Essential Functionality

What are the essential functionalities in Pandas?

The important essential functionalities are as follows:

- *Reindexing*
- *Dropping Entries from an Axis*
- *Indexing, selection and filtering*
- *Integer indexes*
- *Arithmetic and Data alignment*
- *Funtion application and mapping*
- *Sorting indexes and ranking*
- *Identifying the duplicate labels*

In [1]:
```
# import the pandas as pd first
import pandas as pd
import numpy as np
```

Reindexing

Which introduces new index object with values of the original index object remains same

In [2]:
```
# Let's look at the signature of reindex
```

In [3]:
```
pd.Series.reindex?
```

In [4]:
```
# Let's create a Pandas Series object
ob = pd.Series([1, 2, 3, 6], index=['d', 'b', 'a', 'c'])
ob
```

Out[4]:
```
d    1
b    2
a    3
c    6
dtype: int64
```

In [5]:
```
# reindexing without missed values or index labels that are not already present in the original index labels
ob2 = ob.reindex(index=['a', 'b', 'c', 'd'])
ob2
```

Out[5]:
```
a    3
b    2
c    6
d    1
dtype: int64
```

In [6]:
```
# if the new index object in reindex method has new index values then then those  missed  objects  values are filled
# with  NaN or NA
ob3 = ob.reindex(index=['a', 'b', 'c', 'd', 'e'])
ob3
```

Out[6]:
```
a    3.0
b    2.0
c    6.0
d    1.0
e    NaN
dtype: float64
```

In [7]:
```
# filling of values when reindexing using 'ffill' and 'bfill' with the help of method option
ob4 = pd.Series([1, 2, 3], index = [0, 1, 2])
ob4
```

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

In [8]:
```
# without method = 'ffill'
ob4.reindex(index=np.arange(6))
```

Out[8]:
```
0    1.0
1    2.0
2    3.0
3    NaN
4    NaN
5    NaN
dtype: float64
```

In [9]:
```
# with method = 'ffill'
ob4.reindex(index=np.arange(6), method = 'ffill')
```

Out[9]:
```
0    1
1    2
2    3
3    3
4    3
5    3
dtype: int64
```

In [10]:
```
# In DataFrame, reindex can alter either the (row) index, columns, or both.
ob5 = pd.DataFrame(np.arange(9).reshape((3, 3)),
                   index=['a', 'c', 'd'], columns=['Andhra', 'Tamilnadu', 'Kerala'])
ob5
```

Out[10]:

	Andhra	Tamilnadu	Kerala
a	0	1	2
c	3	4	5
d	6	7	8

In [11]:
```
# Let's look at the signature of 'reindex'
pd.DataFrame.reindex?
```

In [12]:
```
# reindexing the DataFrame : 'b' is a new row index name so reindex introduces NaN values for it.
ob6 = ob5.reindex(index=['a', 'b', 'c', 'd'])
ob6
```

Out[12]:

	Andhra	Tamilnadu	Kerala
a	0.0	1.0	2.0
b	NaN	NaN	NaN
c	3.0	4.0	5.0
d	6.0	7.0	8.0

In [13]:
```
# The columns can be reindexed with the columns keyword
capitals = ['Andhra', 'Telangana', 'Kerala']
ob5.reindex(columns=capitals)
```

Out[13]:

	Andhra	Telangana	Kerala
a	0	NaN	2
c	3	NaN	5
d	6	NaN	8

In [14]:
```
# we can also reindex with the loc option
ob5.loc[['a', 'b', 'c', 'd']]
ob5
```

```
C:\Users\user\Anaconda3\lib\site-packages\ipykernel_launcher.py:2: FutureWarning:
Passing list-likes to .loc or [] with any missing label will raise
KeyError in the future, you can use .reindex() as an alternative.

See the documentation here:
https://pandas.pydata.org/pandas-docs/stable/indexing.html#deprecate-loc-reindex-listlike
```

Out[14]:

	Andhra	Tamilnadu	Kerala
a	0	1	2
c	3	4	5
d	6	7	8

Droping Entries From an Axis

'drop' method will return a new object with the indicated value or values deleted from an axis

In [15]:
```
import pandas as pd
```

In [16]:
```
# Let's look at the signature of 'drop' with Series
pd.Series.drop?
```

In [17]:
```
# Let's look at the signature of 'drop' with DataFrame
pd.DataFrame.drop?
```

In [18]:
```
# Let's create a simple Series
data = pd.Series(np.arange(6), index=['a', 'b', 'c', 'd', 'e', 'f'])
data
```

Out[18]:
```
a    0
b    1
c    2
d    3
e    4
f    5
dtype: int32
```

In [19]:
```
# deleting a single row
data.drop('a')
```

Out[19]:
```
b    1
c    2
d    3
e    4
f    5
dtype: int32
```

In [20]:
```
# we can assign it to create new data object
n_data = data.drop('a')
```

In [21]:
```
# deleting multiple rows by passing a list of row names to be deleted
data.drop(['a', 'd'])
```

Out[21]:
```
b    1
c    2
e    4
f    5
dtype: int32
```

In [22]:
```
# In DataFrame, index values can be deleted from either axis
dataframe = pd.DataFrame(np.arange(16).reshape((4, 4)),
                         index=['a', 'b', 'd', 'e'], columns=['Karnataka', 'Andhra', 'Tamilnadu', 'Kerala'])
dataframe
```

Out[22]:

	Karnataka	Andhra	Tamilnadu	Kerala
a	0	1	2	3
b	4	5	6	7
d	8	9	10	11
e	12	13	14	15

In [23]:
```
# Calling drop with a sequence of labels will drop values from the row labels (axis 0)
dataframe.drop(['a', 'e'])
```

Out[23]:

	Karnataka	Andhra	Tamilnadu	Kerala
b	4	5	6	7
d	8	9	10	11

In [24]:
```
# We can drop values from the columns by passing axis=1 or axis='columns'
dataframe.drop('Kerala', axis=1)
```

Out[24]:

	Karnataka	Andhra	Tamilnadu
a	0	1	2
b	4	5	6
d	8	9	10
e	12	13	14

In [25]:
```
# deleting multiple columns: passing axis=1
dataframe.drop(['Kerala', 'Andhra'], axis=1)
```

Out[25]:

	Karnataka	Tamilnadu
a	0	2
b	4	6
d	8	10
e	12	14

In [26]:
```
# by passing axis='columns'
dataframe.drop(['Kerala', 'Tamilnadu'], axis='columns')
```

Out[26]:

	Karnataka	Andhra
a	0	1
b	4	5
d	8	9
e	12	13

In [27]:
```
# without passing 'inplace=True'
# we can get the original data back even after performing an operation like 'drop'
dataframe
```

Out[27]:

	Karnataka	Andhra	Tamilnadu	Kerala
a	0	1	2	3
b	4	5	6	7
d	8	9	10	11
e	12	13	14	15

In [28]:
```
# in order to avoid returning a new object, we can pass 'inplace=True'
# This type of operation can be applied for many functions like 'drop'
dataframe.drop(['Kerala', 'Andhra'], axis=1, inplace=True)
```

In [29]:
```
dataframe
```

Out[29]:

	Karnataka	Tamilnadu
a	0	2
b	4	6
d	8	10
e	12	14

In [30]:
```
# this type of operation with 'inplace=True' is very harmful as it destroys the original data or
# destroys any data that is dropped
dataframe
```

Out[30]:

	Karnataka	Tamilnadu
a	0	2
b	4	6
d	8	10
e	12	14

Integer Indexes

Working with pandas objects indexed by integers is little bit troublesome for users due to some differences with indexing semantics on built-in Python data structures like lists and tuples.

In [31]:
```
series = pd.Series(np.arange(3.))
```

In [32]:
```
print(series)
#series[-1] # if you run this it will cause an error because of potential ambiguity with integer index
# In this case, pandas could “fall back” on integer indexing
```

```
0    0.0
1    1.0
2    2.0
dtype: float64
```

In [33]:
```
# On the other hand, with a non-integer index, there is no potential for ambiguity
series2 = pd.Series(np.arange(3.), index=['a', 'b', 'c'])
series2
```

Out[33]:
```
a    0.0
b    1.0
c    2.0
dtype: float64
```

In [34]:
```
series2[-1]
```

Out[34]:
```
2.0
```

In [35]:
```
#  In order to keep things comfortable, we can use 'loc' and 'iloc' for labels and integers respectively.
```

In [36]:
```
print(series)
```

```
0    0.0
1    1.0
2    2.0
dtype: float64
```

In [37]:
```
series[:1]
```

Out[37]:
```
0    0.0
dtype: float64
```

In [38]:
```
series.loc[:1] # explicit loc includes final index label and its corresponding value
```
Out[38]:
```
0    0.0
1    1.0
dtype: float64
```

In [39]:
```
series.iloc[:1] # implicit loc doesn't include the final index label and its corresponding value
```
Out[39]:
```
0    0.0
dtype: float64
```

In [40]:
```
# with help of 'iloc' we can now use negative index on series with integer index
series.iloc[-1]
```
Out[40]:
```
2.0
```

Arithmetic and Data Alignment

It is very important to know the pandas feature in some applications like arithmetic between objects with different indexes. When you are adding together objects, if any index pairs are not the same, the respective index in the result will be the union of the index pairs.

In [41]:
```
import pandas as pd
import numpy as np
```

In [42]:
```
# Let's create the series
ser1 = pd.Series([7, 5, 4, 1], index=['a', 'c', 'd', 'e'])
ser1
```
Out[42]:
```
a    7
c    5
d    4
e    1
dtype: int64
```

In [43]:
```
ser2 = pd.Series([7, 5, 4, 1, 3], index=['a', 'c', 'e', 'f', 'g'])
ser2
```

Out[43]:
```
a    7
c    5
e    4
f    1
g    3
dtype: int64
```

In [44]:
```
# when we add these two series, the internal data alignment introduces missing values in the label locations that do
n't
# overlap
ser1 + ser2
```

Out[44]:
```
a    14.0
c    10.0
d     NaN
e     5.0
f     NaN
g     NaN
dtype: float64
```

In [45]:
```
# In the case of DataFrame, alignment is performed on both the rows and the columns
```

In [46]:
```
df1 = pd.DataFrame(np.arange(9).reshape((3, 3)),
                   columns=['a', 'c', 'd'], index=['Andhra', 'Tamilnadu', 'Kerala'])
df1
```

Out[46]:

	a	c	d
Andhra	0	1	2
Tamilnadu	3	4	5
Kerala	6	7	8

In [47]:
```
df2 = pd.DataFrame(np.arange(16).reshape((4, 4)),
                   columns=['a', 'b', 'd', 'e'], index=['Karnataka', 'Andhra', 'Tamilnadu', 'Kerala'])
df2
```

Out[47]:

	a	b	d	e
Karnataka	0	1	2	3
Andhra	4	5	6	7
Tamilnadu	8	9	10	11
Kerala	12	13	14	15

In [48]:
```
# when we add two DataFrame's together, the result returns a DataFrame whose index and columns are the unions
# of the ones in each DataFrame:
# Since the 'b', 'c' and 'e' columns are not found in both DataFrame objects, they appear as all missing in the result
print('df1:'); print(df1)
print('df2:'); print(df2)
df1 + df2
```

```
df1:
           a  c  d
Andhra     0  1  2
Tamilnadu  3  4  5
Kerala     6  7  8
df2:
            a   b   d   e
Karnataka   0   1   2   3
Andhra      4   5   6   7
Tamilnadu   8   9  10  11
Kerala     12  13  14  15
```

Out[48]:

	a	b	c	d	e
Andhra	4.0	NaN	NaN	8.0	NaN
Karnataka	NaN	NaN	NaN	NaN	NaN
Kerala	18.0	NaN	NaN	22.0	NaN
Tamilnadu	11.0	NaN	NaN	15.0	NaN

In [49]:
```
# If you add DataFrame objects with no column or row labels in common, the result will contain all null values
```

In [50]:
```
df3 = pd.DataFrame({'A': [1, 2]})
df3
```

Out[50]:

	A
0	1
1	2

In [51]:
```
df4 = pd.DataFrame({'B': [3, 4]})
df4
```

Out[51]:

	B
0	3
1	4

In [52]:
```
df3 + df4
```

Out[52]:

	A	B
0	NaN	NaN
1	NaN	NaN

In [53]:
```
df3 - df4
```

Out[53]:

	A	B
0	NaN	NaN
1	NaN	NaN

Arithmetic methods with fill values

How to use 'fill_value' when an axis label is found in one object but not in the other?

In [54]:
```
df5 = pd.DataFrame(np.arange(12).reshape((3, 4)), columns=list('abcd'))
df5
```

Out[54]:

	a	b	c	d
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11

In [55]:
```
df6 = pd.DataFrame(np.arange(20).reshape((4, 5)), columns=list('abcde'))
df6
```

Out[55]:

	a	b	c	d	e
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14
3	15	16	17	18	19

In [56]:
```
df6.loc[1, 'c'] = np.nan
df6
```

Out[56]:

	a	b	c	d	e
0	0	1	2.0	3	4
1	5	6	NaN	8	9
2	10	11	12.0	13	14
3	15	16	17.0	18	19

In [57]:
```
# adding without using the function 'add' and 'fill_value'
print('df5:'); print(df5);
print('df6:'); print(df6)
df5 + df6
```

```
df5:
   a  b   c   d
0  0  1   2   3
1  4  5   6   7
2  8  9  10  11
df6:
    a   b     c   d   e
0   0   1   2.0   3   4
1   5   6   NaN   8   9
2  10  11  12.0  13  14
3  15  16  17.0  18  19
```

Out[57]:

	a	b	c	d	e
0	*0.0*	*2.0*	*4.0*	*6.0*	*NaN*
1	*9.0*	*11.0*	*NaN*	*15.0*	*NaN*
2	*18.0*	*20.0*	*22.0*	*24.0*	*NaN*
3	*NaN*	*NaN*	*NaN*	*NaN*	*NaN*

In [58]:
```
# Let's look at the signature of one of the arithmetic operation:
pd.DataFrame.add?
# similarly you can check for other arithmetic operations on DataFrame and Series as well.
#pd.DataFrame.sub?
#pd.DataFrame.div?
#pd.DataFrame.radd?   # 'radd' to be discussed soon in this section
```

In [59]:
```
# Using the add method on df5 by passing df6 and an argument, fill_value to fill the missing NaN or NA's
print('df5:'); print(df5)
print('df6:'); print(df6)
df5.add(df6, fill_value=0)
```

```
df5:
   a  b   c   d
0  0  1   2   3
1  4  5   6   7
2  8  9  10  11
df6:
    a   b     c   d   e
0   0   1   2.0   3   4
1   5   6   NaN   8   9
2  10  11  12.0  13  14
3  15  16  17.0  18  19
```

Out[59]:

	a	b	c	d	e
0	0.0	2.0	4.0	6.0	4.0
1	9.0	11.0	6.0	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 [60]:
```
print('Before addition')
print(type(df5.loc[1, 'd'])); print(type(df6.loc[1, 'd'])); print()
print("After addition")
print(type(df5.add(df6, fill_value=0).loc[1, 'd']))
```

```
Before addition
<class 'numpy.int32'>
<class 'numpy.int32'>

After addition
<class 'numpy.float64'>
```

In [61]:
```python
print('df5:'); print(df5)
print('df6:'); print(df6)
df5.add(df6, fill_value=10)
```

```
df5:
   a  b   c   d
0  0  1   2   3
1  4  5   6   7
2  8  9  10  11
df6:
    a   b     c   d   e
0   0   1   2.0   3   4
1   5   6   NaN   8   9
2  10  11  12.0  13  14
3  15  16  17.0  18  19
```

Out[61]:

	a	b	c	d	e
0	0.0	2.0	4.0	6.0	14.0
1	9.0	11.0	16.0	15.0	19.0
2	18.0	20.0	22.0	24.0	24.0
3	25.0	26.0	27.0	28.0	29.0

In [62]:
```
# let's see some more arithmetic aperations
print(df5)
# scalar division
1/df5
```

```
   a  b   c   d
0  0  1   2   3
1  4  5   6   7
2  8  9  10  11
```

Out[62]:

	a	b	c	d
0	inf	1.000000	0.500000	0.333333
1	0.250000	0.200000	0.166667	0.142857
2	0.125000	0.111111	0.100000	0.090909

In [63]:
```
# Multiplication with a scalar value
print(df5)
df5 * 2
```

```
   a  b   c   d
0  0  1   2   3
1  4  5   6   7
2  8  9  10  11
```

Out[63]:

	a	b	c	d
0	0	2	4	6
1	8	10	12	14
2	16	18	20	22

In [64]:
```
# scalar subtraction
print(df5)
df5 - 3
```

```
   a  b   c   d
0  0  1   2   3
1  4  5   6   7
2  8  9  10  11
```

Out[64]:

	a	b	c	d
0	-3	-2	-1	0
1	1	2	3	4
2	5	6	7	8

In [65]:
```
# All thes methods have a counterpart, starting with the letter r
# Ex: radd, rdiv, rmul etc.
# We have already discussed about these in the Theory part. Let's see few examples here.
print(df5)
df5.rdiv(1)
```

```
   a  b   c   d
0  0  1   2   3
1  4  5   6   7
2  8  9  10  11
```

Out[65]:

	a	b	c	d
0	inf	1.000000	0.500000	0.333333
1	0.250000	0.200000	0.166667	0.142857
2	0.125000	0.111111	0.100000	0.090909

In [66]:
```
print(df5)
df5.rmul(2)
```

```
   a  b   c   d
0  0  1   2   3
1  4  5   6   7
2  8  9  10  11
```

Out[66]:

	a	b	c	d
0	0	2	4	6
1	8	10	12	14
2	16	18	20	22

In [67]:
```
print(df5)
df5.rpow(2)
```

```
   a  b   c   d
0  0  1   2   3
1  4  5   6   7
2  8  9  10  11
```

Out[67]:

	a	b	c	d
0	1	2	4	8
1	16	32	64	128
2	256	512	1024	2048

In [68]:
```
print(df5)
df5.radd(10)
```

```
   a  b   c   d
0  0  1   2   3
1  4  5   6   7
2  8  9  10  11
```

Out[68]:

	a	b	c	d
0	10	11	12	13
1	14	15	16	17
2	18	19	20	21

Operations between DataFrame and Series

The operation between DataFrame and Series is known as 'Broadcasting'. In Broadcasting the operation takesplace once per each row

In [69]:
```
# Let's create a DataFrame
df7 = pd.DataFrame(np.arange(12.).reshape((4, 3)),
                   columns=list('bde'), index=['One', 'Two', 'Three', 'Four'])
df7
```

Out[69]:

	b	d	e
One	0.0	1.0	2.0
Two	3.0	4.0	5.0
Three	6.0	7.0	8.0
Four	9.0	10.0	11.0

In [70]:
```
# Let's use 'iloc' to extract one of the row of df7 as a Series
df7_ser = df7.iloc[0]
print(df7_ser); print(type(df7_ser))
```

```
b    0.0
d    1.0
e    2.0
Name: One, dtype: float64
<class 'pandas.core.series.Series'>
```

In [71]:
```
# By default, arithmetic between DataFrame and Series matches the index of the Series on the DataFrame’s columns,
# broadcasting down the rows
# That is Broadcasting from first row down towards the last row.
# Let's subtract df7_ser from each row of df7 DataFrame
print(df7); print(df7_ser)
df7 - df7_ser
```

```
         b     d     e
One    0.0   1.0   2.0
Two    3.0   4.0   5.0
Three  6.0   7.0   8.0
Four   9.0  10.0  11.0
b    0.0
d    1.0
e    2.0
Name: One, dtype: float64
```

Out[71]:

	b	d	e
One	0.0	0.0	0.0
Two	3.0	3.0	3.0
Three	6.0	6.0	6.0
Four	9.0	9.0	9.0

In [72]:
```
# If an index value is not found in either the DataFrame's columns or the Series's index, the objects will be reindexe
d
# to form the union
df7
```

Out[72]:

	b	d	e
One	0.0	1.0	2.0
Two	3.0	4.0	5.0
Three	6.0	7.0	8.0
Four	9.0	10.0	11.0

In [73]:
```
ser2 = pd.Series(range(3), index=['b', 'e', 'f'])
ser2
```

Out[73]:
```
b    0
e    1
f    2
dtype: int64
```

In [74]:
```
# adding together returns a new object with missing rows or columns as NaN
df7 + ser2
```

Out[74]:

	b	d	e	f
One	0.0	NaN	3.0	NaN
Two	3.0	NaN	6.0	NaN
Three	6.0	NaN	9.0	NaN
Four	9.0	NaN	12.0	NaN

In [75]:
```
# In order to 'Broadcast' over the columns of the DataFrame, we need to match the index rows with the columns using
# the arithmetic methods
df7
```

Out[75]:

	b	d	e
One	0.0	1.0	2.0
Two	3.0	4.0	5.0
Three	6.0	7.0	8.0
Four	9.0	10.0	11.0

In [76]:
```
df7_col = df7['b']
df7_col
```

Out[76]:
```
One      0.0
Two      3.0
Three    6.0
Four     9.0
Name: b, dtype: float64
```

In [77]:
```
# Let's pass the (axis='index' or axis=0) to Broadcast over the columns
df7.sub(df7_col, axis='index')
```

Out[77]:

	b	d	e
One	0.0	1.0	2.0
Two	0.0	1.0	2.0
Three	0.0	1.0	2.0
Four	0.0	1.0	2.0

Function Application and Mapping

Similar to like Python's functions which does the work on each element of the object; Pandas also provides 'apply' method which does the same thing on each row or column

In [78]:
```
# signature of 'apply': important to notice about the row and column wise operation
pd.DataFrame.apply?
```

In [79]:
```
# signature of 'applymap': important to notice about the element wise operation
pd.DataFrame.applymap?
```

In [80]:
```
# Let's create a DataFrame
df8 = pd.DataFrame(np.random.randn(4, 3),
                   columns=list('bde'), index=['One', 'Two', 'Three', 'Four'])
df8
```

Out[80]:

	b	d	e
One	0.659410	-1.225919	0.864828
Two	-1.170436	-1.646294	0.214555
Three	0.798262	0.745293	0.604696
Four	-1.100479	-0.094230	1.810666

In [81]:
```
# Let's use the Numpy's abs function on each column(or Series) of the DataFrame
abs(df8)
```

Out[81]:

	b	d	e
One	0.659410	1.225919	0.864828
Two	1.170436	1.646294	0.214555
Three	0.798262	0.745293	0.604696
Four	1.100479	0.094230	1.810666

In [82]:
```
# the 'apply' function by default does the function each column wise
f = lambda x: x.max()
print(df8)
df8.apply(f)
```

```
              b         d         e
One    0.659410 -1.225919  0.864828
Two   -1.170436 -1.646294  0.214555
Three  0.798262  0.745293  0.604696
Four  -1.100479 -0.094230  1.810666
```

Out[82]:
```
b    0.798262
d    0.745293
e    1.810666
dtype: float64
```

In [83]:
```
f = lambda x: x.min()
df8.apply(f)
```

Out[83]:
```
b   -1.170436
d   -1.646294
e    0.214555
dtype: float64
```

In [84]:
```
# we can do vector operation also
f = lambda x: x.max() - x.min()
df8.apply(f)
```

Out[84]:
```
b    1.968698
d    2.391587
e    1.596111
dtype: float64
```

In [85]:
```python
# If we pass axis='columns' to apply, the function will be invoked once per row instead
f = lambda x: x.max()
print(df8)
df8.apply(f, axis='columns')
```

```
              b         d         e
One    0.659410 -1.225919  0.864828
Two   -1.170436 -1.646294  0.214555
Three  0.798262  0.745293  0.604696
Four  -1.100479 -0.094230  1.810666
```

Out[85]:
```
One      0.864828
Two      0.214555
Three    0.798262
Four     1.810666
dtype: float64
```

In [86]:
```python
# similarly you can apply x.min and finally 'f = lambda x: x.max() - x.min()'
f = lambda x: x.max() - x.min()
df8.apply(f, axis='columns')
```

Out[86]:
```
One      2.090747
Two      1.860849
Three    0.193566
Four     2.911146
dtype: float64
```

In [87]:
```
# 'apply' method also accepts user defined function to return a Series with multiple values
def f(x):
    return pd.Series([x.max(), x.min(), x.mean()], index=['max', 'min', 'mean'])
print(df8)
df8.apply(f)
```

```
              b         d         e
One    0.659410 -1.225919  0.864828
Two   -1.170436 -1.646294  0.214555
Three  0.798262  0.745293  0.604696
Four  -1.100479 -0.094230  1.810666
```

Out[87]:

	b	d	e
max	0.798262	0.745293	1.810666
min	-1.170436	-1.646294	0.214555
mean	-0.203311	-0.555287	0.873686

In [88]:
```
# We can also use Python function to do element wise operation with the help of 'applymap' method
```

In [89]:
```
# Let's use this to format the df8 elements round to three digits after the decimal point
f = lambda x: '%.3f' %x
df8.applymap(f)
```

Out[89]:

	b	d	e
One	0.659	-1.226	0.865
Two	-1.170	-1.646	0.215
Three	0.798	0.745	0.605
Four	-1.100	-0.094	1.811

Sorting and Ranking

Sorting will sort the row or column index and return a new sorted object. It uses 'sort_index' method

Ranking assigns ranks from one through the number of valid data points in an array. It uses 'rank' method

Sorting

In [90]:
```
# let's see the signature of each
pd.DataFrame.sort_index?
```

In [91]:
```
pd.DataFrame.rank?
```

In [92]:
```
series = pd.Series(range(6), index=['d', 'a', 'b', 'c', 'f', 'g'])
series
```

Out[92]:
```
d    0
a    1
b    2
c    3
f    4
g    5
dtype: int64
```

In [93]:
```
# sorting looks similar to like sorting in the windows Sort By
series.sort_index(axis=0, level=None, ascending=True)
```

Out[93]:
```
a    1
b    2
c    3
d    0
f    4
g    5
dtype: int64
```

In [94]:
```
# Let's take DataFrame object
df9 = pd.DataFrame(np.random.randn(4, 5),
                   columns=list('bdeac'), index=['1', '3', '2', '4'])
df9
# ['one', 'three', 'two', 'four']
```

Out[94]:

	b	d	e	a	c
1	-1.210677	-0.430911	-0.489966	-1.165102	0.916092
3	0.268974	-0.078269	1.715395	-1.030560	-0.916672
2	-0.072977	1.094294	0.322771	-0.724110	-0.778855
4	0.627308	-0.367973	0.711404	-0.955800	0.664370

In [95]:
```
df9.sort_index(axis=1, level=None, ascending=True)
```

Out[95]:

	a	b	c	d	e
1	-1.165102	-1.210677	0.916092	-0.430911	-0.489966
3	-1.030560	0.268974	-0.916672	-0.078269	1.715395
2	-0.724110	-0.072977	-0.778855	1.094294	0.322771
4	-0.955800	0.627308	0.664370	-0.367973	0.711404

In [96]:
```
df9.sort_index(axis=1, level=None, ascending=False)
```

Out[96]:

	e	d	c	b	a
1	-0.489966	-0.430911	0.916092	-1.210677	-1.165102
3	1.715395	-0.078269	-0.916672	0.268974	-1.030560
2	0.322771	1.094294	-0.778855	-0.072977	-0.724110
4	0.711404	-0.367973	0.664370	0.627308	-0.955800

In [97]:
```
df9.sort_index(axis=0, level=None, ascending=True)
```

Out[97]:

	b	d	e	a	c
1	-1.210677	-0.430911	-0.489966	-1.165102	0.916092
2	-0.072977	1.094294	0.322771	-0.724110	-0.778855
3	0.268974	-0.078269	1.715395	-1.030560	-0.916672
4	0.627308	-0.367973	0.711404	-0.955800	0.664370

In [98]:
```
# Let's create a fixed(or static) values DataFrame to sort on values with sort_index
df10 =pd.DataFrame(np.arange(12).reshape((3, 4)),
             index=['1', '3', '2'],
             columns=['d', 'a', 'b', 'c'])
df10
```

Out[98]:

	d	a	b	c
1	0	1	2	3
3	4	5	6	7
2	8	9	10	11

In [99]:
```
# sorting on integer index
df10.sort_index(axis='index', level=None, ascending=True, kind='mergesort', na_position='last',
                sort_remaining=True, by=None)
```

Out[99]:

	d	a	b	c
1	0	1	2	3
2	8	9	10	11
3	4	5	6	7

In [100]:
```
df10.sort_index(axis='index', level=None, ascending=True, kind='mergesort', na_position='last',
                sort_remaining=True, by=['d'])
```

```
C:\Users\user\Anaconda3\lib\site-packages\ipykernel_launcher.py:2: FutureWarning: by argument to sort_index is deprec
ated, please use .sort_values(by=...)
```

Out[100]:

	d	a	b	c
1	0	1	2	3
3	4	5	6	7
2	8	9	10	11

In [101]:
```
# Let's see the signature of 'sort_values' used to sort by values instead of row or collumns
pd.DataFrame.sort_values?
# pd.DataFrame.sort_values(self, by, axis=0, ascending=True, inplace=False, kind='quicksort', na_position='last')
```

In [102]:
```
print(df9)
df9.sort_values(by=['b'])
```

```
          b         d         e         a         c
1 -1.210677 -0.430911 -0.489966 -1.165102  0.916092
3  0.268974 -0.078269  1.715395 -1.030560 -0.916672
2 -0.072977  1.094294  0.322771 -0.724110 -0.778855
4  0.627308 -0.367973  0.711404 -0.955800  0.664370
```

Out[102]:

	b	d	e	a	c
1	-1.210677	-0.430911	-0.489966	-1.165102	0.916092
2	-0.072977	1.094294	0.322771	-0.724110	-0.778855
3	0.268974	-0.078269	1.715395	-1.030560	-0.916672
4	0.627308	-0.367973	0.711404	-0.955800	0.664370

In [103]:
```
print(df9)
df9.sort_values(by=['d'])
```

```
          b         d         e         a         c
1 -1.210677 -0.430911 -0.489966 -1.165102  0.916092
3  0.268974 -0.078269  1.715395 -1.030560 -0.916672
2 -0.072977  1.094294  0.322771 -0.724110 -0.778855
4  0.627308 -0.367973  0.711404 -0.955800  0.664370
```

Out[103]:

	b	d	e	a	c
1	-1.210677	-0.430911	-0.489966	-1.165102	0.916092
4	0.627308	-0.367973	0.711404	-0.955800	0.664370
3	0.268974	-0.078269	1.715395	-1.030560	-0.916672
2	-0.072977	1.094294	0.322771	-0.724110	-0.778855

Ranking

In [104]:
```
# Let's pick up one earlier DF
df8
```

Out[104]:

	b	d	e
One	0.659410	-1.225919	0.864828
Two	-1.170436	-1.646294	0.214555
Three	0.798262	0.745293	0.604696
Four	-1.100479	-0.094230	1.810666

In [105]:
```
# Let's apply the rank on whole DataFrame first
print(df8)
df8.rank()
```

```
              b         d         e
One    0.659410 -1.225919  0.864828
Two   -1.170436 -1.646294  0.214555
Three  0.798262  0.745293  0.604696
Four  -1.100479 -0.094230  1.810666
```

Out[105]:

	b	d	e
One	3.0	2.0	3.0
Two	1.0	1.0	1.0
Three	4.0	4.0	2.0
Four	2.0	3.0	4.0

In [106]:
```
print(df8)
df8.rank(axis='columns')
```

```
              b         d         e
One    0.659410 -1.225919  0.864828
Two   -1.170436 -1.646294  0.214555
Three  0.798262  0.745293  0.604696
Four  -1.100479 -0.094230  1.810666
```

Out[106]:

	b	d	e
One	2.0	1.0	3.0
Two	2.0	1.0	3.0
Three	3.0	2.0	1.0
Four	1.0	2.0	3.0

In [107]:
```
# Let's take series from the DataFrame
#print(df8['b'])
s = df8.loc[:, 'b']
s
```

Out[107]:
```
One      0.659410
Two     -1.170436
Three    0.798262
Four    -1.100479
Name: b, dtype: float64
```

In [108]:
```
s.rank()
```

Out[108]:
```
One      3.0
Two      1.0
Three    4.0
Four     2.0
Name: b, dtype: float64
```

Axis Indexes with Duplicate Labels

In [109]:
```
# Labels of Pandas inex may not be unique always:
# some times it is neccessary to have duplicate indices
```

In [110]:
```
# Ex
di_s = pd.Series(range(5), index=['a', 'a', 'b', 'b', 'c'])
di_s
```

Out[110]:
```
a    0
a    1
b    2
b    3
c    4
dtype: int64
```

In [111]:
```
# pandas 'is_unique' method will tell us whether the index labels are unique or not
# let's see the signature of the is_unique method. Note that it is a 'CachedProperty'
pd.Index.is_unique?
```

In [112]:
```
di_s.index.is_unique
```

Out[112]:
```
False
```

In [113]:
```
# selection with duplicate index label will select all of the values with that duplicated index label
di_s['a']
```

Out[113]:
```
a    0
a    1
dtype: int64
```

In [114]:
```
di_s['b']
```

Out[114]:
```
b    2
b    3
dtype: int64
```

In [115]:
```
#
df11 = pd.DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b', 'b'])
df11
```

Out[115]:

	0	1	2
a	0.557010	-0.270394	1.340829
a	-2.743338	-1.016061	-0.255532
b	1.951289	-0.203005	-0.087225
b	1.149145	0.068329	-1.018955

In [116]:
```
df11.index.is_unique
```

Out[116]:
```
False
```

In [117]:
```
df11.loc['a']
```

Out[117]:

	0	1	2
a	0.557010	-0.270394	1.340829
a	-2.743338	-1.016061	-0.255532

In [ ]:

In [ ]:

How to Summarise and compute Descriptive Statistics?

In [118]:
```
df12 = pd.DataFrame([[1.4, np.nan], [7.1, -4.5], [np.nan, np.nan], [0.75, -1.3]],
                    index=['a', 'b', 'c', 'd'],
                    columns=['one', 'two'])
df12
```

Out[118]:

	one	two
a	1.40	NaN
b	7.10	-4.5
c	NaN	NaN
d	0.75	-1.3

In [119]:
```
df12.sum()
```

Out[119]:
```
one     9.25
two    -5.80
dtype: float64
```

In [120]:
```
df12.sum(axis='columns')
```

Out[120]:
```
a    1.40
b    2.60
c    0.00
d   -0.55
dtype: float64
```

In [121]:
```
df12.mean(axis='columns', skipna=False)
```

Out[121]:
```
a      NaN
b    1.300
c      NaN
d   -0.275
dtype: float64
```

In [122]:
```
df12.idxmax()
```

Out[122]:
```
one    b
two    d
dtype: object
```

In [123]:
```
df12.cumsum()
```

Out[123]:

	one	two
a	*1.40*	*NaN*
b	*8.50*	*-4.5*
c	*NaN*	*NaN*
d	*9.25*	*-5.8*

In [124]:
```
df12.describe()
```

Out[124]:

	one	two
count	3.000000	2.000000
mean	3.083333	-2.900000
std	3.493685	2.262742
min	0.750000	-4.500000
25%	1.075000	-3.700000
50%	1.400000	-2.900000
75%	4.250000	-2.100000
max	7.100000	-1.300000

In [125]:
```
ser = pd.Series(['a', 'a', 'b', 'c'] * 4)
print(ser)
print(ser.describe())
```

```
0     a
1     a
2     b
3     c
4     a
5     a
6     b
7     c
8     a
9     a
10    b
11    c
12    a
13    a
14    b
15    c
dtype: object
count     16
unique     3
top        a
freq       8
dtype: object
```

In [ ]:

In [ ]:

Unique Values, Value Counts, and Membership

In [126]:
```
# 'unique' which gives us an array of the unique values in a Series
```

In [127]:
```
import pandas as pd
ser_u = pd.Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])
ser_u
```

Out[127]:
```
0    c
1    a
2    d
3    a
4    a
5    b
6    b
7    c
8    c
dtype: object
```

In [128]:
```
# we can use 'sort' method if we want the sorted unique values
uniques = ser_u.unique()
uniques
```

Out[128]:
```
array(['c', 'a', 'd', 'b'], dtype=object)
```

In [129]:
```
# 'value_counts' which computes a Series containing value frequencies
ser_u.value_counts()
```

Out[129]:
```
c    3
a    3
b    2
d    1
dtype: int64
```

In [130]:
```
# isin performs a vectorized set membership check and can be useful in filtering a dataset down to a subset of values
# in a Series or column in a DataFrame
```

In [131]:
```
membership = ser_u.isin(['d', 'c'])
print(ser_u)
membership
```

```
0    c
1    a
2    d
3    a
4    a
5    b
6    b
7    c
8    c
dtype: object
```

Out[131]:
```
0     True
1    False
2     True
3    False
4    False
5    False
6    False
7     True
8     True
dtype: bool
```

In [132]:
```
ser_u[membership]
```

Out[132]:
```
0    c
2    d
7    c
8    c
dtype: object
```

In [133]:
```
# Index.get_indexer method, which gives you an index array from an array of possibly non-distinct values into another
# array of distinct values
```

In [134]:
```
non_dist = pd.Series(['c', 'a', 'b', 'b', 'c', 'a'])
non_dist
```

Out[134]:
```
0    c
1    a
2    b
3    b
4    c
5    a
dtype: object
```

In [135]:
```
dist = pd.Series(['c', 'b', 'a'])
dist
```

Out[135]:
```
0    c
1    b
2    a
dtype: object
```

In [136]:
```
pd.Index(dist).get_indexer(non_dist)
```

Out[136]:
```
array([0, 2, 1, 1, 0, 2], dtype=int32)
```

In [137]:
```
# How to compute histogram on multiple related columns in a DataFrame?
```

In [138]:
```
df13 = pd.DataFrame({'Qu1': [1, 3, 4, 3],
                     'Qu2': [2, 3, 1, 2],
                     'Qu3': [1, 5, 2, 4]})
df13
```

Out[138]:

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

In [139]:
```
# we can pass 'pd.vale_counts' method to 'apply' function of Pandas DataFrame
```

In [140]:
```
histogram = df13.apply(pd.value_counts)
histogram
```

Out[140]:

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

In [141]:
```
# in order to fill NaN values we can use 'fillna=0'
histogram = df13.apply(pd.value_counts).fillna(0)
histogram
```

Out[141]:

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

In [ ]: