Practice5

September 9, 2025

pandas adopts significant parts of NumPy's idiomatic style of array-based computing, especially array-based functions and a preference for data processing without for loops

While pandas adopts many coding idioms from NumPy, the biggest difference is that pandas is designed for working with tabular or heterogeneous data. NumPy, by contrast, is best suited for working with homogeneous numerical array data.

[1]:
```python
from pandas import Series, DataFrame
```

A Series is a one-dimensional array-like object containing a sequence of values (of similar types to NumPy types) and an associated array of data labels, called its index. The simplest Series is formed from only an array of data.

[2]:
```python
import pandas as pd
pd.Series([1,1.5,'a',[1,2,3]])
```

[2]:
```
0            1
1          1.5
2            a
3    [1, 2, 3]
dtype: object
```

You can get the array representation and index object of the Series via its values and index attributes, respectively

[3]:
```python
series=pd.Series([1,1.5,'a',[1,2,3]])
series.index
```

[3]:
```
RangeIndex(start=0, stop=4, step=1)
```

[4]:
```python
series.values
```

[4]:
```
array([1, 1.5, 'a', list([1, 2, 3])], dtype=object)
```

[5]:
```python
list([1, 2, 3])
```

[5]:
```
[1, 2, 3]
```

[6]:
```python
ser=pd.Series(['ankit','summi','kiioo'], index=['a','b','c'])
```

[7]:
```python
ser
```

[7]:
```
a    ankit
b    summi
c    kiioo
dtype: object
```

[8]:
```python
ser.iloc[0]  #accessing first element using index
```

[8]:
```
'ankit'
```

[9]:
```python
ser.loc['a']  #accessing first element using label
```

[9]:
```
'ankit'
```

[10]:
```python
ser['a']
```

[10]:
```
'ankit'
```

[11]:
```python
ser[['a','c']]  #accessing multiple elements using label
```

[11]:
```
a    ankit
c    kiioo
dtype: object
```

[12]:
```python
ser[1]
```

```
/tmp/ipykernel_5585/4267038266.py:1: FutureWarning: Series.__getitem__ treating
keys as positions is deprecated. In a future version, integer keys will always
be treated as labels (consistent with DataFrame behavior). To access a value by
position, use `ser.iloc[pos]`
  ser[1]
```

[12]:
```
'summi'
```

[13]:
```python
ser
```

[13]:
```
a    ankit
b    summi
c    kiioo
dtype: object
```

[14]:
```python
ser['d'] = 'soumi'  #adding new elements
ser
```

[14]:
```
a    ankit
b    summi
```

```
c    kiioo
d    soumi
dtype: object
```

Using NumPy functions or NumPy-like operations, such as filtering with a boolean array, scalar multiplication, or applying math functions, will preserve the index-value link

[15]:
```
ser[(ser=='soumi') | (ser=='kiioo')]
```

[15]:
```
c    kiioo
d    soumi
dtype: object
```

[16]:
```
ser*2
```

[16]:
```
a    ankitankit
b    summisummi
c    kiiookiioo
d    soumisoumi
dtype: object
```

[17]:
```
ser+ser
```

[17]:
```
a    ankitankit
b    summisummi
c    kiiookiioo
d    soumisoumi
dtype: object
```

[18]:
```
ser+'3'
```

[18]:
```
a    ankit3
b    summi3
c    kiioo3
d    soumi3
dtype: object
```

Another way to think about a Series is as a fixed-length, ordered dict, as it is a mapping of index values to data values. It can be used in many contexts where you might use a dict

[19]:
```
'b' in ser
```

[19]:
```
True
```

[20]:
```
'f' in ser
```

[20]:
```
False
```

[21]:
```python
ser1=pd.Series({'a':'ankkit','k':'kiioo'})
ser1
```

[21]:
```
a    ankkit
k     kiioo
dtype: object
```

I will use the terms “missing” or “NA” interchangeably to refer to missing data. The isnull and notnull functions in pandas should be used to detect missing data

[22]:
```python
obj=pd.Series([1,2,None,4,5],index=['a','b','c','d','e'])
print(obj,'\n\n',obj.isnull(),'\n\n',obj.isnull().sum())
```

```
a    1.0
b    2.0
c    NaN
d    4.0
e    5.0
dtype: float64

 a    False
b    False
c     True
d    False
e    False
dtype: bool

 1
```

Both the Series object itself and its index have a name attribute

[23]:
```python
obj.name='My Object'
obj.index.name = 'My Index'
obj
```

[23]:
```
My Index
a    1.0
b    2.0
c    NaN
d    4.0
e    5.0
Name: My Object, dtype: float64
```

[24]:
```python
import numpy as np
obj=pd.Series([1,2,3,4,5],index=['a','b','c','d','e'],dtype=np.int16)
obj
```

[24]:
```
a    1
b    2
c    3
d    4
e    5
dtype: int16
```

Dataframe

A DataFrame represents a rectangular table of data and contains an ordered collection of columns, each of which can be a different value type (numeric, string, boolean, etc.).

There are many ways to construct a DataFrame, though one of the most common is from a dict of equal-length lists or NumPy arrays

[25]:
```
data={'name':['ankit','kiio','summi','soumi'],'institute':
 ↪['ISI','IIIT-D','ECIL','IISc'],'Priority':[4,1,2,3],'Marks':[0,100.0,100.
 ↪0,100.0]}
df=pd.
 ↪DataFrame(data,index=['a','b','c','d'],columns=['name','institute','Marks','Priority'])
df
```

[25]:

	name	institute	Marks	Priority
a	ankit	ISI	0.0	4
b	kiio	IIIT-D	100.0	1
c	summi	ECIL	100.0	2
d	soumi	IISc	100.0	3

[26]:
```
df.head(3)
```

[26]:

	name	institute	Marks	Priority
a	ankit	ISI	0.0	4
b	kiio	IIIT-D	100.0	1
c	summi	ECIL	100.0	2

[27]:
```
df.columns
```

[27]:
```
Index(['name', 'institute', 'Marks', 'Priority'], dtype='object')
```

[28]:
```
df['name'] #dict like notation
```

[28]:
```
a    ankit
b     kiio
c    summi
d    soumi
Name: name, dtype: object
```

[29]:
```
df.name
```

[29]:
```
a    ankit
b     kiio
c    summi
d    soumi
Name: name, dtype: object
```

[30]:
```
# column to numpy array
df.name.values
```

[30]:
```
array(['ankit', 'kiio', 'summi', 'soumi'], dtype=object)
```

[31]:
```
df['name'].values
```

[31]:
```
array(['ankit', 'kiio', 'summi', 'soumi'], dtype=object)
```

[32]:
```
x=pd.Series({'a':1,'b':[1,2]},index=['b','a'])
```

[33]:
```
x
```

[33]:
```
b    [1, 2]
a         1
dtype: object
```

[34]:
```
x['b']
```

[34]:
```
[1, 2]
```

[35]:
```
df.loc[['c','d']]
```

[35]:

	name	institute	Marks	Priority
c	summi	ECIL	100.0	2
d	soumi	IISc	100.0	3

[36]:
```
df.iloc[[0,1]]
```

[36]:

	name	institute	Marks	Priority
a	ankit	ISI	0.0	4
b	kiio	IIIT-D	100.0	1

[37]:
```
df.loc[df['name']=='kiio']
```

[37]:

	name	institute	Marks	Priority
b	kiio	IIIT-D	100.0	1

[38]:
```
df['Marrital Statu']='No'
```

[39]:
```
df
```

[39]:
```
     name institute  Marks  Priority Marrital Statu
a  ankit       ISI    0.0         4             No
b   kiio    IIIT-D  100.0         1             No
c  summi      ECIL  100.0         2             No
d  soumi      IISc  100.0         3             No
```

[40]:
```
df['rank']=np.arange(1,5,1)
print(df)
```

```
     name institute  Marks  Priority Marrital Statu  rank
a  ankit       ISI    0.0         4             No     1
b   kiio    IIIT-D  100.0         1             No     2
c  summi      ECIL  100.0         2             No     3
d  soumi      IISc  100.0         3             No     4
```

[41]:
```
df['rank']=pd.Series([4,1,2,3],index=['a','b','c','d'])
print(df)
```

```
     name institute  Marks  Priority Marrital Statu  rank
a  ankit       ISI    0.0         4             No     4
b   kiio    IIIT-D  100.0         1             No     1
c  summi      ECIL  100.0         2             No     2
d  soumi      IISc  100.0         3             No     3
```

[42]:
```
# You can transpose the DataFrame (swap rows and columns) with similar syntax
 ↪to a NumPy array
df.T
```

[42]:
```
                    a       b      c      d
name            ankit    kiio  summi  soumi
institute         ISI  IIIT-D   ECIL   IISc
Marks             0.0   100.0  100.0  100.0
Priority            4       1      2      3
Marrital Statu     No      No     No     No
rank                4       1      2      3
```

[43]:
```
df.transpose()
```

[43]:
```
                    a       b      c      d
name            ankit    kiio  summi  soumi
institute         ISI  IIIT-D   ECIL   IISc
Marks             0.0   100.0  100.0  100.0
Priority            4       1      2      3
Marrital Statu     No      No     No     No
rank                4       1      2      3
```

a DataFrame's index and columns have their name attributes like series.

[44]:
```
print(df['rank'].values)
print(df.values)
```

```
[4 1 2 3]
[['ankit' 'ISI' 0.0 4 'No' 4]
 ['kiio' 'IIIT-D' 100.0 1 'No' 1]
 ['summi' 'ECIL' 100.0 2 'No' 2]
 ['soumi' 'IISc' 100.0 3 'No' 3]]
```

[45]:
```
df1=pd.DataFrame({'Name':['ankit','kiio'],'Age':[30,25]},index=['a','b'])
new_index=df1.index
df2=pd.DataFrame({'College':['ISI','IIIT-D'],'Rank':[0,1]},index=new_index)
print(df1)
print(df2)
```

```
    Name  Age
a  ankit   30
b   kiio   25
  College  Rank
a     ISI     0
b  IIIT-D     1
```

Index objects are immutable and thus can't be modified by the user

[46]:
```
label=pd.Index(np.arange(2))
df2=pd.DataFrame({'College':['ISI','IIIT-D'],'Rank':[0,1]},index=label)
print(df2)
```

```
  College  Rank
0     ISI     0
1  IIIT-D     1
```

[47]:
```
# reindexing

df1=pd.DataFrame({'Name':['ankit','kiio'],'Age':[30,25]},index=['a','b'])
df1=df1.reindex(['b','a','z'])
print(df1)
```

```
    Name   Age
b   kiio  25.0
a  ankit  30.0
z    NaN   NaN
```

For ordered data like time series, it may be desirable to do some interpolation or filling of values when reindexing. The method option allows us to do this, using a method such as ffill, which forward-fills the values.

[48]:
```
df1=pd.DataFrame({'Name':['ankit','kiio'],'Age':[30,25]},index=['a','b'])
df1=df1.reindex(['b','a','z'],method='ffill')
```

```
print(df1)
```

```
    Name  Age
b   kiio   25
a  ankit   30
z   kiio   25
```

[49]:
```
df=pd.DataFrame([[1.2,'ankit'],[9.7,'kiio'],[9.
 ↪5,'soumi']],index=['a','b','c'],columns=['Grade','Name'])
print(df)
df.drop('a',inplace=True)
print(df)
```

```
   Grade   Name
a    1.2  ankit
b    9.7   kiio
c    9.5  soumi
   Grade   Name
b    9.7   kiio
c    9.5  soumi
```

[50]:
```
df.drop(['Grade'],axis=1)
```

[50]:
```
    Name
b   kiio
c  soumi
```

[51]:
```
df.drop(index=['b','c'])
```

[51]:
```
Empty DataFrame
Columns: [Grade, Name]
Index: []
```

[52]:
```
df=pd.Series(['ankit','kiio','summi','soumi'],index=['a','b','c','d'])
print(df[1:4])
print(df[['b','c']])
```

```
b     kiio
c    summi
d    soumi
dtype: object
b     kiio
c    summi
dtype: object
```

[53]:
```
###### NER (Named Entity Recognition) ######
```

```python
import spacy

# Load the English model
nlp = spacy.load("en_core_web_sm")

text = "Barack Obama was born in Hawaii and was the president of the United␣
↪States."

# Process the text
doc = nlp(text)

# Extract and print named entities
for ent in doc.ents:
    print(ent.text, ent.label_)
```

```
Barack Obama PERSON
Hawaii GPE
the United States GPE
```

[54]:
```python
df=pd.
↪DataFrame(['ankit','kiio','summi','soumi'],index=['a','b','c','d'],columns=['Name'])
print(df[1:4]) # selecting rows
print(df[['Name']]) # selecting columns
print(df[df['Name']=='kiio']) # filtering condition
```

```
    Name
b   kiio
c  summi
d  soumi
    Name
a  ankit
b   kiio
c  summi
d  soumi
   Name
b  kiio
```

[55]:
```python
df=pd.DataFrame([[1.2,'ankit'],[9.7,'kiio'],[9.
↪5,'soumi']],index=['a','b','c'],columns=['Grade','Name'])
print(df)
print(df.loc['a',['Grade','Name']])
```

```
   Grade   Name
a    1.2  ankit
b    9.7   kiio
c    9.5  soumi
Grade      1.2
```

```
Name     ankit
Name: a, dtype: object
```

[56]:
```
print(df.iloc[1,[1,0]])
```

```
Name     kiio
Grade     9.7
Name: b, dtype: object
```

[57]:
```
s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])
s1
```

[57]:
```
a    7.3
c   -2.5
d    3.4
e    1.5
dtype: float64
```

[58]:
```
s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1],index=['a', 'c', 'e', 'f', 'g'])
s2
```

[58]:
```
a   -2.1
c    3.6
e   -1.5
f    4.0
g    3.1
dtype: float64
```

[59]:
```
s1+s2
```

[59]:
```
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

[60]:
```
df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)),columns=list('abcd'))
df1
```

[60]:

	a	b	c	d
0	0.0	1.0	2.0	3.0
1	4.0	5.0	6.0	7.0
2	8.0	9.0	10.0	11.0

[61]:
```
df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)),columns=list('abcde'))
df2
```

```
[61]:       a     b     c     d     e
      0   0.0   1.0   2.0   3.0   4.0
      1   5.0   6.0   7.0   8.0   9.0
      2  10.0  11.0  12.0  13.0  14.0
      3  15.0  16.0  17.0  18.0  19.0
```

```
[62]: df2.loc[1, 'b'] = np.nan
```

```
[63]: df2
```

```
[63]:       a     b     c     d     e
      0   0.0   1.0   2.0   3.0   4.0
      1   5.0   NaN   7.0   8.0   9.0
      2  10.0  11.0  12.0  13.0  14.0
      3  15.0  16.0  17.0  18.0  19.0
```

```
[64]: df1
```

```
[64]:      a    b     c     d
      0  0.0  1.0   2.0   3.0
      1  4.0  5.0   6.0   7.0
      2  8.0  9.0  10.0  11.0
```

```
[65]: df1 + df2
```

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

```
[66]: df1*df2
```

```
[66]:       a     b      c      d   e
      0   0.0   1.0    4.0    9.0 NaN
      1  20.0   NaN   42.0   56.0 NaN
      2  80.0  99.0  120.0  143.0 NaN
      3   NaN   NaN    NaN    NaN NaN
```

```
[67]: df1.add(df2, fill_value=0) # either in df1 or df2, whereever the value is nan,␣
        ↪it will be replaced by zero and
      # then do df1+df2 as usual
```

```
[67]:       a     b     c     d     e
      0   0.0   2.0   4.0   6.0   4.0
      1   9.0   5.0  13.0  15.0   9.0
      2  18.0  20.0  22.0  24.0  14.0
```

```
3  15.0  16.0  17.0  18.0  19.0
```

[68]:
```
arr = np.arange(12.).reshape((3, 4))
arr
```

[68]:
```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]])
```

[69]:
```
arr[0]
```

[69]:
```
array([0., 1., 2., 3.])
```

[70]:
```
arr-arr[0]
```

[70]:
```
array([[0., 0., 0., 0.],
       [4., 4., 4., 4.],
       [8., 8., 8., 8.]])
```

When we subtract arr[0] from arr, the subtraction is performed once for each row. This is referred to as broadcasting.

If you want to instead broadcast over the columns, matching on the rows, you have to use one of the arithmetic methods.

[71]:
```
frame=pd.DataFrame(np.arange(12).
↪reshape(4,3),columns=list('bde'),index=['Utah','Ohio','Texas','Oregon'])
frame
```

[71]:

	b	d	e
Utah	0	1	2
Ohio	3	4	5
Texas	6	7	8
Oregon	9	10	11

[72]:
```
series3 = frame['d']
series3
```

[72]:
```
Utah       1
Ohio       4
Texas      7
Oregon    10
Name: d, dtype: int64
```

[73]:
```
frame.sub(series3, axis='index')
```

[73]:

	b	d	e
Utah	-1	0	1

```
Ohio    -1  0  1
Texas   -1  0  1
Oregon -1  0  1
```

The axis number that you pass is the axis to match on. In this case we mean to match on the DataFrame's row index (axis='index' or axis=0) and broadcast across.

[74]:
```
# NumPy ufuncs (element-wise array methods) also work with pandas objects:

frame = pd.DataFrame(np.random.randn(4, 3), columns=list('bde'),index=['Utah',␣
 ↪'Ohio', 'Texas', 'Oregon'])
frame
```

[74]:
```
               b         d         e
Utah    0.829709 -0.077173 -0.072410
Ohio   -0.328474 -0.322371 -0.581475
Texas   0.619285  0.236105 -1.212218
Oregon  0.043921  1.539648 -0.300668
```

[75]:
```
np.abs(frame)
```

[75]:
```
               b         d         e
Utah    0.829709  0.077173  0.072410
Ohio    0.328474  0.322371  0.581475
Texas   0.619285  0.236105  1.212218
Oregon  0.043921  1.539648  0.300668
```

Another frequent operation is applying a function on one-dimensional arrays to each column or row. DataFrame's apply method does exactly this.

[76]:
```
frame
```

[76]:
```
               b         d         e
Utah    0.829709 -0.077173 -0.072410
Ohio   -0.328474 -0.322371 -0.581475
Texas   0.619285  0.236105 -1.212218
Oregon  0.043921  1.539648 -0.300668
```

[77]:
```
f = lambda x: x.max() - x.min()
frame.apply(f)
```

[77]:
```
b    1.158182
d    1.862019
e    1.139808
dtype: float64
```

[78]:
```python
# If you pass axis='columns' to apply, the function will be invoked once per␣
↪row instead
frame.apply(f, axis='columns')
```

[78]:
```
Utah      0.906882
Ohio      0.259104
Texas     1.831503
Oregon    1.840317
dtype: float64
```

[79]:
```python
def f(x):
    return pd.Series([x.min(), x.max()], index=['min', 'max'])
frame.apply(f)
```

[79]:
```
            b         d         e
min -0.328474 -0.322371 -1.212218
max  0.829709  1.539648 -0.072410
```

[80]:
```python
def f(x):
    return pd.Series([x.min(), x.max()], index=['min', 'max'])
frame.apply(f,axis='columns')
```

[80]:
```
             min       max
Utah   -0.077173  0.829709
Ohio   -0.581475 -0.322371
Texas  -1.212218  0.619285
Oregon -0.300668  1.539648
```

Element-wise Python functions can be used, too. Suppose you wanted to compute a formatted string from each floating-point value in frame. You can do this with apply map.

[81]:
```python
format = lambda x: '%.2f' % x
frame.applymap(format)
```

```
/tmp/ipykernel_5585/1073433956.py:2: FutureWarning: DataFrame.applymap has been
deprecated. Use DataFrame.map instead.
  frame.applymap(format)
```

[81]:
```
            b      d      e
Utah     0.83  -0.08  -0.07
Ohio    -0.33  -0.32  -0.58
Texas    0.62   0.24  -1.21
Oregon   0.04   1.54  -0.30
```

[82]:
```python
frame['e'].map(format)
```

[82]:
```
Utah      -0.07
Ohio      -0.58
Texas     -1.21
Oregon    -0.30
Name: e, dtype: object
```

[83]:
```python
pd.Series([1,3,2,7],index=['d','b','a','c']).sort_index()
```

[83]:
```
a    2
b    3
c    7
d    1
dtype: int64
```

[84]:
```python
frame = pd.DataFrame(np.arange(8).reshape((2, 4)),index=['three',
 ↪'one'],columns=['d', 'a', 'b', 'c'])
frame.sort_index()
```

[84]:

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

[85]:
```python
frame.sort_index(axis=1,ascending=True)
```

[85]:

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

[86]:
```python
obj = pd.Series([4, 7, -3, 2])
obj.sort_values()
```

[86]:
```
2   -3
3    2
0    4
1    7
dtype: int64
```

[87]:
```python
frame = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})
frame
```

[87]:

	b	a
0	4	0
1	7	1
2	-3	0
3	2	1

[88]:
```python
frame.sort_values(by='b')
```

[88]:
```
   b  a
2 -3  0
3  2  1
0  4  0
1  7  1
```

[89]:
```
frame.sort_values(by=['a', 'b']) # first sort by a and then b and when values␣
 ↪of a are same then for that
# values of b will be sorted
```

[89]:
```
   b  a
2 -3  0
0  4  0
3  2  1
1  7  1
```

[91]:
```
df = 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'])
df
```

[91]:
```
    one  two
a  1.40  NaN
b  7.10 -4.5
c   NaN  NaN
d  0.75 -1.3
```

[92]:
```
df.sum()
```

[92]:
```
one    9.25
two   -5.80
dtype: float64
```

[93]:
```
df.sum(axis=1)
```

[93]:
```
a    1.40
b    2.60
c    0.00
d   -0.55
dtype: float64
```

Some methods, like idxmin and idxmax, return indirect statistics like the index value where the minimum or maximum values are attained.

[94]:
```
df.describe()
```

[94]:
```
          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
```

[95]:
```
df.idxmax()
```

[95]:
```
one    b
two    d
dtype: object
```

[96]:
```
df.idxmin()
```

[96]:
```
one    d
two    b
dtype: object
```

Method –> Description

count –> Number of non-NA values
describe –> Compute set of summary statistics for Series or each DataFrame column
min, max –> Compute minimum and maximum values
argmin, argmax –> Compute index locations (integers) at which minimum or maximum value obtained, respectively
idxmin, idxmax –> Compute index labels at which minimum or maximum value obtained, respectively
quantile –> Compute sample quantile ranging from 0 to 1
sum –> Sum of values
mean –> Mean of values
median –> Arithmetic median (50% quantile) of values
mad –> Mean absolute deviation from mean value
prod –> Product of all values
var –> Sample variance of values
std –> Sample standard deviation of values
skew –> Sample skewness (third moment) of values
kurt –> Sample kurtosis (fourth moment) of values
cumsum –> Cumulative sum of values
cummin, cummax –> Cumulative minimum or maximum of values, respectively
cumprod –> Cumulative product of values
diff –> Compute first arithmetic difference (useful for time series)
pct_change –> Compute percent changes

[97]:
```
df.count()
```

[97]:
```
one     3
two     2
dtype: int64
```

[98]:
```
df.min()
```

[98]:
```
one     0.75
two    -4.50
dtype: float64
```

[99]:
```
df.max()
```

[99]:
```
one     7.1
two    -1.3
dtype: float64
```

[100]:
```
df.quantile()
```

[100]:
```
one     1.4
two    -2.9
Name: 0.5, dtype: float64
```

[101]:
```
df.sum()
```

[101]:
```
one     9.25
two    -5.80
dtype: float64
```

[102]:
```
df.mean()
```

[102]:
```
one     3.083333
two    -2.900000
dtype: float64
```

[103]:
```
df.median()
```

[103]:
```
one     1.4
two    -2.9
dtype: float64
```

[104]:
```
df.prod()
```

[104]:
```
one     7.455
two     5.850
dtype: float64
```

[105]:
```
df.var()
```

[105]:
```
one    12.205833
two     5.120000
dtype: float64
```

[106]:
```
df.std()
```

[106]:
```
one    3.493685
two    2.262742
dtype: float64
```

[107]:
```
df.skew()
```

[107]:
```
one    1.664846
two         NaN
dtype: float64
```

[108]:
```
df.kurt()
```

[108]:
```
one   NaN
two   NaN
dtype: float64
```

[109]:
```
df.cumsum()
```

[109]:
```
    one  two
a  1.40  NaN
b  8.50 -4.5
c   NaN  NaN
d  9.25 -5.8
```

[110]:
```
df.cumprod()
```

[110]:
```
     one   two
a  1.400   NaN
b  9.940 -4.50
c    NaN   NaN
d  7.455  5.85
```

[111]:
```
df.cummin()
```

[111]:
```
    one  two
a  1.40  NaN
b  1.40 -4.5
c   NaN  NaN
d  0.75 -4.5
```

[112]:
```
df.cummax()
```

[112]:
```
   one  two
a  1.4  NaN
b  7.1 -4.5
c  NaN  NaN
d  7.1 -1.3
```

[113]:
```
df.diff()
```

[113]:
```
   one  two
a  NaN  NaN
b  5.7  NaN
c  NaN  NaN
d  NaN  NaN
```

[114]:
```
df.pct_change()
```

```
/tmp/ipykernel_5585/890640361.py:1: FutureWarning: The default fill_method='pad'
in DataFrame.pct_change is deprecated and will be removed in a future version.
Either fill in any non-leading NA values prior to calling pct_change or specify
'fill_method=None' to not fill NA values.
  df.pct_change()
```

[114]:
```
        one       two
a       NaN       NaN
b  4.071429       NaN
c  0.000000  0.000000
d -0.894366 -0.711111
```

The corr method of Series computes the correlation of the overlapping, non-NA, aligned-by-index values in two Series. Relatedly, cov computes the covariance

[115]:
```
df['one'].corr(df['two'])
```

[115]: np.float64(-1.0)

[116]:
```
df['one'].cov(df['two'])
```

[116]: np.float64(-10.16)

value_counts –> Return a Series containing unique values as its index and frequencies as its values, ordered count in descending order

[117]:
```
import pandas as pd
data = pd.DataFrame({'Qu1': [1, 3, 4, 3, 4],'Qu2': [2, 3, 1, 2, 3],'Qu3': [1,␣
↪5, 2, 4, 4]})
data
```

[117]:

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

[118]:

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

```
/tmp/ipykernel_5585/1382616601.py:1: FutureWarning: pandas.value_counts is
deprecated and will be removed in a future version. Use
pd.Series(obj).value_counts() instead.
  result = data.apply(pd.value_counts).fillna(0)
```

[118]:

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

[ ]:

```

```