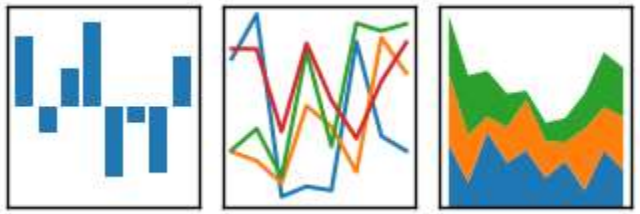
Python-Pandas

Dr. Sarwan Singh

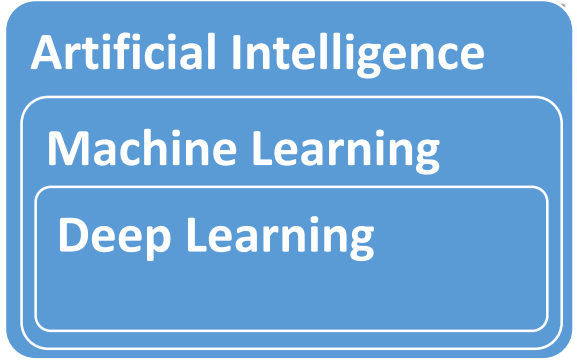
pandas

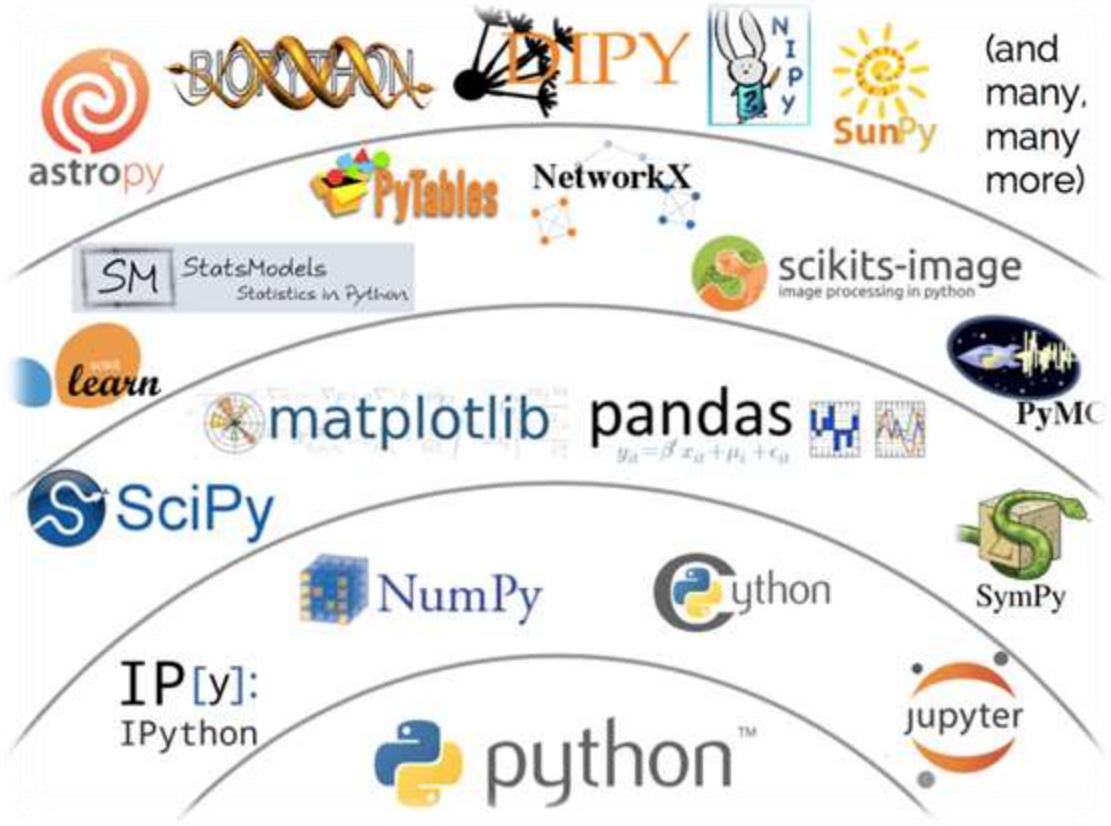
$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$



Agenda

- Introduction – Panel Data System
- History and usage
- Series, DataFrame, Panel
- Sorting





One guiding principle of Python code is that

"explicit is better than implicit"

Introduction

- Pandas is Python package for data analysis.
- Pandas is an open-source Python Library built on top of Numpy
- Provides 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.
- In 2008, developer Wes McKinney started developing pandas when in need of high performance, flexible tool for analysis of data. {at AQR capital Management LLC}
- 30,000 lines of tested Python/Cython code

Introduction

```
import pandas
pandas.__version__
```

```
'0.20.3'
```

- Pandas can accomplish five typical steps in the processing and analysis of data, regardless of the origin of data —

  load → prepare → manipulate → model → analyze.

- Python with Pandas is used in a wide range of fields including academic and commercial domains including finance, economics, Statistics, analytics, etc.
- Pandas is easy to use and powerful, but

  "*with great power comes great responsibility*"

eases data munging

Numpy vs Pandas

- Pandas is designed for working with tabular or heterogeneous data.
- NumPy, is best suited for working with homogeneous numerical array data
- **Numpy** is required by **pandas** (and by virtually all numerical tools for Python)
- numpy consumes (roughtly 1/3) less memory compared to pandas
- numpy generally performs better than pandas for 50K rows or less
- pandas generally performs better than numpy for 500K rows or more
- for 50K to 500K rows, it is a toss up between pandas and numpy depending on the kind of operation

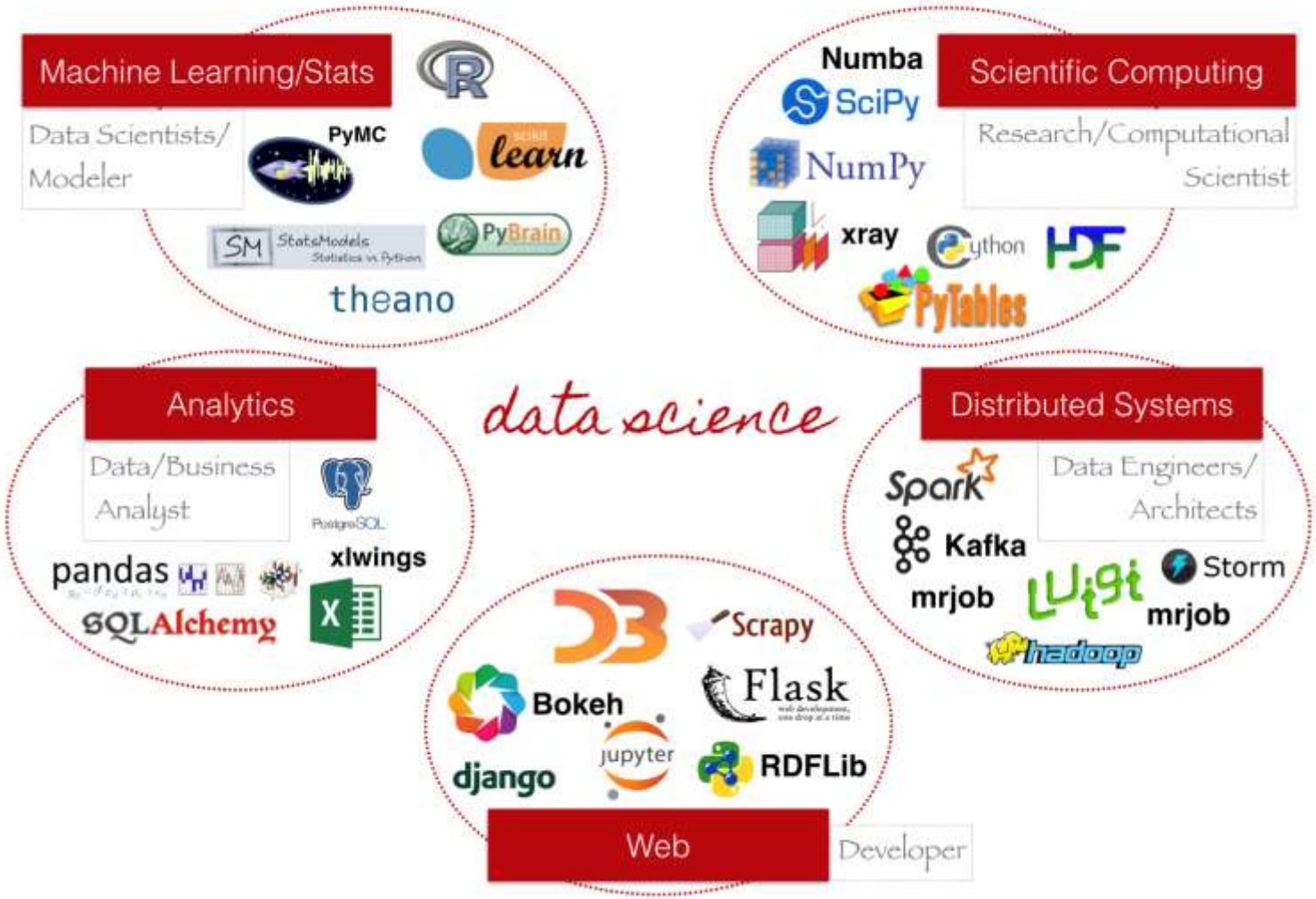
✓Pandas became an open source project in 2010

✓Now has 800 distinct contributors in developer community

Source:gouthamanbalaraman.com

Why pandas

- Big part of Data Science is Data Cleaning.
- Pandas is a power tool for data cleaning



Key Features of Pandas

- 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.

Source : tutorialpoint.com

Pandas and NumPy

- Pandas and NumPy both hold data
- Pandas has column names as well
- Makes it easier to manipulate data

Pandas NumPy Scikit-Learn workflow

- Start with CSV
- Convert to Pandas
- Slice and dice in Pandas
- Convert to NumPy array to feed to scikit-learn

- NumPy is faster than Pandas
- Both are faster than normal Python arrays

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.
- DataFrame is a container of Series, Panel is a container of DataFrame.

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

Pandas - Series

Pandas-Series

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.arrange(n)** if no index is passed.
- **dtype -** dtype is for data type. If None, data type will be inferred
- **copy -** Copy data. Default False

A series can be created using various inputs –

Array, Dict, Scalar value or constant

Create an Empty Series

```
#import the pandas library and aliasing as pd
import pandas as pd
s = pd.Series()
print (s)
```

<u>Essential difference</u> :

*The **NumPy** array has an implicitly defined integer index used to access the values,*

*The **Pandas** Series has an explicitly defined index associated with the values*

Create a Series from ndarray

- If data is an ndarray, then index passed must be of the same length. If no index is passed, then by default index will be **range(n)** ie. 0…..n-1

```
#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)
Or
data = np.array(['a','b','c','d'])
s = pd.Series(data , index=[100,101,102,'indexd'])
print (s)
```

Create a Series from dict

- A **dict** can be passed as input and if no index is specified, then the dictionary keys are taken in a sorted order to construct index.
  If **index** is passed, the values in data corresponding to the labels in the index will be pulled out

```
data = {'a' : 0., 'b' : 1., 'c' : 2.}
s = pd.Series(data)
Or
data = {'a' : 0., 'b' : 1., 'c' : 2.}
s = pd.Series(data, index=['b','c','d','a'])
print (s)
```

Create a Series from Scalar

- If data is a scalar value, an index must be provided. The value will be repeated to match the length of **index**

```
s = pd.Series(5, index=[0, 1, 2, 3])
print (s)
```

Retrieving data from series with position/label(index)

```
s = pd.Series  (  [1,2,3,4,5],  index = ['a','b','c','d','e']  )
#retrieve the first element  -  print s[0]
#retrieve the first three element -  print s[:3]
#retrieve the last three element -  print s[-3:]
#retrieve a single element - print s['a']
#retrieve multiple elements - print s[['a','c','d']]
```

If a label is not contained, an exception is raised.

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

Series as specialized dictionary

```
population_dict = {'Amritsar': 2490656,   'Jalandhar': 2193590, 'Ludhiana' : 3498739,
                   'Hoshiarpur': 1586625 , 'Bathinda': 1388525, 'Fatehgarh Sahib': 600163,
                   'Patiala': 1895686}
population = pd.Series(population_dict)
population
```

```
Amritsar           2490656
Bathinda           1388525
Fatehgarh Sahib     600163
Hoshiarpur         1586625
Jalandhar          2193590
Ludhiana           3498739
Patiala            1895686
dtype: int64
```

```
population['Ludhiana']
```

```
3498739
```

```
population['Bathinda':'Ludhiana']
```

```
Bathinda           1388525
Fatehgarh Sahib     600163
Hoshiarpur         1586625
Jalandhar          2193590
Ludhiana           3498739
dtype: int64
```

```
population[:'Ludhiana']
```

```
Amritsar           2490656
Bathinda           1388525
Fatehgarh Sahib     600163
Hoshiarpur         1586625
Jalandhar          2193590
Ludhiana           3498739
dtype: int64
```

```
population['Ludhiana' :]
```

```
Ludhiana    3498739
Patiala     1895686
dtype: int64
```

```
population[::]
```

```
Amritsar           2490656
Bathinda           1388525
Fatehgarh Sahib     600163
Hoshiarpur         1586625
Jalandhar          2193590
Ludhiana           3498739
Patiala            1895686
dtype: int64
```

```
population[::2]
```

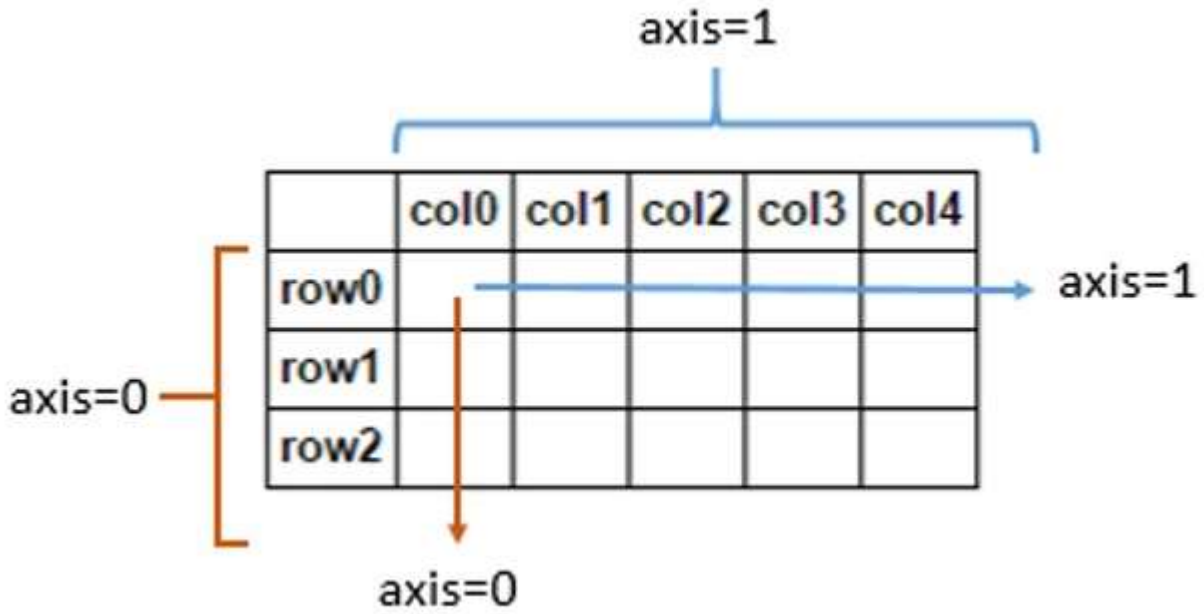
```
Amritsar           2490656
Fatehgarh Sahib     600163
Jalandhar          2193590
Patiala            1895686
dtype: int64
```

DataFrame

- DataFrames are essentially multidimensional arrays with attached row and column labels, and often with heterogeneous types and/or missing data.

Features of DataFrame

- Potentially columns are of different types
- Size – Mutable
- Labeled axes (rows and columns)
- Can Perform Arithmetic operations on rows and columns



pandas.DataFrame

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

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

pandas DataFrame can be created using various inputs like −

- Lists
- dict
- Series
- Numpy ndarrays
- Another DataFrame

- Create an Empty DataFrame

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

- Create a DataFrame from Lists

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

data = [['Amit',10],[‘Sumit',12],[‘Himmat',13]]
df = pd.DataFrame(data,columns=['Name',‘Rollno'])

data = [['Amit',1000],[‘Sumit',1200],[‘Himmat',1350]]
df = pd.DataFrame(data,columns=['Name',‘Fee'],dtype=float)
```

- Create a DataFrame from Dict of ndarrays / Lists

```
data = {'Name':['Amit', 'Anil', 'Sumit', 'Rosy'],'Age':[14,15,13,18]}
df = pd.DataFrame(data)

data = {'Name':['Amit', 'Anil', 'Sumit', 'Rosy'],'Age':[14,15,13,18]}
df = pd.DataFrame(data, index=['rank1','rank2','rank3','rank4'])
```

- Create a DataFrame from List of Dicts

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

data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]
df = pd.DataFrame(data, index=['first', 'second'])

data = [{'a': i, 'b': 2 * i} for i in range(3)]
Df = pd.DataFrame(data)
```

- From a two-dimensional NumPy array.

```
pd.DataFrame(np.random.rand(3, 2),
             columns=['foo', 'bar'],
             index=['a', 'b', 'c'])
```

- From a NumPy structured array.

```
A = np.zeros(3, dtype=[('A', 'i8'), ('B', 'f8')])
```

Create a 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)

print( df('one') )  #column selection

# adding column
df['three'] = pd.Series([10,20,30],index=['a','b','c'])
df['four']  = df['one'] + df['three']
```

Create a DataFrame from Dict of Series (contd…)

```
# deleting column
    del df['three']    or    df.pop('three')
```

Selection by Label - df.loc['b']

Selection by integer location - df.iloc[2]

Slice Row - df[2:4]

Adding a Row - df = df.append(df2)

Deletion of Row - df = df.drop(0)

DataFrame as a generalized NumPy array

```
population_dict = {'Amritsar': 2490656,   'Jalandhar': 2193590, 'Ludhiana' : 3498739,
                   'Hoshiarpur': 1586625 , 'Bathinda': 1388525, 'Fatehgarh Sahib': 600163,
                   'Patiala': 1895686}
population = pd.Series(population_dict)

area_dict = {'Hoshiarpur': 3365  , 'Bathinda': 3385 , 'Fatehgarh Sahib': 1180 ,
              'Amritsar': 2647,   'Jalandhar': 2632 , 'Ludhiana' : 3767 ,
                   'Patiala': 3218 }
area = pd.Series(area_dict)

population
area
```

```
Amritsar           2647
Bathinda           3385
Fatehgarh Sahib    1180
Hoshiarpur         3365
Jalandhar          2632
Ludhiana           3767
Patiala            3218
dtype: int64
```

```
states = pd.DataFrame({'population': population, 'area': area})
states
```

	area	population
Amritsar	2647	2490656
Bathinda	3385	1388525
Fatehgarh Sahib	1180	600163
Hoshiarpur	3365	1586625
Jalandhar	2632	2193590
Ludhiana	3767	3498739
Patiala	3218	1895686

```
states.index
```

```
Index(['Amritsar', 'Bathinda', 'Fatehgarh Sahib', 'Hoshiarpur', 'Jalandhar',
       'Ludhiana', 'Patiala'],
      dtype='object')
```

```
states.columnss
```

```
Index(['area', 'population'], dtype='object')
```

```
states.columns
```

```
Index(['area', 'population'], dtype='object')
```

```
'Amritsar' in states.index
```

```
True
```

```
states.index
```

```
Index(['Amritsar', 'Bathinda', 'Fatehgarh Sahib', 'Hoshiarpur', 'Jalandhar',
       'Ludhiana', 'Patiala'],
      dtype='object')
```

```
states.keys()
```

```
Index(['area', 'population'], dtype='object')
```

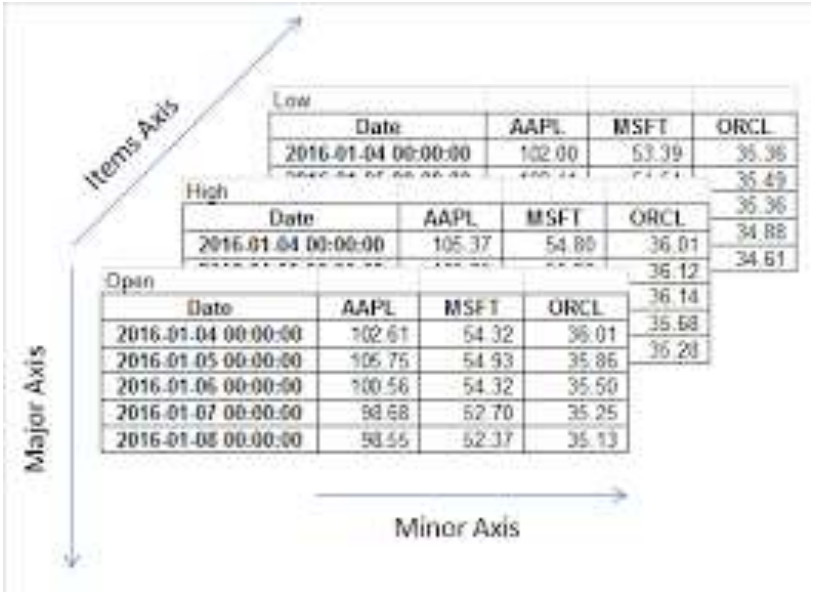
```
list(states.items())
```

```
[('area', Amritsar           2647
  Bathinda           3385
  Fatehgarh Sahib    1180
  Hoshiarpur         3365
  Jalandhar          2632
  Ludhiana           3767
  Patiala            3218
  Name: area, dtype: int64), ('population', Amritsar           2490656
  Bathinda           1388525
  Fatehgarh Sahib     600163
  Hoshiarpur         1586625
  Jalandhar          2193590
```

Panel - 3D container of data.

- pandas.Panel(data, items, major_axis, minor_axis, dtype, copy)

Parameter	Description
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**



Panel can be created – from ndarrays or dict of DataFrames

- Creating empty Panel

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

```
<class 'pandas.core.panel.Panel'>
Dimensions: 0 (items) x 0 (major_axis) x 0 (minor_axis)
Items axis: None
Major_axis axis: None
Minor_axis axis: None
```

- From 3D ndarray

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

```
<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
```

From dict of DataFrame Objects

```
data = {'Item1' : pd.DataFrame(np.random.randn(4, 3)),
        'Item2' : pd.DataFrame(np.random.randn(4, 2))}
p = pd.Panel(data)
```

- Selecting data from Panel

  P['Item1'] or p.major_xs(1) or p.minor_xs(1)

Series Basic Functionality

S.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.
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.

Series Basic Functionality

```
import pandas as pd
import numpy as np

#Create a series with 100 random numbers
s = pd.Series(np.random.randn(6))
print (s)
```

```
0    0.075533
1   -1.006220
2   -0.473522
3   -0.065085
4   -1.310497
5    1.087002
dtype: float64
```

```
print (s.axes)
```

```
[RangeIndex(start=0, stop=6, step=1)]
```

```
print(s.empty)
```

```
False
```

```
print(s.ndim)
```

```
1
```

```
print(s.size)
```

```
6
```

```
print(s.values)
```

```
[ 0.07553301 -1.00621991 -0.47352249 -0.06508496 -1.31049698  1.0870023 ]
```

```
print(s.head(2)) #returns the first n rows
```

```
0    0.075533
1   -1.006220
dtype: float64
```

```
print(s.tail(2)) #returns the last n rows
```

```
4   -1.310497
5    1.087002
dtype: float64
```

DataFrame Basic Functionality

S.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.

DataFrame Basic Functionality

S.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.

```
#Create a Dictionary of series
d = {'Name':pd.Series(['Amit','Jacky','Rohit','Hawa
                                    Singh','Sumit','Ajmal','Jivanjot']),
   'Age':pd.Series([25,26,25,23,30,29,23]),
   'comm':pd.Series([2250.50,1500,1250.50,2005,5000,3500,1850])}
df = pd.DataFrame(d)   #Create a DataFrame
print (df)
```

```
: #Create a Dictionary of series
d = {'Name':pd.Series(['Amit','Jacky','Rohit','Hawa Singh','Sumit','Ajmal','Jivanjot']),
   'Age':pd.Series([25,26,25,23,30,29,23]),
   'comm':pd.Series([2250.50,1500,1250.50,2005,5000,3500,1850])}

df = pd.DataFrame(d)   #Create a DataFrame

print (df)
```

```
   Age        Name    comm
0   25        Amit  2250.5
1   26       Jacky  1500.0
2   25       Rohit  1250.5
3   23  Hawa Singh  2005.0
4   30       Sumit  5000.0
5   29       Ajmal  3500.0
6   23    Jivanjot  1850.0
```

```
print (df)
```

```
   Age          Name    comm
0   25          Amit  2250.5
1   26         Jacky  1500.0
2   25         Rohit  1250.5
3   23   Hawa Singh   2005.0
4   30         Sumit  5000.0
5   29         Ajmal  3500.0
6   23     Jivanjot   1850.0
```

```
df.T  #Returns the transpose of the DataFrame. The rows and columns will interchange.
```

	0	1	2	3	4	5	6
Age	25	26	25	23	30	29	23
Name	Amit	Jacky	Rohit	Hawa Singh	Sumit	Ajmal	Jivanjot
comm	2250.5	1500	1250.5	2005	5000	3500	1850

```
df.axes  #Returns the list of row axis labels and column axis labels.
```

```
[RangeIndex(start=0, stop=7, step=1),
 Index(['Age', 'Name', 'comm'], dtype='object')]
```

```
df.axes  #Returns the list of row axis labels and column axis labels.
```

```
[RangeIndex(start=0, stop=7, step=1),
 Index(['Age', 'Name', 'comm'], dtype='object')]
```

```
df.dtypes #Returns the data type of each column.
```

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

```
df.empty #Returns the Boolean value saying whether the Object is empty or not;df
```

```
False
```

```
df.ndim #Returns the number of dimensions of the object. By definition, DataFrame is a 2D object.
```

```
2
```

```
df.shape #Returns a tuple representing the dimensionality of the DataFrame.
```

```
(7, 3)
```

```
df.size #Returns the number of elements in the DataFrame.
```

```
21
```

```
df.values #Returns the actual data in the DataFrame as an NDarray.
```

```
array([[25, 'Amit', 2250.5],
       [26, 'Jacky', 1500.0],
       [25, 'Rohit', 1250.5],
       [23, 'Hawa Singh', 2005.0],
       [30, 'Sumit', 5000.0],
       [29, 'Ajmal', 3500.0],
       [23, 'Jivanjot', 1850.0]], dtype=object)
```

```
df.head(2) #head() returns the first n rows (observe the index values)
```

	Age	Name	comm
0	25	Amit	2250.5
1	26	Jacky	1500.0

```
df.tail(2)  #returns the last n rows (observe the index values).
```

	Age	Name	comm
5	29	Ajmal	3500.0
6	23	Jivanjot	1850.0

DataFrame - descriptive statistics

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

```
In [22]: print(df)
```

```
   Age         Name    comm
0   25         Amit  2250.5
1   26        Jacky  1500.0
2   25        Rohit  1250.5
3   23  Hawa Singh   2005.0
4   30        Sumit  5000.0
5   29        Ajmal  3500.0
6   23     Jivanjot  1850.0
```

```
In [24]: print(df.sum())
```

```
Age                                                   181
Name    AmitJackyRohitHawa SinghSumitAjmalJivanjot
comm                                                17356
dtype: object
```

```
In [25]: print(df.sum(1)) # axis =1
```

```
0    2275.5
1    1526.0
2    1275.5
3    2028.0
4    5030.0
5    3529.0
6    1873.0
dtype: float64
```

DataFrame – summarizing data

The **describe()** function computes a summary of statistics pertaining to the DataFrame columns.

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)

```
df.describe()
```

	Age	comm
count	7.000000	7.000000
mean	25.857143	2479.428571
std	2.734262	1325.271244
min	23.000000	1250.500000
25%	24.000000	1675.000000
50%	25.000000	2005.000000
75%	27.500000	2875.250000
max	30.000000	5000.000000

```
df.describe(include=['object'])
```

	Name
count	7
unique	7
top	Sumit
freq	1

```
df.describe(include='all')
```

	Age	Name	comm
count	7.000000	7	7.000000
unique	NaN	7	NaN
top	NaN	Sumit	NaN
freq	NaN	1	NaN
mean	25.857143	NaN	2479.428571
std	2.734262	NaN	1325.271244
min	23.000000	NaN	1250.500000
25%	24.000000	NaN	1675.000000
50%	25.000000	NaN	2005.000000
75%	27.500000	NaN	2875.250000
max	30.000000	NaN	5000.000000

DataFrame – as Dictionary

```
punjabd.iloc[:2,:3]
```

	area	population	density
Amritsar	2647	2490656	940.935399
Bathinda	3385	1388525	410.199409

```
punjabd.loc[:'Hoshiarpur',:'population']
```

	area	population
Amritsar	2647	2490656
Bathinda	3385	1388525
Fatehgarh Sahib	1180	600163
Hoshiarpur	3365	1586625

```
punjabd.ix[:3,:]
```

	area	population	density
Amritsar	2647	2490656	940.935399
Bathinda	3385	1388525	410.199409
Fatehgarh Sahib	1180	600163	508.612712

```
punjabd
```

	area	population
Amritsar	2647	2490656
Bathinda	3385	1388525
Fatehgarh Sahib	1180	600163
Hoshiarpur	3365	1586625
Jalandhar	2632	2193590
Ludhiana	3767	3498739
Patiala	3218	1895686

```
punjabd['density'] = punjabd['population'] / punjabd['area']
```

```
punjabd
```

	area	population	density
Amritsar	2647	2490656	940.935399
Bathinda	3385	1388525	410.199409
Fatehgarh Sahib	1180	600163	508.612712
Hoshiarpur	3365	1586625	471.508172
Jalandhar	2632	2193590	833.430851
Ludhiana	3767	3498739	928.786568

punjabd.population is punjabd['population']
Returns - True

Numpy-style data access patterns

```
punjabd['density'] = punjabd['population'] / punjabd['area']
```

```
punjabd
```

	area	population	density
Amritsar	2647	2490656	940.935399
Bathinda	3385	1388525	410.199409
Fatehgarh Sahib	1180	600163	508.612712
Hoshiarpur	3365	1586625	471.508172
Jalandhar	2632	2193590	833.430851
Ludhiana	3767	3498739	928.786568
Patiala	3218	1895686	589.088254

```
punjabd.loc[punjabd.density > 500, ['population', 'density']]
```

	population	density
Amritsar	2490656	940.935399
Fatehgarh Sahib	600163	508.612712
Jalandhar	2193590	833.430851
Ludhiana	3498739	928.786568
Patiala	1895686	589.088254

```
punjabd.iloc[0, 2] = 90
```

```
punjabd
```

	area	population	density
Amritsar	2647	2490656	90.000000
Bathinda	3385	1388525	410.199409
Fatehgarh Sahib	1180	600163	508.612712
Hoshiarpur	3365	1586625	471.508172
Jalandhar	2632	2193590	833.430851
Ludhiana	3767	3498739	928.786568
Patiala	3218	1895686	589.088254

Index alignment in DataFrame

```
import pandas as pd
import numpy as np
rng = np.random.RandomState(42)
A = pd.DataFrame(rng.randint(0, 20, (2, 2)),
                 columns=list('AB'))
```

```
A
```

	A	B
0	6	19
1	14	10

```
B = pd.DataFrame(rng.randint(0, 10, (3, 3)),
                 columns=list('BAC'))
```

```
B
```

	B	A	C
0	7	4	6
1	9	2	6
2	7	4	3

```
A + B
```

	A	B	C
0	10.0	26.0	NaN
1	16.0	19.0	NaN
2	NaN	NaN	NaN

```
fill = A.stack().mean()
A.add(B, fill_value=fill)
```

	A	B	C
0	10.00	26.00	18.25
1	16.00	19.00	18.25
2	16.25	19.25	15.25

Python operator	Pandas method(s)
+	add()
-	sub(), subtract()
*	mul(), multiply()
/	truediv(), div(), divide()
//	floordiv()
%	mod()
**	pow()

MultiIndex

Hierarchical Indexing- MultiIndex series

```
ex = [('Amritsar', 2000), ('Amritsar', 2010),
      ('Hoshiarpur', 2000), ('Hoshiarpur', 2010),
      ('Ludhiana', 2000), ('Ludhiana', 2010)]
ulations = [2123656, 2490656,
            1234625, 1586625,
            2898739,3498739]
 = pd.Series(populations, index=index)
```

```
ritsar, 2000)      2123656
ritsar, 2010)      2490656
shiarpur, 2000)    1234625
shiarpur, 2010)    1586625
dhiana, 2000)      2898739
dhiana, 2010)      3498739
pe: int64
```

```
[('Amritsar', 2010):('Hoshiarpur', 2000)]
```

```
ritsar, 2010)      2490656
shiarpur, 2000)    1234625
pe: int64
```

```
[[i for i in pop.index if i[1] == 2010]] #all values from 2010
```

```
ritsar, 2010)      2490656
shiarpur, 2010)    1586625
dhiana, 2010)      3498739
pe: int64
```

```
index = pd.MultiIndex.from_tuples(index)
```

```
index
```

```
MultiIndex(levels=[['Amritsar', 'Hoshiarpur', 'Ludhiana'], [2000, 2010]],
           labels=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]])
```

```
pop = pop.reindex(index)
pop
```

```
Amritsar    2000    2123656
            2010    2490656
Hoshiarpur  2000    1234625
            2010    1586625
Ludhiana    2000    2898739
            2010    3498739
dtype: int64
```

```
pop[:, 2010]
```

```
Amritsar      2490656
Hoshiarpur    1586625
Ludhiana      3498739
dtype: int64
```

- Data munging is difficult

MultiIndex

```
#unstack() method will quickly convert a multiply
#indexed Series into a conventionally indexed DataFrame
pop_df = pop.unstack()
pop_df
```

	2000	2010
Amritsar	2123656	2490656
Hoshiarpur	1234625	1586625
Ludhiana	2898739	3498739

```
pop_df.stack() #opposite of unstack
```

```
Amritsar    2000    2123656
            2010    2490656
Hoshiarpur  2000    1234625
            2010    1586625
Ludhiana    2000    2898739
            2010    3498739
dtype: int64
```

Methods of MultiIndex Creation

[114]:
```
#straightforward way to construct a multiply indexed Series or DataFrame
df = pd.DataFrame(np.random.rand(4, 2),
         index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
         columns=['data1', 'data2'])
```

[115]:
```
df
```

[115]:

		data1	data2
a	1	0.946748	0.124147
	2	0.030505	0.809607
b	1	0.140180	0.860514
	2	0.488305	0.058456

[117]:
```
#dictionary with appropriate tuples as keys, Pandas will auto-
# matically recognize this and use a MultiIndex by default
data = {('Amritsar', 2000) :2123656,
        ('Amritsar', 2010): 2490656,
        ('Hoshiarpur', 2000): 1234625,
        ('Hoshiarpur', 2010): 1586625,
        ('Ludhiana', 2000) : 2898739,
        ('Ludhiana', 2010) : 3498739
 }
pd.Series(data)
```

t[117]:
```
Amritsar    2000    2123656
            2010    2490656
Hoshiarpur  2000    1234625
            2010    1586625
Ludhiana    2000    2898739
            2010    3498739
dtype: int64
```

Explicit MultiIndex constructors

```
In [118]: pd.MultiIndex.from_arrays([['a', 'a', 'b', 'b'], [1, 2, 1, 2]])

Out[118]: MultiIndex(levels=[['a', 'b'], [1, 2]],
                     labels=[[0, 0, 1, 1], [0, 1, 0, 1]])

In [119]: pd.MultiIndex.from_tuples([('a', 1), ('a', 2), ('b', 1), ('b', 2)])

Out[119]: MultiIndex(levels=[['a', 'b'], [1, 2]],
                     labels=[[0, 0, 1, 1], [0, 1, 0, 1]])

In [120]: # Cartesian product of single indices
          pd.MultiIndex.from_product([['a', 'b'], [1, 2]])

Out[120]: MultiIndex(levels=[['a', 'b'], [1, 2]],
                     labels=[[0, 0, 1, 1], [0, 1, 0, 1]])

In [121]: pd.MultiIndex(levels=[['a', 'b'], [1, 2]],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]])

Out[121]: MultiIndex(levels=[['a', 'b'], [1, 2]],
                     labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

MultiIndex level names

```
In [122]: pop
Out[122]: Amritsar    2000    2123656
                      2010    2490656
          Hoshiarpur  2000    1234625
                      2010    1586625
          Ludhiana    2000    2898739
                      2010    3498739
          dtype: int64

In [123]: pop.index.names = ['Districts of Punjab', 'year']

In [124]: pop
Out[124]: Districts of Punjab  year
          Amritsar             2000    2123656
                               2010    2490656
          Hoshiarpur           2000    1234625
                               2010    1586625
          Ludhiana             2000    2898739
                               2010    3498739
          dtype: int64
```

MultiIndex for columns

- In a DataFrame, the rows and columns are completely symmetric, and just as the rows can have multiple levels of indices, the columns can also have multiple levels

[127]:
```
# hierarchical indices and columns
index = pd.MultiIndex.from_product([[2015, 2016], [1, 2]],
                                   names=['year', 'Quater'])
columns = pd.MultiIndex.from_product([['Brijesh', 'Himmat', 'Surender'], ['Domestic', 'Commercial']],
                                   names=['Employee', 'type'])
```

[128]:
```
# mock some data
data = np.round(np.random.randn(4, 6), 1)
data[:, ::2] *= 10
data += 37
# create the DataFrame
call_data = pd.DataFrame(data, index=index, columns=columns)
call_data
```

[128]:

	Employee	Brijesh		Himmat		Surender	
	type	Domestic	Commercial	Domestic	Commercial	Domestic	Commercial
year	Quater						
2015	1	32.0	37.9	41.0	36.7	52.0	36.8
	2	19.0	37.6	34.0	37.2	45.0	37.7
2016	1	34.0	36.5	37.0	34.9	38.0	37.4
	2	29.0	38.7	34.0	38.1	44.0	34.9

[129]:
```
call_data['Himmat']
```

[129]:

	type	Domestic	Commercial
year	Quater		
2015	1	41.0	36.7
	2	34.0	37.2
2016	1	37.0	34.9
	2	34.0	38.1

Multiply indexed Series

```
In [130]: pop
Out[130]: Districts of Punjab  Year
          Amritsar             2000    2123656
                               2010    2490656
          Hoshiarpur           2000    1234625
                               2010    1586625
          Ludhiana             2000    2898739
                               2010    3498739
          dtype: int64

In [131]: type(pop)
Out[131]: pandas.core.series.Series

In [132]: pop['Amritsar']
Out[132]: Year
          2000    2123656
          2010    2490656
          dtype: int64

In [134]: pop['Ludhiana',2000]
Out[134]: 2898739
```

```
In [135]: pop[:, 2000]
Out[135]: Districts of Punjab
          Amritsar      2123656
          Hoshiarpur    1234625
          Ludhiana      2898739
          dtype: int64

In [137]: pop[pop > 2200000]
Out[137]: Districts of Punjab  Year
          Amritsar             2010    2490656
          Ludhiana             2000    2898739
                               2010    3498739
          dtype: int64

In [138]: pop[['Amritsar', 'Ludhiana']]
Out[138]: Districts of Punjab  Year
          Amritsar             2000    2123656
                               2010    2490656
          Ludhiana             2000    2898739
                               2010    3498739
          dtype: int64
```

Multiply indexed DataFrames

```
[139]: call_data
```

t[139]:

	Employee	Brijesh		Himmat		Surender	
	type	Domestic	Commercial	Domestic	Commercial	Domestic	Commercial
year	Quater						
2015	1	32.0	37.9	41.0	36.7	52.0	36.8
	2	19.0	37.6	34.0	37.2	45.0	37.7
2016	1	34.0	36.5	37.0	34.9	38.0	37.4
	2	29.0	38.7	34.0	38.1	44.0	34.9

```
[141]: call_data['Brijesh','Commercial']
```

```
t[141]: year  Quater
        2015  1         37.9
              2         37.6
        2016  1         36.5
              2         38.7
        Name: (Brijesh, Commercial), dtype: float64
```

```
[142]: call_data.iloc[:2,:2]
```

[142]:

	Employee	Brijesh	
	type	Domestic	Commercial
year	Quater		
2015	1	32.0	37.9
	2	19.0	37.6

```
[144]: call_data.loc[:,('Himmat','Domestic')]
```

```
[144]: year  Quater
       2015  1         41.0
             2         34.0
       2016  1         37.0
             2         34.0
       Name: (Himmat, Domestic), dtype: float64
```

```
[145]: health_data.loc[(:, 1), (:, 'HR')]
```

```
  File "<ipython-input-145-8e3cc151e316>", line 1
    health_data.loc[(:, 1), (:, 'HR')]
                    ^
SyntaxError: invalid syntax
```

Multiply indexed DataFrames - slice()

```
[142]: call_data.iloc[:2,:2]
```

[142]:

	Employee	Brijesh
	type	Domestic	Commercial
year	Quater		
2015	1	32.0	37.9
	2	19.0	37.6

```
[144]: call_data.loc[:,('Himmat','Domestic')]
```

```
[144]: year  Quater
       2015  1         41.0
             2         34.0
       2016  1         37.0
             2         34.0
       Name: (Himmat, Domestic), dtype: float64
```

```
[145]: health_data.loc[(:, 1), (:, 'HR')]
```

```
  File "<ipython-input-145-8e3cc151e316>", line 1
    health_data.loc[(:, 1), (:, 'HR')]
                    ^
SyntaxError: invalid syntax
```

Pandas -Iteration

- basic iteration (for **i** in object) produces –
  - **Series** – values
  - **DataFrame** – column labels
  - **Panel** – item labels
- 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 named tuples

```
In [152]: type(punjabd)
Out[152]: pandas.core.frame.DataFrame

In [154]: for key,value in punjabd.iteritems():
              print (key,value)
```

```
area Amritsar           2647
Bathinda            3385
Fatehgarh Sahib     1180
Hoshiarpur          3365
Jalandhar           2632
Ludhiana            3767
Patiala             3218
Name: area, dtype: int64
population Amritsar           2490656
Bathinda            1388525
Fatehgarh Sahib      600163
Hoshiarpur          1586625
Jalandhar           2193590
Ludhiana            3498739
Patiala             1895686
Name: population, dtype: int64
density Amritsar           940.935399
Bathinda            410.199409
Fatehgarh Sahib     508.612712
Hoshiarpur          471.508172
Jalandhar           833.430851
Ludhiana            928.786568
Patiala             589.088254
Name: density, dtype: float64
```

Pandas -Iteration

- iterrows() returns the iterator yielding each series containing the data in each row.
- itertuples() method will return an iterator yi each row in the DataFrame.

```
In [158]: for row_index,row in punjabd.iterrows():
              print (row_index,row)
```

```
Amritsar area          2.647000e+03
population    2.490656e+06
density       9.409354e+02
Name: Amritsar, dtype: float64
Bathinda area          3.385000e+03
population    1.388525e+06
density       4.101994e+02
Name: Bathinda, dtype: float64
Fatehgarh Sahib area            1180.000000
population    600163.000000
density          508.612712
Name: Fatehgarh Sahib, dtype: float64
Hoshiarpur area          3.365000e+03
population    1.586625e+06
```

```
[159]: for row in punjabd.itertuples():
           print (row)
```

```
Pandas(Index='Amritsar', area=2647, population=2490656, density=940.93539856441259)
Pandas(Index='Bathinda', area=3385, population=1388525, density=410.19940915805023)
Pandas(Index='Fatehgarh Sahib', area=1180, population=600163, density=508.61271186440678)
Pandas(Index='Hoshiarpur', area=3365, population=1586625, density=471.50817236255574)
Pandas(Index='Jalandhar', area=2632, population=2193590, density=833.43085106382978)
Pandas(Index='Ludhiana', area=3767, population=3498739, density=928.78656756039288)
Pandas(Index='Patiala', area=3218, population=1895686, density=589.08825357364822)
```

Sorting

- By label
- By actual value

```
[164]: unsorted_df=pd.DataFrame(np.random.randn(5,2),
                                index=[1,4,2,0,3],columns=['col2','col1'])
       print(unsorted_df)
```

```
       col2      col1
1  0.030141  0.087078
4  0.075662 -1.267849
2 -1.663053 -0.991323
0  0.780041  1.545267
3 -0.220564 -0.420996
```

```
[165]: sorted_df=unsorted_df.sort_index()
       sorted_df
```

t[165]:

	col2	col1
0	0.780041	1.545267
1	0.030141	0.087078
2	-1.663053	-0.991323
3	-0.220564	-0.420996
4	0.075662	-1.267849

```
2]: sorted_df1 = unsorted_df1.sort_values(by='col1')
    sorted_df1
```

2]:

	col2	col1
4	10.075662	8.732151
2	8.336947	9.008677
3	9.779436	9.579004
1	10.030141	10.087078
0	10.780041	11.545267

```
In [169]: unsorted_df1
```

Out[169]:

	col2	col1
1	10.030141	10.087078
4	10.075662	8.732151
2	8.336947	9.008677
0	10.780041	11.545267
3	9.779436	9.579004

```
In [170]: #sort on column name
          sorted_df1=unsorted_df1.sort_index(axis=1)
```

```
In [171]: sorted_df1
```

Out[171]:

	col1	col2
1	10.087078	10.030141
4	8.732151	10.075662
2	9.008677	8.336947
0	11.545267	10.780041
3	9.579004	9.779436

Sorting Algorithm

- **sort_values()** provides a provision to choose the algorithm from mergesort, heapsort and quicksort.
- Mergesort is the only stable algorithm.

```
[176]: sorted_df1 = unsorted_df1.sort_values(by='col1' ,kind='mergesort')

[174]: sorted_df1
```

[174]:

	col2	col1
4	10.075662	8.732151
2	8.336947	9.008677
3	9.779436	9.579004
1	10.030141	10.087078
0	10.780041	11.545267

String Methods

S.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.
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.
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