']}

df = pd.DataFrame(data)

# Displaying the original DataFrame
print("Original DataFrame:")
print(df)

# Dropping the unneeded column
df = df.drop(columns=['Unneeded_Column'])

# Displaying the DataFrame after dropping the column
print("\nDataFrame after streamlining data:")
print(df)
```

```
Original DataFrame:
      Name  Age Unneeded_Column
0    Alice   25               A
1      Bob   30               B
2  Charlie   22               C

DataFrame after streamlining data:
      Name  Age
0    Alice   25
1      Bob   30
2  Charlie   22
```

4.5.3 Dropping Rows with Any Missing Values

> Method one : Using **dropna** function.

In [ ]:
```
import pandas as pd

# Sample DataFrame with missing values
data = {'Name': ['Alice', 'Bob', None, 'Charlie'],
        'Age': [25, 30, None, 22],
        'Salary': [50000, None, 60000, 70000]}
```

```python
df = pd.DataFrame(data)

# Displaying the original DataFrame
print("Original DataFrame:")
print(df)

# Drop the entire row if there is any one missing data
df_clean = df.dropna()
print("\nDataFrame with all rows without missing value ")
print(df_clean)
```

```
Original DataFrame:
      Name   Age   Salary
0    Alice  25.0  50000.0
1      Bob  30.0      NaN
2     None   NaN  60000.0
3  Charlie  22.0  70000.0

DataFrame with all rows without missing value 
      Name   Age   Salary
0    Alice  25.0  50000.0
3  Charlie  22.0  70000.0
```

> Another Method : Using **drop** function; Dropping an entire row containing missing values can be achieved by specifying a single label, index, or column

In [ ]:
```python
import pandas as pd

# Sample DataFrame with missing values
data = {'Name': ['Alice', 'Bob', None, 'Charlie'],
        'Age': [25, 30, None, 22],
        'Salary': [50000, None, 60000, 70000]}

df = pd.DataFrame(data)

# Displaying the original DataFrame
print("Original DataFrame:")
print(df)

# Drop the entire row if there is any one missing data
df_clean = df.drop(2,  axis=0)
print("\nDataFrame with all rows without missing value ")
print(df_clean)
```

```
Original DataFrame:
      Name   Age   Salary
0    Alice  25.0  50000.0
1      Bob  30.0      NaN
2     None   NaN  60000.0
3  Charlie  22.0  70000.0

DataFrame with all rows without missing value
      Name   Age   Salary
0    Alice  25.0  50000.0
1      Bob  30.0      NaN
3  Charlie  22.0  70000.0
```

4.5.4 Detecting All Missing Values

- Detecting missing values is crucial for maintaining data integrity, and it can be achieved using the **isnull()** function.
- Additionally, there is another function named **notnull()**, which, when used in reverse, indicates false wherever there is a missing value.

In [ ]:

```
import pandas as pd

# Sample DataFrame with missing values
data = {'Name': ['Alice', 'Bob', None, 'Charlie'],
        'Age': [25, 30, None, 22],
        'Salary': [50000, None, 60000, 70000]}

df = pd.DataFrame(data)

# Displaying the original DataFrame
print("Original DataFrame:")
print(df)

# Checking for missing values
missing_values = df.isnull()

# Displaying the DataFrame with missing value indicators
print("\nDataFrame with Missing Value Indicators:")
print(missing_values)

# Show rows where any one of them have missing value in the Salary column
rows_with_missing = df[df["Salary"].isnull()]
print("\nDataFrame with all rows contain Missing Value ")
print(rows_with_missing)
```

```
Original DataFrame:
      Name   Age   Salary
0    Alice  25.0  50000.0
1      Bob  30.0      NaN
2     None   NaN  60000.0
3  Charlie  22.0  70000.0

DataFrame with Missing Value Indicators:
    Name    Age  Salary
0  False  False   False
1  False  False    True
2   True   True   False
3  False  False   False

DataFrame with all rows contain Missing Value
  Name   Age  Salary
1  Bob  30.0     NaN
```

4.5.5 Eliminating Duplicates

Duplicate data can distort your analysis. The **drop_duplicates()** function is instrumental in preserving data integrity.

In [ ]:
```
import pandas as pd

# Sample DataFrame with duplicate data
data = {'Name': ['Alice', 'Bob', 'Alice', 'Charlie', 'Bob'],
        'Age': [25, 30, 25, 22, 30],
        'Salary': [50000, 60000, 50000, 70000, 60000]}

df = pd.DataFrame(data)

# Displaying the original DataFrame
print("Original DataFrame:")
print(df)

# Dropping duplicates based on all columns
df_no_duplicates = df.drop_duplicates()

# Displaying the DataFrame after dropping duplicates
print("\nDataFrame after dropping duplicates:")
print(df_no_duplicates)
```

```
Original DataFrame:
      Name  Age  Salary
0    Alice   25   50000
1      Bob   30   60000
2    Alice   25   50000
3  Charlie   22   70000
4      Bob   30   60000

DataFrame after dropping duplicates:
      Name  Age  Salary
0    Alice   25   50000
1      Bob   30   60000
3  Charlie   22   70000
```

> Another example to eliminating duplicates

In [ ]:
```python
# create DataFrame with duplicate entries
df = pd.DataFrame({'k1':['one']*3 + ['two']*4, 'k2':[1,1,2,3,3,4,4]})
# Displaying the original DataFrame
print("Original DataFrame:")
print(df)

# see the duplicate entries
duplicates = df.duplicated()
# Displaying the duplicates to look at them
print("\nThe duplicate entries :")
print(duplicates)

# Dropping duplicates based on all columns
df_no_duplicates = df.drop_duplicates()

# Displaying the DataFrame after dropping duplicates
print("\nDataFrame after dropping duplicates:")
print(df_no_duplicates)
```

```
Original DataFrame:
    k1  k2
0  one   1
1  one   1
2  one   2
3  two   3
4  two   3
5  two   4
6  two   4

The duplicate entries :
0    False
1     True
2    False
3    False
4     True
5    False
6     True
dtype: bool

DataFrame after dropping duplicates:
    k1  k2
0  one   1
2  one   2
3  two   3
5  two   4
```

4.5.6 Control Eliminating Duplicates

- Keep The last entry ;

> The drop_duplicates command currently removes the last entry. If there is a need to retain the last entry, the 'keep' keyword can be employed.

In [ ]:
```
df.drop_duplicates(keep="last")
```

Out[ ]:

	k1	k2
1	one	1
2	one	2
4	two	3
6	two	4

> We can eliminate all duplicate values based on specific columns as well.

In [ ]:
```
# create DataFrame with duplicate entries
df = pd.DataFrame({'k1':['one']*3 + ['two']*4, 'k2':[1,1,2,3,3,4,4]})
# Displaying the original DataFrame
print("Original DataFrame:")
print(df)

# drop duplicate entries based on k1 only
df_k1 = df.drop_duplicates(['k1'])
# Displaying the df after drop the duplicates based on k1
print("\n The DF without duplicate (k1 based) :")
print(df_k1)


# drop if k1 and k2 column matched
df_k1_k2 =df.drop_duplicates(['k1', 'k2'])
# Displaying the df after drop the duplicates based on k1 and k2
print("\n The DF without duplicate (k1_k2 based) :")
print(df_k1_k2)
```

```
Original DataFrame:
    k1  k2
0  one   1
1  one   1
2  one   2
3  two   3
4  two   3
5  two   4
6  two   4

 The DF without duplicate (k1 based) :
    k1  k2
0  one   1
3  two   3

 The DF without duplicate (k1_k2 based) :
    k1  k2
0  one   1
2  one   2
3  two   3
5  two   4
```

4.5.7 Transforming Data Using the replace() Function

The replace() function can customize your dataset by substituting values. It is commonly used for data cleaning, allowing correction of mislabeled data or handling outliers.

In [ ]:
```
import pandas as pd

# Sample DataFrame with a column containing specific values to be replace
data = {'Category': ['Fruit', 'Vegetable', 'Fruit', 'Meat', 'Vegetable',
        'Price': [2.5, 1.8, 3.0, 5.5, 2.0, 3.2, 6.0]}
```

```
df = pd.DataFrame(data)

# Displaying the original DataFrame
print("Original DataFrame:")
print(df)

# Replacing 'Fruit' with 'Healthy Snack' in the 'Category' column
df['Category'] = df['Category'].replace('Fruit', 'Healthy Snack')

# Displaying the DataFrame after replacement
print("\nDataFrame after Replacement:")
print(df)
```

```
Original DataFrame:
    Category  Price
0      Fruit    2.5
1  Vegetable    1.8
2      Fruit    3.0
3       Meat    5.5
4  Vegetable    2.0
5      Fruit    3.2
6       Meat    6.0

DataFrame after Replacement:
        Category  Price
0  Healthy Snack    2.5
1      Vegetable    1.8
2  Healthy Snack    3.0
3           Meat    5.5
4      Vegetable    2.0
5  Healthy Snack    3.2
6           Meat    6.0
```

> Another example for data transformation by replacing

In [ ]:

```
# create DataFrame with duplicate entries
df = pd.DataFrame({'k1':['one']*3 + ['two']*4, 'k2':[1,1,2,3,3,4,4]})
# Displaying the original DataFrame
print("Original DataFrame:")
print(df)

df_replaced=df.replace(['one', 3], ['One', '30'])
# Displaying the DataFrame after replacement
print("\nDataFrame after Replacement:")
print(df_replaced)
```

```
Original DataFrame:
    k1  k2
0  one   1
1  one   1
2  one   2
3  two   3
4  two   3
5  two   4
6  two   4

DataFrame after Replacement:
    k1  k2
0  One   1
1  One   1
2  One   2
3  two  30
4  two  30
5  two   4
6  two   4
```

4.5.8 Analysis of Categorical Data

A) Identifying Distinct Elements Using unique()

The unique function reveals the uniqueness within your data, providing insights into its diversity.

In [ ]:
```
import pandas as pd

# Sample DataFrame with a column containing repeated elements
data = {'Category': ['Fruit', 'Vegetable', 'Fruit', 'Meat', 'Vegetable',

df = pd.DataFrame(data)

# Displaying the original DataFrame
print("Original DataFrame:")
print(df)

# Extracting unique elements from the 'Category' column
unique_categories = df['Category'].unique()

# Displaying the unique elements
print("\nUnique Categories:")
print(unique_categories)
```

```
Original DataFrame:
    Category
0      Fruit
1  Vegetable
2      Fruit
3       Meat
4  Vegetable
5      Fruit
6       Meat

Unique Categories:
['Fruit' 'Vegetable' 'Meat']
```

B) Counting Uniqueness with nunique()

The **nunique()** function offers a quick method to comprehend the variation within your dataset.

In [ ]:
```
import pandas as pd

# Sample DataFrame with a column containing repeated elements
data = {'Category': ['Fruit', 'Vegetable', 'Fruit', 'Meat', 'Vegetable',

df = pd.DataFrame(data)

# Displaying the original DataFrame
print("Original DataFrame:")
print(df)

# Counting the number of unique elements in the 'Category' column
num_unique_categories = df['Category'].nunique()

# Displaying the number of unique elements
print("\nNumber of Unique Categories:", num_unique_categories)
```

```
Original DataFrame:
    Category
0      Fruit
1  Vegetable
2      Fruit
3       Meat
4  Vegetable
5      Fruit
6       Meat

Number of Unique Categories: 3
```

5. Multi Index

- MultiIndex: Used to create a hierarchical index for a DataFrame. This implies that the DataFrame's index can have multiple levels instead of just a single level.

5.1 Creating a 3D hierarchical table using MultiIndex or other methods.

5.1.1 Creating a hierarchical table with a MultiIndex

In [ ]:
```
import pandas as pd

index = [('California', 2000),('California', 2010),
         ('New York', 2000),('New York', 2010),
         ('Texas', 2000),('Texas', 2010)]

populations = [10000,15000,
               20000,25000,
               30000,35000]

index = pd.MultiIndex.from_tuples(index)
pop = pd.Series(populations, index=index)
pop = pop.reindex(index)

print(pop)
```

```
California  2000    10000
            2010    15000
New York    2000    20000
            2010    25000
Texas       2000    30000
            2010    35000
dtype: int64
```

5.1.2 Creating a hierarchical table with a DataFrame

The concept involves adding an index in 2D, considering the existing columns as 1D, resulting in a total of 3D structure.

In [ ]:
```
import pandas as pd
import numpy as np
df = pd.DataFrame(np.random.rand(4, 2),
                  index=[['a', 'a', 'b', 'b'],
                         [1, 2, 1, 2]],columns=['income', 'profit'])
print(df)
```

```
       income    profit
a 1  0.434272  0.849378
  2  0.930367  0.951611
b 1  0.874657  0.133630
  2  0.160487  0.983924
```

5.1.3 Creating a hierarchical table with a dictionary

The approach involves structuring the data in a dictionary, organizing it with nested dictionaries to achieve a hierarchical arrangement.

In [ ]:
```
import pandas as pd
data = {('California', 2000): 10000,('California', 2010):15000,
        ('Texas', 2000): 20000,('Texas', 2010): 25000,
        ('New York', 2000): 30000,('New York', 2010): 35000}
df = pd.Series(data)
print(df)
```

```
California  2000    10000
            2010    15000
Texas       2000    20000
            2010    25000
New York    2000    30000
            2010    35000
dtype: int64
```

5.2 Searching in the 3D Table by Year

In [ ]:
```
import pandas as pd

index = [('California', 2000),('California', 2010),
         ('New York', 2000),('New York', 2010),
         ('Texas', 2000),('Texas', 2010)]

populations = [10000,15000,
               20000,25000,
               30000,35000]

index = pd.MultiIndex.from_tuples(index)
pop = pd.Series(populations, index=index)
pop = pop.reindex(index)

print(pop[:, 2010])
```

```
California    15000
New York      25000
Texas         35000
dtype: int64
```

5.3 Transforming the hierarchical table to a regular format using the unstack method.

In [ ]:
```
import pandas as pd

index = [('California', 2000),('California', 2010),
         ('New York', 2000),('New York', 2010),
```

```
         ('Texas', 2000),('Texas', 2010)]

populations = [10000,15000,
               20000,25000,
               30000,35000]

index = pd.MultiIndex.from_tuples(index)
pop = pd.Series(populations, index=index)
pop = pop.reindex(index)

print(pop.unstack())
```

```
             2000   2010
California  10000  15000
New York    20000  25000
Texas       30000  35000
```

5.4 Adding one more column to the DataFrame

In [ ]:

```
import pandas as pd

index = [('California', 2000),('California', 2010),
         ('New York', 2000),('New York', 2010),
         ('Texas', 2000),('Texas', 2010)]

populations = [10000,15000,
               20000,25000,
               30000,35000]

index = pd.MultiIndex.from_tuples(index)
pop = pd.Series(populations, index=index)
pop = pop.reindex(index)

pop_df = pd.DataFrame({'total': pop,
                       'under18':[9267089, 9284094,4687374,
                                  4318033,5906301, 6879014]})

print(pop)
print(pop_df)
```

```
California  2000    10000
            2010    15000
New York    2000    20000
            2010    25000
Texas       2000    30000
            2010    35000
dtype: int64
                 total  under18
California 2000  10000  9267089
           2010  15000  9284094
New York   2000  20000  4687374
           2010  25000  4318033
Texas      2000  30000  5906301
           2010  35000  6879014
```

5.5 Examples of Complicated Tables with MultiIndex

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

index = pd.MultiIndex.from_product([[2013, 2014],[1, 2]],
                                   names=['year', 'visit'])

columns = pd.MultiIndex.from_product([['Bob', 'Guido', 'Sue'],
                                      ['HR', 'Temp']],
                                     names=['subject', 'type'])
data = np.round(np.random.randn(4, 6))
health_data = pd.DataFrame(data, index=index, columns=columns)
print(health_data)
```

```
subject      Bob      Guido       Sue
type          HR Temp    HR Temp   HR Temp
year visit
2013 1      -1.0  1.0  -0.0  0.0 -1.0  1.0
     2       0.0 -1.0  -2.0 -1.0 -0.0  1.0
2014 1      -0.0  0.0   0.0 -1.0 -0.0  0.0
     2       0.0 -0.0   1.0 -1.0 -0.0  2.0
```

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

index = pd.MultiIndex.from_product([[2013, 2014],[1, 2]],
                                   names=['year', 'visit'])

columns = pd.MultiIndex.from_product([['Bob', 'Guido', 'Sue'],
                                      ['HR', 'Temp']],
                                     names=['subject', 'type'])
data = np.round(np.random.randn(4, 6))
health_data = pd.DataFrame(data, index=index, columns=columns)

print(health_data['Guido', 'HR'])
```

```
year  visit
2013  1       -1.0
      2        0.0
2014  1       -0.0
      2        1.0
Name: (Guido, HR), dtype: float64
```

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

index = pd.MultiIndex.from_product([[2013, 2014],[1, 2]],
                                   names=['year', 'visit'])

columns = pd.MultiIndex.from_product([['Bob', 'Guido', 'Sue'],
                                      ['HR', 'Temp']],
                                    names=['subject', 'type'])
data = np.round(np.random.randn(4, 6))
health_data = pd.DataFrame(data, index=index, columns=columns)

print(health_data.loc[:, ('Bob', 'HR')])
```

```
year  visit
2013  1       -2.0
      2       -0.0
2014  1       -1.0
      2       -0.0
Name: (Bob, HR), dtype: float64
```

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

index = pd.MultiIndex.from_product([[2013, 2014],[1, 2]],
                                   names=['year', 'visit'])

columns = pd.MultiIndex.from_product([['Bob', 'Guido', 'Sue'],
                                      ['HR', 'Temp']],
                                    names=['subject', 'type'])
data = np.round(np.random.randn(4, 6))
health_data = pd.DataFrame(data, index=index, columns=columns)

idx = pd.IndexSlice
print(health_data.loc[idx[:, 1], idx[:, 'HR']])
```

```
subject      Bob Guido  Sue
type          HR    HR   HR
year visit
2013 1       1.0   2.0 -0.0
2014 1      -0.0  -1.0 -1.0
```

6. String in pandas

- Pandas handles strings through various methods, offering a range of functionalities for string manipulation.
- It's important to note that you can access the data as a series and then use the str class, which contains numerous functions for applying operations element-wise and returning the result for each element.

In [ ]:
```
import pandas as pd
data = ['peter', 'Paul', 'MARY', '15' , '  ' ]
print( pd.Series(data).str.len()) # length
print( pd.Series(data).str.startswith('p'))# is it start with "p"
print( pd.Series(data).str.endswith('Y')) #does is end with "r"
print( pd.Series(data).str.find('t')) # find this letter
print( pd.Series(data).str.rfind('A'))# find it from right
```

```
0    5
1    4
2    4
3    2
4    2
dtype: int64
0     True
1    False
2    False
3    False
4    False
dtype: bool
0    False
1    False
2     True
3    False
4    False
dtype: bool
0    2
1   -1
2   -1
3   -1
4   -1
dtype: int64
0   -1
1   -1
2    1
3   -1
4   -1
dtype: int64
```

In [ ]:
```
import pandas as pd
data = ['peter', 'Paul', 'MARY', '15' , '  ' ]
print( pd.Series(data).str.rjust(20))# adjust from right
print( pd.Series(data).str.ljust(50))# adjust from left
print( pd.Series(data).str.center(10))# make it center
print( pd.Series(data).str.zfill(5))# fill zeros
```

```
0                    peter
1                     Paul
2                     MARY
3                       15
4                         
dtype: object
0    peter                                        ...
1    Paul                                         ...
2    MARY                                         ...
3    15                                           ...
4                                                 ...
dtype: object
0      peter
1       Paul
2       MARY
3         15
4           
dtype: object
0    peter
1    0Paul
2    0MARY
3    00015
4    000
dtype: object
```

In [ ]:

```python
import pandas as pd
data = ['peter', 'Paul', 'MARY', '15' , '  ' ]
print( pd.Series(data).str.isupper())# is it all calpital
print( pd.Series(data).str.islower())# is it lower ?
print( pd.Series(data).str.istitle())# is like like "This"
print( pd.Series(data).str.isspace())# is it all spaces ?
print( pd.Series(data).str.isdigit())# is it numbers ?
print( pd.Series(data).str.isalpha())# is it all letters ?
print( pd.Series(data).str.isalnum())# is it not got any spaces ?
print( pd.Series(data).str.isdecimal())# is it decimals ?
print( pd.Series(data).str.isnumeric()) # is it number
print( pd.Series(data).str.upper())# make it capital
print( pd.Series(data).str.capitalize())# make it like 'This'
print( pd.Series(data).str.lower())# make it lower
print( pd.Series(data).str.swapcase())# switch capital & small
```

```
0    False
1    False
2     True
3    False
4    False
dtype: bool
0     True
1    False
2    False
3    False
4    False
dtype: bool
0    False
```

```
1     True
2    False
3    False
4    False
dtype: bool
0    False
1    False
2    False
3    False
4     True
dtype: bool
0    False
1    False
2    False
3     True
4    False
dtype: bool
0     True
1     True
2     True
3    False
4    False
dtype: bool
0     True
1     True
2     True
3     True
4    False
dtype: bool
0    False
1    False
2    False
3     True
4    False
dtype: bool
0    False
1    False
2    False
3     True
4    False
dtype: bool
0    PETER
1     PAUL
2     MARY
3       15
4         
dtype: object
0    Peter
1     Paul
2     Mary
3       15
4         
dtype: object
```

```
0    peter
1     paul
2     mary
3       15
4
dtype: object
0    PETER
1     pAUL
2     mary
3       15
4
dtype: object
```

7. Date & Time

7.1 Write Date in Pandas

- In Pandas, dates are typically represented in the format of year, month, day, hour, minute, and second.
- This format allows for easy access to individual components such as year, month, day, hour, minute, and second.

In [ ]:
```python
import pandas as pd

x = pd.to_datetime("4th of July, 2018")

print(x)
```

```
2018-07-04 00:00:00
```

7.2 Range between past and future dates

In [ ]:
```python
import pandas as pd
import numpy as np
x = pd.to_datetime("4th of July, 2018")

y = x + pd.to_timedelta(np.arange(20),'D')
z = x - pd.to_timedelta(np.arange(20),'D')

print(x)
print(y)
print(z)
```

```
2018-07-04 00:00:00
DatetimeIndex(['2018-07-04', '2018-07-05', '2018-07-06', '2018-07-07',
               '2018-07-08', '2018-07-09', '2018-07-10', '2018-07-11',
               '2018-07-12', '2018-07-13', '2018-07-14', '2018-07-15',
               '2018-07-16', '2018-07-17', '2018-07-18', '2018-07-19',
               '2018-07-20', '2018-07-21', '2018-07-22', '2018-07-23'],
              dtype='datetime64[ns]', freq=None)
DatetimeIndex(['2018-07-04', '2018-07-03', '2018-07-02', '2018-07-01',
               '2018-06-30', '2018-06-29', '2018-06-28', '2018-06-27',
               '2018-06-26', '2018-06-25', '2018-06-24', '2018-06-23',
               '2018-06-22', '2018-06-21', '2018-06-20', '2018-06-19',
               '2018-06-18', '2018-06-17', '2018-06-16', '2018-06-15'],
              dtype='datetime64[ns]', freq=None)
```

7.3 Write Dates & Tables

7.3.1 Creating a Table for Dates

In [ ]:
```
import pandas as pd
index = pd.DatetimeIndex(['2014-07-04', '2014-08-04',
                          '2015-07-04','2015-08-04'])
data = pd.Series([0, 1, 2, 3], index=index)
print(data)
```

```
2014-07-04    0
2014-08-04    1
2015-07-04    2
2015-08-04    3
dtype: int64
```

7.3.2 Filtering based on dates: days, months, or years

In [ ]:
```
import pandas as pd
index = pd.DatetimeIndex(['2011-03-12', '2012-08-21',
                          '2013-07-11','2014-11-08'])
data = pd.Series([0, 1, 2, 3], index=index)
print(data['2011-01-01':'2012-12-31'])
```

```
2011-03-12    0
2012-08-21    1
dtype: int64
```

In [ ]:
```
import pandas as pd
index = pd.DatetimeIndex(['2011-03-12', '2012-08-21',
                          '2013-07-11','2014-11-08'])

data = pd.Series([0, 1, 2, 3], index=index)
print(data['2012'])
```

```
2012-08-21    1
dtype: int64
```

In [ ]:
```
import pandas as pd
index = pd.DatetimeIndex(['2011-03-12', '2012-08-21',
                          '2013-07-11','2014-11-08'])

data = pd.Series([0, 1, 2, 3], index=index)
print(data['2012-08'])
```

```
2012-08-21    1
dtype: int64
```

7.3.3 Creating a table for dates with a range between two dates or starting from a specific day.

In [ ]:
```
import pandas as pd
data = pd.date_range('2011-12-25', '2012-01-08')
print(data)
```

```
DatetimeIndex(['2011-12-25', '2011-12-26', '2011-12-27', '2011-12-28',
               '2011-12-29', '2011-12-30', '2011-12-31', '2012-01-01',
               '2012-01-02', '2012-01-03', '2012-01-04', '2012-01-05',
               '2012-01-06', '2012-01-07', '2012-01-08'],
              dtype='datetime64[ns]', freq='D')
```

In [ ]:
```
import pandas as pd
data = pd.date_range('2011-12-25', periods=8)
print(data)
```

```
DatetimeIndex(['2011-12-25', '2011-12-26', '2011-12-27', '2011-12-28',
               '2011-12-29', '2011-12-30', '2011-12-31', '2012-01-01'],
              dtype='datetime64[ns]', freq='D')
```

In [ ]:
```
import pandas as pd
data = pd.date_range('2011-12-25', periods=8, freq='H')
print(data)
```

```
DatetimeIndex(['2011-12-25 00:00:00', '2011-12-25 01:00:00',
               '2011-12-25 02:00:00', '2011-12-25 03:00:00',
               '2011-12-25 04:00:00', '2011-12-25 05:00:00',
               '2011-12-25 06:00:00', '2011-12-25 07:00:00'],
              dtype='datetime64[ns]', freq='H')
```

7.3.4 Creating a table for dates with a range specified by hours only, excluding dates.

In [ ]:
```
import pandas as pd
data = pd.timedelta_range(0, periods=10, freq='H')
print(data)
```

```
TimedeltaIndex(['0 days 00:00:00', '0 days 01:00:00', '0 days 02:00:00',
                '0 days 03:00:00', '0 days 04:00:00', '0 days 05:00:00',
                '0 days 06:00:00', '0 days 07:00:00', '0 days 08:00:00',
                '0 days 09:00:00'],
               dtype='timedelta64[ns]', freq='H')
```

In [ ]:
```
import pandas as pd
data = pd.timedelta_range(0, periods=9, freq="2H30T40S")
print(data)
```

```
TimedeltaIndex(['0 days 00:00:00', '0 days 02:30:40', '0 days 05:01:20',
                '0 days 07:32:00', '0 days 10:02:40', '0 days 12:33:20',
                '0 days 15:04:00', '0 days 17:34:40', '0 days 20:05:20'],
               dtype='timedelta64[ns]', freq='9040S')
```

In [ ]:
```
import pandas as pd
from pandas.tseries.offsets import BDay
data = pd.date_range('2018-07-01', periods=5, freq=BDay())

print(data)
```

```
DatetimeIndex(['2018-07-02', '2018-07-03', '2018-07-04', '2018-07-05',
               '2018-07-06'],
              dtype='datetime64[ns]', freq='B')
```

8. File Handling (CSV files)

The **CSV** (Comma-Separated Values) file --> is a text file format that uses commas to separate values

8.1 Read CSV or Excel file

> (Comma-Separated Values) is a text file format that uses commas to separate values

In [ ]:
```
iris = 'http://archive.ics.uci.edu/ml/machine-learning-databases/iris/iri
df = pd.read_csv(iris, sep=',')
df.columns = ["sepal_length", "sepal_width", "petal_length", "petal_width
print(df.head(10))
```

```
   sepal_length  sepal_width  petal_length  petal_width        class
0           4.9          3.0           1.4          0.2  Iris-setosa
1           4.7          3.2           1.3          0.2  Iris-setosa
2           4.6          3.1           1.5          0.2  Iris-setosa
3           5.0          3.6           1.4          0.2  Iris-setosa
4           5.4          3.9           1.7          0.4  Iris-setosa
5           4.6          3.4           1.4          0.3  Iris-setosa
6           5.0          3.4           1.5          0.2  Iris-setosa
7           4.4          2.9           1.4          0.2  Iris-setosa
8           4.9          3.1           1.5          0.1  Iris-setosa
9           5.4          3.7           1.5          0.2  Iris-setosa
```

In [ ]:
```
import pandas as pd

grades1 = pd.read_csv('dataframe1.csv')
grades2 = pd.read_excel('dataframe1.xlsx')

print(grades1)
print(grades2)
```

```
  Unnamed: 0  Math  Physics  French  Chemistry
0          a     1        6      11         16
1          b     2        7      12         17
2          c     3        8      13         18
3          d     4        9      14         19
4          e     5       10      15         20
  Unnamed: 0  Math  Physics  French  Chemistry
0          a     1        6      11         16
1          b     2        7      12         17
2          c     3        8      13         18
3          d     4        9      14         19
4          e     5       10      15         20
```

8.2 Save DF as CSV or Excel file

In [ ]:
```
import pandas as pd
w = pd.Series({'a':1 ,'b':2 ,'c':3 ,'d':4 ,'e':5})
x = pd.Series({'a':6 ,'b':7 ,'c':8 ,'d':9 ,'e':10})
y = pd.Series({'a':11 ,'b':12 ,'c':13 ,'d':14 ,'e':15})
z = pd.Series({'a':16 ,'b':17 ,'c':18 ,'d':19 ,'e':20})
grades = pd.DataFrame({'Math':w,'Physics':x,'French':y,'Chemistry':z})

print(grades)
grades.to_csv('dataframe1.csv')
```

```
   Math  Physics  French  Chemistry
a     1        6      11         16
b     2        7      12         17
c     3        8      13         18
d     4        9      14         19
e     5       10      15         20
```

In [ ]:
```
import pandas as pd
```

```
w = pd.Series({'a':1 ,'b':2 ,'c':3 ,'d':4 ,'e':5})
x = pd.Series({'a':6 ,'b':7 ,'c':8 ,'d':9 ,'e':10})
y = pd.Series({'a':11 ,'b':12 ,'c':13 ,'d':14 ,'e':15})
z = pd.Series({'a':16 ,'b':17 ,'c':18 ,'d':19 ,'e':20})

grades = pd.DataFrame({'Math':w,'Physics':x,'French':y,'Chemistry':z})

print(grades)
grades.to_excel('dataframe1.xlsx',sheet_name ='Sheet1')
```

```
   Math  Physics  French  Chemistry
a     1        6      11         16
b     2        7      12         17
c     3        8      13         18
d     4        9      14         19
e     5       10      15         20
```

8.3 Applying different methods while reading the DataFrame

8.3.1 Specify one column to be the index column

In [ ]:
```
import pandas as pd
data=pd.read_csv("dataframe1.csv",  index_col= "Math")
print(data)
```

```
     Unnamed: 0  Physics  French  Chemistry
Math
1             a        6      11         16
2             b        7      12         17
3             c        8      13         18
4             d        9      14         19
5             e       10      15         20
```

8.3.2 Skip some rows

> Example for reading data and displaying from a CSV file:

- In this example, we skip the first three rows in the CSV file since they contain comments that I want to keep in the CSV file. Adjust the number accordingly based on your specific file.

In [ ]:
```
import pandas as pd
filename= "pima-indians-diabetes.csv"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', '
df = pd.read_csv(filename, skiprows=9, names = names)
# df.head(10)
description = data.describe()
```

```
print(description)
```

```
         Physics     French  Chemistry
count   5.000000   5.000000   5.000000
mean    8.000000  13.000000  18.000000
std     1.581139   1.581139   1.581139
min     6.000000  11.000000  16.000000
25%     7.000000  12.000000  17.000000
50%     8.000000  13.000000  18.000000
75%     9.000000  14.000000  19.000000
max    10.000000  15.000000  20.000000
```

8.3.3 Classify Data

> Example for classifying this data based on class (0,0) using groupby.

In [ ]:
```
# Class Distribution
import pandas as pd
filename= "pima-indians-diabetes.csv"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', '
data = pd.read_csv(filename, skiprows=9, names = names)
class_counts = data.groupby('class').size()
print(class_counts)
```

```
class
0    496
1    263
dtype: int64
```

8.3.4 Correlation and Skew

- Example for correlation of reading data from a file and skew:
    - A positive skew indicates that the distribution is skewed to the right, with more values above the mean than below the mean.
    - Conversely, a negative skew indicates that the distribution is skewed to the left, with more values below the mean than above the mean.

In [ ]:
```
import pandas as pd
filename= "pima-indians-diabetes.csv"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', '
data = pd.read_csv(filename, skiprows=9, names = names)
correlations = data.corr(method='pearson')
print(correlations)
skew = data.skew()
print(skew)
```

```
            preg      plas      pres      skin      test      mass      ped
i  \
preg   1.000000  0.127728  0.149506 -0.074168 -0.067703  0.019442 -0.02701
2
plas   0.127728  1.000000  0.154069  0.056797  0.327046  0.224216  0.13566
7
pres   0.149506  0.154069  1.000000  0.207468  0.089140  0.289950  0.04819
1
skin  -0.074168  0.056797  0.207468  1.000000  0.433753  0.392981  0.18311
9
test  -0.067703  0.327046  0.089140  0.433753  1.000000  0.198878  0.19044
5
mass   0.019442  0.224216  0.289950  0.392981  0.198878  1.000000  0.13321
6
pedi  -0.027012  0.135667  0.048191  0.183119  0.190445  0.133216  1.00000
0
age    0.549304  0.257701  0.240014 -0.120604 -0.051139  0.035393  0.03323
9
class  0.225405  0.464388  0.067412  0.068956  0.124995  0.293990  0.16712
7

            age     class
preg   0.549304  0.225405
plas   0.257701  0.464388
pres   0.240014  0.067412
skin  -0.120604  0.068956
test  -0.051139  0.124995
mass   0.035393  0.293990
pedi   0.033239  0.167127
age    1.000000  0.235132
class  0.235132  1.000000
preg     0.905962
plas     0.162860
pres    -1.850360
skin     0.115403
test     2.266363
mass    -0.436700
pedi     1.828278
age      1.131371
class    0.646393
dtype: float64
```

The Key Resources :

- Codes and examples in this notebook based heavily on Hesham Asem Python Playlist for ML.