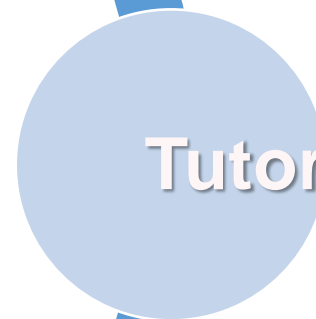




**Python 3x
Programing**



Lesson PANDAS



Tutor Mrs. Mỹ Linh



Time 90 mins

Content

- Python Pandas
- Series
- DataFrame
- Panel
- Basic Functionality
- Descriptive Statistics
- Function Application
- Reindexing
- Iteration
- Sorting
- Working with Text Data
- Options & Customization
- Indexing & Selecting Data
- Statistical Functions
- Window Functions
- Aggregations
- Missing Data
- GroupBy
- Merging/Joining
- Concatenation
- Date Functionality
- Timedelta
- Categorical Data
- Visualization
- IO Tools
- Sparse Data
- Caveats & Gotchas

Introduction to Pandas

Python Pandas

- Pandas is an open-source Python Library providing high-performance data manipulation and analysis tool using its powerful data structures. The name Pandas is derived from the word Panel Data – an Econometrics from Multidimensional data.
- To use Pandas, must **import pandas as pd**
- Pandas deals with the following three data structures
 - Series: dimension = 1
 - DataFrame: dimension = 2
 - Panel: dimension = 3
- Fast and efficient DataFrame object with default and customized indexing.
- Tools for loading data into in-memory data objects from different file formats.
- Data alignment and integrated handling of missing data.
- Reshaping and pivoting of date sets.
- Label-based slicing, indexing and subsetting of large data sets.
- Columns from a data structure can be deleted or inserted.
- Group by data for aggregation and transformations.
- High performance merging and joining of data.
- Time Series functionality.

Python Pandas - Series

- Create: **pandas.Series(data, index, dtype, copy)**
 - **Data:** data takes various forms like ndarray, list, constants
 - **Index:** Index values must be unique and hashable, same length as data. Default np.arange(n) if no index is passed.
 - **Dtype:** dtype is for data type. If None, data type will be inferred
 - **Copy:** Copy data. Default False

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
data = np.array(['a','b','c','d'])
s = pd.Series(data)
print(s)
```

0	a
1	b
2	c
3	d

- Retrieve Data Using Label

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve a single element

print (s[0])
print (s['a'])
print(s[1:5])
```

→	1	
	1	
	b	2
	c	3
	d	4
	e	5

Python Pandas - DataFrame

- **Create:**
 - `pandas.DataFrame(data, index, columns, dtype, copy)`
 - Columns: For column labels, the optional default syntax is - `np.arange(n)`. This is only true if no index is passed.
- Creating dataframe many ways
- Adding column
- Delete column
- Row Selection, Addition, and Deletion

Example – Create Dataframe

```
import pandas as pd
data = [1,2,3,4,5]
df = pd.DataFrame(data)
print(df)
```

```
print('More dataframe example')
```

```
data = [['Alex',10],['Bob',12],['Clarke',13]]
df1 = pd.DataFrame(data,columns=['Name','Age'])
print (df1)
```

```
data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}
df2 = pd.DataFrame(data)
print (df2)
```

```
data = [{'a': 1, 'b': 2},{ 'a': 5, 'b': 10, 'c': 20}]
df3 = pd.DataFrame(data)
print(df3)
```

```
data = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
       'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df4 = pd.DataFrame(data)
print(df4)
```

```
0
0  1
1  2
2  3
3  4
4  5

More dataframe example

      Name  Age
0    Alex   10
1     Bob   12
2  Clarke   13

      Age  Name
0    28   Tom
1    34  Jack
2    29  Steve
3    42  Ricky

      a  b  c
0  1  2  NaN
1  5 10 20.0

      one  two
a  1.0    1
b  2.0    2
c  3.0    3
d  NaN    4
```

Column Addition

```
import pandas as pd
```

```
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),  
     'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}
```

```
df = pd.DataFrame(d)
```

```
# Adding a new column to an existing DataFrame object with c  
    passing new series
```

```
print ("Adding a new column by passing as Series:")  
df['three']=pd.Series([10,20,30],index=['a','b','c'])  
print df
```

```
print ("Adding a new column using the existing columns in DataFrame:")  
df['four']=df['one']+df['three']  
  
print (df)
```

Adding a new column by passing as Series:

	one	two	three
a	1.0	1	10.0
b	2.0	2	20.0
c	3.0	3	30.0
d	NaN	4	NaN

Adding a new column using the existing columns in DataFrame:

	one	two	three	four
a	1.0	1	10.0	11.0
b	2.0	2	20.0	22.0
c	3.0	3	30.0	33.0
d	NaN	4	NaN	NaN

Column Deletion

```
# Using the previous DataFrame, we will delete a column
# using del function
import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd']),
     'three' : pd.Series([10,20,30], index=['a','b','c'])}

df = pd.DataFrame(d)
print ("Our dataframe is:")
print df

# using del function
print ("Deleting the first column using DEL function:")
del df['one']
print df

# using pop function
print ("Deleting another column using POP function:")
df.pop('two')
print df
```

Our dataframe is:

	one	three	two
a	1.0	10.0	1
b	2.0	20.0	2
c	3.0	30.0	3
d	NaN	NaN	4

Deleting the first column using DEL function:

	three	two
a	10.0	1
b	20.0	2
c	30.0	3
d	NaN	4

Deleting another column using POP function:

	three
a	10.0
b	20.0
c	30.0
d	NaN

Row Selection, Addition, and Deletion

```
import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)

print(df)

print('Selection by Label')
print (df.loc['b'])

print('Selection by Label')
print(df.iloc[2])

print('Slice Rows')
print (df[2:4])

print('Addition of Rows')
df2 = pd.DataFrame([[5, 6], [7, 8]], columns = ['a', 'b'])
df = df.append(df2)
print(df)

print('Drop rows with label ....')
df = df.drop(0)
df = df.drop('d')
df = df.drop('c')
df = df.drop(1)
print (df)
```

	one	two
a	1.0	1
b	2.0	2
c	3.0	3
d	NaN	4

Selection by Label

one	2.0
two	2.0

Name: b, dtype: float64

Selection by Label

one	3.0
two	3.0

Name: c, dtype: float64

Slice Rows

	one	two
c	3.0	3
d	NaN	4

Addition of Rows

	a	b	one	two
a	NaN	NaN	1.0	1.0
b	NaN	NaN	2.0	2.0
c	NaN	NaN	3.0	3.0
d	NaN	NaN	NaN	4.0
0	5.0	6.0	NaN	NaN
1	7.0	8.0	NaN	NaN

Drop rows with label

	a	b	one	two
a	NaN	NaN	1.0	1.0
b	NaN	NaN	2.0	2.0

Python Pandas - Panel

- Create: **pandas.Panel(data, items, major_axis, minor_axis, dtype, copy)**
- Data: Data takes various forms like ndarray, series, map, lists, dict, constants and also another DataFrame
- Items: axis=0
- Major_axis: axis=1
- Minor_axis: axis=2
- Dtype: Data type of each column
- Copy: Copy data. Default, false

```
# creating an empty panel
```

```
import pandas as pd
```

```
import numpy as np
```

```
data = np.random.rand(2,4,5)
```

```
p = pd.Panel(data)
```

```
print(p)
```

```
<class 'pandas.core.panel.Panel'>
```

```
Dimensions: 2 (items) x 4 (major_axis) x 5 (minor_axis)
```

```
Items axis: 0 to 1
```

```
Major_axis axis: 0 to 3
```

```
Minor_axis axis: 0 to 4
```

Example - From 3D ndarray

```
# creating an empty panel
import pandas as pd
import numpy as np
data = {'Item1' : pd.DataFrame(np.random.randn(4, 3)),
        'Item2' : pd.DataFrame(np.random.randn(4, 2))}
```

```
p = pd.Panel(data)
```

```
print(p)
```

```
print (p['Item1'])
```

```
print (p['Item2'])
```

```
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 4 (major_axis) x 3 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 0 to 3
Minor_axis axis: 0 to 2
```

	0	1	2
0	0.386301	0.937950	1.331670
1	-0.838045	-0.758695	-1.086383
2	0.278756	-0.402047	1.628165
3	0.673774	-0.432396	0.485400

	0	1	2
0	0.654729	0.836816	NaN
1	-0.010171	2.285872	NaN
2	-0.013479	-1.614701	NaN
3	-0.431427	1.147201	NaN

Series Basic Functionality

Sr.No.	Attribute or Method & Description
1	axes Returns a list of the row axis labels
2	dtype Returns the dtype of the object.
3	empty Returns True if series is empty.
4	ndim Returns the number of dimensions of the underlying data, by definition 1.

Series Basic Functionality

5	size Returns the number of elements in the underlying data.
6	values Returns the Series as ndarray.
7	head() Returns the first n rows.
8	tail() Returns the last n rows.

DataFrame Basic Functionality

Sr.No.	Attribute or Method & Description
1	T Transposes rows and columns.
2	axes Returns a list with the row axis labels and column axis labels as the only members.
3	dtypes Returns the dtypes in this object.
4	empty True if NDFrame is entirely empty [no items]; if any of the axes are of length 0.
5	ndim Number of axes / array dimensions.

DataFrame Basic Functionality

6	shape Returns a tuple representing the dimensionality of the DataFrame.
7	size Number of elements in the NDFrame.
8	values Numpy representation of NDFrame.
9	head() Returns the first n rows.
10	tail() Returns last n rows.

Function Application

- To apply your own or another library's functions to Pandas objects, you should be aware of the three important methods. The appropriate method to use depends on whether your function expects to operate on an entire DataFrame, row- or column-wise, or element wise.
 - Table wise Function Application: `pipe()`
 - Row or Column Wise Function Application: `apply()`
 - Element wise Function Application: `applymap()`

Function Application

- Suppose df is data frame and adder is function

```
def adder(num1,num2):  
    return num1+num2
```

- df = df.pipe(adder,2)

	Age	Score	Salary
0	29.0	10.2	2502.0
1	24.0	11.0	1002.0
2	47.0	9.6	502.0

	Age	Score	Salary
0	27.0	8.2	2500.0
1	22.0	9.0	1000.0
2	45.0	7.6	500.0

- df = df.apply(np.mean)

Age	31.333333
Score	8.266667
Salary	1333.333333
dtype:	float64

```
df = df['Salary'].map(lambda x:x*10)  
#On Series data
```

0	25000.0
1	10000.0
2	5000.0
Name:	Salary, dtype: float64

- df = df.apply(np.mean, axis = 1)

0	845.066667
1	343.666667
2	184.200000
dtype:	float64

```
df = df.applymap(lambda x:x*10)
```

	Age	Score	Salary
0	270.0	82.0	25000.0
1	220.0	90.0	10000.0
2	450.0	76.0	5000.0

```
df = df.apply(lambda x: x.max() - x.min())
```

Age	23.0
Score	1.4
Salary	2000.0
dtype:	float64

Re-indexing

- Reindexing changes the row labels and column labels of a DataFrame. To reindex means to conform the data to match a given set of labels along a particular axis.
- Multiple operations can be accomplished through indexing like –
 - Reorder the existing data to match a new set of labels.
 - Insert missing value (NA) markers in label locations where no data for the label existed.

Example

```
import pandas as pd
import numpy as np

N=20

df = pd.DataFrame({
    'A': pd.date_range(start='2016-01-01',periods=N,freq='D'),
    'x': np.linspace(0,stop=N-1,num=N),
    'y': np.random.rand(N),
    'C': np.random.choice(['Low','Medium','High'],N).tolist(),
    'D': np.random.normal(100, 10, size=(N)).tolist()
})

print(df)

print(' ')
print('=====')
print(' ')

#reindex the DataFrame
df_reindexed = df.reindex(index=[0,2,5], columns=['A', 'C', 'B'])

print(df_reindexed)
```

	A	C	D	x	y
0	2016-01-01	Medium	102.633441	0.0	0.736833
1	2016-01-02	High	104.292073	1.0	0.362471
2	2016-01-03	Medium	99.963524	2.0	0.841574
3	2016-01-04	Low	93.014575	3.0	0.917657
4	2016-01-05	Medium	104.145754	4.0	0.825684
5	2016-01-06	Medium	82.369978	5.0	0.188074
6	2016-01-07	Medium	107.769696	6.0	0.786211
7	2016-01-08	Low	106.559870	7.0	0.811664
8	2016-01-09	Low	92.231004	8.0	0.333942
9	2016-01-10	Medium	104.774480	9.0	0.252270
10	2016-01-11	Low	87.682552	10.0	0.858438
11	2016-01-12	Low	97.608726	11.0	0.468110
12	2016-01-13	Low	107.884712	12.0	0.505663
13	2016-01-14	Low	98.667701	13.0	0.707653
14	2016-01-15	Medium	84.764679	14.0	0.843188
15	2016-01-16	Medium	106.471790	15.0	0.535894
16	2016-01-17	High	90.854407	16.0	0.539273
17	2016-01-18	Medium	105.756181	17.0	0.323379
18	2016-01-19	Low	84.810183	18.0	0.654856
19	2016-01-20	Medium	105.197082	19.0	0.063267

=====

	A	C	B
0	2016-01-01	Medium	NaN
2	2016-01-03	Medium	NaN
5	2016-01-06	Medium	NaN

Re-index to Align with Other Objects

```
import pandas as pd
import numpy as np

df1 = pd.DataFrame(np.random.randn(10,3),columns=['col1','col2','col3'])
df2 = pd.DataFrame(np.random.randn(7,3),columns=['col1','col2','col3'])

df1 = df1.reindex_like(df2)
print df1
```

	col1	col2	col3
0	-2.467652	-1.211687	-0.391761
1	-0.287396	0.522350	0.562512
2	-0.255409	-0.483250	1.866258
3	-1.150467	-0.646493	-0.222462
4	0.152768	-2.056643	1.877233
5	-1.155997	1.528719	-1.343719
6	-1.015606	-1.245936	-0.295275

Filling while ReIndexing

- `reindex()` takes an optional parameter `method` which is a filling method with values as follows –
 - `pad/ffill` – Fill values forward
 - `bfill/backfill` – Fill values backward
 - `nearest` – Fill from the nearest index values

Example

```
import pandas as pd
import numpy as np

df1 = pd.DataFrame(np.random.randn(6,3),columns=['col1','col2','col3'])
df2 = pd.DataFrame(np.random.randn(2,3),columns=['col1','col2','col3'])

print(df1)
print(' ')
print(df2)

print(' ')

# Padding NAN's
print df2.reindex_like(df1)

print(' ')

# Now Fill the NAN's with preceding Values
print ("Data Frame with Forward Fill:")
print df2.reindex_like(df1,method='ffill')
```

	col1	col2	col3
0	0.477280	-0.440055	-1.239634
1	0.801811	0.388711	-0.345307
2	-0.745925	-0.287503	0.271269
3	-0.228431	-0.562865	-1.621816
4	-1.332601	1.451127	-0.459078
5	0.492429	0.695719	-0.322964

	col1	col2	col3
0	-0.891543	-0.364250	1.071647
1	-0.401467	0.191972	0.264598

	col1	col2	col3
0	-0.891543	-0.364250	1.071647
1	-0.401467	0.191972	0.264598
2	NaN	NaN	NaN
3	NaN	NaN	NaN
4	NaN	NaN	NaN
5	NaN	NaN	NaN

Data Frame with Forward Fill:

	col1	col2	col3
0	-0.891543	-0.364250	1.071647
1	-0.401467	0.191972	0.264598
2	-0.401467	0.191972	0.264598
3	-0.401467	0.191972	0.264598
4	-0.401467	0.191972	0.264598
5	-0.401467	0.191972	0.264598

Limits on Filling while Re-indexing

- The limit argument provides additional control over filling while reindexing. Limit specifies the maximum count of consecutive matches.

```
import pandas as pd
import numpy as np

df1 = pd.DataFrame(np.random.randn(6,3),columns=['col1','col2','col3'])
df2 = pd.DataFrame(np.random.randn(2,3),columns=['col1','col2','col3'])

# Padding NAN's
print df2.reindex_like(df1)

# Now Fill the NAN's with preceding Values
print ("Data Frame with Forward Fill limiting to 1:")
print df2.reindex_like(df1,method='ffill',limit=1)
```

	col1	col2	col3
0	0.247784	2.128727	0.702576
1	-0.055713	-0.021732	-0.174577
2	NaN	NaN	NaN
3	NaN	NaN	NaN
4	NaN	NaN	NaN
5	NaN	NaN	NaN

Data Frame with Forward Fill limiting to 1:

	col1	col2	col3
0	0.247784	2.128727	0.702576
1	-0.055713	-0.021732	-0.174577
2	-0.055713	-0.021732	-0.174577
3	NaN	NaN	NaN
4	NaN	NaN	NaN
5	NaN	NaN	NaN

Renaming

- The rename() method allows you to relabel an axis based on some mapping (a dict or Series) or an arbitrary function.

```
import pandas as pd
import numpy as np

df1 = pd.DataFrame(np.random.randn(6,3),columns=['col1','col2','col3'])
print df1

print ("After renaming the rows and columns:")
print df1.rename(columns={'col1' : 'c1', 'col2' : 'c2'},
index = {0 : 'apple', 1 : 'banana', 2 : 'durian'})
```

	col1	col2	col3
0	0.486791	0.105759	1.540122
1	-0.990237	1.007885	-0.217896
2	-0.483855	-1.645027	-1.194113
3	-0.122316	0.566277	-0.366028
4	-0.231524	-0.721172	-0.112007
5	0.438810	0.000225	0.435479

After renaming the rows and columns:

	c1	c2	col3
apple	0.486791	0.105759	1.540122
banana	-0.990237	1.007885	-0.217896
durian	-0.483855	-1.645027	-1.194113
3	-0.122316	0.566277	-0.366028
4	-0.231524	-0.721172	-0.112007
5	0.438810	0.000225	0.435479

ITERATION

- The behavior of basic iteration over Pandas objects depends on the type. When iterating over a Series, it is regarded as array-like, and basic iteration produces the values. Other data structures, like DataFrame and Panel, follow the **dict-like** convention of iterating over the **keys** of the objects.
- In short, basic iteration (for **i** in object) produces –
 - **Series** – values
 - **DataFrame** – column labels
 - **Panel** – item labels

ITERATOR COLUMN

- Iterating a DataFrame gives column names

```
import pandas as pd
import numpy as np
```

```
N=20
```

```
df = pd.DataFrame({
    'A': pd.date_range(start='2016-01-01', periods=N, freq='D'),
    'x': np.linspace(0, stop=N-1, num=N),
    'y': np.random.rand(N),
    'C': np.random.choice(['Low', 'Medium', 'High'], N).tolist(),
    'D': np.random.normal(100, 10, size=(N)).tolist()
})
```

```
for col in df:
    print(col)
```



A

C

D

x

y

ITERATOR ROWS

- To iterate over the rows of the DataFrame, we can use the following functions –
 - **iteritems()** – to iterate over the (key,value) pairs
 - **iterrows()** – iterate over the rows as (index,series) pairs
 - **itertuples()** – iterate over the rows as namedtuples

iteritems()

- Iterates over each column as key, value pair with label as key and column value as a Series object.

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(4,3),columns=['col1','col2','col3'])

print(df)

print(' ')

for key,value in df.iteritems():
    print(key,value)
```

```
...      col1      col2      col3
0 -1.064935  2.037650 -1.091317
1 -1.820371  0.981087 -0.685399
2  0.109807 -0.648325  0.254567
3 -0.905518 -0.437735 -0.096516

('col1', 0   -1.064935
1   -1.820371
2    0.109807
3   -0.905518
Name: col1, dtype: float64)
('col2', 0    2.037650
1    0.981087
2   -0.648325
3   -0.437735
Name: col2, dtype: float64)
('col3', 0   -1.091317
1   -0.685399
2    0.254567
3   -0.096516
Name: col3, dtype: float64)
```

iterrows()

- iterrows() returns the iterator yielding each index value along with a series containing the data in each row.

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(4,3),columns = ['col1','col2','col3'])

print(df)

print(' ')

for row_index,row in df.iterrows():
    print(row_index,row)
```

	col1	col2	col3
0	3.186601	2.278300	0.980039
1	1.227548	0.895289	-0.524095
2	0.168116	-0.021478	-1.476323
3	-0.427900	0.009018	0.347493

```
(0, col1    3.186601
col2    2.278300
col3    0.980039
Name: 0, dtype: float64)
(1, col1    1.227548
col2    0.895289
col3   -0.524095
Name: 1, dtype: float64)
(2, col1    0.168116
col2   -0.021478
col3   -1.476323
Name: 2, dtype: float64)
(3, col1   -0.427900
col2    0.009018
col3    0.347493
Name: 3, dtype: float64)
```

itertuples()

- `itertuples()` method will return an iterator yielding a named tuple for each row in the DataFrame. The first element of the tuple will be the row's corresponding index value, while the remaining values are the row values.

```
import pandas as pd
import numpy as np
```

```
df = pd.DataFrame(np.random.randn(4,3), columns = ['col1', 'col2', 'col3'])
```

```
print(df)
```

```
print(' ')
```

```
for row in df.itertuples():
    print(row)
```

	col1	col2	col3
0	-1.222998	-0.060763	-0.175401
1	0.609082	0.248033	-1.267356
2	-1.060177	-0.023235	0.875370
3	1.575262	0.770238	-0.049036

```
Pandas(Index=0, col1=-1.2229981344593324, col2=-0.060762990782568256, col3=-0.17540080460233923)
Pandas(Index=1, col1=0.60908214159083529, col2=0.24803265674889541, col3=-1.2673562738450024)
Pandas(Index=2, col1=-1.0601774305508465, col2=-0.023234683852895711, col3=0.8753702029249425)
Pandas(Index=3, col1=1.5752618196577792, col2=0.77023815349641189, col3=-0.049036106021165177)
```

Example

```
import pandas as pd
import numpy as np
```

```
df = pd.DataFrame(np.random.randn(4,3),columns = ['col1','col2','col3'])
```

```
print(df)
```

```
print(' ')
```

```
for index, row in df.iterrows():
    row['a'] = 10
print(df)
```

	col1	col2	col3
0	0.558406	0.722226	1.270489
1	2.213536	-0.448291	0.617900
2	-0.758190	-0.293903	0.904212
3	1.461615	0.031728	0.417533

	col1	col2	col3
0	0.558406	0.722226	1.270489
1	2.213536	-0.448291	0.617900
2	-0.758190	-0.293903	0.904212
3	1.461615	0.031728	0.417533

Sorting

- There are two kinds of sorting available in Pandas. They are –
 - By label
 - By Actual Value
- Look at data generating randomly

```
import pandas as pd
import numpy as np

df=pd.DataFrame(np.random.randn(10,2),
index=[1,4,6,2,3,5,9,8,0,7],
columns=['col2','col1'])
```



	col2	col1
1	0.197920	-0.502069
4	1.610500	-1.253438
6	0.329770	-1.862410
2	0.798931	-0.823565
3	-0.412609	-1.244844
5	1.492556	-0.124418
9	-0.344938	-1.154500
8	1.694326	0.298172
0	0.000128	-1.884862
7	-1.541107	1.006505

Sorting Example

```
sorted_df_1=df.sort_index()
```

```
print(sorted_df_1)
```



	col2	col1
0	0.000128	-1.884862
1	0.197920	-0.502069
2	0.798931	-0.823565
3	-0.412609	-1.244844
4	1.610500	-1.253438
5	1.492556	-0.124418
6	0.329770	-1.862410
7	-1.541107	1.006505
8	1.694326	0.298172
9	-0.344938	-1.154500

```
sorted_df_2 = df.sort_index(ascending=False)
```

```
print(sorted_df_2)
```



	col2	col1
9	-0.344938	-1.154500
8	1.694326	0.298172
7	-1.541107	1.006505
6	0.329770	-1.862410
5	1.492556	-0.124418
4	1.610500	-1.253438
3	-0.412609	-1.244844
2	0.798931	-0.823565
1	0.197920	-0.502069
0	0.000128	-1.884862

Sorting Example

```
sorted_df_3=df.sort_index(axis=1)
```

```
print(sorted_df_3)
```



	col1	col2
1	-0.502069	0.197920
4	-1.253438	1.610500
6	-1.862410	0.329770
2	-0.823565	0.798931
3	-1.244844	-0.412609
5	-0.124418	1.492556
9	-1.154500	-0.344938
8	0.298172	1.694326
0	-1.884862	0.000128
7	1.006505	-1.541107

```
sorted_df_4 = df.sort_values(by='col1')
```

```
print(sorted_df_4)
```



	col2	col1
0	0.000128	-1.884862
6	0.329770	-1.862410
4	1.610500	-1.253438
3	-0.412609	-1.244844
9	-0.344938	-1.154500
2	0.798931	-0.823565
1	0.197920	-0.502069
5	1.492556	-0.124418
8	1.694326	0.298172
7	-1.541107	1.006505

Sorting Example

```
sorted_df_5 = df.sort_values(by='col1', kind='mergesort')
```

```
print(sorted_df_5)
```



	col2	col1
0	0.000128	-1.884862
6	0.329770	-1.862410
4	1.610500	-1.253438
3	-0.412609	-1.244844
9	-0.344938	-1.154500
2	0.798931	-0.823565
1	0.197920	-0.502069
5	1.492556	-0.124418
8	1.694326	0.298172
7	-1.541107	1.006505

```
sorted_df_6 = df.sort_values(by=['col1', 'col2'])
```

```
print(sorted_df_6)
```



	col2	col1
0	0.000128	-1.884862
6	0.329770	-1.862410
4	1.610500	-1.253438
3	-0.412609	-1.244844
9	-0.344938	-1.154500
2	0.798931	-0.823565
1	0.197920	-0.502069
5	1.492556	-0.124418
8	1.694326	0.298172
7	-1.541107	1.006505

Working with Text Data

- Pandas provides a set of string functions which make it easy to operate on string data. Most importantly, these functions ignore (or exclude) missing/NaN values.

Sr.No	Function & Description
1	lower() Converts strings in the Series/Index to lower case.
2	upper() Converts strings in the Series/Index to upper case.
3	len() Computes String length().
4	strip() Helps strip whitespace(including newline) from each string in the Series/index from both the sides.
5	split(' ') Splits each string with the given pattern.
6	cat(sep=' ') Concatenates the series/index elements with given separator.
7	get_dummies() Returns the DataFrame with One-Hot Encoded values.

Working with Text Data

8	contains(pattern) Returns a Boolean value True for each element if the substring contains in the element, else False.
9	replace(a,b) Replaces the value a with the value b .
10	repeat(value) Repeats each element with specified number of times.
11	count(pattern) Returns count of appearance of pattern in each element.
12	startswith(pattern) Returns true if the element in the Series/Index starts with the pattern.
13	endswith(pattern) Returns true if the element in the Series/Index ends with the pattern.
14	find(pattern) Returns the first position of the first occurrence of the pattern.

Working with Text Data

15	findall(pattern) Returns a list of all occurrence of the pattern.
16	swapcase Swaps the case lower/upper.
17	islower() Checks whether all characters in each string in the Series/Index in lower case or not. Returns Boolean
18	isupper() Checks whether all characters in each string in the Series/Index in upper case or not. Returns Boolean.
19	isnumeric() Checks whether all characters in each string in the Series/Index are numeric. Returns Boolean.

Options and Customization

- `get_option(param)`: `get_option` takes a single parameter and returns the value as given in the table
- `set_option(param,value)`: `set_option` takes two arguments and sets the value to the parameter as shown table
- `reset_option(param)`: takes an argument and sets the value back to the default value.
- `describe_option(param)`: `describe_option` prints the description of the argument.
- `option_context()`: `option_context` context manager is used to set the option in with statement temporarily. Option values are restored automatically when you exit the with block

Sr.No	Parameter & Description
1	display.max_rows Displays maximum number of rows to display
2	2 display.max_columns Displays maximum number of columns to display
3	display.expand_frame_repr Displays DataFrames to Stretch Pages
4	display.max_colwidth Displays maximum column width
5	display.precision Displays precision for decimal numbers

Indexing and Selecting Data in Pandas

Indexing and Selecting Data

- The Python and NumPy indexing operators "[]" and attribute operator "." provide quick and easy access to Pandas data structures across a wide range of use cases. However, since the type of the data to be accessed isn't known in advance, directly using standard operators has some optimization limits. For production code, we recommend that you take advantage of the optimized pandas data access methods explained.
- Pandas now supports three types of Multi-axes indexing; the three types are mentioned in the following table.

Sr.No	Indexing & Description
1	.loc() Label based
2	.iloc() Integer based
3	.ix() Both Label and Integer based

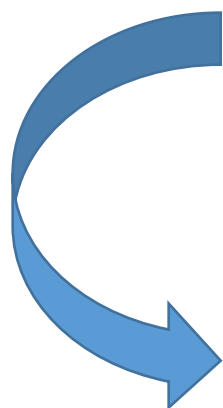
.loc()

- Pandas provide various methods to have purely label based indexing. When slicing, the start bound is also included. Integers are valid labels, but they refer to the label and not the position.
- **.loc()** has multiple access methods like:
 - A single scalar label
 - A list of labels
 - A slice object
 - A Boolean array
- **loc** takes two single/list/range operator separated by ','. The first one indicates the row and the second one indicates columns.

.loc() Example

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
```

```
df = pd.DataFrame(np.random.randn(8, 4),
index = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'],
columns = ['A', 'B', 'C', 'D'])
```



	A	B	C	D
a	1.695355	0.462850	-0.644750	1.339618
b	-0.224149	-0.830238	-0.183428	1.272660
c	-1.691320	-0.729269	-1.635839	-0.395096
d	0.308963	-0.977447	-0.446715	-1.427920
e	-0.912702	0.628778	1.460212	0.588769
f	0.732504	-0.214279	0.498870	-1.508137
g	1.301601	-1.564609	-0.058068	-0.612667
h	0.729417	2.626195	0.401886	0.290472

.loc() Example

```
#select all rows for a specific column
print(df.loc[:, 'A'])
```

	A	B	C	D
a	1.695355	0.462850	-0.644750	1.339618
b	-0.224149	-0.830238	-0.183428	1.272660
c	-1.691320	-0.729269	-1.635839	-0.395096
d	0.308963	-0.977447	-0.446715	-1.427920
e	-0.912702	0.628778	1.460212	0.588769
f	0.732504	-0.214279	0.498870	-1.508137
g	1.301601	-1.564609	-0.058068	-0.612667
h	0.729417	2.626195	0.401886	0.290472

```
a    1.695355
b   -0.224149
c   -1.691320
d    0.308963
e   -0.912702
f    0.732504
g    1.301601
h    0.729417
Name: A, dtype: float64
```

.loc() Example

```
# Select all rows for multiple columns, say list[]
print(df.loc[:,['A','C']])
```

	A	B	C	D		A	C
a	1.695355	0.462850	-0.644750	1.339618	a	1.695355	-0.644750
b	-0.224149	-0.830238	-0.183428	1.272660	b	-0.224149	-0.183428
c	-1.691320	-0.729269	-1.635839	-0.395096	c	-1.691320	-1.635839
d	0.308963	-0.977447	-0.446715	-1.427920	d	0.308963	-0.446715
e	-0.912702	0.628778	1.460212	0.588769	e	-0.912702	1.460212
f	0.732504	-0.214279	0.498870	-1.508137	f	0.732504	0.498870
g	1.301601	-1.564609	-0.058068	-0.612667	g	1.301601	-0.058068
h	0.729417	2.626195	0.401886	0.290472	h	0.729417	0.401886

.loc() Example

```
# Select few rows for multiple columns, say list[]
print(df.loc[['a','b','f','h'],['A','C']])
```

	A	B	C	D
a	1.695355	0.462850	-0.644750	1.339618
b	-0.224149	-0.830238	-0.183428	1.272660
c	-1.691320	-0.729269	-1.635839	-0.395096
d	0.308963	-0.977447	-0.446715	-1.427920
e	-0.912702	0.628778	1.460212	0.588769
f	0.732504	-0.214279	0.498870	-1.508137
g	1.301601	-1.564609	-0.058068	-0.612667
h	0.729417	2.626195	0.401886	0.290472

	A	C
a	1.695355	-0.644750
b	-0.224149	-0.183428
f	0.732504	0.498870
h	0.729417	0.401886

.loc() Example

```
# Select range of rows for all columns
print(df.loc['a':'h'])
```

	A	B	C	D
a	1.695355	0.462850	-0.644750	1.339618
b	-0.224149	-0.830238	-0.183428	1.272660
c	-1.691320	-0.729269	-1.635839	-0.395096
d	0.308963	-0.977447	-0.446715	-1.427920
e	-0.912702	0.628778	1.460212	0.588769
f	0.732504	-0.214279	0.498870	-1.508137
g	1.301601	-1.564609	-0.058068	-0.612667
h	0.729417	2.626195	0.401886	0.290472

	A	B	C	D
a	1.695355	0.462850	-0.644750	1.339618
b	-0.224149	-0.830238	-0.183428	1.272660
c	-1.691320	-0.729269	-1.635839	-0.395096
d	0.308963	-0.977447	-0.446715	-1.427920
e	-0.912702	0.628778	1.460212	0.588769
f	0.732504	-0.214279	0.498870	-1.508137
g	1.301601	-1.564609	-0.058068	-0.612667
h	0.729417	2.626195	0.401886	0.290472

.loc() Example

```
# for getting values with a boolean array
print(df.loc['a']>0)
```

	A	B	C	D
a	1.695355	0.462850	-0.644750	1.339618
b	-0.224149	-0.830238	-0.183428	1.272660
c	-1.691320	-0.729269	-1.635839	-0.395096
d	0.308963	-0.977447	-0.446715	-1.427920
e	-0.912702	0.628778	1.460212	0.588769
f	0.732504	-0.214279	0.498870	-1.508137
g	1.301601	-1.564609	-0.058068	-0.612667
h	0.729417	2.626195	0.401886	0.290472

A True
B True
C False
D True
Name: a, dtype: bool

.iloc()

- Pandas provide various methods in order to get purely integer based indexing. Like python and numpy, these are **0-based** indexing.
- The various access methods are as follows:
 - An Integer
 - A list of integers
 - A range of values

.iloc() Example

```
# import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
```

```
df = pd.DataFrame(np.random.randn(8, 4), columns = ['A', 'B', 'C', 'D'])
```

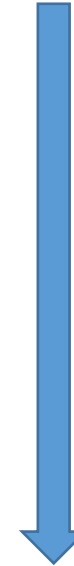
```
#print
print(df)
```

	A	B	C	D
0	0.176753	-0.775588	0.776157	-0.557363
1	1.201638	-1.326106	-1.184976	-0.766993
2	0.411747	1.221240	0.531377	-1.802342
3	0.038384	0.455340	-0.183154	1.001551
4	1.411158	0.245044	0.550815	-0.373446
5	0.638930	-1.200300	-0.429424	-0.533487
6	-0.318424	-0.888178	-0.179147	-0.519741
7	-0.855873	-0.574493	-0.657266	-1.156689

.iloc() Example

```
# select all rows for a specific column  
print(df.iloc[:4])
```

	A	B	C	D
0	0.176753	-0.775588	0.776157	-0.557363
1	1.201638	-1.326106	-1.184976	-0.766993
2	0.411747	1.221240	0.531377	-1.802342
3	0.038384	0.455340	-0.183154	1.001551
4	1.411158	0.245044	0.550815	-0.373446
5	0.638930	-1.200300	-0.429424	-0.533487
6	-0.318424	-0.888178	-0.179147	-0.519741
7	-0.855873	-0.574493	-0.657266	-1.156689



	A	B	C	D
0	0.176753	-0.775588	0.776157	-0.557363
1	1.201638	-1.326106	-1.184976	-0.766993
2	0.411747	1.221240	0.531377	-1.802342
3	0.038384	0.455340	-0.183154	1.001551

.iloc() Example

	A	B	C	D
0	0.176753	-0.775588	0.776157	-0.557363
1	1.201638	-1.326106	-1.184976	-0.766993
2	0.411747	1.221240	0.531377	-1.802342
3	0.038384	0.455340	-0.183154	1.001551

```
# Integer slicing
print(df.iloc[:4])
print(df.iloc[1:5, 2:4])
```

	C	D
1	-1.184976	-0.766993
2	0.531377	-1.802342
3	-0.183154	1.001551
4	0.550815	-0.373446

	A	B	C	D
0	0.176753	-0.775588	0.776157	-0.557363
1	1.201638	-1.326106	-1.184976	-0.766993
2	0.411747	1.221240	0.531377	-1.802342
3	0.038384	0.455340	-0.183154	1.001551
4	1.411158	0.245044	0.550815	-0.373446
5	0.638930	-1.200300	-0.429424	-0.533487
6	-0.318424	-0.888178	-0.179147	-0.519741
7	-0.855873	-0.574493	-0.657266	-1.156689

.iloc() Example

	B	D
1	-1.326106	-0.766993
3	0.455340	1.001551
5	-1.200300	-0.533487

	A	B	C	D
1	1.201638	-1.326106	-1.184976	-0.766993
2	0.411747	1.221240	0.531377	-1.802342

	A	B	C	D
0	0.176753	-0.775588	0.776157	-0.557363
1	1.201638	-1.326106	-1.184976	-0.766993
2	0.411747	1.221240	0.531377	-1.802342
3	0.038384	0.455340	-0.183154	1.001551
4	1.411158	0.245044	0.550815	-0.373446
5	0.638930	-1.200300	-0.429424	-0.533487
6	-0.318424	-0.888178	-0.179147	-0.519741
7	-0.855873	-0.574493	-0.657266	-1.156689

```
# Slicing through list of values
print(df.iloc[[1, 3, 5], [1, 3]])
print(df.iloc[1:3, :])
print(df.iloc[:, 1:3])
```

	B	C
0	-0.775588	0.776157
1	-1.326106	-1.184976
2	1.221240	0.531377
3	0.455340	-0.183154
4	0.245044	0.550815
5	-1.200300	-0.429424
6	-0.888178	-0.179147
7	-0.574493	-0.657266

.ix()

- Besides pure label based and integer based, Pandas provides a hybrid method for selections and subsetting the object using the .ix() operator.

```
import pandas as pd
import numpy as np
```

```
df = pd.DataFrame(np.random.randn(8, 4), columns = ['A', 'B', 'C', 'D'])
```

```
#df
print(df)
```



	A	B	C	D
0	0.378898	0.133670	0.136070	0.127399
1	-0.499039	2.357291	-1.150006	0.935712
2	-2.329760	-0.380842	0.063687	1.551128
3	-1.400219	0.317153	0.651748	-1.084645
4	-0.252273	-0.652334	-1.204376	1.341390
5	0.623551	-0.820163	0.610148	0.894935
6	-1.855714	-0.442705	0.665694	0.374564
7	-1.282746	-0.646424	-0.021149	0.006043

.ix() Example

```
# Integer slicing  
print(df.ix[:4])
```

	A	B	C	D
0	0.378898	0.133670	0.136070	0.127399
1	-0.499039	2.357291	-1.150006	0.935712
2	-2.329760	-0.380842	0.063687	1.551128
3	-1.400219	0.317153	0.651748	-1.084645
4	-0.252273	-0.652334	-1.204376	1.341390

```
# Index slicing  
print(df.ix[:, 'A'])
```

0	0.378898
1	-0.499039
2	-2.329760
3	-1.400219
4	-0.252273
5	0.623551
6	-1.855714
7	-1.282746

Name: A, dtype: float64

	A	B	C	D
0	0.378898	0.133670	0.136070	0.127399
1	-0.499039	2.357291	-1.150006	0.935712
2	-2.329760	-0.380842	0.063687	1.551128
3	-1.400219	0.317153	0.651748	-1.084645
4	-0.252273	-0.652334	-1.204376	1.341390
5	0.623551	-0.820163	0.610148	0.894935
6	-1.855714	-0.442705	0.665694	0.374564
7	-1.282746	-0.646424	-0.021149	0.006043

Use of Notations

- Getting values from the Pandas object with Multi-axes indexing uses the following notation
- **Note:** `.iloc()` & `.ix()` applies the same indexing options and Return value.

Object	Indexers	Return Type
Series	<code>s.loc[indexer]</code>	Scalar value
DataFrame	<code>df.loc[row_index,col_index]</code>	Series object
Panel	<code>p.loc[item_index,major_index, minor_index]</code>	<code>p.loc[item_index,major_index, minor_index]</code>

(Example 1) Use the basic indexing operator '[']'

```
import pandas as pd
import numpy as np
df = pd.DataFrame(np.random.randn(8, 4), columns = ['A', 'B', 'C', 'D'])

print(df)
```

	A	B	C	D
0	0.170981	-1.474156	0.544007	-1.918815
1	-0.183828	0.322550	0.443701	-0.531228
2	0.114509	-0.473415	-1.736726	-1.137762
3	-0.630348	-0.268956	0.981704	1.121474
4	-0.121557	-1.798246	0.551525	-0.072194
5	0.345434	-1.330808	1.411509	1.100317
6	0.117264	1.135388	-1.672977	-0.126768
7	-0.622439	0.918388	0.936736	0.945120

(Example 1) Use the basic indexing operator '[']'

```
0    0.170981
1   -0.183828
2    0.114509
3   -0.630348
4   -0.121557
5    0.345434
6    0.117264
7   -0.622439
Name: A, dtype: float64
```

`print(df['A'])`

`print(df[['A', 'B']])`

`print(df[2:2])`

```
      A      B
0  0.170981 -1.474156
1 -0.183828  0.322550
2  0.114509 -0.473415
3 -0.630348 -0.268956
4 -0.121557 -1.798246
5  0.345434 -1.330808
6  0.117264  1.135388
7 -0.622439  0.918388
Empty DataFrame
Columns: [A, B, C, D]
Index: []
```

```
      A      B      C      D
0  0.170981 -1.474156  0.544007 -1.918815
1 -0.183828  0.322550  0.443701 -0.531228
2  0.114509 -0.473415 -1.736726 -1.137762
3 -0.630348 -0.268956  0.981704  1.121474
4 -0.121557 -1.798246  0.551525 -0.072194
5  0.345434 -1.330808  1.411509  1.100317
6  0.117264  1.135388 -1.672977 -0.126768
7 -0.622439  0.918388  0.936736  0.945120
```

Empty DataFrame
Columns: [A, B, C, D]
Index: []

Sort, Filter, Aggregation, Grouping, Pivot, Concatenation, Merge/Join in Pandas

Sort

- Sort theo 1 column, mặc định là tăng dần: `df.sort_values(by='TOTAL')`

	ID	USER_ID	PRODUCT_ID	SUBTOTAL	TAX	TOTAL	DISCOUNT	CREATED_AT	QUANTITY
92	93	17	15	25.098764	0.00	25.175195	NaN	2017-06-18T11:15:50.035	4
75	76	15	185	26.384667	1.72	28.098903	NaN	2016-12-19T19:40:17.782	2
5	6	1	60	29.802148	1.64	31.441679	NaN	2019-11-06T16:38:50.134	3
70	71	12	161	31.727470	1.27	32.940866	NaN	2017-09-01T11:51:46.788	4
69	70	12	22	32.136780	1.29	33.418084	NaN	2019-11-21T11:21:36.739	3

- Sort theo thứ tự giảm dần: `df.sort_values(by='TOTAL', ascending=False)`
- Sort theo nhiều trường: `df.sort_values(by=['QUANTITY', 'TOTAL'])`
- Sort nhiều trường theo thứ tự khác nhau: `df.sort_values(by=['QUANTITY', 'TOTAL'], ascending=[True, False])`

Filter (lọc dữ liệu)

- Filter lấy ra các cột của dataframe: `df.filter(items=['USER_ID', 'TAX'])`
- Filter lấy ra các cột theo regular expression: `df.filter(regex='T$', axis=1)`

	DISCOUNT	CREATED_AT
0	NaN	2019-02-11T21:40:27.892
1	NaN	2018-05-15T08:04:04.58
2	6.416679	2019-12-06T22:22:48.544
3	NaN	2019-08-22T16:30:42.392
4	NaN	2018-10-10T03:34:47.309
...

Filter

- Filter các row chứa ký tự: `df.filter(like='bbi', axis=0)`
- Filter các row theo biểu thức so sánh
 - Ví dụ lấy tất cả các order có TOTAL lớn hơn 100: `df[df['TOTAL'] > 100]`
- Filter theo một hàm tự định nghĩa

```
def custom(tax, total):  
    return ( total - tax > 100)
```

```
df[custom(df['TAX'], df['TOTAL'])]
```

Aggregation

```
# importing pandas package
import pandas as pd

# making data frame from csv file
df = pd.read_csv("nba.csv")

# printing the first 10 rows of the dataframe
print(df[:10])
```

Name	Team	Number	Position	Age	Height	Weight	College	Salary
Avery Bradley	Boston Celtics	0	PG	25	180	77.2	Texas	7730337
Jae Crowder	Boston Celtics	99	SF	25	172	65	Georgia State	6796117
John Holland	Boston Celtics	30	SG	27	165	55	Boston University	
R.J. Hunter	Boston Celtics	28	SG	22	177	85	Georgia State	1148640
Jonas Jerebko	Boston Celtics	8	PF	29	198	100		5000000
Bojan Bogdanovic	Brooklyn Nets	44	SG	27	150	52		3425510
Markel Brown	Brooklyn Nets	22	SG	24	188	90	Oklahoma State	845059
Arron Afflalo	New York Knicks	4	SG	30	175	70	UCLA	8000000
Lou Amundson	New York Knicks	17	PF	33	171	72	UCLA	1635476
Elton Brand	Philadelphia 76ers	42	PF	37	158	60	UCLA	
Isaiah Canaan	Philadelphia 76ers	0	PG	25	179	70	UCLA	947276
Robert Covington	Philadelphia 76ers	33	SF	25	180	78	Georgia State	1000000
Joel Embiid	Philadelphia 76ers	21	C	22	179	76	Texas	4626960
Bismack Biyombo	Toronto Raptors	8	C	23	169	68.5		2814000
Bruno Caboclo	Toronto Raptors	20	SF	20	170	73.5		1524000

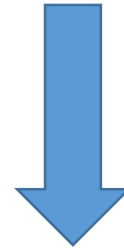
Example

Applying aggregation across all the columns

sum and min will be found for each

```
print(df.agg(['sum', 'min']))
```

```
print(df.agg([np.sum, np.min]))
```



	Name	...	Salary			
sum	Avery Bradley	Jae Crowder	John Holland	R.J. Hunter	...	45493375.0
min				Arron Afflalo	...	845059.0

Example

We are going to find aggregation for these columns

```
newAggre = df.aggregate({"Number":['sum', 'min'],  
                          "Age":['max', 'min'],  
                          "Weight":['min', 'sum'],  
                          "Salary":['sum']})
```

```
print(newAggre)
```

	Number	Age	Weight	Salary
max	NaN	37.0	NaN	NaN
min	0.0	20.0	52.0	NaN
sum	376.0	NaN	1092.2	45493375.0

Example

```
#Apply Different Functions to Different Columns of a Dataframe  
print(df.aggreate({'Age' : np.sum, 'Salary' : np.mean}))
```

```
Age          3.940000e+02  
Salary       3.499490e+06  
dtype: float64
```

Example

```
print(df[['Number', 'Age', 'Weight', 'Salary']].aggregate(np.sum))
```

```
Number          376.0  
Age             394.0  
Weight         1092.2  
Salary        45493375.0  
dtype: float64
```

Group

```
print(df.groupby('Team'))
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x000001F036F69F98>
```

Group

```
print(df.groupby('Team').groups)
```

```
{'Boston Celtics': Int64Index([0, 1, 2, 3, 4], dtype='int64'), 'Brooklyn Nets': Int64Index([5, 6], dtype='int64'), 'New York  
Knicks': Int64Index([7, 8], dtype='int64'), 'Philadelphia 76ers': Int64Index([9, 10, 11, 12], dtype='int64'), 'Toronto Raptors':  
Int64Index([13, 14], dtype='int64')}
```

Group

```
print(df.groupby( 'Team' )[ 'Age' ].agg([ 'count' ]))
```

	count
Team	
Boston Celtics	5
Brooklyn Nets	2
New York Knicks	2
Philadelphia 76ers	4
Toronto Raptors	2

Group

```
print(df.groupby('Team')['Age'].agg(['count', 'min', 'max', 'mean']))
```

	count	min	max	mean
Team				
Boston Celtics	5	22.0	29.0	25.60
Brooklyn Nets	2	24.0	27.0	25.50
New York Knicks	2	30.0	33.0	31.50
Philadelphia 76ers	4	22.0	37.0	27.25
Toronto Raptors	2	20.0	23.0	21.50

Group

```
print(df.groupby( 'Team')[ 'Age', 'Weight' ].agg([ 'count', 'min', 'max', 'mean' ]))
```

Team	Age				Weight			
	count	min	max	mean	count	min	max	mean
Boston Celtics	5	22.0	29.0	25.60	5	55.0	100.0	76.44
Brooklyn Nets	2	24.0	27.0	25.50	2	52.0	90.0	71.00
New York Knicks	2	30.0	33.0	31.50	2	70.0	72.0	71.00
Philadelphia 76ers	4	22.0	37.0	27.25	4	60.0	78.0	71.00
Toronto Raptors	2	20.0	23.0	21.50	2	68.5	73.5	71.00

Group

```
print(df.groupby(['Team','College'])['Age','Weight'].agg(['count','min','max','mean']))
```

		Age		Weight				
		count	min	max	...	min	max	mean
Team	College							
Boston Celtics	Boston University	1	27.0	27.0	...	55.0	55.0	55.0
	Georgia State	2	22.0	25.0	...	65.0	85.0	75.0
	Texas	1	25.0	25.0	...	77.2	77.2	77.2
Brooklyn Nets	Oklahoma State	1	24.0	24.0	...	90.0	90.0	90.0
New York Knicks	UCLA	2	30.0	33.0	...	70.0	72.0	71.0
Philadelphia 76ers	Georgia State	1	25.0	25.0	...	78.0	78.0	78.0
	Texas	1	22.0	22.0	...	76.0	76.0	76.0
	UCLA	2	25.0	37.0	...	60.0	70.0	65.0

Grouping with user-define function

- Chẳng hạn group lại theo Team và lấy ra tổng số tuổi của 10 bản ghi đầu tiên

```
def custom_aggregate(series):  
    return series.head(10).sum()
```

```
df.groupby(['Team'])['Age'].agg(custom_aggregate)
```

Pivot

- One of the most common tasks in data science is to manipulate the data frame we have to a specific format.
- Give data about life expectancy (expectancy refers to the number of years a person is expected to live based on the statistical average. Life expectancy varies by geographical area and by era.)
- Python Pandas function `pivot_table` help us with the summarization and conversion of dataframe in long form to dataframe in wide form, in a variety of complex scenarios.

continent	year	lifeExp	continent	Africa	Americas	Asia	Europe	Oceania
Europe	1972	69.210	year					
Asia	1992	75.190	1952	30.000	37.579	28.801	43.585	69.12
Asia	1987	53.914	1957	31.570	40.696	30.332	48.079	70.26
Americas	1962	70.210	1962	32.767	43.428	31.997	52.098	70.93
Europe	1967	69.610	1967	34.113	45.032	34.020	54.336	71.10
			1972	35.400	46.714	36.088	57.005	71.89

Raw data: df

Pivot

Pandas Simple Pivot

- A simple example of Python Pivot using a dataframe with just two columns. Let us subset our dataframe to contain just two columns, continent and lifeExp

continent	lifeExp
Africa	72.301
Africa	57.678
Europe	68.000
Europe	64.030
Asia	37.373



continent	Africa	Americas	Asia	Europe
lifeExp	48.86533	64.658737	60.064903	71.903686

```
pd.pivot_table(df[['continent','lifeExp']], values='lifeExp', columns='continent')
```

Pandas pivot_table on a data frame with three columns

- Pandas pivot_table gets more useful when we try to summarize and convert a tall data frame with more than two variables into a wide data frame. Use three columns; continent, year, and lifeExp

```
pd.pivot_table(df[['continent', 'year','lifeExp']], values='lifeExp', index=['year'], columns='continent')
```

	continent	year	lifeExp		continent	Africa	Americas	Asia	Europe	Oceania
0	Asia	1952	28.801		year					
1	Asia	1957	30.332		1952	39.135500	53.27984	46.314394	64.408500	69.2550
2	Asia	1962	31.997		1957	41.266346	55.96028	49.318544	66.703067	70.2950
3	Asia	1967	34.020		1962	43.319442	58.39876	51.563223	68.539233	71.0850
4	Asia	1972	36.088		1967	45.334538	60.41092	54.663640	69.737600	71.3100
...		1972	47.450942	62.39492	57.319269	70.775033	71.9100
1699	Africa	1987	62.351		1977	49.580423	64.39156	59.610556	71.937767	72.8550
1700	Africa	1992	60.377		1982	51.592865	66.22884	62.617939	72.806400	74.2900
1701	Africa	1997	46.809		1987	53.344788	68.09072	64.851182	73.642167	75.3200
1702	Africa	2002	39.989		1992	53.629577	69.56836	66.537212	74.440100	76.9450
1703	Africa	2007	43.487		1997	53.598269	71.15048	68.020515	75.505167	78.1900
					2002	53.325231	72.42204	69.233879	76.700600	79.7400
					2007	54.806038	73.60812	70.728485	77.648600	80.7195

Pandas pivot_table with Different Aggregating Function

- Pivot_table uses mean function for aggregating or summarizing data by default. We can change the aggregating function, if needed.
- For example, we can use aggfunc='max' to compute “maximum” lifeExp instead of “mean” lifeExp for each year and continent values.

`pd.pivot_table(df[['continent', 'year','lifeExp']], values='lifeExp', index=['year'], columns='continent',aggfunc='max')`

	continent	year	lifeExp
0	Asia	1952	28.801
1	Asia	1957	30.332
2	Asia	1962	31.997
3	Asia	1967	34.020
4	Asia	1972	36.088
...
1699	Africa	1987	62.351
1700	Africa	1992	60.377
1701	Africa	1997	46.809
1702	Africa	2002	39.989
1703	Africa	2007	43.487




	continent	Africa	Americas	Asia	Europe	Oceania
year						
1952		52.724	68.750	65.390	72.670	69.390
1957		58.089	69.960	67.840	73.470	70.330
1962		60.246	71.300	69.390	73.680	71.240
1967		61.557	72.130	71.430	74.160	71.520
1972		64.274	72.880	73.420	74.720	71.930
1977		67.064	74.210	75.380	76.110	73.490
1982		69.885	75.760	77.110	76.990	74.740
1987		71.913	76.860	78.670	77.410	76.320
1992		73.615	77.950	79.360	78.770	77.560
1997		74.772	78.610	80.690	79.390	78.830
2002		75.744	79.770	82.000	80.620	80.370
2007		76.442	80.653	82.603	81.757	81.235

Pandas pivot_table with Different Aggregating Function

- `pd.pivot_table(df[['continent', 'year', 'lifeExp']], values='lifeExp', index=['year'], columns='continent', aggfunc=[min,max])`

	continent	year	lifeExp										
0	Asia	1952	28.801										
1	Asia	1957	30.332										
2	Asia	1962	31.997										
3	Asia	1967	34.020										
4	Asia	1972	36.088										
...										
1699	Africa	1987	62.351										
1700	Africa	1992	60.377										
1701	Africa	1997	46.809										
1702	Africa	2002	39.989										
1703	Africa	2007	43.487										



		min				max							
	continent	Africa	Americas	Asia	Europe	...	Americas	Asia	Europe	Oceania			
	year					...							
	1952	30.000	37.579	28.801	43.585	...	68.750	65.390	72.670	69.390			
	1957	31.570	40.696	30.332	48.079	...	69.960	67.840	73.470	70.330			
	1962	32.767	43.428	31.997	52.098	...	71.300	69.390	73.680	71.240			
	1967	34.113	45.032	34.020	54.336	...	72.130	71.430	74.160	71.520			
	1972	35.400	46.714	36.088	57.005	...	72.880	73.420	74.720	71.930			
	1977	36.788	49.923	31.220	59.507	...	74.210	75.380	76.110	73.490			
	1982	38.445	51.461	39.854	61.036	...	75.760	77.110	76.990	74.740			
	1987	39.906	53.636	40.822	63.108	...	76.860	78.670	77.410	76.320			
	1992	23.599	55.089	41.674	66.146	...	77.950	79.360	78.770	77.560			
	1997	36.087	56.671	41.763	68.835	...	78.610	80.690	79.390	78.830			
	2002	39.193	58.137	42.129	70.845	...	79.770	82.000	80.620	80.370			
	2007	39.613	60.916	43.828	71.777	...	80.653	82.603	81.757	81.235			

Melt

- Pandas melt() function is used to change the DataFrame format from wide to long. It's used to create a specific format of the DataFrame object where one or more columns work as identifiers. All the remaining columns are treated as values and unpivoted to the row axis and only two columns – variable and value.

```
import pandas as pd

d1 = {"Name": ["Pankaj", "Lisa", "David"], "ID": [1, 2, 3], "Role": ["CEO", "Editor", "Author"]}

df = pd.DataFrame(d1)

print(df)

df_melted = pd.melt(df, id_vars=["ID"], value_vars=["Name", "Role"])

print(df_melted)
```

	Name	ID	Role
0	Pankaj	1	CEO
1	Lisa	2	Editor
2	David	3	Author



	ID	variable	value
0	1	Name	Pankaj
1	2	Name	Lisa
2	3	Name	David
3	1	Role	CEO
4	2	Role	Editor
5	3	Role	Author

Concatenation

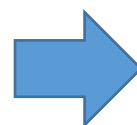
```
df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                    'B': ['B0', 'B1', 'B2', 'B3'],
                    'C': ['C0', 'C1', 'C2', 'C3'],
                    'D': ['D0', 'D1', 'D2', 'D3']},
                    index=[0, 1, 2, 3])

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)
```



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

df2				
	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

df3				
	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Result				
	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

Advanced Concatenation

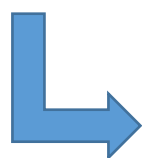
```
df4 = pd.DataFrame({'B': ['B2', 'B3', 'B6', 'B7'],  
                    'D': ['D2', 'D3', 'D6', 'D7'],  
                    'F': ['F2', 'F3', 'F6', 'F7']},  
                    index=[2, 3, 6, 7])
```

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



df1					df4				Result							

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



df1					df4				Result							
	A	B	C	D		B	D	F		A	B	C	D	B	D	F
0	A0	B0	C0	D0	2	B2	D2	F2	2	A2	B2	C2	D2	B2	D2	F2
1	A1	B1	C1	D1	3	B3	D3	F3	3	A3	B3	C3	D3	B3	D3	F3
2	A2	B2	C2	D2	6	B6	D6	F6								
3	A3	B3	C3	D3	7	B7	D7	F7								

Advanced Concatenation

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

df1					df4				Result							
	A	B	C	D		B	D	F		A	B	C	D	B	D	F
0	A0	B0	C0	D0	2	B2	D2	F2	0	A0	B0	C0	D0	NaN	NaN	NaN
1	A1	B1	C1	D1	3	B3	D3	F3	1	A1	B1	C1	D1	NaN	NaN	NaN
2	A2	B2	C2	D2	6	B6	D6	F6	2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	7	B7	D7	F7	3	A3	B3	C3	D3	B3	D3	F3

https://pandas.pydata.org/pandas-docs/stable/user_guide/merging.html

Merging

```
left = pd.DataFrame({'key1': ['K0', 'K0', 'K1', 'K2'],  
                    'key2': ['K0', 'K1', 'K0', 'K1'],  
                    'A': ['A0', 'A1', 'A2', 'A3'],  
                    'B': ['B0', 'B1', 'B2', 'B3']})  
  
right = pd.DataFrame({'key1': ['K0', 'K1', 'K1', 'K2'],  
                    'key2': ['K0', 'K0', 'K0', 'K0'],  
                    'C': ['C0', 'C1', 'C2', 'C3'],  
                    'D': ['D0', 'D1', 'D2', 'D3']})  
  
result = pd.merge(left, right, on=['key1', 'key2'])
```

left				
	key1	key2	A	B
0	K0	K0	A0	B0
1	K0	K1	A1	B1
2	K1	K0	A2	B2
3	K2	K1	A3	B3

right				
	key1	key2	C	D
0	K0	K0	C0	D0
1	K1	K0	C1	D1
2	K1	K0	C2	D2
3	K2	K0	C3	D3

Result						
	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K1	K0	A2	B2	C1	D1
2	K1	K0	A2	B2	C2	D2

Joining

```
left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],  
                    'B': ['B0', 'B1', 'B2']},  
                    index=['K0', 'K1', 'K2'])  
  
right = pd.DataFrame({'C': ['C0', 'C2', 'C3'],  
                     'D': ['D0', 'D2', 'D3']},  
                     index=['K0', 'K2', 'K3'])  
  
result = left.join(right)
```

left			right			Result				
	A	B		C	D		A	B	C	D
K0	A0	B0	K0	C0	D0	K0	A0	B0	C0	D0
K1	A1	B1	K2	C2	D2	K1	A1	B1	NaN	NaN
K2	A2	B2	K3	C3	D3	K2	A2	B2	C2	D2

Data Manipulation in Pandas

Regex

```
import re

pattern = '^a...s$'
test_string = 'abyss'
result = re.match(pattern, test_string)

if result:
    print("Search successful.")
else:
    print("Search unsuccessful.")
```

A Regular Expression (Regex) is a sequence of characters that defines a search pattern.

```
import re

string = '39801 356, 2102 1111'

# Three digit number followed by space followed by two digit number
pattern = '(\d{3}) (\d{2})'

# match variable contains a Match object.
match = re.search(pattern, string)

if match:
    print(match.group())
else:
    print("pattern not found")
```


Date Functionality

```
import pandas as pd
```

```
df = pd.date_range('1/1/2020', periods=25)
```

```
print(df)
```

```
DatetimeIndex(['2020-01-01', '2020-01-02', '2020-01-03', '2020-01-04',  
              '2020-01-05', '2020-01-06', '2020-01-07', '2020-01-08',  
              '2020-01-09', '2020-01-10', '2020-01-11', '2020-01-12',  
              '2020-01-13', '2020-01-14', '2020-01-15', '2020-01-16',  
              '2020-01-17', '2020-01-18', '2020-01-19', '2020-01-20',  
              '2020-01-21', '2020-01-22', '2020-01-23', '2020-01-24',  
              '2020-01-25'],  
              dtype='datetime64[ns]', freq='D')
```

Date functionality

```
import pandas as pd

df = pd.date_range('01/20/2020', periods=5, freq = 'M')

print(df)
```



```
DatetimeIndex(['2020-01-31', '2020-02-29', '2020-03-31', '2020-04-30',
               '2020-05-31'],
              dtype='datetime64[ns]', freq='M')
```

Time Delta

- Time deltas are differences in times, expressed in difference units, for example, days, hours, minutes, seconds.
- They can be both positive and negative.

Example

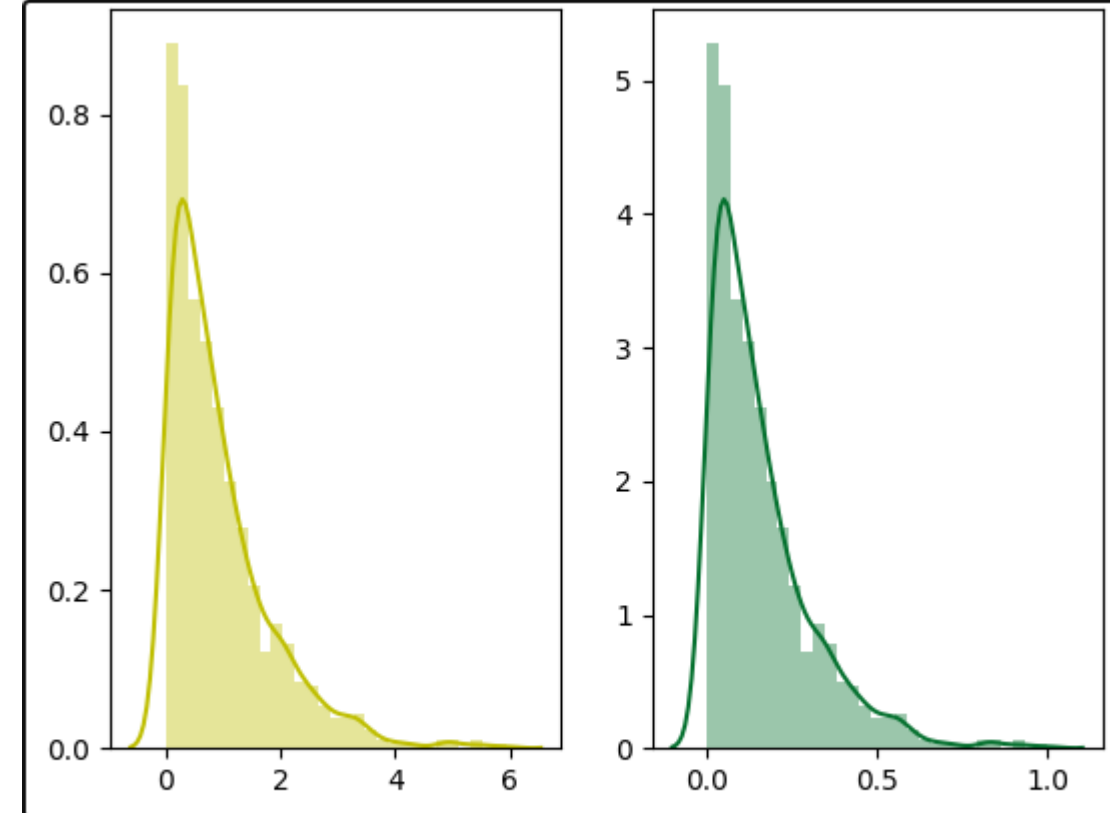
Name	Code	Output
By passing a string literal, we can create a timedelta object.	<code>pd.Timedelta('2 days 2 hours 15 minutes 30 seconds')</code>	2 days 02:15:30
By passing an integer value with the unit, an argument creates a Timedelta object.	<code>pd.Timedelta(6,unit='h')</code>	0 days 06:00:00
Data offsets such as - weeks, days, hours, minutes, seconds, milliseconds, microseconds, nanoseconds	<code>pd.Timedelta(days=2)</code>	2 days 00:00:00
Convert a scalar, array, list, or series from a recognized timedelta format/ value into a Timedelta type. It will construct Series if the input is a Series, a scalar if the input is scalar-like, otherwise will output a TimedeltaIndex.	<code>pd.Timedelta(days=2)</code>	2 days 00:00:00

Example

Name	Code	Output																			
Operate on Series/ Data Frames and construct timedelta64[ns] Series through subtraction operations on datetime64[ns] Series, or Timestamps.	s = pd.Series(pd.date_range('2012-1-1', periods=3, freq='D')) td = pd.Series([pd.Timedelta(days=i) for i in range(3)]) df = pd.DataFrame(dict(A = s, B = td))	<table><tr><th></th><th>A</th><th>B</th></tr><tr><td>0</td><td>2012-01-01</td><td>0 days</td></tr><tr><td>1</td><td>2012-01-02</td><td>1 days</td></tr><tr><td>2</td><td>2012-01-03</td><td>2 days</td></tr></table>		A	B	0	2012-01-01	0 days	1	2012-01-02	1 days	2	2012-01-03	2 days							
	A	B																			
0	2012-01-01	0 days																			
1	2012-01-02	1 days																			
2	2012-01-03	2 days																			
Addition Operations	s = pd.Series(pd.date_range('2012-1-1', periods=3, freq='D')) td = pd.Series([pd.Timedelta(days=i) for i in range(3)]) df = pd.DataFrame(dict(A = s, B = td)) df['C']=df['A']+df['B']	<table><tr><th></th><th>A</th><th>B</th><th>C</th></tr><tr><td>0</td><td>2012-01-01</td><td>0 days</td><td>2012-01-01</td></tr><tr><td>1</td><td>2012-01-02</td><td>1 days</td><td>2012-01-03</td></tr><tr><td>2</td><td>2012-01-03</td><td>2 days</td><td>2012-01-05</td></tr></table>		A	B	C	0	2012-01-01	0 days	2012-01-01	1	2012-01-02	1 days	2012-01-03	2	2012-01-03	2 days	2012-01-05			
	A	B	C																		
0	2012-01-01	0 days	2012-01-01																		
1	2012-01-02	1 days	2012-01-03																		
2	2012-01-03	2 days	2012-01-05																		
Subtraction Operation	s = pd.Series(pd.date_range('2012-1-1', periods=3, freq='D')) td = pd.Series([pd.Timedelta(days=i) for i in range(3)]) df = pd.DataFrame(dict(A = s, B = td)) df['C']=df['A']+df['B'] df['D']=df['C']+df['B']	<table><tr><th></th><th>A</th><th>B</th><th>C</th></tr><tr><td>0</td><td>2012-01-01</td><td>0 days</td><td>2012-01-01</td><td>2012-01-01</td></tr><tr><td>1</td><td>2012-01-02</td><td>1 days</td><td>2012-01-03</td><td>2012-01-04</td></tr><tr><td>2</td><td>2012-01-03</td><td>2 days</td><td>2012-01-05</td><td>2012-01-07</td></tr></table>		A	B	C	0	2012-01-01	0 days	2012-01-01	2012-01-01	1	2012-01-02	1 days	2012-01-03	2012-01-04	2	2012-01-03	2 days	2012-01-05	2012-01-07
	A	B	C																		
0	2012-01-01	0 days	2012-01-01	2012-01-01																	
1	2012-01-02	1 days	2012-01-03	2012-01-04																	
2	2012-01-03	2 days	2012-01-05	2012-01-07																	

Normalization

- Normalization refers to rescaling real-valued numeric attributes into a **0 to 1** range.
- Data normalization is used in machine learning **to make model training less sensitive to the scale of features**. This allows our model to converge to better weights and, in turn, leads to a more accurate model.



Left: Original Data, Right: Normalized Data

```
from sklearn import preprocessing
import numpy as np
```

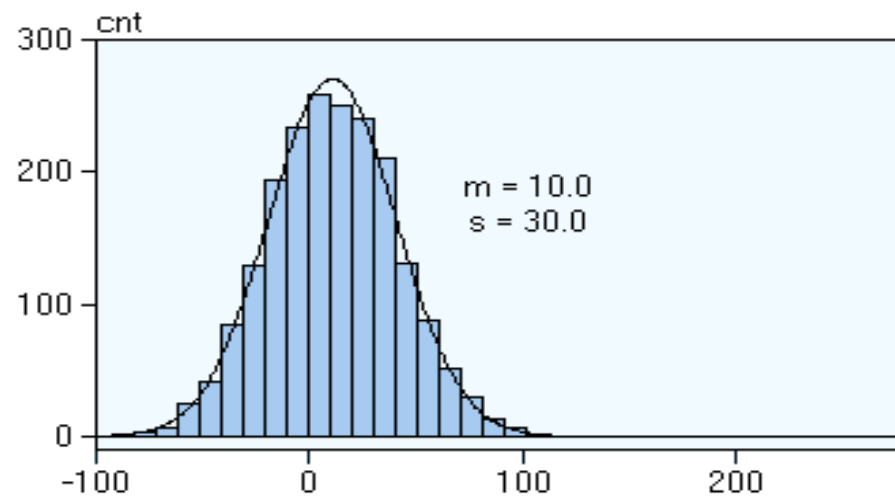
```
a = np.random.random((1, 4))
a = a*20
print("Data = ", a)
```

```
# normalize the data attributes
normalized = preprocessing.normalize(a)
print("Normalized Data = ", normalized)
```

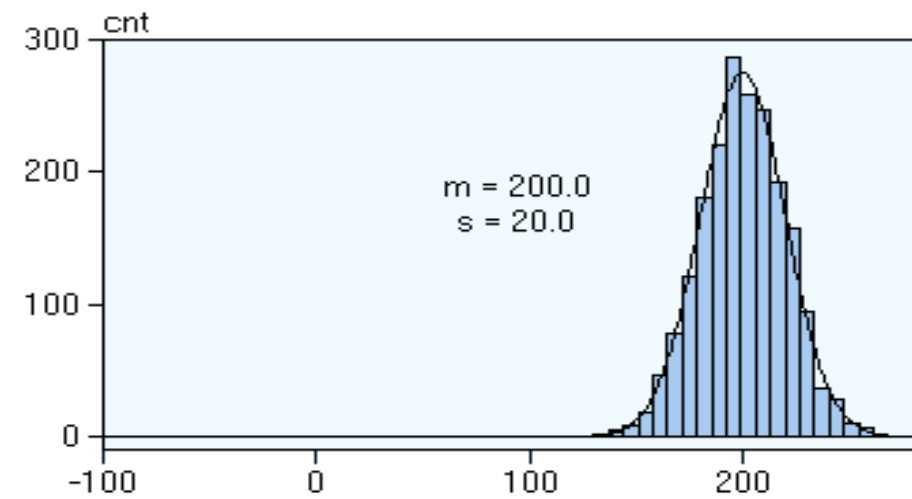
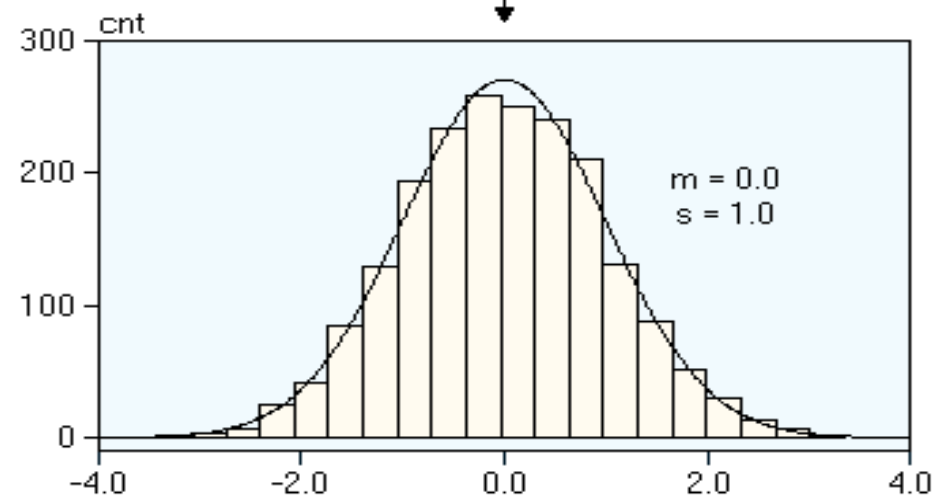
$$z = \frac{x - \min(x)}{\max(x) - \min(x)}$$

```
Data = [[10.52384526 14.14223072  9.48558746  2.08542834]]
Normalized Data = [[0.52288283 0.70266423 0.47129644 0.10361561]]
```

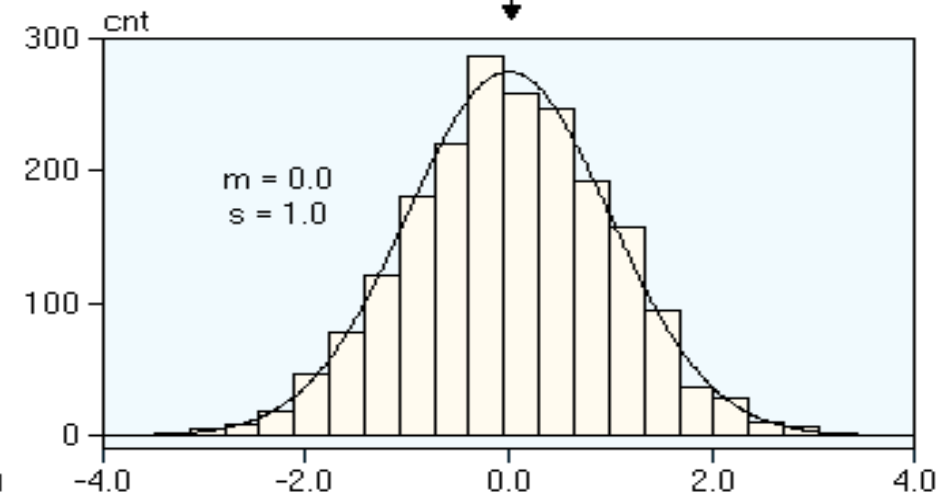
Standardization



Standardisation



Standardisation



comparable distributions
($m = 0.0$, $s = 1.0$)

Missing Data Handle

- Missing Data can occur when no information is provided for one or more items or for a whole unit. Missing Data is a very big problem in real life scenario. Missing Data can also refer to as **NA(Not Available)** values in pandas. In DataFrame sometimes many datasets simply arrive with missing data, either because it exists and was not collected or it never existed.
- In Pandas missing data is represented by two value:
 - **None**: None is a Python singleton object that is often used for missing data in Python code.
 - **NaN** : NaN (an acronym for Not a Number), is a special floating-point value recognized by all systems that use the standard IEEE floating-point representation
- Pandas treat None and NaN as essentially interchangeable for indicating missing or null values. To facilitate this convention, there are several useful functions for detecting, removing, and replacing null values in Pandas DataFrame :
 - **isnull()**
 - **notnull()**
 - **dropna()**
 - **fillna()**
 - **replace()**
 - **interpolate()**

isnull()

	First Score	Second Score	Third Score
0	100.0	30.0	NaN
1	90.0	45.0	40.0
2	NaN	56.0	80.0
3	95.0	NaN	98.0

	First Score	Second Score	Third Score
0	False	False	True
1	False	False	False
2	True	False	False
3	False	True	False

notnull()

	First Score	Second Score	Third Score
0	100.0	30.0	NaN
1	90.0	45.0	40.0
2	NaN	56.0	80.0
3	95.0	NaN	98.0

	First Score	Second Score	Third Score
0	True	True	False
1	True	True	True
2	False	True	True
3	True	False	True

Filling Missing Data

	First Score	Second Score	Third Score
0	100.0	30.0	NaN
1	90.0	45.0	40.0
2	NaN	56.0	80.0
3	95.0	NaN	98.0

#1

```
# filling missing value using fillna()  
df.fillna(0)
```

	First Score	Second Score	Third Score
0	100.0	30.0	0.0
1	90.0	45.0	40.0
2	0.0	56.0	80.0
3	95.0	0.0	98.0

#2

```
# filling a missing value with  
# previous ones  
df.fillna(method = 'pad')
```

	First Score	Second Score	Third Score
0	100.0	30.0	NaN
1	90.0	45.0	40.0
2	90.0	56.0	80.0
3	95.0	56.0	98.0

#3

```
# filling null value using fillna() function  
df.fillna(method = 'bfill')
```

	First Score	Second Score	Third Score
0	100.0	30.0	40.0
1	90.0	45.0	40.0
2	95.0	56.0	80.0
3	95.0	NaN	98.0

Interpolate

	A	B	C	D
0	12.0	NaN	20.0	14.0
1	4.0	2.0	16.0	3.0
2	5.0	54.0	NaN	NaN
3	NaN	3.0	3.0	NaN
4	1.0	NaN	8.0	6.0



	A	B	C	D
0	12.0	NaN	20.0	14.0
1	4.0	2.0	16.0	3.0
2	5.0	54.0	9.5	4.0
3	3.0	3.0	3.0	5.0
4	1.0	3.0	8.0	6.0

dropna()

	First Score	Second Score	Third Score	Fourth Score
0	100.0	30.0	52	NaN
1	90.0	NaN	40	NaN
2	NaN	45.0	80	NaN
3	95.0	56.0	98	65.0



	First Score	Second Score	Third Score	Fourth Score
3	95.0	56.0	98	65.0

dropna()

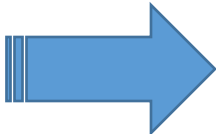
	First Score	Second Score	Third Score	Fourth Score
0	100.0	30.0	52.0	NaN
1	NaN	NaN	NaN	NaN
2	NaN	45.0	80.0	NaN
3	95.0	56.0	98.0	65.0



	First Score	Second Score	Third Score	Fourth Score
0	100.0	30.0	52.0	NaN
2	NaN	45.0	80.0	NaN
3	95.0	56.0	98.0	65.0

dropna()

	First Score	Second Score	Third Score	Fourth Score
0	100.0	30.0	52.0	60
1	NaN	NaN	NaN	67
2	NaN	45.0	80.0	68
3	95.0	56.0	98.0	65



	Fourth Score
0	60
1	67
2	68
3	65

Window Functions

- .rolling() Function
- .expanding() Function
- .ewm() Function

.rolling() Function

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(10, 4),
index = pd.date_range('1/1/2000', periods=10),
columns = ['A', 'B', 'C', 'D'])
```

```
print(df)
```

```
print(df.rolling(window=3).mean())
```

	A	B	C	D
2000-01-01	-0.692862	-0.563469	-1.172630	0.062225
2000-01-02	1.013375	1.948667	-0.311838	-0.550377
2000-01-03	0.907432	-2.203215	0.163166	0.099759
2000-01-04	1.416964	0.526772	1.269941	-0.074776
2000-01-05	-1.562144	-0.002639	1.008214	-0.898144
2000-01-06	-0.137106	-0.228348	-1.523341	-0.231084
2000-01-07	-1.023343	1.231910	0.214703	-1.000276
2000-01-08	-0.647357	-1.959665	-0.002584	0.857234
2000-01-09	-0.970856	0.460758	-1.452498	0.642333
2000-01-10	-0.708996	-1.539174	-0.899433	-0.339364

	A	B	C	D
2000-01-01	NaN	NaN	NaN	NaN
2000-01-02	NaN	NaN	NaN	NaN
2000-01-03	0.409315	-0.272673	-0.440434	-0.129464
2000-01-04	1.112590	0.090741	0.373756	-0.175131
2000-01-05	0.254084	-0.559694	0.813774	-0.291054
2000-01-06	-0.094095	0.098595	0.251605	-0.401335
2000-01-07	-0.907531	0.333641	-0.100141	-0.709835
2000-01-08	-0.602602	-0.318701	-0.437074	-0.124709
2000-01-09	-0.880519	-0.088999	-0.413460	0.166431
2000-01-10	-0.775736	-1.012694	-0.784838	0.386735

.expanding() Function

```
import pandas as pd
import numpy as np
```

```
df = pd.DataFrame([
    ['a', 1],
    ['a', 2],
    ['a', 3],
    ['b', 5],
    ['b', 6],
    ['b', 7],
    ['b', 8],
    ['c', 10],
    ['c', 11],
    ['c', 12],
    ['c', 13]
], columns = ['category', 'value'])
```

```
print(df)
```

```
print(df.value.expanding(min_periods=3).sum())
```

```
print(df.value.expanding(min_periods=3).mean())
```

9/10/2021

	category	value
0	a	1
1	a	2
2	a	3
3	b	5
4	b	6
5	b	7
6	b	8
7	c	10
8	c	11
9	c	12
10	c	13

0	NaN
1	NaN
2	6.0
3	11.0
4	17.0
5	24.0
6	32.0
7	42.0
8	53.0
9	65.0
10	78.0

Name: value, dtype: float64


0	NaN
1	NaN
2	2.000000
3	2.750000
4	3.400000
5	4.000000
6	4.571429
7	5.250000
8	5.888889
9	6.500000
10	7.090909

Name: value, dtype: float64

EWM

- Ewm is applied on a series of data. Specify any of the com, span, halflife argument and apply the appropriate statistical function on top of it. It assigns the weights exponentially.
- Using to make data smooth to handle noise data

	Stock_ABC_Corp
2020-01-01	12.5
2020-01-02	15.0
2020-01-03	17.0
2020-01-04	10.2
2020-01-05	20.5
2020-01-06	16.1
2020-01-07	14.2
2020-01-08	19.7
2020-01-09	20.0
2020-01-10	2.8


`df.ewm(com=0.5).mean()`

	Stock_ABC_Corp
2020-01-01	12.500000
2020-01-02	14.375000
2020-01-03	16.192308
2020-01-04	12.147500
2020-01-05	17.738843
2020-01-06	16.644780
2020-01-07	15.014181
2020-01-08	18.138537
2020-01-09	19.379575
2020-01-10	8.326338

Data Analysis in Pandas

Descriptive Statistics

- Most of these are aggregations like `sum()`, `mean()`, but some of them, like `sumsum()`, produce an object of the same size.
- These methods take an axis argument, just like `ndarray.{sum, std, ...}`, but the axis can be specified by name or integer.
 - DataFrame – “index” (axis=0, default), “columns” (axis=1)

Sr.No.	Function	Description
1	<code>count()</code>	Number of non-null observations
2	<code>sum()</code>	Sum of values
3	<code>mean()</code>	Mean of Values
4	<code>median()</code>	Median of Values
5	<code>mode()</code>	Mode of values
6	<code>std()</code>	Standard Deviation of the Values
7	<code>min()</code>	Minimum Value
8	<code>max()</code>	Maximum Value
9	<code>abs()</code>	Absolute Value
10	<code>prod()</code>	Product of Values
11	<code>cumsum()</code>	Cumulative Sum
12	<code>cumprod()</code>	Cumulative Product

Example

#Create a Dictionary of series

```
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack',  
    'Lee','David','Gasper','Betina','Andres']),  
    'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46]),  
    'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8,3.78,2.98,4.80,4.10,3.65])  
}
```

#Create a DataFrame

```
df = pd.DataFrame(d)  
print df.sum()
```



```
Age                                     382  
Name      TomJamesRickyVinSteveSmithJackLeeDavidGasperBe...  
Rating                                     44.92  
dtype: object
```

#Create a DataFrame

```
df = pd.DataFrame(d)  
print df.sum(1)
```



```
0      29.23  
1      29.24  
2      28.98  
3      25.56  
4      33.20  
5      33.60  
6      26.80  
7      37.78  
8      42.98  
9      34.80  
10     55.10  
11     49.65  
dtype: float64
```

Summarizing Data

- The describe() function computes a summary of statistics pertaining to the Data Frame columns.

```
import pandas as pd
import numpy as np

#Create a Dictionary of series
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack',
    'Lee','David','Gasper','Betina','Andres']),
    'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46]),
    'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8,3.78,2.98,4.80,4.10,3.65])
}

#Create a DataFrame
df = pd.DataFrame(d)
print df.describe()
```

	Age	Rating
count	12.000000	12.000000
mean	31.833333	3.743333
std	9.232682	0.661628
min	23.000000	2.560000
25%	25.000000	3.230000
50%	29.500000	3.790000
75%	35.500000	4.132500
max	51.000000	4.800000

Summarizing Data with include

- This function gives the mean, std and IQR values. And, function excludes the character columns and given summary about numeric columns. 'include' is the argument which is used to pass necessary information regarding what columns need to be considered for summarizing. Takes the list of values; by default, 'number'.
 - object – Summarizes String columns
 - number – Summarizes Numeric columns
 - all – Summarizes all columns together (Should not pass it as a list value)

describe(include=['object'])

- #Create a DataFrame
- `df = pd.DataFrame(d)`
- `print df.describe(include=['object'])`

```
              Name  
count          12  
unique          12  
top           Ricky  
freq           1
```

describe(include='all')

```
#Create a DataFrame
df = pd.DataFrame(d)
print df. describe(include='all')
```

	Age	Name	Rating
count	12.000000	12	12.000000
unique	NaN	12	NaN
top	NaN	Ricky	NaN
freq	NaN	1	NaN
mean	31.833333	NaN	3.743333
std	9.232682	NaN	0.661628
min	23.000000	NaN	2.560000
25%	25.000000	NaN	3.230000
50%	29.500000	NaN	3.790000
75%	35.500000	NaN	4.132500
max	51.000000	NaN	4.800000

Statistical Functions

- Statistical methods help in the understanding and analyzing the behavior of data.
- Some useful functions:
 - Percent change
 - Covariance
 - Correlation
 - Data Ranking

Percent_change

- Series, DataFrames and Panel, all have the function **pct_change()**.
- This function compares every element with its prior element and computes the change percentage.
- Formulas: $value_n = (x_n - x_{n-1}) : (x_{n-1})$

```
s = pd.Series([1,2,3,4,5,4])  
  
print(s.pct_change())
```

	0	1
0	NaN	
1	1.000000	
2	0.500000	
3	0.333333	
4	0.250000	
5	-0.200000	

dtype: float64

```
df = pd.DataFrame(np.random.randn(5, 2))  
  
print(df)
```

	0	1
0	NaN	NaN
1	-2.308832	21.970813
2	-1.434196	0.135932
3	-4.361687	-0.638241
4	-1.006457	-0.148755

```
print(df.pct_change())
```

Co-variance

- Covariance is applied on **series data**. The Series object has a method **cov** to compute **covariance between series objects**. **NA** will be excluded automatically.
- The covariance formula is similar to the formula for deals with the calculation of data points from the average value in a dataset. For example, the covariance between two random variables X and Y can be calculated using the following formula (for population → left) or (for sample → right):

$$\text{Cov}(X, Y) = \frac{\sum (X_i - \bar{X})(Y_j - \bar{Y})}{n}$$

$$\text{Cov}(X, Y) = \frac{\sum (X_i - \bar{X})(Y_j - \bar{Y})}{n-1}$$

```
import pandas as pd
import numpy as np
```

```
s1 = pd.Series([1,2,3,4,5,4,1,1,1,2])
s2 = pd.Series([2,2,1,2,3,4,5,6,8,0])
```



```
-1.3555555555555556
```

```
print(s1.cov(s2))
```


Correlation Value

- The correlation coefficient is a value that indicates the strength of the relationship. The coefficient can take any values from -1 to 1. The interpretations of the values are:
 - **-1:** Perfect negative correlation. The variables tend to move in opposite directions (i.e., when one variable increases, the other variable decreases).
 - **0:** No correlation. The variables do not have a relationship with each other.
 - **1:** Perfect positive correlation. The variables tend to move in the same direction (i.e., when one variable increases, the other variable also increases).

```
import pandas as pd
import numpy as np

frame = pd.DataFrame(np.random.randn(10, 5),
columns=['a', 'b', 'c', 'd', 'e'])

print(frame)

print(" ")

print(frame['a'].corr(frame['b']))

print(" ")

print(frame.corr())
```

	a	b	c	d	e
0	1.420436	0.016242	0.788225	0.326262	-0.590108
1	-1.334494	1.004633	-0.314193	-0.531058	1.523258
2	-0.650768	-0.015261	-1.661331	-0.637942	-1.003861
3	0.146829	-1.498770	-1.044598	0.535980	0.736555
4	0.673129	1.023355	-0.953526	-1.453017	0.885171
5	-0.027378	-1.197029	0.615263	0.185669	-1.108388
6	0.434728	1.874489	0.760801	-0.811918	1.492091
7	0.510840	-0.373273	1.131043	-2.957110	-0.419172
8	-0.257535	0.957575	0.396334	0.303238	-0.952718
9	1.750641	1.061318	0.726307	2.061660	-1.244804

0.074593980183

	a	b	c	d	e
a	1.000000	0.074594	0.473045	0.284992	-0.306381
b	0.074594	1.000000	0.169951	-0.009250	0.330528
c	0.473045	0.169951	1.000000	0.018195	-0.224682
d	0.284992	-0.009250	0.018195	1.000000	-0.336748
e	-0.306381	0.330528	-0.224682	-0.336748	1.000000

Data Ranking

- Data Ranking produces ranking for each element in the array of elements. In case of ties, assigns the mean rank.

```
import pandas as pd
import numpy as np
s = pd.Series([9,0,2,0,3,5,4], index=list('abcdefg'))
```

```
print(s)
```

```
s['d'] = s['b'] # so there's a tie
```

```
print(s.rank())
```

```
a    7.0
b    1.5
c    3.0
d    1.5
e    4.0
f    6.0
g    5.0
dtype: float64
```

```
a    9
b    0
c    2
d    0
e    3
f    5
g    4
dtype: int64
- -
```

Data Ranking – More Example

```
a      9
b      0
c      2
d      0
e      3
f      5
g      4
dtype: int64
```

```
print(s.rank(method = 'min'))
```

```
print(s.rank(method = 'max'))
```

```
print(s.rank(method = 'first'))
```

```
a      7.0
b      1.0
c      3.0
d      1.0
e      4.0
f      6.0
g      5.0
dtype: float64
a      7.0
b      2.0
c      3.0
d      2.0
e      4.0
f      6.0
g      5.0
dtype: float64
a      7.0
b      1.0
c      3.0
d      2.0
e      4.0
f      6.0
g      5.0
dtype: float64
```

Categorical Data

- Data includes the text columns, which are repetitive. Features like gender, country, and codes are always repetitive. These are the examples for categorical data.
 - Categorical variables can take on only a limited, and usually fixed number of possible values. Besides the fixed length, categorical data might have an order but cannot perform numerical operation. Categorical are a Pandas data type.
- The categorical data type is useful in the following cases –
 - A **string variable** consisting of only a few different values. Converting such a string variable to a categorical variable will save some memory.
 - The **lexical order of a variable** is not the same as the logical order (“one”, “two”, “three”). By converting to a categorical and specifying an order on the categories, sorting and min/max will use the logical order instead of the lexical order.
 - As a signal to other python libraries that this column should be treated as a **categorical variable** (e.g. to use suitable statistical methods or plot types)

Example

```
import pandas as pd
```

```
s = pd.Series(["a","b","c","a"], dtype="category")
```

```
print(s)
```

```
0    a
```

```
1    b
```

```
2    c
```

```
3    a
```

```
dtype: category
```

```
Categories (3, object): [a, b, c]
```

```
cat = cat=pd.Categorical(['a','b','c','a','b','c','d'], ['c', 'b', 'a'])
```

```
print(cat)
```

```
[a, b, c, a, b, c, NaN]
```

```
Categories (3, object): [c, b, a]
```

Comparison of Categorical Data

```
cat = pd.Series([1,2,3]).astype("category", categories=[1,2,3], ordered=True)
cat1 = pd.Series([2,2,2]).astype("category", categories=[1,2,3], ordered=True)

print(cat>cat1)
```

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

Visualization

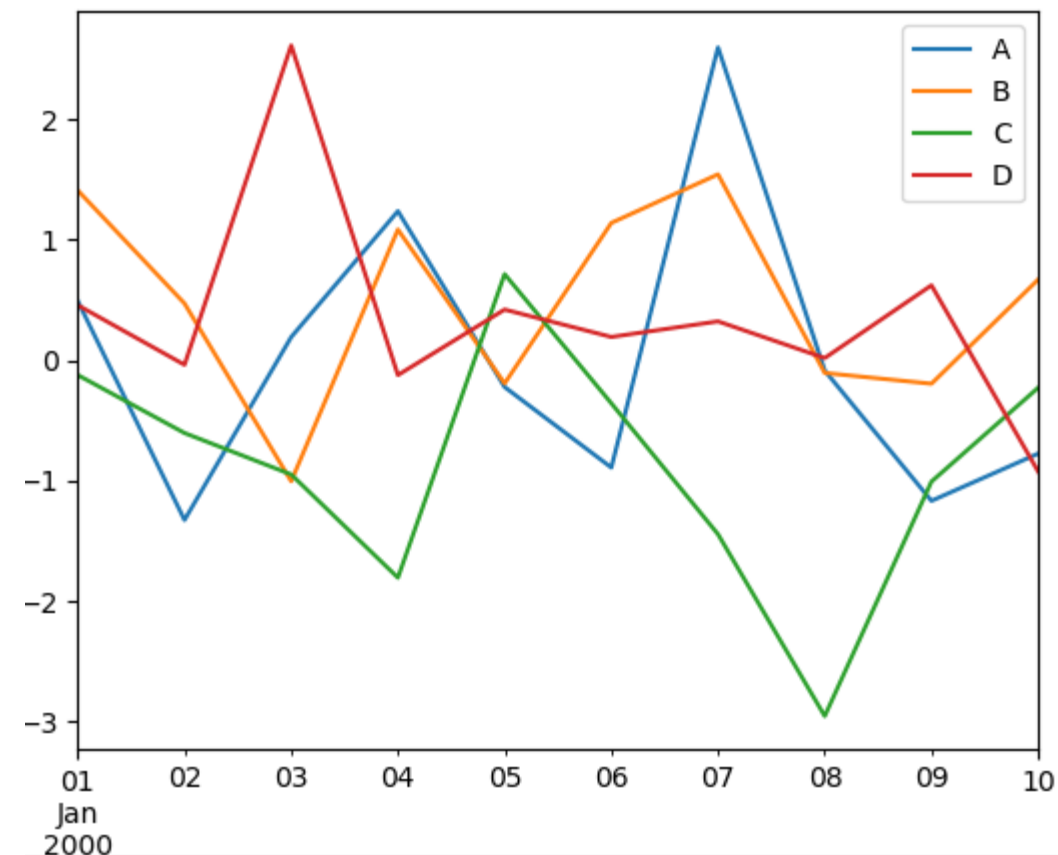
- Plotting methods allow a handful of plot styles other than the default line plot. These methods can be provided as the kind keyword argument to plot(). These include –
 - bar or barh for bar plots
 - hist for histogram
 - box for boxplot
 - 'area' for area plots
 - 'scatter' for scatter plots

Plotting

```
import pandas as pd
import numpy as np

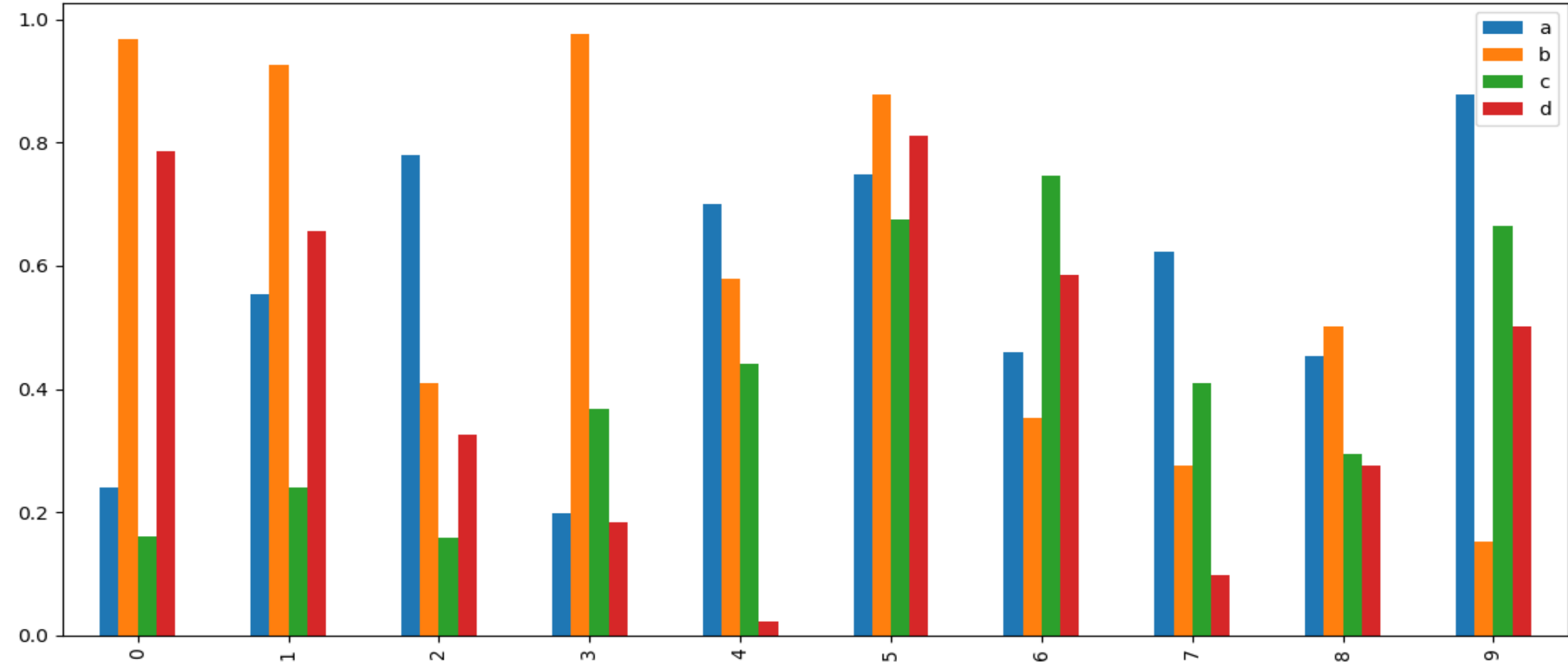
df = pd.DataFrame(np.random.randn(10,4),
                  index=pd.date_range('1/1/2000',periods=10),
                  columns=list('ABCD'))

df.plot()
```



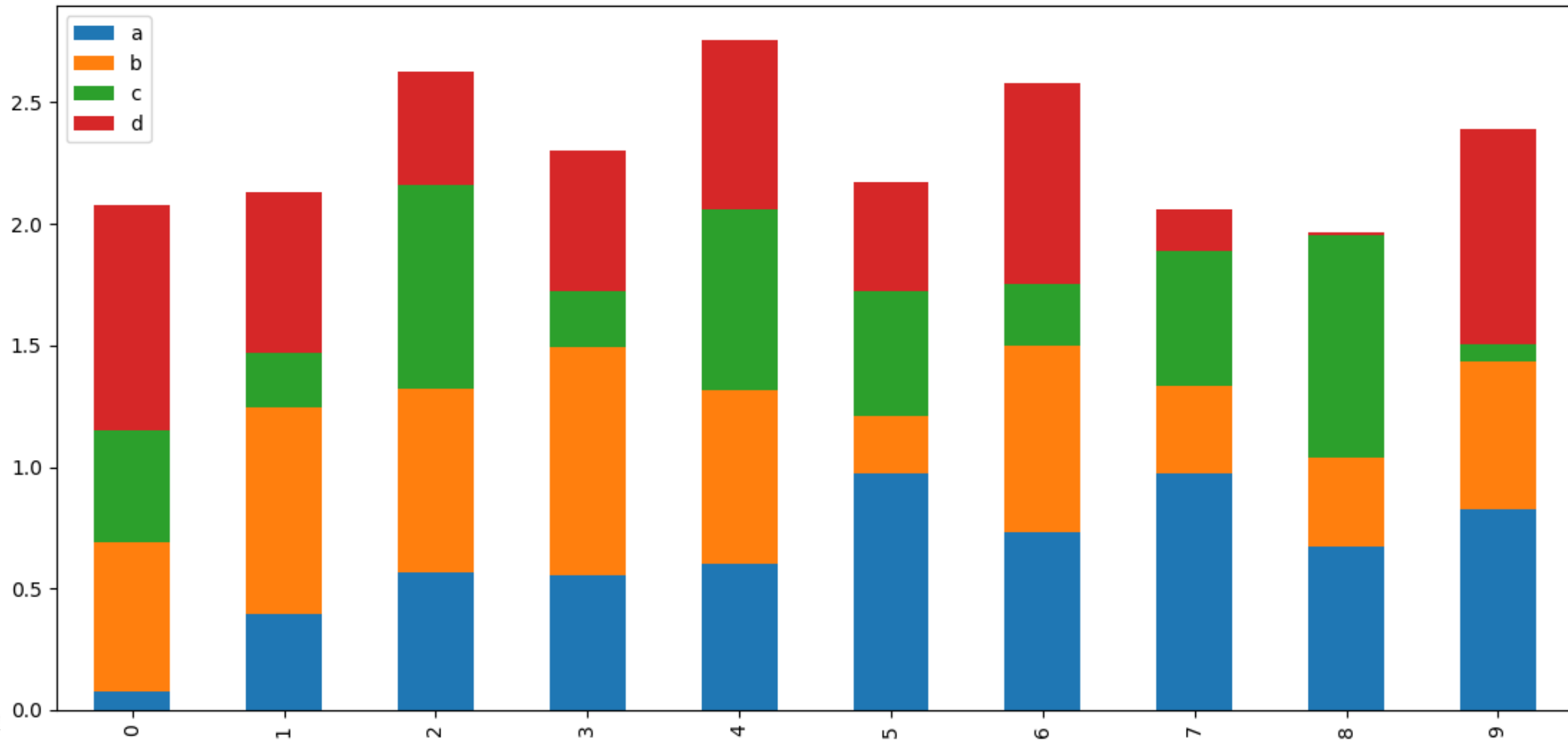
Bar Plotting

```
df = pd.DataFrame(np.random.rand(10,4),columns=['a','b','c','d'])  
df.plot.bar()
```

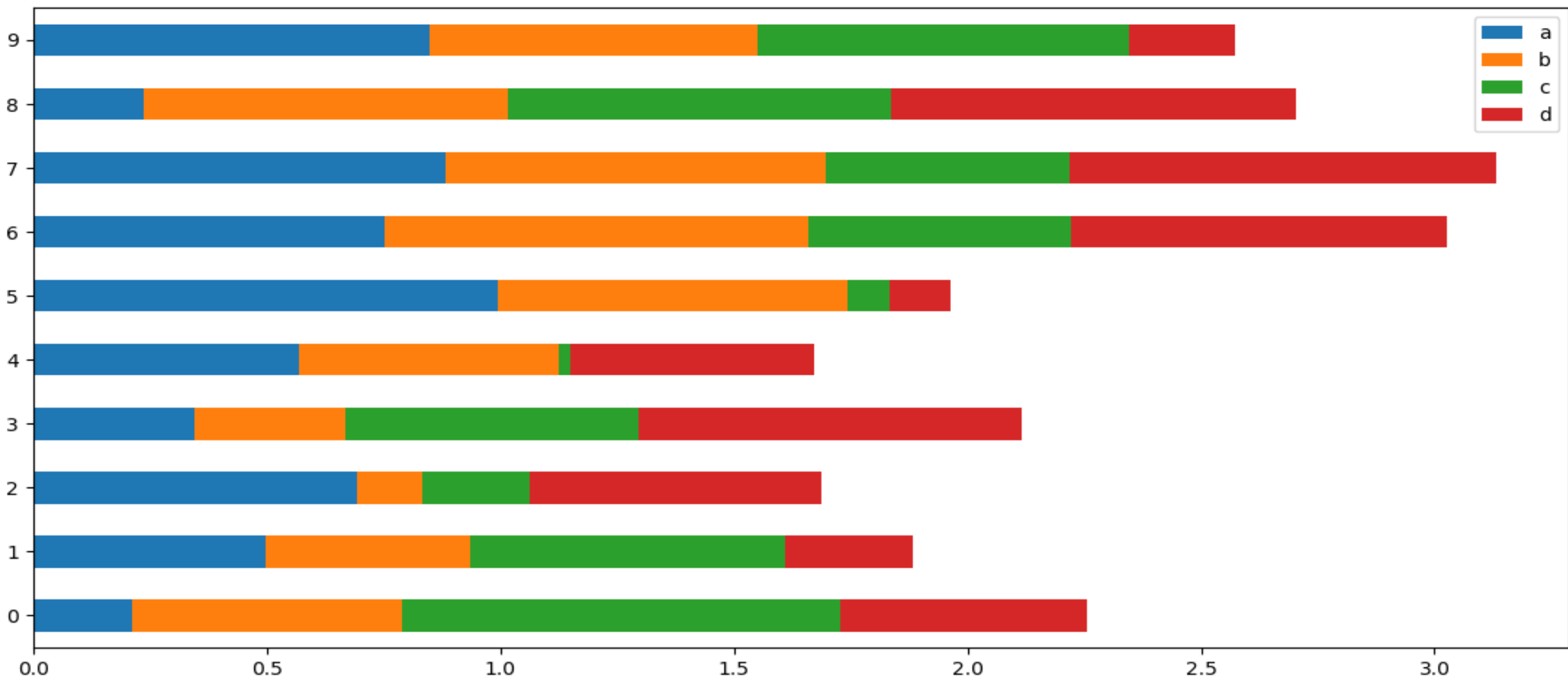


Stacked Bar Plotting

```
df = pd.DataFrame(np.random.rand(10,4),columns=['a','b','c','d'])  
df.plot.bar(stacked=True)
```



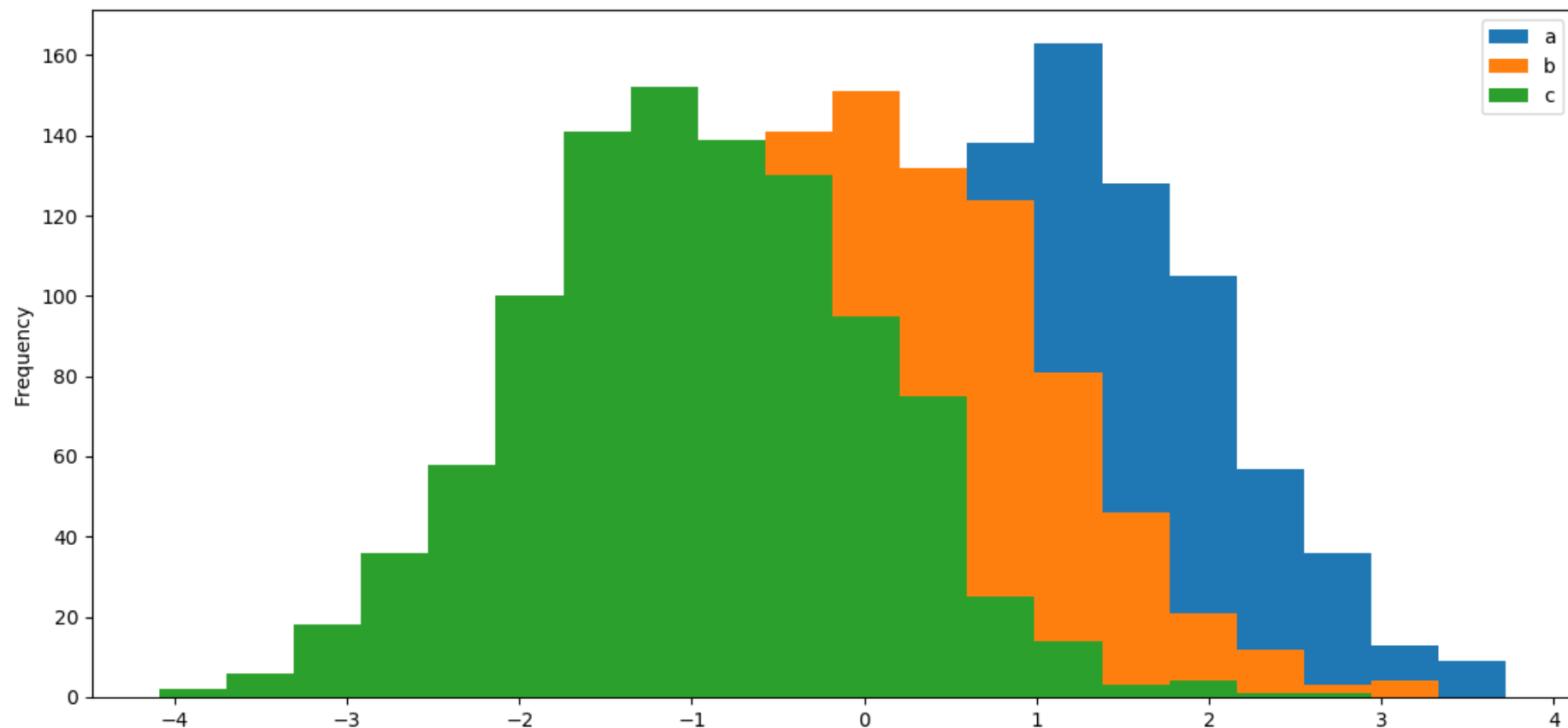
Horizontal Bar Plot `df = pd.DataFrame(np.random.rand(10,4),columns=['a','b','c','d'])`
`df.plot.barh(stacked=True)`



Histogram in same plot

```
df = pd.DataFrame({'a':np.random.randn(1000)+1,'b':np.random.randn(1000),'c':np.random.randn(1000) - 1}, columns=['a', 'b', 'c'])
```

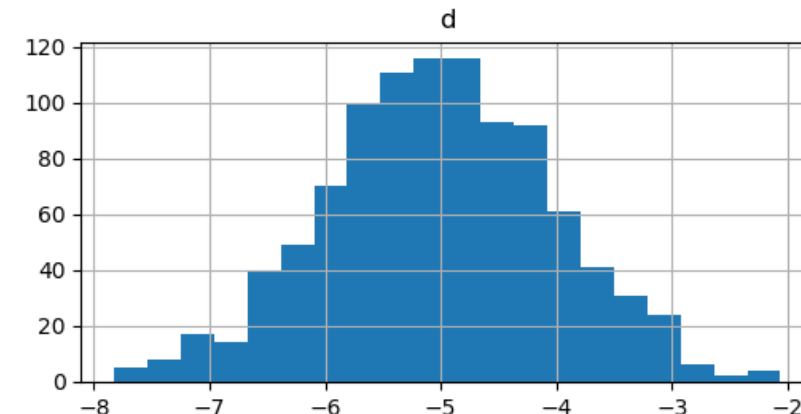
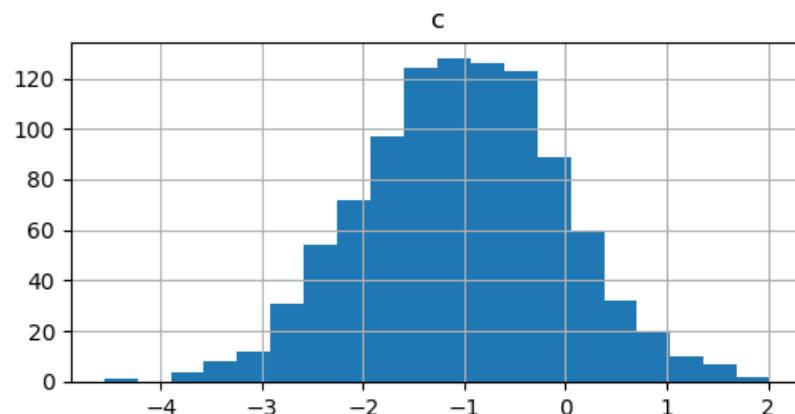
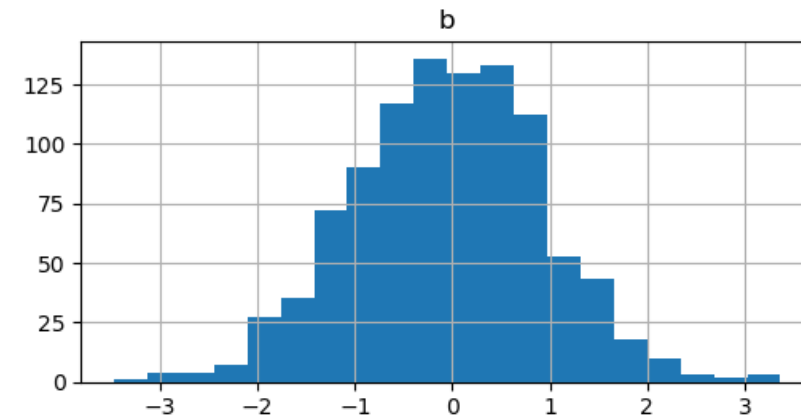
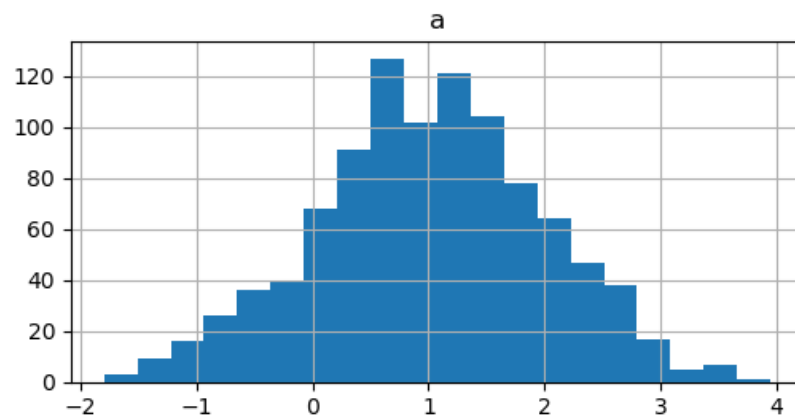
```
df.plot.hist(bins=20)
```



Plot different histograms for each column

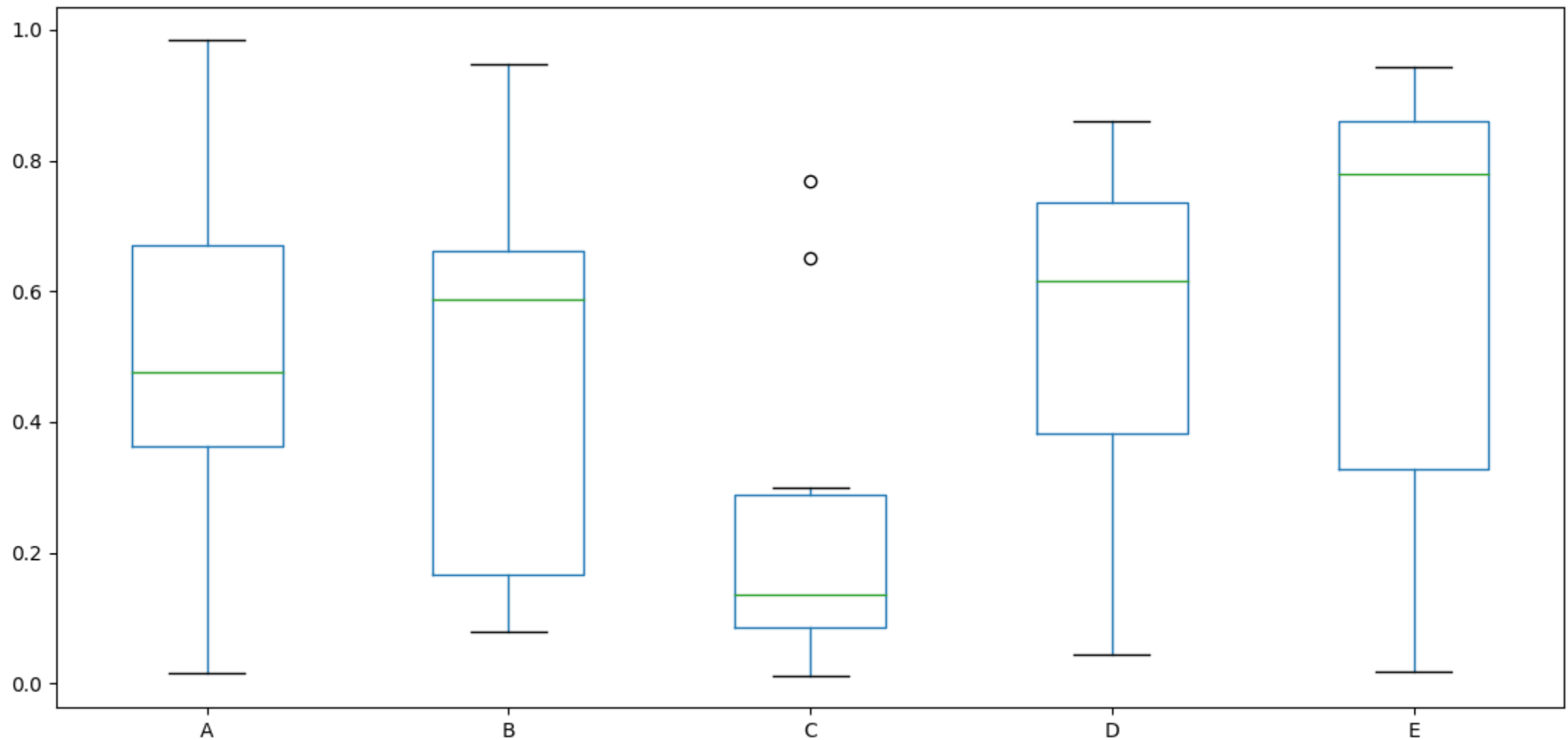
```
df=pd.DataFrame({'a':np.random.randn(1000)+1,'b':np.random.randn(1000),  
                 'c':np.random.randn(1000) - 1, 'd':np.random.randn(1000) - 5}, columns=['a', 'b', 'c', 'd'])
```

```
df.hist(bins=20)
```



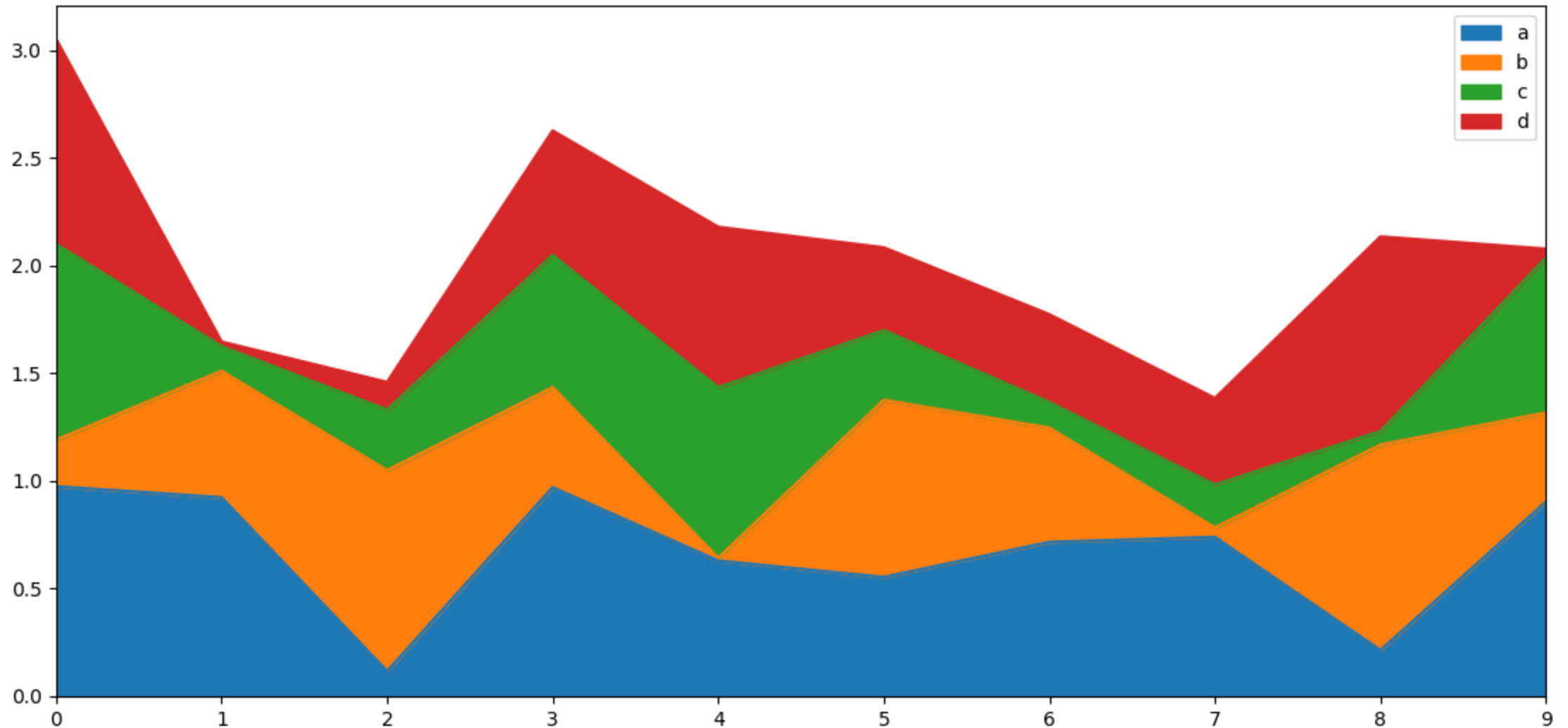
Box Plots

```
df = pd.DataFrame(np.random.rand(10, 5), columns=['A', 'B', 'C', 'D', 'E'])  
df.plot.box()
```



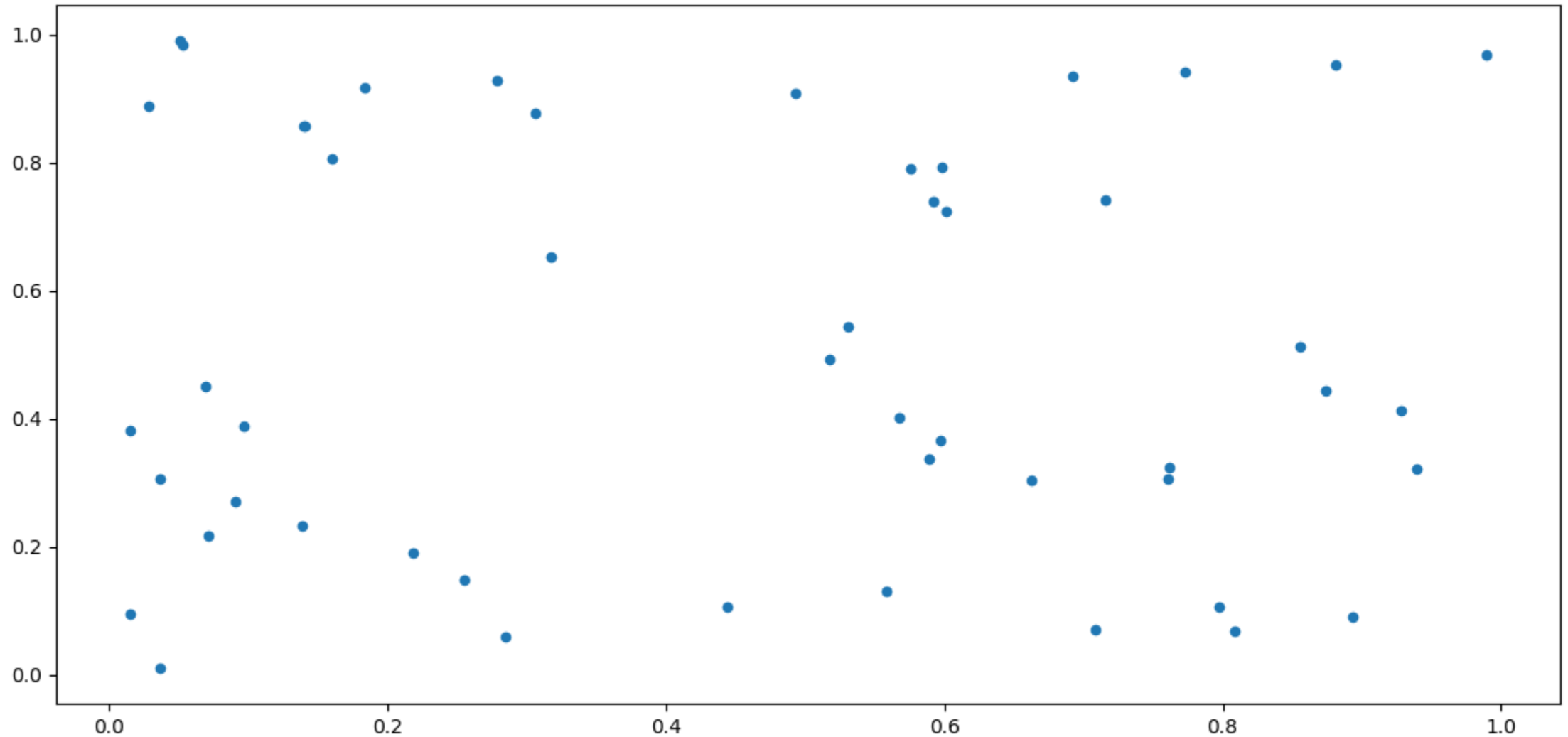
Area Plot

```
df = pd.DataFrame(np.random.rand(10, 4), columns=['a', 'b', 'c', 'd'])  
df.plot.area()
```



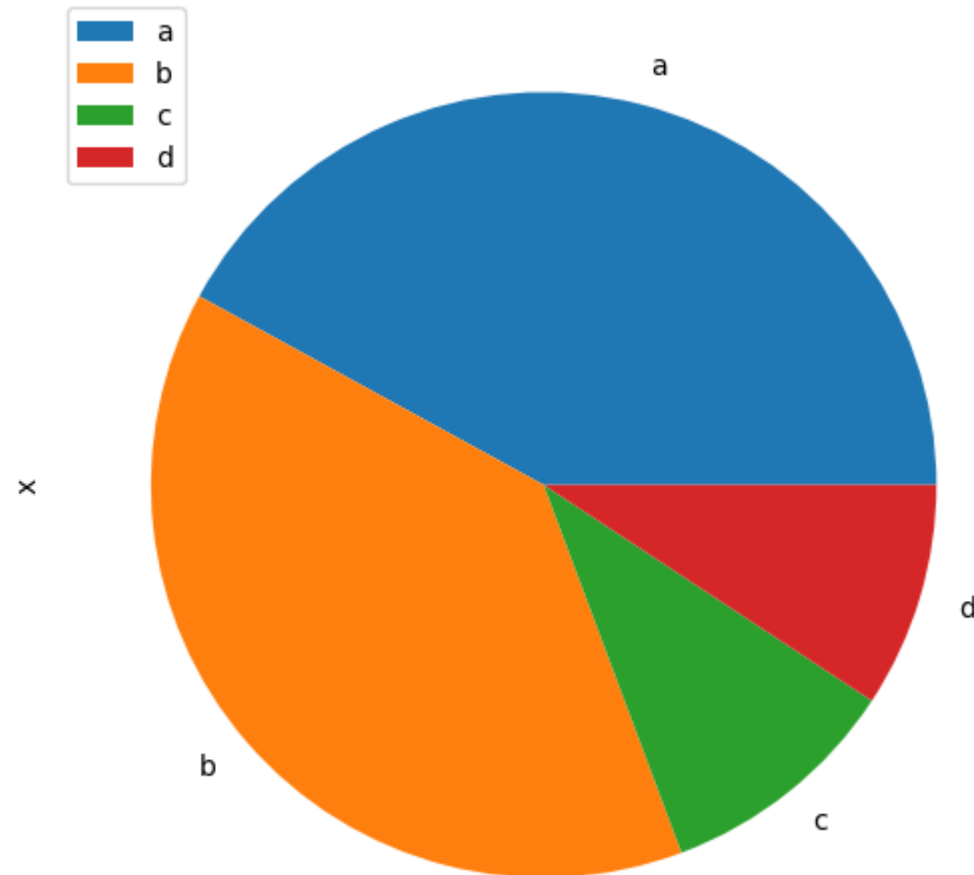
Scatter Plots

```
df = pd.DataFrame(np.random.rand(50, 4), columns=['a', 'b', 'c', 'd'])  
df.plot.scatter(x='a', y='b')
```



Pie Chart

```
df = pd.DataFrame(3 * np.random.rand(4), index=['a', 'b', 'c', 'd'], columns=['x'])  
df.plot.pie(subplots=True)
```



IO Tools

- The two workhorse functions for reading text files (or the flat files) are `read_csv()` and `read_table()`. They both use the same parsing code to intelligently convert tabular data into a `DataFrame` object
- Example: The **temp.csv** file data looks like

```
S.No,Name,Age,City,Salary
1,Tom,28,Toronto,20000
2,Lee,32,HongKong,3000
3,Steven,43,Bay Area,8300
4,Ram,38,Hyderabad,3900
```

```
pandas.read_csv(filepath_or_buffer, sep=',', delimiter=None, header='infer',
names=None, index_col=None, usecols=None
```

```
pandas.read_csv(filepath_or_buffer, sep='\t', delimiter=None, header='infer',
names=None, index_col=None, usecols=None
```

Example

- `df=pd.read_csv("temp.csv")`
- `df=pd.read_csv("temp.csv",index_col=['S.No'])`
- `df = pd.read_csv("temp.csv", dtype={'Salary': np.float64})`
- `df=pd.read_csv("temp.csv", names=['a', 'b', 'c','d','e'])`

	a	b	c	d	e
0	S.No	Name	Age	City	Salary
1	1	Tom	28	Toronto	20000
2	2	Lee	32	HongKong	3000
3	3	Steven	43	Bay Area	8300
4	4	Ram	38	Hyderabad	3900

`df=pd.read_csv("temp.csv",names=['a','b','c','d','e'],header=0)`

→ What is about

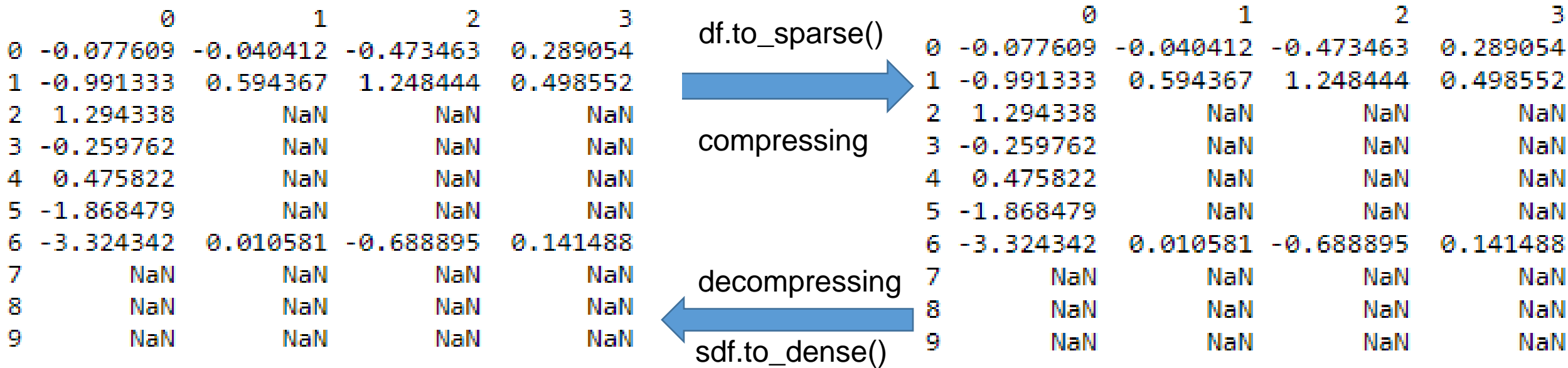
- `df=pd.read_csv("temp.csv", skiprows=2)`

	2	Lee	32	HongKong	3000
0	3	Steven	43	Bay Area	8300
1	4	Ram	38	Hyderabad	3900

Sparse Data

- Sparse objects are “compressed” when any data matching a specific value (NaN / missing value, though any value can be chosen) is omitted. A special SparseIndex object tracks where data has been “sparsified”.
- **Using to compress data to improve memory if data is sparse**
- **Use for Series data and Data Frame**
- Sparse data should have the same dtype as its dense representation. Currently, float64, int64 and bool dtypes are supported. Depending on the original dtype, fill_value default changes.
 - float64 – np.nan
 - int64 – 0
 - bool – False

Example



sdf.density → density = 0.4

Caveats & Gotchas

- Caveats means warning and gotcha means an unseen problem.
- Pandas follows the numpy convention of raising an error when you try to convert something to a bool. This happens in an if or when using the Boolean operations, and, or, or not. It is not clear what the result should be. Should it be True because it is not zerolength? False because there are False values? It is unclear, so instead, Pandas raises a ValueError.
- Series data
 - .empty
 - .bool()
 - .item()
 - .any()
 - .all()
 - Bitwise Boolean
 - Isin

Example

```
if pd.Series([False, True, False]).any():  
    print("I am any")
```



I am any

```
print pd.Series([True]).bool()
```



True

```
s = pd.Series(range(5))  
print s==4
```



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

```
s = pd.Series(list('abc'))  
s = s.isin(['a', 'c', 'e'])  
print s
```



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

Comparison with SQL

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

Query	T-SQL	Pandas
SELECT	SELECT total_bill, tip, smoker, time FROM tips LIMIT 5;	tips[['total_bill', 'tip', 'smoker', 'time']].head(5)
WHERE	SELECT * FROM tips WHERE time = 'Dinner' LIMIT 5;	tips[tips['time'] == 'Dinner'].head(5)
GROUP BY	SELECT sex, count(*) FROM tips GROUP BY sex;	tips.groupby('sex').size()
TOP N ROWs	SELECT * FROM tips LIMIT 5 ;	tips.head(5)

Mastering Pandas - To master data manipulation in Python using Pandas, here's what you need to learn:

- read csv
- set index
- reset index
- loc
- iloc
- drop
- dropna
- fillna
- assign
- filter
- query
- rename
- sort values
- agg
- groupby
- concat
- merge
- pivot
- melt

THANK YOU
Q & A