Hi guys this is Akshansh (+91 8384891269, akshanshofficial@gmail.com (mailto:akshanshofficial@gmail.com)) again. First of all, thank you for staying in touch with me for this long. In addition of this, I would like to thank you for overwhelming response and support so far. We have already discussed Python Basics from lecture 1 to lecture 15, then we got to know the numerical analysis by NumPy in lecture 16. Now we are moving on to something very crucial and important in terms of knowledge and personal gain. This is something that's is kind of prerequisite for the Jobs as well.

The pandas library - An Introduction

With this lecture note, you can finally get into the heart of datascience. This fantastic python library is a perfect tool for anyone who wants to practie data analyis using Python as a programming language.

There are mainly two data structure called Series and DataFrame. I urge you to practice this lecture until you get familiar with these two. These are too imortant as oxygen.

Pandas : The Python Data Analysis Library (skip if you don't want to read this topic but general information is good, atleast you should know what you are dealing with)

Pandas is an open source library in Python for highly specialized data analysis. It is being used to study and analyze data sets for statistical purpose of analysis and decision making.

This was designed by Wes McKinney in 2008, later in 2012 Sien Chang, his colleague, was added to the development. Together they founded one of the most used libbraries in the Python Community.

This Python package is designed on the basis of the NumPy library. This choice, I can say, was critical to the success and the rapid spread of pandas. This choice made pandas compatible with most of the other modules.

Installation Guide

In windows with Anaconda

conda install pandas

In linux

On debian and Ubuntu distributions-

sudo apt-get install python-pandas

on OpenSuse and Fedora

zypper in python-pandas

Getting started with Pandas

Remember pythonistas, i told you that pandas is based on numpy, so we have to import pandas along side with numpy. I am importing both of libraries here. np for NumPy and pd for Pandas is general conventional method in python community and I am going with same.

In [1]:

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

see there's no erroe, that means both of libraries have been succesfully called.

#INTRODUCTION TO PANDAS DATA STRUCTURES

there are mainly two primary data strutures in pandas - Series and DataFrame

in series, as you will see, this is like one dimentional data and DataFrame is more complex and it is designed to contain cases with several dimensions.

"THE SERIES"

it represents one dimentional data structures similarlyto an array but with some additional features. Let's me declare a series for you. This is how it looks like.

```
|index|value|
|  0  | 12  |
|  1  | -4  |
|  2  |  7  |
|  3  |  9  |
```

The above structure represents a series object in python.

# declaring a Series

simply call the Series() constructure passing an argument of array that contains the values

In [2]:

```
s=pd.Series([12,-4,7,9])
```

let's print our series

In [3]:

```
s
```

Out[3]:

```
0    12
1    -4
2     7
3     9
dtype: int64
```

first col is index and other col is value.

(TBR) Means to be remember, put this under your pillow -

(TBR) - pass an index array if you want to change the index like this

In [4]:

```
s=pd.Series([12,-4,7,9], index=['a','b','c','d'])
```

In [5]:

```
s
```

Out[5]:

```
a    12
b    -4
c     7
d     9
dtype: int64
```

see index has been changed.

this is how you can access values and index of Series

In [6]:

```
s.values
```

Out[6]:

```
array([12, -4,  7,  9])
```

In [7]:

```
s.index
```

Out[7]:

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

see values are stored in one dimentional array. This is what i meant when I said, pass an argument as an array.

Selecting the internal elelments

In [8]:

```
s[2] #prints index 2
```

Out[8]:

```
7
```

In [9]:

```
s['b'] #means print what is at index 'b'
```

Out[9]:

```
-4
```

Series is like one dimentional array so you can acess multiple items as you used to do in array in lecture 16 Numpy

In [10]:

```
s[0:2] #prints index 0 to 1 (2-1)
```

Out[10]:

```
a    12
b    -4
dtype: int64
```

In [11]:

```
s[['b','d']] #this is how you can print multiple index values
```

Out[11]:

```
b   -4
d    9
dtype: int64
```

assigning values to the Elements

In [12]:

```
s[1]=0 # change value to 0 at index 1
```

In [13]:

```
s
```

Out[13]:

```
a    12
b     0
c     7
d     9
dtype: int64
```

see index1 that is 'b' has been changed to 0

In [14]:

```
s['b']=1  #index'b' changed to 1
```

In [15]:

```
s
```

Out[15]:

```
a    12
b     1
c     7
d     9
dtype: int64
```

DEFINING SERIES FROM NUMPY ARRAYS AND OTHER SERIES

I told you that Series takes an argument of one dimentional array. So instead of passing whole array, why don't we create an array first and then pass is into the Series.

In [16]:

```
arr=np.arange(5)
```

In [17]:

```
arr
```

Out[17]:

```
array([0, 1, 2, 3, 4])
```

In [18]:

```
s3=pd.Series(arr)
```

In [19]:

```
s3
```

Out[19]:

```
0    0
1    1
2    2
3    3
4    4
dtype: int64
```

see whole array was feeded to Series and hence array was converted to Series as well. You can pass a Series in another Series too, because Series will take it as one dimentional array. (thank me later, ring me if you want to transfer money to my account for this awesome tip)

In [20]:

```
s
```

Out[20]:

```
a    12
b     1
c     7
d     9
dtype: int64
```

In [21]:

```
s4=pd.Series(s)
```

In [22]:

```
s4
```

Out[22]:

```
a    12
b     1
c     7
d     9
dtype: int64
```

(TBR) - CHANGING AN ELEMENT IN ARRAY WILL CHANGE IT IN SERIES TOO THAT'S GENUINE

# FILTERING VALUES FROM A SERIES

Thanks to the choice of NumPy library as the base for the development of the pandas libraries as a result, for its data structures, many operations applicable to NumPy arrays are extended to Series. One of these is the filteration of the values, it is just like we used to do with arrays.

In [23]:

```
s
```

Out[23]:

```
a    12
b     1
c     7
d     9
dtype: int64
```

In [24]:

```
s>8  #prints boolean
```

Out[24]:

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

In [25]:

```
s[s>8]  #print result
```

Out[25]:

```
a    12
d     9
dtype: int64
```

OPERATIONS AND MATHEMATICAL FUNCTIONS

Other operations such as (+,-,*,/) or mathematical functions that are applicable to NumPy array can be extended to Series as well.

In [26]:

```
s  #prints s series we created before
```

Out[26]:

```
a    12
b     1
c     7
d     9
dtype: int64
```

In [27]:

```
s/2  #divide each value by 2
```

Out[27]:

```
a    6.0
b    0.5
c    3.5
d    4.5
dtype: float64
```

you can use mathematical NumPy functions too on series

In [28]:

```
np.log(s)  #taking logarithm of each value of Series
```

Out[28]:

```
a    2.484907
b    0.000000
c    1.945910
d    2.197225
dtype: float64
```

Evaluating values

most of the time, your series will have duplicate values too. You may need to have information on what are the sample contianed, how many duplicates arre there or whether a value is present or not in series.

Let's create a Series in of duplicates

In [29]:

```
serd=pd.Series([1,0,2,1,2,3], index=['white','white','blue','green','green','yellow
```

In [30]:

```
serd
```

Out[30]:

```
white     1
white     0
blue      2
green     1
green     2
yellow    3
dtype: int64
```

(TBR)- unique() will remove duplicates

In [31]:

```
serd.unique()
```

Out[31]:

```
array([1, 0, 2, 3])
```

value_counts() function, will return the unique value and will tell you how many time it is in Series

In [32]:

```
serd.value_counts()
```

Out[32]:

```
2    2
1    2
3    1
0    1
dtype: int64
```

isin() function

you pass a list of input in list, and if it is there it will show you True otherwise False. Let's check where is 0 and 3 in Series serd (Series of Duplicate)

In [33]:

```
serd.isin([0,3])
```

Out[33]:

```
white     False
white      True
blue      False
green     False
green     False
yellow     True
dtype: bool
```

it just informed us True False, but we don't know where is exactly 0 and 3. So this is how we use it to know exact index

In [34]:

```
serd[serd.isin([0,3])]
```

Out[34]:

```
white     0
yellow    3
dtype: int64
```

this has done the needful, right. This is one of the most important feature that will help you alot in finding values in Series.

# NaN (not a number) values

When values is not a number it represents NaN values. Generally, these NaN values are a problem and must be managed in some way, specially during data analysis. Theses values gives troubles whenever you are persforming actions on whole dataset. This is also called missing data too.

despite their problematic naturem however, pandas allows to explicitly define and add this value in data structure, such as Series. Within the array containing the vlue you enter np.Nan wherever you want to define a missing value.

(TBR) - while declaring an array to feed in to Series, if you want to pass a missing value, write np.NaN

In [35]:

```
s2=pd.Series([5,-3,np.NaN,14])
```

In [36]:

```
s2 #prints s2
```

Out[36]:

```
0     5.0
1    -3.0
2     NaN
3    14.0
dtype: float64
```

see np.NaN has been printed as NaN at index 2

if you want to check where is NaN or where is not NaN, use isnull() and notnull() functions

In [37]:

```
s2.isnull() #prints True if NaN is there
```

Out[37]:

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

In [38]:

```
s2.notnull() #prints True is NaN is not there
```

Out[38]:

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

In [39]:

```
s2[s2.notnull()]  #this remove NaN values
```

Out[39]:

```
0     5.0
1    -3.0
3    14.0
dtype: float64
```

In [40]:

```
s2[s2.isnull()]  #finds full values for you with index too
```

Out[40]:

```
2   NaN
dtype: float64
```

Series as Dictionaries

so far I've told you that, arrays gets passed as Input to a Series, what if I pass a dictionay? Dictinary has two inputs, keys and values, so in this case, keys become index and values become value :-P

In [41]:

```
mydict={'red':2000,'blue':1000,'yellow':500,'orange':1000}
```

In [42]:

```
myseries=pd.Series(mydict)  #passing dictionary as argument
```

In [43]:

```
myseries
```

Out[43]:

```
red       2000
blue      1000
yellow     500
orange    1000
dtype: int64
```

you can even change the index like i told you earlier by using index and pass a list

Operations between Series

let's create another series, with the help of dictionary.

In [44]:

```
mydict2={'red':400,'yellow':1000,'black':700}
```

In [45]:

```
myseries2=pd.Series(mydict2)
```

In [46]:

```
myseries2
```

Out[46]:

```
red        400
yellow    1000
black      700
dtype: int64
```

In [47]:

```
myseries  #we created this earlier you remember it?
```

Out[47]:

```
red       2000
blue      1000
yellow     500
orange    1000
dtype: int64
```

adding Series

In [48]:

```
myseries+myseries2
```

Out[48]:

```
black        NaN
blue         NaN
orange       NaN
red       2400.0
yellow    1500.0
dtype: float64
```

red yellow was common between those two series that's why only these got added, rest of fucntions can be used like this. Do if you want to use them. In series this is all we could have discussed. Let's move towards more important part of the pandas librabry that is DataFrame.

"DATAFRAME"

DataFrame is a tabular data structure very similar to Spreadsheet (like excel spreadsheets). This data structure is designed to extend the case of the Series to multiple dimensions.

In fact, the DataFrame consists of an ordered collection of columns each of which can contain a value of different type (numerica,string, boolean etc)

```
        DATAFRAME
      [-----columns------]
index  color  object price
  0    blue   ball    1.2
  1    green  pen     1.0
  2    yellow pencil  0.6
  3    red    paper   0.9
  4    white  mug     1.7
```

defining a DataFrame

most common way to ccreate a new DataFrame is precisely to pass a dictionary object ot DataFrame() constructor. This dict object contains a key for each column that we want to define, with an array of values

In [49]:

```
data={'color':['blue','green','yellow','red','white'],
     'object':['ball','pen','pencil','paper','mug'],
     'price':[1.2,1.0,0.6,0.9,1.7]}
```

what is did, i created a datset named data in which all the information is passed. data it self is a dictionary in which 'color' 'object' and 'price' are the keys and these will behave like columns name in our DataFrame and values of color object and price is given as list.

In [50]:

```
frame=pd.DataFrame(data)  #passed data to DataFrame as input
```

In [51]:

```
frame #printing just created frame
```

Out[51]:

	color	object	price
0	blue	ball	1.2
1	green	pen	1.0
2	yellow	pencil	0.6
3	red	paper	0.9
4	white	mug	1.7

see this is how we create a DataFrame and you can see that, color, object, price those were keys is our dictionary have become columns name, and the very first line is automatically generated and that is index.

you can also access specific col too

In [52]:

```
pd.DataFrame(data, columns=['object'])  #only print object
```

Out[52]:

	object
0	ball
1	pen
2	pencil
3	paper
4	mug

let's print more cols, increse the number of elements you passed in the list assigned to columns

In [53]:

```
pd.DataFrame(data,columns=['object','price'])
```

Out[53]:

	object	price
0	ball	1.2
1	pen	1.0
2	pencil	0.6
3	paper	0.9
4	mug	1.7

we can also change the index like we did in Series

In [54]:

```
frame2=pd.DataFrame(data,index=['one','two','three','four','five'])
```

In [55]:

```
frame2
```

Out[55]:

	color	object	price
one	blue	ball	1.2
two	green	pen	1.0
three	yellow	pencil	0.6
four	red	paper	0.9
five	white	mug	1.7

see index has been changed too. Easy Peasy right?

theres is another way to define a DataFrame by using index and columns

let's create another DataFrame named frame3 using index and columns

In [56]:

```
frame3=pd.DataFrame(np.arange(16).reshape((4,4)),
                    index=['red','blue','yellow','white'],
                    columns=['ball','pen','pencil','paper'])
```

In [57]:

```
frame3  #prints frame3
```

Out[57]:

	ball	pen	pencil	paper
red	0	1	2	3
blue	4	5	6	7
yellow	8	9	10	11
white	12	13	14	15

selecting elements

In [58]:

```
frame3.columns  #prints cols
```

Out[58]:

```
Index(['ball', 'pen', 'pencil', 'paper'], dtype='object')
```

In [59]:

```
frame3.index  # prints index
```

Out[59]:

```
Index(['red', 'blue', 'yellow', 'white'], dtype='object')
```

In [60]:

```
frame3.values  #prints values of array
```

Out[60]:

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

In [61]:

```
frame3['ball']  #prints col ball in frame3
```

Out[61]:

```
red        0
blue       4
yellow     8
white     12
Name: ball, dtype: int64
```

(TBR) return value is Series in this case. There is another way to access cols and that is out favorite dot method

In [62]:

```
frame3.ball
```

Out[62]:

```
red        0
blue       4
yellow     8
white     12
Name: ball, dtype: int64
```

if you remember, we have earlier created a DataFrame named, frame.

In [63]:

```
frame
```

Out[63]:

	color	object	price
0	blue	ball	1.2
1	green	pen	1.0
2	yellow	pencil	0.6
3	red	paper	0.9
4	white	mug	1.7

suppose you want to select multiple rows from the DataFrame

In [64]:

```
frame[0:3]  #will print rows, index 0 to 2 (3-1)
```

Out[64]:

	color	object	price
0	blue	ball	1.2
1	green	pen	1.0
2	yellow	pencil	0.6

(TBR)- To access cols from DataFrame you write frame[col] and for accessing rows you write frame[startingIndex : endingIndex]

If you want to access a single value from DataFrame like you want to access what's col 'object' has at row 3

In [65]:

```
frame['object'][3]
```

Out[65]:

```
'paper'
```

that means col object has paper in row 3

Assigning values

you can use name attribute to assign new labels to index and columns

In [66]:

```
frame
```

Out[66]:

	color	object	price
0	blue	ball	1.2
1	green	pen	1.0
2	yellow	pencil	0.6
3	red	paper	0.9
4	white	mug	1.7

In [67]:

```
frame.index.name='INDEX'
```

In [68]:

```
frame
```

Out[68]:

	color	object	price
INDEX			
0	blue	ball	1.2
1	green	pen	1.0
2	yellow	pencil	0.6
3	red	paper	0.9
4	white	mug	1.7

see index has got the name of INDEX. Same can be done for the cols too, suppose You want to name cols as COLUMN

In [69]:

```
frame.columns.name='COLUMNS'
```

In [70]:

```
frame
```

Out[70]:

COLUMNS	color	object	price
INDEX			
0	blue	ball	1.2
1	green	pen	1.0
2	yellow	pencil	0.6
3	red	paper	0.9
4	white	mug	1.7

see, COLUMNS and INDEX has been named there.

In [71]:

```
frame #let's check frame
```

Out[71]:

COLUMNS	color	object	price
INDEX			
0	blue	ball	1.2
1	green	pen	1.0
2	yellow	pencil	0.6
3	red	paper	0.9
4	white	mug	1.7

(TBR)- So if you made changes to a DataFrame, it changes permanently

Adding a new column

one of the best feature if the data strucute of pandas is their high flexibility, In fact you can always intervene at any level to change the internal data structure. For example, a very common operation is to add a new column.

you can simply do this by assigning a value to the 'new'

if we remember, we access col like frame[col] so will be passing new like this.

In [72]:

```
frame['new']=12  #CREATED a col named new in which each value is 12
```

In [73]:

```
frame
```

Out[73]:

COLUMNS	color	object	price	new
INDEX				
0	blue	ball	1.2	12
1	green	pen	1.0	12
2	yellow	pencil	0.6	12
3	red	paper	0.9	12
4	white	mug	1.7	12

you can see that a new column has been added named 'new' and each value in that col is 12

you can pass a list to update values in col, like I am updating values at col new

In [74]:

```
frame['new']=[3,1.3,2.2,0.8,1.1]
```

In [75]:

```
frame
```

Out[75]:

COLUMNS	color	object	price	new
INDEX				
0	blue	ball	1.2	3.0
1	green	pen	1.0	1.3
2	yellow	pencil	0.6	2.2
3	red	paper	0.9	0.8
4	white	mug	1.7	1.1

see that values has been updated there in col 'new'

there is another way to update, you can use Series too. Suppose we have a series

In [76]:

```
ser=pd.Series(np.arange(5))
```

In [77]:

```
ser
```

Out[77]:

```
0    0
1    1
2    2
3    3
4    4
dtype: int64
```

In [78]:

```
frame['new']=ser  #assigning series to col new
```

In [79]:

```
frame
```

Out[79]:

COLUMNS	color	object	price	new
INDEX				
0	blue	ball	1.2	0
1	green	pen	1.0	1
2	yellow	pencil	0.6	2
3	red	paper	0.9	3
4	white	mug	1.7	4

now see that, col 'new' has been updated again and this time, series has filled it's values

What if you want to change a single value

suppose I want to change in col price at row 2, and assigning a new value to 3.3 (previosly is is 0.6, look above)

In [80]:

```
frame['price'][2]=3.3
```

```
/home/akshansh/.local/lib/python3.6/site-packages/ipykernel_launcher.p
y:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas
-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
 (https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.htm
l#returning-a-view-versus-a-copy)
  """Entry point for launching an IPython kernel.
```

In [81]:

```
frame #see what we got now
```

Out[81]:

COLUMNS	color	object	price	new
INDEX				
0	blue	ball	1.2	0
1	green	pen	1.0	1
2	yellow	pencil	3.3	2
3	red	paper	0.9	3
4	white	mug	1.7	4

see carefully, at row number (index) 2, col price has changed. Wasn't that easy?

Membership of a value

remember isin() from the series? Yeah? It is as applicable too in DataFrame too

In [82]:

```
frame
```

Out[82]:

COLUMNS	color	object	price	new
INDEX				
0	blue	ball	1.2	0
1	green	pen	1.0	1
2	yellow	pencil	3.3	2
3	red	paper	0.9	3
4	white	mug	1.7	4

In [83]:

```
frame.isin([1.0,'pen'])  #prints True where is 1 and pen
```

Out[83]:

```
COLUMNS  color  object  price    new
  INDEX
      0  False   False  False  False
      1  False    True   True   True
      2  False   False  False  False
      3  False   False  False  False
      4  False   False  False  False
```

that's not a good view, definitely not me. It might be for you but not me, let's check something else

In [84]:

```
frame[frame.isin([1.0,'pen'])]  #getting True values
```

Out[84]:

```
COLUMNS  color  object  price  new
  INDEX
      0    NaN     NaN    NaN  NaN
      1    NaN     pen    1.0  1.0
      2    NaN     NaN    NaN  NaN
      3    NaN     NaN    NaN  NaN
      4    NaN     NaN    NaN  NaN
```

now this looks good, isn't it?

Deleting a column

My mom once said, it is hard to build but always easy to destroy. And she was talking about the respect. We built new col by hard work and now we are going to delete. It will be easy

In [85]:

```
del frame['new']
```

In [86]:

```
frame #see what we got now
```

Out[86]:

COLUMNS	color	object	price
INDEX			
0	blue	ball	1.2
1	green	pen	1.0
2	yellow	pencil	3.3
3	red	paper	0.9
4	white	mug	1.7

col 'new' has gone now

Filtering

Even for a DataFrame you can apply the filtering through the application of certain conditions, for example if you want to get all the values smaller than a certain number, for example

In [87]:

```
frame3 #We crreated this earlier you remember nah?
```

Out[87]:

	ball	pen	pencil	paper
red	0	1	2	3
blue	4	5	6	7
yellow	8	9	10	11
white	12	13	14	15

In [88]:

```
frame3<12
```

Out[88]:

	ball	pen	pencil	paper
red	True	True	True	True
blue	True	True	True	True
yellow	True	True	True	True
white	False	False	False	False

that's not what we looking for,let's dig bit more deep

In [89]:

```
frame3[frame3<12]
```

Out[89]:

	ball	pen	pencil	paper
red	0.0	1.0	2.0	3.0
blue	4.0	5.0	6.0	7.0
yellow	8.0	9.0	10.0	11.0
white	NaN	NaN	NaN	NaN

see all the values in frame3 which were less than 12 have been printed and other has become NaN

Transposition of a DataFrame

In [90]:

```
frame3
```

Out[90]:

	ball	pen	pencil	paper
red	0	1	2	3
blue	4	5	6	7
yellow	8	9	10	11
white	12	13	14	15

In [91]:

```
frame3.T #transposing
```

Out[91]:

	red	blue	yellow	white
ball	0	4	8	12
pen	1	5	9	13
pencil	2	6	10	14
paper	3	7	11	15

observe that, rows have become col and col has become rows

Playing with index Objects

you have seen basic operations on Series and DataFrames. Let's play with the index of each. Sometimes you have to change, alter and do other shit stuff with index. That's why I created a whole new topic of this.

let's declare a series (you haven't forgotten series, right?)

In [92]:

```
ser=pd.Series([5,0,3,8,4],index=['red','blue','yellow','white','green'])
```

In [93]:

```
ser #prints Series
```

Out[93]:

```
red       5
blue      0
yellow    3
white     8
green     4
dtype: int64
```

In [94]:

```
ser.index #prints index of Series
```

Out[94]:

```
Index(['red', 'blue', 'yellow', 'white', 'green'], dtype='object')
```

(TBR) - Unlike other elements in Pandas Series and DataFrame, Index is Immutable, that means it can't be changed. (unless you put an extra effort)

In [95]:

```
ser.idxmin()  #print index of min value
```

Out[95]:

```
'blue'
```

In [96]:

```
ser.idxmax()  #prints index of max value
```

Out[96]:

```
'white'
```

what if index has duplicate labels

In [97]:

```
serd=pd.Series(range(4), index=['white','white','green','blue'])
```

In [98]:

```
serd
```

Out[98]:

```
white    0
white    1
green    2
blue     3
dtype: int64
```

In [99]:

```
serd['white']  #prints all the white in index of Series
```

Out[99]:

```
white    0
white    1
dtype: int64
```

how can you check if Series has duplicate or all are uique

No need to take tension, let pully handle it, (ha ha ha, it was a joke, if you are physics student, you would have got it). Anyways, let's check if Series is unique or not?

In [100]:

```
serd.index.is_unique
```

Out[100]:

```
False
```

Means our serd is not unique, it has duplicates. That's you know it.

reindex()

sometime you've to cahnge the order of the index, reindex() help you in this. Pandas got no chill man, it has everything for everything. A rich library, should I say padas as Bill Gates of Python :-P

In [101]:

```
ser
```

Out[101]:

```
red       5
blue      0
yellow    3
white     8
green     4
dtype: int64
```

let's reindex it

In [102]:

```
ser.reindex(['blue','red','white','yellow','green','orange'])
```

Out[102]:

```
blue      0.0
red       5.0
white     8.0
yellow    3.0
green     4.0
orange    NaN
dtype: float64
```

see order of index has been changed. Didn't you notice, NaN is in fron of orange, yes because orange was not in origianl ser Series. Pandas was not able to find it's value, so it added NaN to it.

You see it is not sorted, or in random order so how do we sort?

In [103]:

```
ser3=pd.Series([1,5,6,3],index=[0,3,5,6])
```

In [104]:

```
ser3
```

Out[104]:

```
0    1
3    5
5    6
6    3
dtype: int64
```

as you can see in this example, the index column in not a perfect sequence of number,in fact there are missing values too (1,2 and 4). A common need would be to perform an interpolarion in order to obtain the complere sequence of number. To acheive this you will ise the reindexing with method option set to ffill.

In [105]:

```
ser3.reindex(range(6),method='ffill')
```

Out[105]:

```
0    1
1    1
2    1
3    5
4    5
5    6
dtype: int64
```

you can see that, index which were not present earlier now have been added, and by convetion lowest value in series is assigned to them. What if you want to assign greatest value to these missing index.

In [106]:

```
ser3.reindex(range(6),method='bfill')
```

Out[106]:

```
0    1
1    5
2    5
3    5
4    6
5    6
dtype: int64
```

using ffill in DataFrame

In [107]:

```
frame.reindex(range(5),method='ffill', columns=['colors','price','new','object'])
```

Out[107]:

COLUMNS	colors	price	new	object
INDEX				
0	blue	1.2	blue	ball
1	green	1.0	green	pen
2	yellow	3.3	yellow	pencil
3	red	0.9	red	paper
4	white	1.7	white	mug

Dropping

In [108]:

```
ser
```

Out[108]:

```
red       5
blue      0
yellow    3
white     8
green     4
dtype: int64
```

In [109]:

```
ser.drop('yellow')  #will drop yellow index
```

Out[109]:

```
red      5
blue     0
white    8
green    4
dtype: int64
```

```
In [110]:
ser #no change in orginal series drop is temporary
Out[110]:
red       5
blue      0
yellow    3
white     8
green     4
dtype: int64
```

pass list of index you want to drop

```
In [111]:
ser.drop(['blue','white'])
Out[111]:
red       5
yellow    3
green     4
dtype: int64
```

blue and white have been dropped

drop on DataFrame

```
In [112]:
frame=pd.DataFrame(np.arange(16).reshape((4,4)),
                   index=['red','blue','yellow','white'],
                   columns=['ball','pen','pencil','paper'])
```

```
In [113]:
frame
Out[113]:
```

	ball	pen	pencil	paper
red	0	1	2	3
blue	4	5	6	7
yellow	8	9	10	11
white	12	13	14	15

to delete rows, we just pass the indexes of rows

In [114]:

```
frame.drop(['blue','yellow'])
```

Out[114]:

	ball	pen	pencil	paper
red	0	1	2	3
white	12	13	14	15

blue and yellow are no more there.

In [115]:

```
frame.drop(['blue','yellow'], axis=0)  #axis=0 means row
```

Out[115]:

	ball	pen	pencil	paper
red	0	1	2	3
white	12	13	14	15

to delete columns, just pass the column name in a list to drop but always specify that axis=1.

In [116]:

```
frame.drop(['pen','pencil'],axis=1)
```

Out[116]:

	ball	paper
red	0	3
blue	4	7
yellow	8	11
white	12	15

column name pen and pencil has dropped from the DataFrame

Airthmatics operation on two dataframes

addition of two Series has been already discussed earlier in Series, let's talk about addition of DataFrames

In [117]:

```
frame1=pd.DataFrame(np.arange(16).reshape((4,4)),
                   index=['red','blue','yellow','white'],
                   columns=['ball','pen','pencil','paper'])
```

In [118]:

```
frame1
```

Out[118]:

	ball	pen	pencil	paper
red	0	1	2	3
blue	4	5	6	7
yellow	8	9	10	11
white	12	13	14	15

In [119]:

```
frame2=pd.DataFrame(np.arange(12).reshape((4,3)),
                   index=['blue','green','white','yellow'],
                   columns=['mug','pen','ball'])
```

In [120]:

```
frame2
```

Out[120]:

	mug	pen	ball
blue	0	1	2
green	3	4	5
white	6	7	8
yellow	9	10	11

In [121]:

```
frame1+frame2
```

Out[121]:

	ball	mug	paper	pen	pencil
blue	6.0	NaN	NaN	6.0	NaN
green	NaN	NaN	NaN	NaN	NaN
red	NaN	NaN	NaN	NaN	NaN
white	20.0	NaN	NaN	20.0	NaN
yellow	19.0	NaN	NaN	19.0	NaN

common index (things) has been added and rest of them which are not common has become NaN

In [122]:

```
frame1-frame2
```

Out[122]:

	ball	mug	paper	pen	pencil
blue	2.0	NaN	NaN	4.0	NaN
green	NaN	NaN	NaN	NaN	NaN
red	NaN	NaN	NaN	NaN	NaN
white	4.0	NaN	NaN	6.0	NaN
yellow	-3.0	NaN	NaN	-1.0	NaN

In [123]:

```
frame1/2  #dividing frame1 by 2
```

Out[123]:

	ball	pen	pencil	paper
red	0.0	0.5	1.0	1.5
blue	2.0	2.5	3.0	3.5
yellow	4.0	4.5	5.0	5.5
white	6.0	6.5	7.0	7.5

In [124]:

```
frame2/2  #dividing frame2 by 2
```

Out[124]:

	mug	pen	ball
blue	0.0	0.5	1.0
green	1.5	2.0	2.5
white	3.0	3.5	4.0
yellow	4.5	5.0	5.5

In [125]:

```
frame2*2  #multiplied by 2 in each elelment od frame 2
```

Out[125]:

	mug	pen	ball
blue	0	2	4
green	6	8	10
white	12	14	16
yellow	18	20	22

In [126]:

```
frame1*2  #each element of frame1 is multiplied by 2
```

Out[126]:

	ball	pen	pencil	paper
red	0	2	4	6
blue	8	10	12	14
yellow	16	18	20	22
white	24	26	28	30

In [127]:

```
frame1*frame2  #multiplying both frames
```

Out[127]:

	ball	mug	paper	pen	pencil
blue	8.0	NaN	NaN	5.0	NaN
green	NaN	NaN	NaN	NaN	NaN
red	NaN	NaN	NaN	NaN	NaN
white	96.0	NaN	NaN	91.0	NaN
yellow	88.0	NaN	NaN	90.0	NaN

operation btween data structures

you have seen basic operation on these two data types in pandas. Let's move bit ahead and talk about the other methods those include two or more than two dataframes.

flexible Airthmatic methods

add() sub() div() mul()

add()

In [128]:

```
frame1
```

Out[128]:

	ball	pen	pencil	paper
red	0	1	2	3
blue	4	5	6	7
yellow	8	9	10	11
white	12	13	14	15

In [129]:

```
frame2
```

Out[129]:

	mug	pen	ball
blue	0	1	2
green	3	4	5
white	6	7	8
yellow	9	10	11

In [130]:

```
frame1.add(frame2)
```

Out[130]:

	ball	mug	paper	pen	pencil
blue	6.0	NaN	NaN	6.0	NaN
green	NaN	NaN	NaN	NaN	NaN
red	NaN	NaN	NaN	NaN	NaN
white	20.0	NaN	NaN	20.0	NaN
yellow	19.0	NaN	NaN	19.0	NaN

sub()

In [132]:

```
frame1.sub(frame2)
```

Out[132]:

	ball	mug	paper	pen	pencil
blue	2.0	NaN	NaN	4.0	NaN
green	NaN	NaN	NaN	NaN	NaN
red	NaN	NaN	NaN	NaN	NaN
white	4.0	NaN	NaN	6.0	NaN
yellow	-3.0	NaN	NaN	-1.0	NaN

In [133]:

```
frame2.sub(frame1)
```

Out[133]:

	ball	mug	paper	pen	pencil
blue	-2.0	NaN	NaN	-4.0	NaN
green	NaN	NaN	NaN	NaN	NaN
red	NaN	NaN	NaN	NaN	NaN
white	-4.0	NaN	NaN	-6.0	NaN
yellow	3.0	NaN	NaN	1.0	NaN

div()

In [134]:

```
frame1.div(frame2)
```

Out[134]:

	ball	mug	paper	pen	pencil
blue	2.000000	NaN	NaN	5.000000	NaN
green	NaN	NaN	NaN	NaN	NaN
red	NaN	NaN	NaN	NaN	NaN
white	1.500000	NaN	NaN	1.857143	NaN
yellow	0.727273	NaN	NaN	0.900000	NaN

In [135]:

```
frame2.div(frame1)
```

Out[135]:

	ball	mug	paper	pen	pencil
blue	0.500000	NaN	NaN	0.200000	NaN
green	NaN	NaN	NaN	NaN	NaN
red	NaN	NaN	NaN	NaN	NaN
white	0.666667	NaN	NaN	0.538462	NaN
yellow	1.375000	NaN	NaN	1.111111	NaN

mul()

In [137]:

```
frame1.mul(frame2)
```

Out[137]:

	ball	mug	paper	pen	pencil
blue	8.0	NaN	NaN	5.0	NaN
green	NaN	NaN	NaN	NaN	NaN
red	NaN	NaN	NaN	NaN	NaN
white	96.0	NaN	NaN	91.0	NaN
yellow	88.0	NaN	NaN	90.0	NaN

In [138]:

```
frame2.mul(frame1)
```

Out[138]:

	ball	mug	paper	pen	pencil
blue	8.0	NaN	NaN	5.0	NaN
green	NaN	NaN	NaN	NaN	NaN
red	NaN	NaN	NaN	NaN	NaN
white	96.0	NaN	NaN	91.0	NaN
yellow	88.0	NaN	NaN	90.0	NaN

(TBR) you can also perfrom operations between DataFrames and Series

In [140]:

```
frame=pd.DataFrame(np.arange(16).reshape((4,4)),
                  index=['red','blue','yelloe','white'],
                  columns=['ball','pen','pencil','paper'])
```

In [141]:

```
frame
```

Out[141]:

	ball	pen	pencil	paper
red	0	1	2	3
blue	4	5	6	7
yelloe	8	9	10	11
white	12	13	14	15

let's create a Series where cols of DataFrame is Index

In [142]:

```
ser=pd.Series(np.arange(4), index=['ball','pen','pencil','paper'])
```

In [143]:

```
ser
```

Out[143]:

```
ball      0
pen       1
pencil    2
paper     3
dtype: int64
```

In [144]:

```
frame-ser  #reduces the items in Ser from DataFrame
```

Out[144]:

	ball	pen	pencil	paper
red	0	0	0	0
blue	4	4	4	4
yelloe	8	8	8	8
white	12	12	12	12

In [145]:

```
frame+ser
```

Out[145]:

	ball	pen	pencil	paper
red	0	2	4	6
blue	4	6	8	10
yelloe	8	10	12	14
white	12	14	16	18

In [147]:

```
frame.add(ser)  #can also perform the same action
```

Out[147]:

	ball	pen	pencil	paper
red	0	2	4	6
blue	4	6	8	10
yelloe	8	10	12	14
white	12	14	16	18

PANDAS LIBRARY FUNCTIONS

FUNCTIONS THOSE WORK ON ELEMENTS"

In [149]:

```
frame
```

Out[149]:

	ball	pen	pencil	paper
red	0	1	2	3
blue	4	5	6	7
yelloe	8	9	10	11
white	12	13	14	15

(TBR)- Pandas library is buit on the foundation of Numpy so all tha numpy functions will be eligible here

In [150]:

```
np.sqrt(frame) #taking square root of each element
```

Out[150]:

	ball	pen	pencil	paper
red	0.000000	1.000000	1.414214	1.732051
blue	2.000000	2.236068	2.449490	2.645751
yelloe	2.828427	3.000000	3.162278	3.316625
white	3.464102	3.605551	3.741657	3.872983

functions by rows and cols

you can also define the function in your own way and then apply it to whole DataFrame. Suppose I am describing a function of my own-

In [151]:

```
def my_func(x):
    return x.max() -x.min()
```

using apply() function to apply my own functio to the DataFrame

In [153]:

```
frame
```

Out[153]:

	ball	pen	pencil	paper
red	0	1	2	3
blue	4	5	6	7
yelloe	8	9	10	11
white	12	13	14	15

In [152]:

```
frame.apply(my_func)
```

Out[152]:

```
ball      12
pen       12
pencil    12
paper     12
dtype: int64
```

so what is did, i created a function where we get max value - min value (colors doesn't matter, i don't support racism). and in the answer what you get is, a single column will all the desired value.

In [154]:

```
frame.apply(my_func, axis=0)  #axis 0 means apply on rows
```

Out[154]:

```
ball      12
pen       12
pencil    12
paper     12
dtype: int64
```

you can apply it to the columns too,

In [156]:

```
frame.apply(my_func, axis=1) #axis=1 means cols
```

Out[156]:

```
red       3
blue      3
yelloe    3
white     3
dtype: int64
```

what happened, it looked wax value in col wise and then reduced the min value for each index

Let me show you another way around, that's something beautiful

In [158]:

```
def my_func(x):
    return pd.Series([x.min(),x.max()],index=['min value','max value'])
```

I created a function here that returns, series on x.min() and x.max() where index will be min value and max value

let's apply it to the DataFrame

In [159]:

```
frame.apply(my_func)
```

Out[159]:

	ball	pen	pencil	paper
min value	0	1	2	3
max value	12	13	14	15

sum()

In [160]:

```
frame
```

Out[160]:

	ball	pen	pencil	paper
red	0	1	2	3
blue	4	5	6	7
yelloe	8	9	10	11
white	12	13	14	15

In [162]:

```
frame.sum()  #prints total of a col
```

Out[162]:

```
ball      24
pen       28
pencil    32
paper     36
dtype: int64
```

mean()

In [164]:

```
frame.mean()  #mean value of a col will be printed
```

Out[164]:

```
ball      6.0
pen       7.0
pencil    8.0
paper     9.0
dtype: float64
```

describe()

In [165]:

```
frame.describe()  #wide view of stats of DataFrame
```

Out[165]:

	ball	pen	pencil	paper
count	4.000000	4.000000	4.000000	4.000000
mean	6.000000	7.000000	8.000000	9.000000
std	5.163978	5.163978	5.163978	5.163978
min	0.000000	1.000000	2.000000	3.000000
25%	3.000000	4.000000	5.000000	6.000000
50%	6.000000	7.000000	8.000000	9.000000
75%	9.000000	10.000000	11.000000	12.000000
max	12.000000	13.000000	14.000000	15.000000

Sorting and Ranking

In [168]:

```
ser=pd.Series([5,0,3,8,4],index=['red','blue','yellow','white','green'])
```

In [169]:

```
ser
```

Out[169]:

```
red       5
blue      0
yellow    3
white     8
green     4
dtype: int64
```

In [171]:

```
ser.sort_index()  #sort index alphabatically
```

Out[171]:

```
blue      0
green     4
red       5
white     8
yellow    3
dtype: int64
```

In [172]:

```
ser.sort_index(ascending=False) # reverses the aplphabatical order
```

Out[172]:

```
yellow    3
white     8
red       5
green     4
blue      0
dtype: int64
```

sorting in DataFrame

In [173]:

```
frame
```

Out[173]:

	ball	pen	pencil	paper
red	0	1	2	3
blue	4	5	6	7
yelloe	8	9	10	11
white	12	13	14	15

In [175]:

```
frame.sort_index()  #sorts index alphabatically
```

Out[175]:

	ball	pen	pencil	paper
blue	4	5	6	7
red	0	1	2	3
white	12	13	14	15
yelloe	8	9	10	11

In [177]:

```
frame.sort_index(ascending=False) #reverse sorting
```

Out[177]:

	ball	pen	pencil	paper
yelloe	8	9	10	11
white	12	13	14	15
red	0	1	2	3
blue	4	5	6	7

what if you want to sort cols instead of rows (index)

In [192]:

```
frame.sort_index(axis=1)
```

Out[192]:

	ball	paper	pen	pencil
red	0	3	1	2
blue	4	7	5	6
yelloe	8	11	9	10
white	12	15	13	14

look at the cols, alphabatically sorted now

correlation and covariance

two important statistical calculations are correlation and covariance, expressed in pandas by corr() and cov(). I suggest you to read more about these in mathematics books of BSc Statistics

In [196]:

```
frame
```

Out[196]:

	ball	pen	pencil	paper
red	0	1	2	3
blue	4	5	6	7
yelloe	8	9	10	11
white	12	13	14	15

In [197]:

```
frame.corr()
```

Out[197]:

	ball	pen	pencil	paper
ball	1.0	1.0	1.0	1.0
pen	1.0	1.0	1.0	1.0
pencil	1.0	1.0	1.0	1.0
paper	1.0	1.0	1.0	1.0

In [198]:

```
frame.cov()
```

Out[198]:

	ball	pen	pencil	paper
ball	26.666667	26.666667	26.666667	26.666667
pen	26.666667	26.666667	26.666667	26.666667
pencil	26.666667	26.666667	26.666667	26.666667
paper	26.666667	26.666667	26.666667	26.666667

NaN type value

when some thing is missing in Data It is Usually depicted as NaN not a number type va;ue. Let's find out what is it and how it fucntions

assigning a NaN value

i've told you earlier in Series that it is assigned by np.NaN value

In [199]:

```
ser = pd.Series([0,1,2,np.NaN,9], index=['red','blue','yellow','white','green']
```

In [200]:

```
ser
```

Out[200]:

```
red       0.0
blue      1.0
yellow    2.0
white     NaN
green     9.0
dtype: float64
```

you can see that NaN value is assigned to white

In [201]:

```
ser['red']
```

Out[201]:

```
0.0
```

In [202]:

```
ser['yellow']
```

Out[202]:

```
2.0
```

In [203]:

```
ser['white']
```

Out[203]:

```
nan
```

that means nothing there on white index

filtering the NaN value

In [204]:

```
ser
```

Out[204]:

```
red       0.0
blue      1.0
yellow    2.0
white     NaN
green     9.0
dtype: float64
```

In [205]:

```
ser.dropna()  #drop NaN index
```

Out[205]:

```
red       0.0
blue      1.0
yellow    2.0
green     9.0
dtype: float64
```

since white was NaN index, it has been dropped by dropna() method

In [206]:

```
ser.notnull() #prints which are not NaN
```

Out[206]:

```
red        True
blue       True
yellow     True
white     False
green      True
dtype: bool
```

that is not looking good, let's get the values

In [207]:

```
ser[ser.notnull()]
```

Out[207]:

```
red       0.0
blue      1.0
yellow    2.0
green     9.0
dtype: float64
```

now looks great.

In [208]:

```
frame3 = pd.DataFrame([[6,np.nan,6],[np.nan,np.nan,np.nan],[2,np.nan,5]],
                            index = ['blue','green','red'],
                            columns = ['ball','mug','pen'])
```

In [210]:

```
frame3
```

Out[210]:

	ball	mug	pen
blue	6.0	NaN	6.0
green	NaN	NaN	NaN
red	2.0	NaN	5.0

In [211]:

```
frame3.dropna()
```

Out[211]:

ball	mug	pen

this didn't work as we expected, we should specify how to here

In [212]:

```
frame3.dropna(how='all')
```

Out[212]:

	ball	mug	pen
blue	6.0	NaN	6.0
red	2.0	NaN	5.0

now, you will see that row called, green which has all the values in NaN type, has been removed. You will not want to remove all the NaN value because ther are other values too and you have to play with them. But a row full of NaN values, you can't play with that, so you better drop it, right?

filling Nan Occurrances

In [213]:

```
frame3
```

Out[213]:

	ball	mug	pen
blue	6.0	NaN	6.0
green	NaN	NaN	NaN
red	2.0	NaN	5.0

In [215]:

```
frame3.fillna(0)
```

Out[215]:

	ball	mug	pen
blue	6.0	0.0	6.0
green	0.0	0.0	0.0
red	2.0	0.0	5.0

see NaN values has been filled by 0

In [216]:

```
frame3.fillna(1)
```

Out[216]:

	ball	mug	pen
blue	6.0	1.0	6.0
green	1.0	1.0	1.0
red	2.0	1.0	5.0

all NaN values has been filled by 1, you can do whatever you want to do

In [217]:

```
frame3.fillna("nothing here")
```

Out[217]:

	ball	mug	pen
blue	6	nothing here	6
green	nothing here	nothing here	nothing here
red	2	nothing here	5

In [ ]:

```

```