**Pandas DataFrame overview**

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

**DataFrame**

In [ ]:

In [14]:

**import** **pandas** **as** **pd**

apples = pd.Series([3,2,0,1])

oranges = pd.Series([3,4,7,8])

*#print(apples, oranges)*

data = {"apples": apples, "oranges": oranges}

fruits\_df = pd.DataFrame(data)

print(fruits\_df)

apples oranges

0 3 3

1 2 4

2 0 7

3 1 8

**keep in mind, Indexes**

In [2]:

**import** **pandas** **as** **pd**

apples = pd.Series([3,2,0,1], ["a", "b", "c", "d"] )

oranges = pd.Series([3,2,0,1], index = ["mon", "tue", "wed", "thr"])

*#print(apples, oranges)*

data = {"apples": apples, "oranges": oranges}

fruits\_df = pd.DataFrame(data)

print(fruits\_df)

*# index not matched*

apples oranges

a 3.0 NaN

b 2.0 NaN

c 0.0 NaN

d 1.0 NaN

mon NaN 3.0

thr NaN 1.0

tue NaN 2.0

wed NaN 0.0

In [4]:

**import** **pandas** **as** **pd**

apples = pd.Series([3,2,0,1] , index = ["mon", "tue", "wed", "thr"] )

oranges = pd.Series([3,2,0,1], index = ["mon", "tue", "wed", "thr"])

*#print(apples,"\n", oranges)*

data = {"apples": apples, "oranges": oranges}

fruits\_df = pd.DataFrame(data)

print(fruits\_df)

apples oranges

mon 3 3

tue 2 2

wed 0 0

thr 1 1

In [5]:

state = ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada']

data = {'state': state ,

'year' : [2000, 2001, 2002, 2001, 2002, 2003],

'pop' : [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}

state\_pop\_df = pd.DataFrame(data

, index = ['1st', '2nd', '3rd',

'4th', '5th','6th'] )

print(state\_pop\_df)

state year pop

1st Ohio 2000 1.5

2nd Ohio 2001 1.7

3rd Ohio 2002 3.6

4th Nevada 2001 2.4

5th Nevada 2002 2.9

6th Nevada 2003 3.2

In [12]:

*#state\_pop\_df.head()*

state\_pop\_df =pd.DataFrame(data, columns=['year', 'state', 'pop'])

print(state\_pop\_df)

x =10

x

year state pop

0 2000 Ohio 1.5

1 2001 Ohio 1.7

2 2002 Ohio 3.6

3 2001 Nevada 2.4

4 2002 Nevada 2.9

5 2003 Nevada 3.2

Out[12]:

10

In [19]:

frame2 = pd.DataFrame(data,

columns=['year', 'state', 'pop', 'debt'],

index=['one', 'two', 'three', 'four','five', 'six'])

frame2.head()

Out[19]:

|  | **year** | **state** | **pop** | **debt** |
| --- | --- | --- | --- | --- |
| **one** | 2000 | Ohio | 1.5 | NaN |
| **two** | 2001 | Ohio | 1.7 | NaN |
| **three** | 2002 | Ohio | 3.6 | NaN |
| **four** | 2001 | Nevada | 2.4 | NaN |
| **five** | 2002 | Nevada | 2.9 | NaN |

In [20]:

print(frame2.columns)

print( frame2.index )

Index(['year', 'state', 'pop', 'debt'], dtype='object')

Index(['one', 'two', 'three', 'four', 'five', 'six'], dtype='object')

**A column in a DataFrame can be retrieved as a Series either by dict-like notation or by attribute:**

In [21]:

*#print(data)*

*#data['state']*

*# this is dictinary like notation to access or extract*

*# dataframe column data*

frame2["state"]

Out[21]:

one Ohio

two Ohio

three Ohio

four Nevada

five Nevada

six Nevada

Name: state, dtype: object

In [22]:

*# there frame2 is another method , attribute style of accessing*

*# dataframe data.*

frame2.state

Out[22]:

one Ohio

two Ohio

three Ohio

four Nevada

five Nevada

six Nevada

Name: state, dtype: object

In [23]:

print(frame2.loc['two'] )

state\_pop\_df.head()

frame2.head()

year 2001

state Ohio

pop 1.7

debt NaN

Name: two, dtype: object

Out[23]:

|  | **year** | **state** | **pop** | **debt** |
| --- | --- | --- | --- | --- |
| **one** | 2000 | Ohio | 1.5 | NaN |
| **two** | 2001 | Ohio | 1.7 | NaN |
| **three** | 2002 | Ohio | 3.6 | NaN |
| **four** | 2001 | Nevada | 2.4 | NaN |
| **five** | 2002 | Nevada | 2.9 | NaN |

**Columns can be modified by assignment. For example, the empty 'debt' column could be assigned a scalar value or an array of values:**

In [24]:

frame2['debt'] = 16.5

frame2.head()

Out[24]:

|  | **year** | **state** | **pop** | **debt** |
| --- | --- | --- | --- | --- |
| **one** | 2000 | Ohio | 1.5 | 16.5 |
| **two** | 2001 | Ohio | 1.7 | 16.5 |
| **three** | 2002 | Ohio | 3.6 | 16.5 |
| **four** | 2001 | Nevada | 2.4 | 16.5 |
| **five** | 2002 | Nevada | 2.9 | 16.5 |

In [13]:

*# important cell for codes review*

**import** **numpy** **as** **np**

**import** **pandas** **as** **pd**

state = ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada']

data = {'state': state ,

'year' : [2000, 2001, 2002, 2001, 2002, 2003],

'pop' : [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}

frame2 = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'],

index=['one', 'two', 'three', 'four','five', 'six'])

ln = len(frame2) *# finding now of rows in dataframe*

*#print(frame2)*

rng = np.arange(ln)

print(rng)

frame2['debt'] = rng

frame2

[0 1 2 3 4 5]

Out[13]:

|  | **year** | **state** | **pop** | **debt** |
| --- | --- | --- | --- | --- |
| **one** | 2000 | Ohio | 1.5 | 0 |
| **two** | 2001 | Ohio | 1.7 | 1 |
| **three** | 2002 | Ohio | 3.6 | 2 |
| **four** | 2001 | Nevada | 2.4 | 3 |
| **five** | 2002 | Nevada | 2.9 | 4 |
| **six** | 2003 | Nevada | 3.2 | 5 |

**point to remember , inserting new values, needs to be match in length of dataFrame (number of elements)**

In [18]:

val = pd.Series([-1.2, -1.5, -1.7, -1.2, -1.5, -1.7, 2.2],

index=['one', 'two', 'three', 'four', 'five',

'six', 'seven'])

frame2['debt'] = val

frame2

Out[18]:

|  | **year** | **state** | **pop** | **debt** |
| --- | --- | --- | --- | --- |
| **one** | 2000 | Ohio | 1.5 | -1.2 |
| **two** | 2001 | Ohio | 1.7 | -1.5 |
| **three** | 2002 | Ohio | 3.6 | -1.7 |
| **four** | 2001 | Nevada | 2.4 | -1.2 |
| **five** | 2002 | Nevada | 2.9 | -1.5 |
| **six** | 2003 | Nevada | 3.2 | -1.7 |

In [11]:

val = pd.Series([-1.2, -1.5, -1.7, 2.6], index=['two', 'four', 'five', 'six'])

frame2['debt'] = val

frame2

Out[11]:

|  | **year** | **state** | **pop** | **debt** |
| --- | --- | --- | --- | --- |
| **one** | 2000 | Ohio | 1.5 | NaN |
| **two** | 2001 | Ohio | 1.7 | -1.2 |
| **three** | 2002 | Ohio | 3.6 | NaN |
| **four** | 2001 | Nevada | 2.4 | -1.5 |
| **five** | 2002 | Nevada | 2.9 | -1.7 |
| **six** | 2003 | Nevada | 3.2 | 2.6 |

**remmeber! index length must be equal to values ( or , do not provide index)**

In [28]:

*# Mind Teaser !!*

*#Assigning a column that already exist will \_\_\_\_\_\_\_\_ and*

*# assigning a column that does not exist will \_\_\_\_\_\_\_\_*

In [25]:

*# how to reindex and usie of ffill value of parameter method*

**import** **pandas** **as** **pd**

obj3 = pd.Series(['blue', 'purple', 'yellow'], index=[0, 3, 6])

print( obj3 )

*# might create a new rows*

*#obj3 =obj3.reindex(range(9) )*

*#print(obj3)*

obj3 =obj3.reindex(range(9), method='ffill' )

print(obj3)

*#obj3 =obj3.reindex(range(2,11), method="ffill")*

*#obj3*

0 blue

3 purple

6 yellow

dtype: object

0 blue

1 blue

2 blue

3 purple

4 purple

5 purple

6 yellow

7 yellow

8 yellow

dtype: object

In [32]:

**import** **numpy** **as** **np**

**import** **pandas** **as** **pd**

states = pd.DataFrame(np.arange(9).reshape((3, 3) ),

index=['a', 'c', 'd'], columns=['Ohio', 'Texas', 'California'])

print(states)

*# for your own working, run following statement without ffill*

states = states.reindex(['a', 'b', 'c', 'd'],method = 'ffill')

print(states)

Ohio Texas California

a 0 1 2

c 3 4 5

d 6 7 8

Ohio Texas California

a 0 1 2

b 0 1 2

c 3 4 5

d 6 7 8

In [84]:

*# column name changing using reindex method*

states = pd.DataFrame(np.arange(9).reshape((3, 3) ),

index=['a', 'c', 'd'], columns=['Ohio', 'Texas', 'California'])

print(states)

states\_name = ['Texas', 'Utah', 'California','Ohio']

*# can we use ffill parameter in column reindex mode ?*

states =states.reindex(columns=states\_name)

print(states)

Ohio Texas California

a 0 1 2

c 3 4 5

d 6 7 8

Texas Utah California Ohio

a 1 NaN 2 0

c 4 NaN 5 3

d 7 NaN 8 6

**Deleting data (row or column from dataframe)**

In [11]:

**import** **numpy** **as** **np**

**import** **pandas** **as** **pd**

data\_df = pd.DataFrame(np.arange(16).reshape((4, 4)),

index=['Ohio', 'Colorado', 'Utah', 'New York'],

columns=['one', 'two', 'three', 'four'])

print(data\_df)

one two three four

Ohio 0 1 2 3

Colorado 4 5 6 7

Utah 8 9 10 11

New York 12 13 14 15

In [35]:

**import** **numpy** **as** **np**

**import** **pandas** **as** **pd**

data\_df = pd.DataFrame(np.arange(16).reshape((4, 4)),

index=['Ohio', 'Colorado', 'Utah', 'New York'],

columns=['one', 'two', 'three', 'four'])

print(data\_df, "**\n**")

*#You can drop values from the columns by passing axis=1*

*#or axis='columns' :*

data\_df = data\_df.drop('two', axis=1)

*#data\_df.drop('two', axis=1, inplace = True)*

print(data\_df)

*# if i want to remove a row for example 'Colorado',*

*# how can we do that*

one two three four

Ohio 0 1 2 3

Colorado 4 5 6 7

Utah 8 9 10 11

New York 12 13 14 15

one three four

Ohio 0 2 3

Colorado 4 6 7

Utah 8 10 11

New York 12 14 15

**Indexing, Selection, and Filtering**

In [34]:

**import** **numpy** **as** **np**

**import** **pandas** **as** **pd**

data\_df = pd.DataFrame(np.arange(16).reshape((4, 4)),

index=['Ohio', 'Colorado', 'Utah', 'New York'],

columns=['one', 'two', 'three', 'four'])

print(data\_df, "**\n**")

df2 = data\_df[ ["one", "three"] ]

*#print(df2)*

*#print( data\_df[2:] ) # same like numpy*

*#print(data\_df["one"] )# dicitonary like style of accessing data*

*#print(data\_df.one[2:]) # filter on both row and column*

*# Conditonal Selection*

*#print ( data\_df.three[data\_df['three'] > 5] )*

*#print(data\_df, "\n")*

*#f2 = data\_df["three"] > 5 # boolean dataframe*

*#print(df2)*

print( data\_df[ data\_df["three"] > 5 ] )

one two three four

Ohio 0 1 2 3

Colorado 4 5 6 7

Utah 8 9 10 11

New York 12 13 14 15

one two three four

Colorado 4 5 6 7

Utah 8 9 10 11

New York 12 13 14 15

**Selection with loc and iloc**

In [38]:

**import** **numpy** **as** **np**

**import** **pandas** **as** **pd**

data\_df = pd.DataFrame(np.arange(16).reshape((4, 4)),

index=['Ohio', 'Colorado', 'Utah', 'New York'],

columns=['one', 'two', 'three', 'four'])

print(data\_df, "**\n**")

*# in the loc method specify row label first*

*# then specify column names*

*# remember! mutiple column names require array notation*

*#print( data\_df.loc[['Colorado','Ohio'], ['two', 'three'] ] )*

*#print("\n")*

print( data\_df.iloc[2:, [3, 0, 1] ] ) *# using number instead of labels*

*#print( data\_df.iloc[:])*

*#print ( data\_df.iloc[ :3 , :3 ] )*

*#print( data\_df.iloc[:, :3])*

one two three four

Ohio 0 1 2 3

Colorado 4 5 6 7

Utah 8 9 10 11

New York 12 13 14 15

four one two

Utah 11 8 9

New York 15 12 13

**Arithmetic and Data Alignment**

In [91]:

*#print(list('bcd'))*

df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)),

columns=list('bcd'),

index=['Ohio', 'Texas', 'Colorado'])

df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)),

columns=list('bde'),

index=['Utah', 'Ohio', 'Texas', 'Oregon'])

print(df1)

print(df2)

print()

*# applying plus operation between two data frames*

df3 = df1 + df2

print(df3)

*# your work is to fill all Nan values of this df3 with a number,*

*# choice of number is yours*

b c d

Ohio 0.0 1.0 2.0

Texas 3.0 4.0 5.0

Colorado 6.0 7.0 8.0

b d e

Utah 0.0 1.0 2.0

Ohio 3.0 4.0 5.0

Texas 6.0 7.0 8.0

Oregon 9.0 10.0 11.0

b c d e

Colorado NaN NaN NaN NaN

Ohio 3.0 NaN 6.0 NaN

Oregon NaN NaN NaN NaN

Texas 9.0 NaN 12.0 NaN

Utah NaN NaN NaN NaN

**Arithmetic methods with fill values**

In [73]:

df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)),

columns=list('abcd'))

df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)),

columns=list('abcde'))

print(df1)

*#print(df2)*

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

print(df2)

df3 = df1 + df2

print()

print("direct + operation without fill\_value")

*#print(df3)*

print("--------")

print()

*# We can use add method for filling NaN cells with a value*

*# Nan will be replaced by 0 and then addition operation will apply*

print("addition using a method with replacing Nan with 0")

df3 = df1.add(df2, fill\_value=0)

print(df3)

a b c d

0 0.0 1.0 2.0 3.0

1 4.0 5.0 6.0 7.0

2 8.0 9.0 10.0 11.0

a b c d e

0 0.0 1.0 2.0 3.0 4.0

1 5.0 NaN 7.0 8.0 9.0

2 10.0 11.0 12.0 13.0 14.0

3 15.0 16.0 17.0 18.0 19.0

direct + operation without fill\_value

--------

addition using a method with replacing Nan with 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

**Operations between DataFrame and Series**

In [23]:

**import** **numpy** **as** **np**

**import** **pandas** **as** **pd**

frame = pd.DataFrame(np.arange(12.).reshape((4, 3)),

columns=list('bde'),

index=['Utah', 'Ohio', 'Texas', 'Oregon'])

*# iloc syntax*

*# iloc[start\_row\_pos:end\_row\_pos, start\_column\_pos: end\_col\_positin ]*

series = frame.iloc[0]

print(frame)

*#print(series)*

*#print(series.values)*

*#print(series.index)*

*#print(frame - series)*

*# step 1: run above commands after uncomment*

*# step 2: comment above prints except the line contains print(frame)*

*# write following lines*

*# and run the cell again*

print("---- using new series")

series = pd.Series([1,2,3], index =list('bde') )

print(series.index, series.values)

print(frame - series)

*# dateframe column names will be match with series index*

*# because its rows wise broadcasting operation*

b d e

Utah 0.0 1.0 2.0

Ohio 3.0 4.0 5.0

Texas 6.0 7.0 8.0

Oregon 9.0 10.0 11.0

---- using new series

Index(['b', 'd', 'e'], dtype='object') [1 2 3]

b d e

Utah -1.0 -1.0 -1.0

Ohio 2.0 2.0 2.0

Texas 5.0 5.0 5.0

Oregon 8.0 8.0 8.0

**Function Application and Mapping**

In [48]:

**import** **numpy** **as** **np**

**import** **pandas** **as** **pd**

frame = pd.DataFrame(np.random.randn(4, 3),

columns=list('bde'),

index=['Utah', 'Ohio', 'Texas', 'Oregon'])

print(frame)

*#print(np.abs(frame))*

*#print(frame["d"]. min())*

*#print(frame["d"].max())*

*#print(frame["d"].max() - frame["d"]. min())*

f = **lambda** x: x.max() - x.min()

df = frame.apply(f)

print(df, type(df))

*#df = frame.apply(f, axis=1)*

*#print(df)*

*#def min\_max(x):*

*# return pd.Series( [x.max() - x.min() ], index=['min-max'])*

*#df = frame.apply(min\_max)*

*#print(df, type(df))*

b d e

Utah -0.441234 -1.156622 -2.867987

Ohio 1.045428 -0.103549 0.916670

Texas 1.636705 1.113381 -1.008535

Oregon 0.326248 1.374434 -0.205099

b 2.077939

d 2.531056

e 3.784657

dtype: float64 <class 'pandas.core.series.Series'>

In [49]:

*# Sorting and Ranking*

**import** **numpy** **as** **np**

**import** **pandas** **as** **pd**

frame = pd.DataFrame(np.arange(8).reshape((2, 4)),

index=['three', 'one'],

columns=['d', 'a', 'b', 'c'])

print(frame)

print()

print(frame.sort\_index(axis=1, ascending=**True**))

print()

print( frame.sort\_index())

*# defaults in sort: axis = 0, ascending = True*

d a b c

three 0 1 2 3

one 4 5 6 7

a b c d

three 1 2 3 0

one 5 6 7 4

d a b c

one 4 5 6 7

three 0 1 2 3

In [54]:

*# sort by values*

print( frame.sort\_values(by='b') )

print(frame.rank(ascending=**False**, method='max'))

print(frame.rank(ascending=**True**, method='min'))

print( frame.rank(axis='columns'))

*#check details from the book*

*'''*

*'average' Default: assign the average rank to each entry in the equal group*

*'min'*

*'max'*

*'first'*

*'dense'*

*Use the minimum rank for the whole group*

*Use the maximum rank for the whole group*

*Assign ranks in the order the values appear in the data*

*Like method='min' , but ranks always increase by 1 in between groups rather than the number of equal*

*elements in a group*

*'''*

d a b c

three 0 1 2 3

one 4 5 6 7

d a b c

one 4 5 6 7

three 0 1 2 3

In [55]:

*#Summarizing and Computing Descriptive Statistics*

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'])

print(df)

print()

print( df.sum())

print()

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

*# note: use of skipna*

one two

a 1.40 NaN

b 7.10 -4.5

c NaN NaN

d 0.75 -1.3

one 9.25

two -5.80

dtype: float64

a 1.40

b 2.60

c 0.00

d -0.55

dtype: float64

In [27]:

print(df)

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

print()

print(x)

one two

a 1.40 NaN

b 7.10 -4.5

c NaN NaN

d 0.75 -1.3

a NaN

b 1.300

c NaN

d -0.275

dtype: float64

In [57]:

*# unique values*

df = pd.DataFrame([

[1.4,1.4, 1.5, np.nan], [7.1, -4.5, 1.5, 1.4],

[1.4, np.nan, 0.5, np.nan], [0.75, -1.3, 1.3, np.nan]

], index=['a', 'b', 'c', 'd'], columns=['one', 'two', 'three', 'four'])

print(df)

print()

print(df['one'].unique(), df['two'].unique() )

df['one'].value\_counts()

*#end of chapter 1*

one two three four

a 1.40 1.4 1.5 NaN

b 7.10 -4.5 1.5 1.4

c 1.40 NaN 0.5 NaN

d 0.75 -1.3 1.3 NaN

[1.4 7.1 0.75] [ 1.4 -4.5 nan -1.3]

Out[57]:

1.40 2

0.75 1

7.10 1

Name: one, dtype: int64