Data Manipulation with Python (Pandas)

Introduction to Pandas

- Pandas is an open-source, BSD-licensed Python library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.
- Python with Pandas is used in a wide range of fields including academic and commercial domains including finance, economics, Statistics, analytics, etc.
- 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 sub setting 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.

Data Structures

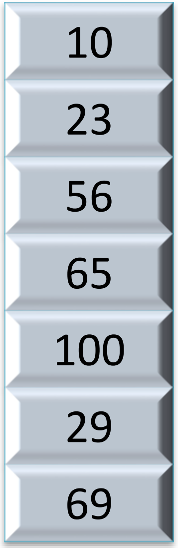
- Pandas deals with the following three data structures –
  - Series
  - DataFrame
  - Panel
- These data structures are built on top of Numpy array, which means they are fast.
- The best way to think of these data structures is that the higher dimensional data structure is a container of its lower dimensional data structure. For example, DataFrame is a container of Series, Panel is a container of DataFrame.

Data Structure	Dimensions	Description
Series	1	1D labeled homogeneous array, sizeimmutable.
Data Frames	2	General 2D labeled, size-mutable tabular structure with potentially heterogeneously typed columns.
Panel	3	General 3D labeled, size-mutable array.

Series

- Series is a one-dimensional array like structure with homogeneous data.
  For example, the following series is a collection of integers 10, 23, 56, …

- **Key Points**
  - Homogeneous data
  - Size Immutable
  - Values of Data Mutable




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

```
#Create an Empty Series
s = pd.Series()
print (s)
```

```
Series([], dtype: float64)
```

```
print (type(s))
```

```
<class 'pandas.core.series.Series'>
```

pandas.Series

- Series is a one-dimensional labeled array capable of holding data of any type (integer, string, float, python objects, etc.).
- The axis labels are collectively called index.
- A pandas Series can be created using the following constructor –

  pandas.Series(data, index, dtype, copy)

- The parameters of the constructor are as follows –

Parameter	Description
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

pandas.Series

- Creation of Series from *Array*

```
data = np.array(['a','b','c','d'])
print (data)
```

```
['a' 'b' 'c' 'd']
```

```
# Series from ndarray
s = pd.Series(data)
print (s)
```

```
0     a
1     b
2     c
3     d
dtype: object
```

```
# Series from ndarray with index parameter
s = pd.Series(data,index=[100,101,102,103])
print (s)
```

```
100     a
101     b
102     c
103     d
dtype: object
```

pandas.Series

- Creation of Series from *Dictionary*

```
data = {'a' : 0, 'b' : 1, 'c' : 2}
print (data)
```

```
{'a': 0, 'b': 1, 'c': 2}
```

```
s = pd.Series(data)
print (s)
```

```
a    0
b    1
c    2
dtype: int64
```

```
# Series from dict with index parameter
s = pd.Series(data, index=['b', 'c', 'd', 'a'])
print (s)
```

```
b    1.0
c    2.0
d    NaN
a    0.0
dtype: float64
```

pandas.Series

- Accessing Data from Series with Position

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

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

```
#retrieve the first element
print (s[0])
```

```
1
```

```
#retrieve the first three element
print (s[:3])
```

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

```
#retrieve the last three element
print (s[-3:])
```

```
c    3
d    4
e    5
dtype: int64
```

pandas.Series

- Retrieve Data Using Label (Index)

```
#retrieve a single element
print (s['a'])
```

```
1
```

```
#retrieve multiple elements
print (s[['a','c','d']])
```

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

```
#retrieve non existant index
print (s['f'])
```

```
KeyError: 'f'
```

Data Operations

- Series Basic Functionality

S. No	Attributes or Methods	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.
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.

Data Operations

axes

```
#Create a series with random numbers
s = pd.Series(np.random.randn(8))
print (s)
```

```
0   -1.184971
1    1.086263
2    1.077182
3   -1.585128
4   -0.188483
5    1.301081
6   -0.551787
7   -0.172077
dtype: float64
```

```
print ("The axes are:")
print (s.axes)
```

```
The axes are:
[RangeIndex(start=0, stop=8, step=1)]
```

```
s2 = pd.Series(np.random.randn(4),index=[11,12,13,14])
print (s2)
```

```
11   -1.557158
12   -0.518703
13   -0.951047
14   -0.247724
dtype: float64
```

```
print ("The axes are:")
print (s2.axes)
```

```
The axes are:
[Int64Index([11, 12, 13, 14], dtype='int64')]
```

Data Operations

dtype

```
print(s.dtype)
```

```
float64
```

ndim

```
print ("The dimensions of the object:")
print (s.ndim)
```

```
The dimensions of the object:
1
```

size

```
print ("The size of the object:")
print (s.size)
```

```
The size of the object:
8
```

empty

```
se = pd.Series()
print (se)
```

```
Series([], dtype: float64)
```

```
print ("Is the Object empty?")
print (se.empty)
```

```
Is the Object empty?
True
```

```
print ("Is the Object empty?")
print (s.empty)
```

```
Is the Object empty?
False
```

Data Operations

values

```
print ("The actual data series is:")
print (s.values)
```

```
The actual data series is:
[-1.1849715   1.0862629   1.07718216 -1.58512769 -0.18848313  1.30108131
 -0.55178674 -0.17207681]
```

head

```
print ("The first two elements of series:")
print (s.head(2))
```

```
The first two elements of the data series:
0   -1.184971
1    1.086263
dtype: float64
```

tail

```
print ("The last two elements of series:")
print (s.tail(2))
```

```
The last two elements of the data series:
6   -0.551787
7   -0.172077
dtype: float64
```

DataFrame

- A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns.

- **Features of DataFrame**

  Potentially columns are of different types

  Size – Mutable

  Labeled axes (rows and columns)

  Can Perform Arithmetic operations on rows and columns

DataFrame

Structure

- Let us represents the data of a sales team of an organization with their overall performance rating. The data is represented in rows and columns.
- Each column represents an attribute and each row represents a person.

Name	Age	Gender	Rating
Ketan	32	Male	3.45
Ram	28	Female	4.6
Amit	45	Male	3.9
Deepti	38	Female	2

DataFrame

- A pandas DataFrame can be created using the following constructor –

pandas.DataFrame(data, index, columns, dtype, copy)

Parameter	Description
data	data takes various forms like ndarray, series, map, lists, dict, constants and also another DataFrame.
index	For the row labels, the Index to be used for the resulting frame is Optional Default np.arange(n) if no index is passed.
columns	For column labels, the optional default syntax is - np.arange(n). This is only true if no index is passed.
dtype	Data type of each column.
copy	This command (or whatever it is) is used for copying of data, if the default is False.

DataFrame

- DataFrame from Lists

```
data = [11,12,13,14,15]
df = pd.DataFrame(data)
print (df)
```

```
    0
0  11
1  12
2  13
3  14
4  15
```

```
data = [['Alok',10],['Bhushan',12],['Chitra',13]]

df = pd.DataFrame(data,columns=['Name','Age'])
print (df)
```

```
      Name  Age
0     Alok   10
1  Bhushan   12
2   Chitra   13
```

```
df = pd.DataFrame(data,columns=['Name','Age'],dtype=float)
print (df)
```

```
      Name   Age
0     Alok  10.0
1  Bhushan  12.0
2   Chitra  13.0
```

DataFrame

- DataFrame from Dict of Lists

```
data = {'Name':['Swapnil', 'Rahul', 'Viraj', 'Pranav'],
        'Age':[28,34,29,42]}
print(data)
```

```
{'Name': ['Swapnil', 'Rahul', 'Viraj', 'Pranav'], 'Age': [28, 34, 29, 42]}
```

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

```
      Name  Age
0  Swapnil   28
1    Rahul   34
2    Viraj   29
3   Pranav   42
```

```
df = pd.DataFrame(data, index=['rank1','rank2','rank3','rank4'])
print (df)
```

```
          Name  Age
rank1  Swapnil   28
rank2    Rahul   34
rank3    Viraj   29
rank4   Pranav   42
```

DataFrame

- DataFrame from Dict of Series

```
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)
```

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

Column Selection

```
print (df ['one'])
```

```
a    1.0
b    2.0
c    3.0
d    NaN
Name: one, dtype: float64
```

DataFrame

Column Addition

- Adding a new column to an existing DataFrame object with column label by passing new series

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

```
   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:"
df['four']=df['one']+df['three']
print (df)
```

```
   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
```

DataFrame

Column Deletion

```
# using del function
del df['one']
print (df)
```

```
   two  three  four
a    1   10.0  11.0
b    2   20.0  22.0
c    3   30.0  33.0
d    4    NaN   NaN
```

```
# using pop function
x = df.pop('two')
print (x)
```

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

```
print (df)
```

```
   three  four
a   10.0  11.0
b   20.0  22.0
c   30.0  33.0
d    NaN   NaN
```

DataFrame

Column Selection

```
#Selection by Label

print (df.loc['b'])
```

```
three    20.0
four     22.0
Name: b, dtype: float64
```

```
#Selection by integer location

print (df.iloc[2])
```

```
three    30.0
four     33.0
Name: c, dtype: float64
```

```
#Slice Rows

print (df[2:4])
```

```
   three  four
c   30.0  33.0
d    NaN   NaN
```

DataFrame

append()

```
df = pd.DataFrame([[1, 2], [3, 4]],
                  columns = ['a','b'])
print(df)
```

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

```
df2 = pd.DataFrame([[5, 6], [7, 8]],
                   columns = ['a','b'])
print(df2)
```

```
   a  b
0  5  6
1  7  8
```

```
df3 = df.append(df2)
print (df3)
```

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

```
df4 = df.append(df2,ignore_index = True)
print (df4)
```

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

DataFrame

Deletion of Rows

```
df = pd.DataFrame([[1, 2], [3, 4]], columns = ['a','b'])
print(df)
```

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

```
# Drop rows with label 0
df = df.drop(0)
print (df)
```

```
   a  b
1  3  4
```

Data Operations

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.
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.

Data Operations

dtype

```
print (df.dtypes)
```

```
Name          object
Age            int64
Rating       float64
dtype: object
```

ndim

```
print (df.ndim)
```

```
2
```

shape

```
print (df.shape)
```

```
(7, 3)
```

size

```
print (df.size)
```

```
21
```

values

```
print (df.values)
```

```
[['Pranay' 25 4.23]
 ['Swapnil' 26 3.24]
 ['Ayush' 25 3.98]
 ['Viraj' 23 2.56]
 ['Bhushan' 30 3.2]
 ['Ashwin' 29 4.6]
 ['Vishal' 23 3.8]]
```

head

```
print (df.head(2))
```

```
      Name  Age  Rating
0   Pranay   25    4.23
1  Swapnil   26    3.24
```

tail

```
print (df.tail(2))
```

```
     Name  Age  Rating
5  Ashwin   29     4.6
6  Vishal   23     3.8
```

Essential Functionality

- A critical method on pandas objects is **reindex()**, which means to create a new object with the data conformed to a new index.
- 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.

Argument	Description
ffill or pad	Fill (or carry) values forward
bfill or backfill	Fill (or carry) values backward

Re-indexing

- With DataFrame, reindex can alter either the (row) index, columns, or both.

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

```
d    4.5
b    7.2
a   -5.3
c    3.6
dtype: float64
```

- When passed just a sequence, the rows are **reindexed** in the result.

```
obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])
obj2
```

```
a   -5.3
b    7.2
c    3.6
d    4.5
e    NaN
dtype: float64
```

Re-indexing

```
obj3 = pd.Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])
print(obj3)
```

```
0      blue
2    purple
4    yellow
dtype: object
```

```
obj3.reindex(range(6), method='ffill')
```

```
0      blue
1      blue
2    purple
3    purple
4    yellow
5    yellow
dtype: object
```

Re-indexing

- The columns can be **reindexed** using the **columns** keyword. Both can be **reindexed** in one shot also.

```
frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
                     index=['a', 'c', 'd'],
                     columns=['Nagpur', 'Raipur', 'Hyderabad'])
print(frame)
```

```
   Nagpur  Raipur  Hyderabad
a       0       1          2
c       3       4          5
d       6       7          8
```

```
frame2 = frame.reindex(['a', 'b', 'c', 'd'])
frame2
```

	Nagpur	Raipur	Hyderabad
a	0.0	1.0	2.0
b	NaN	NaN	NaN
c	3.0	4.0	5.0
d	6.0	7.0	8.0

Re-indexing

```
states = ['Raipur', 'Indore', 'Hyderabad']
frame.reindex(columns=states)
```

	Raipur	Indore	Hyderabad
a	1	NaN	2
c	4	NaN	5
d	7	NaN	8

```
frame
```

	Nagpur	Raipur	Hyderabad
a	0	1	2
c	3	4	5
d	6	7	8

```
frame.reindex(index=['a', 'b', 'c', 'd'],
              columns=states)
```

	Raipur	Indore	Hyderabad
a	1.0	NaN	2.0
b	NaN	NaN	NaN
c	4.0	NaN	5.0
d	7.0	NaN	8.0

Re-indexing

- Dropping one or more entries from an axis is easy if one have an index array or list without those entries. As that can require a bit of set logic, the **drop** method will return a new object with the indicated value or values deleted from an axis.

```
obj = pd.Series(np.arange(5), index=['a', 'b', 'c', 'd', 'e'])
obj
```

```
a    0
b    1
c    2
d    3
e    4
dtype: int32
```

```
new_obj = obj.drop('c')
new_obj
```

```
a    0
b    1
d    3
e    4
dtype: int32
```

```
obj.drop(['d', 'c'])
```

```
a    0
b    1
e    4
dtype: int32
```

Re-indexing

- With DataFrame, index values can be deleted from either axis.

```
data = pd.DataFrame(np.arange(16).reshape(4, 4),
                    index=['Nagpur', 'Raipur', 'Hyderabad', 'Indore'],
                    columns=['one', 'two', 'three', 'four'])
data
```

	one	two	three	four
Nagpur	0	1	2	3
Raipur	4	5	6	7
Hyderabad	8	9	10	11
Indore	12	13	14	15

```
data.drop(['Raipur', 'Nagpur'])
```

	one	two	three	four
Hyderabad	8	9	10	11
Indore	12	13	14	15

Re-indexing

```
data.drop('two', axis = 1)
```

	one	three	four
Nagpur	0	2	3
Raipur	4	6	7
Hyderabad	8	10	11
Indore	12	14	15

```
data.drop(['two', 'four'], axis = 'columns')
```

	one	three
Nagpur	0	2
Raipur	4	6
Hyderabad	8	10
Indore	12	14

Arithmetic and data alignment

- One of the most important pandas features is the behavior of arithmetic between objects with different indexes.

- When adding together objects, if any index pairs are not the same, the respective index in the result will be the union of the index pairs

- The internal data alignment introduces NAN values in the indices that don't overlap.

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

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

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

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

```
s1 + s2
```

```
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
```

Arithmetic and data alignment

- In arithmetic operations **between differently-indexed objects**, you might want to fill with a special value, like 0, when an axis label is found in one object but not the other.

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

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

```
df2 = pd.DataFrame(np.arange(20).reshape(4, 5),columns=list('abcde'))
df2.loc[1, 'b'] = np.nan
df2
```

	a	b	c	d	e
0	0	1.0	2	3	4
1	5	NaN	7	8	9
2	10	11.0	12	13	14
3	15	16.0	17	18	19

```
df1 + df2
```

	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

Arithmetic and data alignment

- Relatively, when **reindexing** a Series or DataFrame, one can also specify a different fill value.

```
df1.add(df2, fill_value=0)
```

	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

```
df1.reindex(columns=df2.columns, fill_value=0)
```

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

Descriptive Statistics with Pandas

- Pandas objects are equipped with a set of common mathematical and statistical methods. Most of these fall into the category of reductions or summary statistics, methods that extract a single value (like the sum or mean) from a Series or a Series of values from the rows or columns of a DataFrame. Compared with the equivalent methods of NumPy arrays, they are all built from the ground up to exclude missing data.

```
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
```

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

```
df.sum()
```

```
one     9.25
two    -5.80
dtype: float64
```

Descriptive Statistics with Pandas

- Calling DataFrame's sum method returns a Series containing column sums. Passing *axis=1* sums over the rows instead.
- NA values are excluded unless the entire slice (row or column in this case) is NA. This can be disabled using the **skipna** option.

```
df.sum(axis='columns')
```

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

```
df.mean(axis='columns', skipna=False)
```

```
a       NaN
b     1.300
c       NaN
d    -0.275
dtype: float64
```

```
df.cumsum()
```

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

Descriptive Statistics with Pandas

- Other method is ***describe***, producing multiple summary statistics in one shot. On non-numeric data, describe produces alternate summary statistics.

```
df.describe()
```

	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

```
obj = pd.Series(['a', 'a', 'b', 'c'] * 4)
obj
```

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

```
obj.describe()
```

```
count     16
unique     3
top        a
freq       8
dtype: object
```

Correlation and Covariance

- Let's consider some DataFrames of stock prices and volumes obtained from Yahoo! Finance.

```
price = pd.read_pickle('yahoo_price.pkl')
volume = pd.read_pickle('yahoo_volume.pkl')
```

```
price.head()
```

Date	AAPL	GOOG	IBM	MSFT
2010-01-04	27.990226	313.062468	113.304536	25.884104
2010-01-05	28.038618	311.683844	111.935822	25.892466
2010-01-06	27.592626	303.826685	111.208683	25.733566
2010-01-07	27.541619	296.753749	110.823732	25.465944
2010-01-08	27.724725	300.709808	111.935822	25.641571

```
volume.head()
```

Date	AAPL	GOOG	IBM	MSFT
2010-01-04	123432400	3927000	6155300	38409100
2010-01-05	150476200	6031900	6841400	49749600
2010-01-06	138040000	7987100	5605300	58182400
2010-01-07	119282800	12876600	5840600	50559700
2010-01-08	111902700	9483900	4197200	51197400

Correlation and Covariance

- First we will compute percent changes of the prices.

```
returns = price.pct_change()
returns.tail()
```

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

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

```
returns['MSFT'].corr(returns['IBM'])
```

```
0.4997636114415114
```

```
returns['MSFT'].cov(returns['IBM'])
```

```
8.870655479703546e-05
```

Correlation and Covariance

- DataFrame's **corr** and **cov** methods, on the other hand, return a full correlation or covariance matrix as a DataFrame, respectively.

```
returns.corr()
```

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

```
returns.cov()
```

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

Correlation and Covariance

- Using DataFrame’s **corrwith** method, one can compute pairwise correlations between a DataFrame’s columns or rows with another Series or DataFrame.

```
returns.corrwith(returns['IBM'])
```

```
AAPL    0.386817
GOOG    0.405099
IBM     1.000000
MSFT    0.499764
dtype: float64
```

Unique Values, Value Counts, Membership

- There is another class of related methods extracts information about the values contained in a one-dimensional Series.

- The first function is **unique**, which gives one an array of the unique values in a Series. The unique values are not necessarily returned in sorted order, but could be sorted after the fact if needed (uniques.sort()). Relatedly, value_counts computes a Series containing value frequencies.

```
obj = pd.Series(['c','a','d','a','a','b','b','c','c'])
obj
```

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

```
uniques = obj.unique()
uniques
```

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

```
obj.value_counts()
```

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

Unique Values, Value Counts, Membership

- The Series is sorted by value in descending order as a convenience. **value_counts** is also available as a top-level pandas method that can be used with any array or sequence.

```
pd.value_counts(obj.values, sort=False)
```

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

```
mask = obj.isin(['b', 'c'])
mask
```

```
0     True
1    False
2    False
3    False
4    False
5     True
6     True
7     True
8     True
dtype: bool
```

```
obj[mask]
```

```
0    c
5    b
6    b
7    c
8    c
dtype: object
```

- **isin** is responsible for vectorized set membership and can be very useful in filtering a data set down to a subset of values in a Series or column in a DataFrame.

Data Loading, Storage, & File Formats

- The tools & libraries for data analysis are of little use if one can't easily import and export data in Python. We will focused on input and output with pandas objects, though there are of course numerous tools in other libraries to aid in this process.
- Input and output typically falls into a few main categories:
    - Reading text files and other more efficient on-disk formats
    - Loading data from databases
    - Interacting with network sources like web APIs.
- Python pandas features a number of functions for reading tabular data as a DataFrame object, though **read_csv()** is likely the one used the most.

Data Loading, Storage, & File Formats

```
df = pd.read_csv("temp.txt")
df
```

	S.No	Name	Age	City	Salary	DOB
0	1	Vishal	NaN	Nagpur	20000	22-12-1998
1	2	Pranay	32.0	Mumbai	3000	23-02-1991
2	3	Akshay	43.0	Banglore	8300	12-05-1985
3	4	Ram	38.0	Hyderabad	3900	01-12-1992

```
print (df.shape)
```

(4, 6)

```
df = pd.read_csv("temp.txt",usecols = ['Name','Age'])
df
```

	Name	Age
0	Vishal	NaN
1	Pranay	32.0
2	Akshay	43.0
3	Ram	38.0

Data Loading, Storage, & File Formats

```
#----------------- custom index -----------------
df = pd.read_csv("temp.txt",index_col=['S.No'])
df
```

	Name	Age	City	Salary	DOB
S.No					
1	Vishal	NaN	Nagpur	20000	22-12-1998
2	Pranay	32.0	Mumbai	3000	23-02-1991
3	Akshay	43.0	Banglore	8300	12-05-1985
4	Ram	38.0	Hyderabad	3900	01-12-1992

```
df.shape
```

```
(4, 5)
```

```
df.dtypes
```

```
Name       object
Age       float64
City       object
Salary      int64
DOB        object
dtype: object
```

Data Loading, Storage, & File Formats

```
#------------------ Converters --------------------
date_cols = ['DOB']
df = pd.read_csv("temp.txt",parse_dates=date_cols)
df
```

	S.No	Name	Age	City	Salary	DOB
0	1	Vishal	NaN	Nagpur	20000	1998-12-22
1	2	Pranay	32.0	Mumbai	3000	1991-02-23
2	3	Akshay	43.0	Banglore	8300	1985-12-05
3	4	Ram	38.0	Hyderabad	3900	1992-01-12

```
df.dtypes
```

```
S.No                int64
Name               object
Age               float64
City               object
Salary              int64
DOB        datetime64[ns]
dtype: object
```

```
df['DOB'].dt.year
```

```
0    1998
1    1991
2    1985
3    1992
Name: DOB, dtype: int64
```

Data Loading, Storage, & File Formats

```
#---------------- Header_names -----------------

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

	a	b	c	d	e	f
0	S.No	Name	Age	City	Salary	DOB
1	1	Vishal	NaN	Nagpur	20000	22-12-1998
2	2	Pranay	32	Mumbai	3000	23-02-1991
3	3	Akshay	43	Banglore	8300	12-05-1985
4	4	Ram	38	Hyderabad	3900	01-12-1992

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

	a	b	c	d	e	f
0	1	Vishal	NaN	Nagpur	20000	22-12-1998
1	2	Pranay	32.0	Mumbai	3000	23-02-1991
2	3	Akshay	43.0	Banglore	8300	12-05-1985
3	4	Ram	38.0	Hyderabad	3900	01-12-1992

Data Loading, Storage, & File Formats

```
#------------------ skiprows --------------------
df=pd.read_csv("temp.txt", skiprows = 2,
               names=['a', 'b', 'c','d','e','f'],
               header=0)
df
```

	a	b	c	d	e	f
0	3	Akshay	43	Banglore	8300	12-05-1985
1	4	Ram	38	Hyderabad	3900	01-12-1992

```
df = pd.read_csv("temp.txt")
df.loc[0,'Age'] = 21
df
```

	S.No	Name	Age	City	Salary	DOB
0	1	Vishal	21.0	Nagpur	20000	22-12-1998
1	2	Pranay	32.0	Mumbai	3000	23-02-1991
2	3	Akshay	43.0	Banglore	8300	12-05-1985
3	4	Ram	38.0	Hyderabad	3900	01-12-1992

```
df.to_csv("df.txt",index = False)
```

JSON , HTML , Excel & Web API

- JSON (short for JavaScript Object Notation) has become one of the standard formats for sending data by HTTP request between web browsers and other applications. It is a much more flexible data format than a tabular text form like CSV.
- There are several Python libraries for reading and writing JSON data.
- One can pass a list of JSON objects to the DataFrame constructor and select a subset of the data fields to convert a JSON object or list of objects to a DataFrame.

JSON , HTML , Excel & Web API

```python
data = pd.read_json('example.json')
data
```

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

```python
data.to_json()
```

```
'{"a":{"0":1,"1":4,"2":7},"b":{"0":2,"1":5,"2":8},"c":{"0":3,"1":6,"2":9}}'
```

```python
data.to_json(orient='records')
```

```
'[{"a":1,"b":2,"c":3},{"a":4,"b":5,"c":6},{"a":7,"b":8,"c":9}]'
```

JSON , HTML , Excel & Web API

- HTML data can be read by read_html() method of Pandas which reads HTML tables into a list of DataFrame objects.

```
tables = pd.read_html('fdic_failed_bank_list.html')
tables
```

```
[                              Bank Name             City  ST   CERT  \
 0                          Allied Bank         Mulberry  AR     91
 1         The Woodbury Banking Company         Woodbury  GA  11297
 2               First CornerStone Bank  King of Prussia  PA  35312
 3                   Trust Company Bank          Memphis  TN   9956
 4           North Milwaukee State Bank        Milwaukee  WI  20364
 ..                                 ...              ...  ..    ...
 542                 Superior Bank, FSB         Hinsdale  IL  32646
 543                Malta National Bank            Malta  OH   6629
 544    First Alliance Bank & Trust Co.       Manchester  NH  34264
 545  National State Bank of Metropolis       Metropolis  IL   3815
 546                   Bank of Honolulu         Honolulu  HI  21029

 [547 rows x 7 columns]]
```

JSON , HTML , Excel & Web API

```
failures = tables[0]
failures.head()
```

	Bank Name	City	ST	CERT	Acquiring Institution	Closing Date	Updated Date
0	Allied Bank	Mulberry	AR	91	Today's Bank	September 23, 2016	November 17, 2016
1	The Woodbury Banking Company	Woodbury	GA	11297	United Bank	August 19, 2016	November 17, 2016
2	First CornerStone Bank	King of Prussia	PA	35312	First-Citizens Bank & Trust Company	May 6, 2016	September 6, 2016
3	Trust Company Bank	Memphis	TN	9956	The Bank of Fayette County	April 29, 2016	September 6, 2016
4	North Milwaukee State Bank	Milwaukee	WI	20364	First-Citizens Bank & Trust Company	March 11, 2016	June 16, 2016

JSON , HTML , Excel & Web API

```
close_timestamps = pd.to_datetime(failures['Closing Date'])
close_timestamps.dt.year.value_counts()
```

```
2010    157
2009    140
2011     92
2012     51
2008     25
2013     24
2014     18
2002     11
2015      8
2016      5
2004      4
2001      4
2007      3
2003      3
2000      2
Name: Closing Date, dtype: int64
```

JSON , HTML , Excel & Web API

- Python pandas also supports reading tabular data stored in Excel 2003 (and higher) files using the **read_excel()**.

```
frame = pd.read_excel('ex1.xlsx', 'Sheet1')
frame
```

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

```
frame.to_excel('ex2.xlsx')
```

JSON , HTML , Excel & Web API

- Many websites have public APIs providing data feeds via JSON or some other format. One easy-to-use method that can be used is the **requests** package .

```
import requests
url = 'https://api.github.com/repos/pandas-dev/pandas/issues'
resp = requests.get(url)
resp
```

```
<Response [200]>
```

```
data = resp.json()
data[0]['title']
```

```
'API: expected result of concat of SparseArray with Categorical?'
```

JSON , HTML , Excel & Web API

```
issues = pd.DataFrame(data, columns=['number', 'title',
                                      'labels', 'state'])
issues
```

	number	title	labels	state
0	34459	API: expected result of concat of SparseArray ...	[{'id': 1741841389, 'node_id': 'MDU6TGFiZWwxNz...	open
1	34458	CLN: Clean csv files in test data GH34427	[]	open
2	34457	API: SparseArray.astype behaviour to always pr...	[{'id': 35818298, 'node_id': 'MDU6TGFiZWwzNTgx...	open
3	34456	BUG: behaviour of astype_nansafe(..., copy=Fals...	[{'id': 76811, 'node_id': 'MDU6TGFiZWw3NjgxMQ=...	open
4	34455	BUG: Groupby.apply raises KeyError for Float64...	[{'id': 697792067, 'node_id': 'MDU6TGFiZWw2OTc...	open
5	34454	TST/REF: refactor the arithmetic tests for Int...	[{'id': 1817503692, 'node_id': 'MDU6TGFiZWwxOD...	open
6	34453	[ENH] Allow pad, backfill and cumcount in grou...	[]	open