In [1]:
```
import numpy as np
import pandas as pd
```

Class4: 3rd January, 2021, Sunday

The Pandas DataFrame Object

A DataFrame represents a rectangular table of data and contains an ordered collec- tion of columns, each of which can be a different value type (numeric, string, boolean, etc.).

Series object single dimension ki hoti hai, jabke dataframe object rectangular dimension ka hota hai

Creating a DataFrame from scratch

In [2]:
```
# Create a DataFrame from a 2d ndarray

# df = pd.DataFrame(np.array([[10, 11, 12, 13, 12], [20, 21, 22, 23]]))  error dega>> no of cols !=

df = pd.DataFrame(np.array([[10, 11, 12, 13], [20, 21, 22, 23]]))
df

# default row and column indexes

# number of columns dono mai equal hone chahiyen warna ye error dega
# series mai sirf row index hota hai jabke dataframe main row and col index dono hain
```

Out[2]:

	0	1	2	3
0	10	11	12	13
1	20	21	22	23

In [3]:
```
# Create a DataFrame from a list of series objects

df1 = pd.DataFrame([pd.Series(np.arange(10,15)),
                    pd.Series(np.arange(15,20))])

df1

# default row and col indexes
```

Out[3]:

	0	1	2	3	4
0	10	11	12	13	14
1	15	16	17	18	19

In [4]:
```
# Create a DataFrame from two Series objects
# and a dictionary

s1 = pd.Series(np.arange(1, 6, 1))

s2 = pd.Series(np.arange(6, 11, 1))

df2 = pd.DataFrame({'boys': s1, 'girls': s2})
df2

# keys hamari automatically col indices banjati hain
```

Out[4]:

	boys	girls
0	1	6
1	2	7
2	3	8
3	4	9
4	5	10

In [5]:
```
data = {'name': ["Shahid", "Sadiq", "Kashif", "Rashid"], 'age': [20,26,36,40], 'grades': ["A","B","A","B"]}

data = pd.DataFrame(data)

data
```

Out[5]:

	name	age	grades
0	Shahid	20	A
1	Sadiq	26	B
2	Kashif	36	A
3	Rashid	40	B

In [6]:
```
# specify column names

df3 = pd.DataFrame(np.array([[10, 11], [20, 21]]), columns=['apples', 'oranges'])
df3

# jab hum frame dictionary sai nhi banarhe to cols ko labels is tarah denge
```

Out[6]:

	apples	oranges
0	10	11
1	20	21

In [7]:
```
# create a DataFrame with named columns and rows

df4 = pd.DataFrame(np.array([[10,11,12,13], [20,21,22,23]]),
                   index=['apples', 'oranges'],
                   columns=['Mon','Tue','Wed','Thu'])

df4
```

Out[7]:

	Mon	Tue	Wed	Thu
apples	10	11	12	13
oranges	20	21	22	23

In [8]:
```
# demonstrate alignment during creation

s3 = pd.Series(np.arange(12, 14), index=[1,2])

df5 = pd.DataFrame({'c1': s1, 'c2': s2, 'c3': s3})
df5
```

Out[8]:

	c1	c2	c3
0	1	6	NaN
1	2	7	12.0
2	3	8	13.0
3	4	9	NaN
4	5	10	NaN

Examples of creating data frames

In [9]:
```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002, 2003],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}

frame = pd.DataFrame(data)
frame
```

Out[9]:

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9
5	Nevada	2003	3.2

In [10]:
```
# ab hum ye chchte hain ke ye state pehle nhi aye second pe aye pehle year ajaye, to mujhe columns indices ko shuffle..
# ..karna hoga, or ye hamein new frame return karta hai, original ko modify nhi karta

pd.DataFrame(frame, columns=['year', 'state', 'pop'])   # in-place nhi hoga
```

Out[10]:

	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

In [11]:
```
# Values kese uthaynge cols ki, 2 tarah sai uthti hain values
# ye wala format sirf us soorat mai execute hoga jab cols ese define hue hon jese hum apna variable define karte hain
# ..jese hum variable mai "-" use nhi karte agar humne dash use kia to ye format error dega, dash wale keliye neechay wala..
# ..format use karenge

frame.year
```

Out[11]:
```
0    2000
1    2001
2    2002
3    2001
4    2002
5    2003
Name: year, dtype: int64
```

In [12]:
```
frame['pop']
```

Out[12]:
```
0    1.5
1    1.7
2    3.6
3    2.4
4    2.9
5    3.2
Name: pop, dtype: float64
```

In [13]:
```
# If u pass a col that isn't contained in the dict(debt), it will appear with missing values in the result:

frame2 = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
                            index=['one', 'two', 'three', 'four', 'five', 'six'])

frame2

# ab yahan new col aa to gya lekin ismai value kese seed karen? kia tareeqe hain ?
```

Out[13]:

	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
six	2003	Nevada	3.2	NaN

In [14]:
```
# 1st method is mai ne poore col ko call kia or usmai hardcode value dedi

frame2.debt = 100
frame2
```

Out[14]:

	year	state	pop	debt
one	2000	Ohio	1.5	100
two	2001	Ohio	1.7	100
three	2002	Ohio	3.6	100
four	2001	Nevada	2.4	100
five	2002	Nevada	2.9	100
six	2003	Nevada	3.2	100

In [15]:
```
# 2nd method is debt ke col ko call karo or range create kardo

frame2['debt'] = np.arange(6)
frame2
```

Out[15]:

	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

In [16]:
```
# 3rd method is humne val series banayi

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

Out[16]:
```
two    -1.2
four   -1.5
five   -1.7
dtype: float64
```

In [17]:
```
frame2['debt'] = val

frame2
```

Out[17]:

	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	NaN

In [18]:
```
# Adding more cols to dataframe

frame2['eastern'] = frame2.state == 'Ohio'   # true / false

frame2
```

Out[18]:

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

In [19]:
```
frame2['greaterThan2'] = frame2['pop'] > 2
frame2
```

Out[19]:

	year	state	pop	debt	eastern	greaterThan2
one	2000	Ohio	1.5	NaN	True	False
two	2001	Ohio	1.7	-1.2	True	False
three	2002	Ohio	3.6	NaN	True	True
four	2001	Nevada	2.4	-1.5	False	True
five	2002	Nevada	2.9	-1.7	False	True
six	2003	Nevada	3.2	NaN	False	True

In [20]:
```
# delete in-place 'eastern' col
del frame2['eastern']
frame2
```

Out[20]:

	year	state	pop	debt	greaterThan2
one	2000	Ohio	1.5	NaN	False
two	2001	Ohio	1.7	-1.2	False
three	2002	Ohio	3.6	NaN	True
four	2001	Nevada	2.4	-1.5	True
five	2002	Nevada	2.9	-1.7	True
six	2003	Nevada	3.2	NaN	True

In [21]:
```
data = {'name':["Asad","Saad","Fahad", 'Ali'],
        'age':[23,34,23,21],
        'aiForEveryOne':[89,78,90,98],
        'python':[78,89,87,89],
        'git': [90,98,87,86],
        'numpy':[98,87,98,99]}

data = pd.DataFrame(data)
data

# I want to add total marks and percentage cols as well
```

Out[21]:

	name	age	aiForEveryOne	python	git	numpy
0	Asad	23	89	78	90	98
1	Saad	34	78	89	98	87
2	Fahad	23	90	87	87	98
3	Ali	21	98	89	86	99

In [22]:
```
# total col
data['Total'] = data[['aiForEveryOne', 'python', 'git', 'numpy']].sum(axis=1)

# percentage col
data['Percentage'] = data['Total'] / 400 * 100

data
```

Out[22]:

	name	age	aiForEveryOne	python	git	numpy	Total	Percentage
0	Asad	23	89	78	90	98	355	88.75
1	Saad	34	78	89	98	87	352	88.00
2	Fahad	23	90	87	87	98	362	90.50
3	Ali	21	98	89	86	99	372	93.00

In [23]:
```
data['Grade'] = np.where((data['Total'] >= 360), 'A', 'B')
data
```

Out[23]:

	name	age	aiForEveryOne	python	git	numpy	Total	Percentage	Grade
0	Asad	23	89	78	90	98	355	88.75	B
1	Saad	34	78	89	98	87	352	88.00	B
2	Fahad	23	90	87	87	98	362	90.50	A
3	Ali	21	98	89	86	99	372	93.00	A

In [24]:
```
#Another common form of data is a nested dict of dicts:

# pop = {'Nevada': {2001: 2.4, 2002: 2.9},

#        'Ohio':   {2000: 1.5, 2001: 1.7, 2002: 3.6}}

pop = {
    'Nevada': {
        2001: 2.4,
        2002: 2.9
    },
    'Ohio': {
        2000: 1.5,
        2001: 1.7,
        2002: 3.6
    }
}

df3 =pd.DataFrame(pop)
df3

# jo outermost key hoti hai wo cols banjate hain or jo inner keys hoti hain wo rows banjati hain
```

Out[24]:

	Nevada	Ohio
2001	2.4	1.7
2002	2.9	3.6
2000	NaN	1.5

If the nested dict is passed to the DataFrame, pandas will interpret the outer dict keys as the columns and the inner keys as the row indices

In [25]:
```
# data frames can be transposed

df3.T     #pass by value
```

Out[25]:

	2001	2002	2000
Nevada	2.4	2.9	NaN
Ohio	1.7	3.6	1.5

In [26]:
```
df3      #original is not changed
```

Out[26]:

	Nevada	Ohio
2001	2.4	1.7
2002	2.9	3.6
2000	NaN	1.5

In [27]:
```
pop
```

Out[27]:
```
{'Nevada': {2001: 2.4, 2002: 2.9}, 'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

In [28]:
```
pop1 = pd.DataFrame(pop, index=[2001,2002,2003])
pop1
```

Out[28]:

	Nevada	Ohio
2001	2.4	1.7
2002	2.9	3.6
2003	NaN	NaN

In [29]:
```
# pdata = {'Ohio': df3['Ohio'][:-1],

#          'Nevada': df3['Nevada'][:2]
#          }


# df3 ke frame main jao or ohio ke col mai yahan tak data slice karke le ao
pdata = {
    'Ohio': df3['Ohio'][:-1],
    'Nevada': df3['Nevada'][:2]
}

pd.DataFrame(pdata)
```

Out[29]:

	Ohio	Nevada
2001	1.7	2.4
2002	3.6	2.9

In [30]:
```
df3
```

Out[30]:

	Nevada	Ohio
2001	2.4	1.7
2002	2.9	3.6
2000	NaN	1.5

In [31]:
```
# hum indexes ke naam bhi deskate hain
df3.index.name = 'year'
df3.columns.name = 'state_names'

df3
```

Out[31]:

state_names	Nevada	Ohio
year		
2001	2.4	1.7
2002	2.9	3.6
2000	NaN	1.5

Index Objects

pandas's Index objects are responsible for holding the axis labels and other metadata (like the axis name or names). Any array or other sequence of labels you use when constructing a Series or DataFrame is internally converted to an Index:

```
Index jo hum dete hain usmain bhi operations perform hosakte hain
```

In [32]:
```
obj = pd.Series(range(3), index=['a', 'b', 'c'])
obj
```

Out[32]:
```
a    0
b    1
c    2
dtype: int64
```

In [33]:
```
# jo iski index hian wo bhi 1 object hai
index = obj.index
index
```

Out[33]:
```
Index(['a', 'b', 'c'], dtype='object')
```

In [34]:
```
# Slice index
index[1:]
```

Out[34]:
```
Index(['b', 'c'], dtype='object')
```

In [35]:
```
index[1] = 'd'   # indices are immutable  means indices ka object immutable hota hai
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-35-dc69b677ecb7> in <module>
----> 1 index[1] = 'd'   # indices are immutable  means indices ka object immutable hota hai

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\indexes\base.py in __setitem__(self, key, value)
   3908
   3909     def __setitem__(self, key, value):
-> 3910         raise TypeError("Index does not support mutable operations")
   3911
   3912     def __getitem__(self, key):

TypeError: Index does not support mutable operations
```

In [36]:
```
# yahan mai ne 1 list create kari or uso type caste kardiya index ke object main

# created an ndarray that is immutable
# coz created via Index function and index are immutable
labels = pd.Index(["a","b","c","d","e","f"])

labels
```

Out[36]:
```
Index(['a', 'b', 'c', 'd', 'e', 'f'], dtype='object')
```

In [37]:
```
# immutable hone ki wajh sai value change nhi hogi
labels[0] = "z"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-37-f6802854a9ac> in <module>
      1 # immutable hone ki wajh sai value change nhi hogi
----> 2 labels[0] = "z"

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\indexes\base.py in __setitem__(self, key, value)
   3908
   3909     def __setitem__(self, key, value):
-> 3910         raise TypeError("Index does not support mutable operations")
   3911
   3912     def __getitem__(self, key):

TypeError: Index does not support mutable operations
```

In [38]:
```
print(frame)
```

```
    state  year  pop
0    Ohio  2000  1.5
1    Ohio  2001  1.7
2    Ohio  2002  3.6
3  Nevada  2001  2.4
4  Nevada  2002  2.9
5  Nevada  2003  3.2
```

In [39]:
```
# mai apne labels ko apne index mai fit kar rha hun qk uski type index thi
frame.index = labels
frame
```

Out[39]:

	state	year	pop
a	Ohio	2000	1.5
b	Ohio	2001	1.7
c	Ohio	2002	3.6
d	Nevada	2001	2.4
e	Nevada	2002	2.9
f	Nevada	2003	3.2

In [40]:
```
frame.index    # is index type object
```

Out[40]:
```
Index(['a', 'b', 'c', 'd', 'e', 'f'], dtype='object')
```

In [41]:
```
frame.columns   # is also index type object iska matlab hua ke col ka naam bhi change nhi karsakte
```

Out[41]:
```
Index(['state', 'year', 'pop'], dtype='object')
```

Essential Functionality

In [42]:
```
frame2['debt']=np.arange(6)
print("The frame is", end="\n\n")

print(frame2,end="\n\n")

print("The row indices are", end="\n\n")

print(frame2.index,end="\n\n")

print("The col indeces are",end="\n\n")

print(frame2.columns)
```

```
The frame is

       year   state  pop  debt  greaterThan2
one    2000    Ohio  1.5     0         False
two    2001    Ohio  1.7     1         False
three  2002    Ohio  3.6     2          True
four   2001  Nevada  2.4     3          True
five   2002  Nevada  2.9     4          True
six    2003  Nevada  3.2     5          True

The row indices are

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

The col indeces are

Index(['year', 'state', 'pop', 'debt', 'greaterThan2'], dtype='object')
```

In [43]:
```
#### By default row are reindexed via reindex function  ####
# yaani agar mai nhi bata rha ke kia reindex karna hai to ye automtically isko row samjhega

reindex_frame = frame2.reindex(['five','two', 'three', 'six', 'four','one','seven'])
```

In [44]:
```
reindex_frame
```

Out[44]:

	year	state	pop	debt	greaterThan2
five	2002.0	Nevada	2.9	4.0	True
two	2001.0	Ohio	1.7	1.0	False
three	2002.0	Ohio	3.6	2.0	True
six	2003.0	Nevada	3.2	5.0	True
four	2001.0	Nevada	2.4	3.0	True
one	2000.0	Ohio	1.5	0.0	False
seven	NaN	NaN	NaN	NaN	NaN

The columns can be reindexed with the columns keyword:

In [45]:
```
reindex_frame = frame2.reindex(columns=['pop','year','imports', 'debt', 'state',"exports" ])
```

In [46]:
```
reindex_frame
```

Out[46]:

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

Dropping Entries from an Axis

In [47]:
```
reindex_frame
```

Out[47]:

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

In [48]:
```
row_dropped_frame = reindex_frame.drop(['three','six'])   # not dropping inplace
                                # by default dropping row labels axis =0
row_dropped_frame

# drop or delete mai farq hai
```

Out[48]:

	pop	year	imports	debt	state	exports
one	1.5	2000	NaN	0	Ohio	NaN
two	1.7	2001	NaN	1	Ohio	NaN
four	2.4	2001	NaN	3	Nevada	NaN
five	2.9	2002	NaN	4	Nevada	NaN

In [49]:
```
col_dropped_frame = reindex_frame.drop(['imports','exports'],axis=1)
col_dropped_frame
```

Out[49]:

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

Another Example

In [50]:
```
index = ['Firefox', 'Chrome', 'Safari', 'IE10', 'Konqueror']
df = pd.DataFrame({
    'http_status': [200,200,404,404,301],
    'response_time': [0.04, 0.02, 0.07, 0.08, 1.0]
    },index=index)
df
```

Out[50]:

	http_status	response_time
Firefox	200	0.04
Chrome	200	0.02
Safari	404	0.07
IE10	404	0.08
Konqueror	301	1.00

Create a new index and reindex the dataframe. By default values in the new index that do not have corresponding records in the dataframe are assigned `NaN` .

In [51]:
```
new_index= ['Safari', 'Iceweasel', 'Comodo Dragon', 'IE10','Chrome']
df.reindex(new_index)
```

Out[51]:

	http_status	response_time
Safari	404.0	0.07
Iceweasel	NaN	NaN
Comodo Dragon	NaN	NaN
IE10	404.0	0.08
Chrome	200.0	0.02

We can fill in the missing values by passing a value to the keyword `fill_value` . Because the index is not monotonically increasing or decreasing, we cannot use arguments to the keyword `method` to fill the `NaN` values.

In [52]:
```
df.reindex(new_index, fill_value=0)
```

Out[52]:

	http_status	response_time
Safari	404	0.07
Iceweasel	0	0.00
Comodo Dragon	0	0.00
IE10	404	0.08
Chrome	200	0.02

In [53]:
```
df.reindex(new_index, fill_value='missing')
```

Out[53]:

	http_status	response_time
Safari	404	0.07
Iceweasel	missing	missing
Comodo Dragon	missing	missing
IE10	404	0.08
Chrome	200	0.02

In [54]:
```
#We can also reindex the columns.

df.reindex(columns=['http_status', 'user_agent'])
```

Out[54]:

	http_status	user_agent
Firefox	200	NaN
Chrome	200	NaN
Safari	404	NaN
IE10	404	NaN
Konqueror	301	NaN

In [55]:
```
# Or we can use "axis-style" keyword arguments
df.reindex(['http_status', 'user_agent'], axis="columns")
```

Out[55]:

	http_status	user_agent
Firefox	200	NaN
Chrome	200	NaN
Safari	404	NaN
IE10	404	NaN
Konqueror	301	NaN

To further illustrate the filling functionality in `reindex`, we will create a dataframe with a monotonically increasing index (for example, a sequence of dates)

In [56]:
```
# create index through date, pehli sai shuru karo or period 6 honge or D means Day-wise karna
date_index = pd.date_range('1/1/2010', periods=6, freq='D')

df2 = pd.DataFrame({"prices": [100, 101, np.nan, 100, 89, 88]},index=date_index)
df2
```

Out[56]:

	prices
2010-01-01	100.0
2010-01-02	101.0
2010-01-03	NaN
2010-01-04	100.0
2010-01-05	89.0
2010-01-06	88.0

Suppose we decide to expand the dataframe to cover a wider date range.

In [57]:
```
date_index2 = pd.date_range('12/29/2009', periods=10, freq='D')
df2.reindex(date_index2)
```

Out[57]:

	prices
2009-12-29	NaN
2009-12-30	NaN
2009-12-31	NaN
2010-01-01	100.0
2010-01-02	101.0
2010-01-03	NaN
2010-01-04	100.0
2010-01-05	89.0
2010-01-06	88.0
2010-01-07	NaN

The index entries that did not have a value in the original data frame (for example, '2009-12-29') are by default filled with `NaN`. If desired, we can fill in the missing values using one of several options.

For example, to back-propagate the last valid value to fill the `NaN` values, pass `bfill` as an argument to the `method` keyword.

In [58]:
```
df2.reindex(date_index2, method="bfill")
```

Out[58]:

	prices
2009-12-29	100.0
2009-12-30	100.0
2009-12-31	100.0
2010-01-01	100.0
2010-01-02	101.0
2010-01-03	NaN
2010-01-04	100.0
2010-01-05	89.0
2010-01-06	88.0
2010-01-07	NaN

Please note that the `NaN` value present in the original dataframe (at index value 2010-01-03) will not be filled by any of the value propagation schemes. This is because filling while reindexing does not look at dataframe values, but only compares the original and desired indexes. If you do want to fill in the `NaN` values present in the original dataframe, use the `fillna()` method.

Indexing, Selection, and Filtering

In [69]:
```
data = pd.DataFrame(np.arange(40).reshape((10, 4)),
    index=['Ohio', 'Colorado', 'Washington','Nebraska','Utah', 'New York','California', 'Texas', 'Georgia',
'Alaska'],
    columns=['Jan', 'Feb', 'Mar', 'Apr'])
data
```

Out[69]:

	Jan	Feb	Mar	Apr
Ohio	0	1	2	3
Colorado	4	5	6	7
Washington	8	9	10	11
Nebraska	12	13	14	15
Utah	16	17	18	19
New York	20	21	22	23
California	24	25	26	27
Texas	28	29	30	31
Georgia	32	33	34	35
Alaska	36	37	38	39

In [70]:
```
# getting a single col
data['Jan']
```

Out[70]:
```
Ohio          0
Colorado      4
Washington    8
Nebraska     12
Utah         16
New York     20
California   24
Texas        28
Georgia      32
Alaska       36
Name: Jan, dtype: int32
```

In [71]:
```
#getting multiple cols
data[['Jan', 'Apr']]
```

Out[71]:

	Jan	Apr
Ohio	0	3
Colorado	4	7
Washington	8	11
Nebraska	12	15
Utah	16	19
New York	20	23
California	24	27
Texas	28	31
Georgia	32	35
Alaska	36	39

In [72]:
```
#integer based
data[:2]  #slicing rows starts from 0 & take two rows

# NOTE:  end parameter is exclusive in integer based
```

Out[72]:

	Jan	Feb	Mar	Apr
Ohio	0	1	2	3
Colorado	4	5	6	7

In [73]:
```
#label based
data["Utah":"Texas"]  #slicing rows starts from "Utah" & goto "Texas"

# NOTE:  end parameter is not exclusive as in label based
```

Out[73]:

	Jan	Feb	Mar	Apr
Utah	16	17	18	19
New York	20	21	22	23
California	24	25	26	27
Texas	28	29	30	31

In [74]:
```
data[2:6,0:2]   # Slicing Subsets of Rows and Columns either by label index
                # or by integer indexing is not possible, we have some other sol

# row and cols ko provide karke mein frame main sai frame ko extract karun chaahe wo label index sai ho ya in
teger
# index sai ho wo possible nhi hai
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-74-59af7fe07849> in <module>
----> 1 data[2:6,0:2]   # Slicing Subsets of Rows and Columns either by label index
      2                 # or by integer indexing is not possible, we have some other sol

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\frame.py in __getitem__(self, key)
   2798             if self.columns.nlevels > 1:
   2799                 return self._getitem_multilevel(key)
-> 2800             indexer = self.columns.get_loc(key)
   2801             if is_integer(indexer):
   2802                 indexer = [indexer]

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\indexes\base.py in get_loc(self, key, method, toleranc
e)
   2644                 )
   2645             try:
-> 2646                 return self._engine.get_loc(key)
   2647             except KeyError:
   2648                 return self._engine.get_loc(self._maybe_cast_indexer(key))

pandas\_libs\index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas\_libs\index.pyx in pandas._libs.index.IndexEngine.get_loc()

TypeError: '(slice(2, 6, None), slice(0, 2, None))' is an invalid key
```

In [75]:
```
data["Utah":"Texas", "Jan":'Mar']    # Slicing Subsets of Rows and Columns either by label index
                                     # or by integer indexing isnot possible, we have some other sol
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-75-c5ae50bb7ba0> in <module>
----> 1 data["Utah":"Texas", "Jan":'Mar']    # Slicing Subsets of Rows and Columns either by label index
      2                                     # or by integer indexing isnot possible, we have some other sol

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\frame.py in __getitem__(self, key)
   2798             if self.columns.nlevels > 1:
   2799                 return self._getitem_multilevel(key)
-> 2800             indexer = self.columns.get_loc(key)
   2801             if is_integer(indexer):
   2802                 indexer = [indexer]

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\indexes\base.py in get_loc(self, key, method, toleranc
e)
   2644                 )
   2645             try:
-> 2646                 return self._engine.get_loc(key)
   2647             except KeyError:
   2648                 return self._engine.get_loc(self._maybe_cast_indexer(key))

pandas\_libs\index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas\_libs\index.pyx in pandas._libs.index.IndexEngine.get_loc()

TypeError: '(slice('Utah', 'Texas', None), slice('Jan', 'Mar', None))' is an invalid key
```

In [76]:
```
# NOTE: frame mai sai frame slice karte hue hamein loc (label base) or iloc (integer base) use karna
# ..parega warna slice nhi hoga error ayega, or loc mai last index inclusive hoga or iloc mai last
# ..index exclusive hoga
```

We can select specific ranges of our data in both the row and column directions using either label or integer-based indexing.

loc is primarily label based indexing. Integers may be used but they are interpreted as a label.

iloc is primarily integer based indexing To select a subset of rows and columns from our DataFrame, we can use the iloc method.

In [77]:
```
# use if loc (label based)

data.loc["Utah":"Texas", "Jan":'Mar']
```

Out[77]:

	Jan	Feb	Mar
Utah	16	17	18
New York	20	21	22
California	24	25	26
Texas	28	29	30

In [78]:
```
#use if iloc (integer based)

data.iloc[2:6,0:2]
```

Out[78]:

	Jan	Feb
Washington	8	9
Nebraska	12	13
Utah	16	17
New York	20	21

In [79]:
```
# DF.div(n)

a = pd.DataFrame({"p":[2,4,6]})   # frame / 2
a.div(2)
```

Out[79]:

	p
0	1.0
1	2.0
2	3.0

In [81]:
```
# DF.rdiv(n)
# reverse
a = pd.DataFrame({"p":[2,4,6]})    # 2 / frame
a.rdiv(2)
```

Out[81]:

	p
0	1.000000
1	0.500000
2	0.333333

In [82]:
```
# select all the data from the month of march that have value greater than 15

data['Mar'] > 15
```

Out[82]:
```
Ohio          False
Colorado      False
Washington    False
Nebraska      False
Utah           True
New York       True
California     True
Texas          True
Georgia        True
Alaska         True
Name: Mar, dtype: bool
```

In [83]:
```
data[data['Mar'] > 20]
```

Out[83]:

	Jan	Feb	Mar	Apr
New York	20	21	22	23
California	24	25	26	27
Texas	28	29	30	31
Georgia	32	33	34	35
Alaska	36	37	38	39

In [84]:
```
data[data < 5] = 0
data
```

Out[84]:

	Jan	Feb	Mar	Apr
Ohio	0	0	0	0
Colorado	0	5	6	7
Washington	8	9	10	11
Nebraska	12	13	14	15
Utah	16	17	18	19
New York	20	21	22	23
California	24	25	26	27
Texas	28	29	30	31
Georgia	32	33	34	35
Alaska	36	37	38	39

Function Application and Mapping

functions ko apply karen or usko data ke saath map karden, function banayen, or us functino ko data ke saathmap karden, yaani wo function automaticaaly hamare saare data par apply hojaye, ye kaam hum aam tor par for loop ke through karte hain

In [85]:
```
frame = np.abs(
                pd.DataFrame(
                        np.random.randn(4, 3),
                        columns=list('bde'),
                        index=['Utah', 'Ohio', 'Texas', 'Oregon']))
frame
```

Out[85]:

	b	d	e
Utah	0.727765	1.314090	0.130133
Ohio	0.078893	0.157854	0.437879
Texas	2.351619	0.956398	1.405248
Oregon	0.197710	0.006349	0.678122

In [86]:
```
# x is the variable in which oura data will come, and what we have to with data is x.max() - x.min()

f = lambda x: x.max() - x.min()  # subtract the min value of each col from max of each col
```

In [88]:
```
frame.apply(f,axis="rows")  # row or 0 for each row wise
```

Out[88]:
```
b    2.272726
d    1.307741
e    1.275115
dtype: float64
```

In [93]:
```
print(2.351619 - 0.078893)
print(1.314090 - 0.006349)
print(1.405248 - 0.130133)
```

```
2.272726
1.307741
1.275115
```

In [94]:
```
frame.apply(f,axis="columns")  # columns or 1 for each col wise
```

Out[94]:
```
Utah      1.183956
Ohio      0.358987
Texas     1.395221
Oregon    0.671773
dtype: float64
```

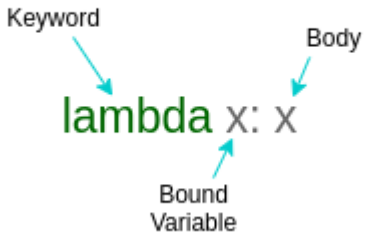
In [98]:
```
print(1.314090 - 0.130133)
print(0.437879 - 0.078893)
print(2.351619 - 0.956398)
print(0.678122 - 0.006349)
```

```
1.183957
0.358986
1.3952209999999998
0.671773
```

What are Lambda Functions?

A ***lambda*** *function is a small function containing a single expression. Lambda functions can also act as anonymous functions where they don't require any name. These are very helpful when we have to perform small tasks with less code.*

Lambda functions are handy and used in many programming languages but we'll be focusing on using them in Python here. In Python, lambda functions have the following syntax:



IIFEs using lambda functions

IIFEs are Immediately Invoked Function Expressions. These are functions that are executed as soon as they are created. IIFEs require no explicit call to invoke the function. In Python, IIFEs can be created using the lambda function.

Here, created an IIFE that returns the cube of a number:

```
In [100]: (Lambda x: x*x*x)(10)
```

```
Out[100]: 1000
```

```
In [101]: # awesome
```

Application of Lambda Functions with Different Functions

created a random dataset that contains information about a family of 5 people with their id, names, ages, and income per month. I will be using this dataframe to show you how to apply lambda functions using different functions on a dataframe in Python.

In [117]:
```
df=pd.DataFrame({
                 'id':[1,2,3,4,5],
                 'name':['Asad','Saad','Numi','Roman','Maria'],
                 'age':[20,25,15,10,30],
                 'income':[4000,7000,200,0,10000]})
df
```

Out[117]:

	id	name	age	income
0	1	Asad	20	4000
1	2	Saad	25	7000
2	3	Numi	15	200
3	4	Roman	10	0
4	5	Maria	30	10000

In [118]:
```
# Without lambda we can do this by either of this approach:

# df['age'] = df['age'] + 3   # it will also work
df['age'] = [ age + 3 for age in df['age'] ]
df
```

Out[118]:

	id	name	age	income
0	1	Asad	23	4000
1	2	Saad	28	7000
2	3	Numi	18	200
3	4	Roman	13	0
4	5	Maria	33	10000

Application of Lambda with Apply

Let's say we have got an error in the age variable. We recorded ages with a difference of 3 years. So, to remove this error from the Pandas dataframe, we have to add three years to every person's age. We can do this with the ***apply() function*** *in Pandas.*

apply() function *in Pandas calls the lambda function and applies it to every row or column of the dataframe and returns a modified copy of the dataframe:*

In [119]:
```
df['age']=df.apply(lambda x: x['age']+3,axis='columns')  # on frame
```

In [120]:
```
df
```

Out[120]:

	id	name	age	income
0	1	Asad	26	4000
1	2	Saad	31	7000
2	3	Numi	21	200
3	4	Roman	16	0
4	5	Maria	36	10000

In [121]:
```
df['age']=df['age'].apply(lambda x: x+3) #on particular series
```

In [122]:
```
df
```

Out[122]:

	id	name	age	income
0	1	Asad	29	4000
1	2	Saad	34	7000
2	3	Numi	24	200
3	4	Roman	19	0
4	5	Maria	39	10000

Application of Lambda with Filter

Now, let's see how many of these people are above the age of 18.

*We can do this using the **filter() function**.*

*The **filter() function** takes a lambda function and a Pandas series and applies the lambda function on the series and filters the data.*

In [123]:
```
list(filter(lambda x: x>18, df['age']))
```

Out[123]:
```
[29, 34, 24, 19, 39]
```

Application of Lambda with Map

You'll be able to relate to the next statement. □ It's performance appraisal time and the income of all the employees gets increased by 20%. This means we have to increase the salary of each person by 20% in our Pandas dataframe.

We can do this using the map() function. This map() function maps the series according to input correspondence. It is very helpful when we have to substitute a series with other values.

In [124]:
```
df['income']=list(map(lambda x: int(x+x*0.2),df['income']))

df
```

Out[124]:

	id	name	age	income
0	1	Asad	29	4800
1	2	Saad	34	8400
2	3	Numi	24	240
3	4	Roman	19	0
4	5	Maria	39	12000

Conditional Statements using Lambda Functions

Lambda functions also support conditional statements, such as if..else. This makes lambda functions very powerful.

Let's say in the family dataframe we have to categorize people into 'Adult' or 'Child'. For this, we can simply apply the lambda function to our dataframe:

In [125]:
```
df['category']=df['age'].apply(lambda x: 'Adult' if x>=18 else 'Child')

df
```

Out[125]:

	id	name	age	income	category
0	1	Asad	29	4800	Adult
1	2	Saad	34	8400	Adult
2	3	Numi	24	240	Adult
3	4	Roman	19	0	Adult
4	5	Maria	39	12000	Adult

Lambda with Reduce

Now, let's see the total income of the family. To calculate this, we can use the reduce() function in Python. It is used to apply a particular function to the list of elements in the sequence. The reduce() function is defined in the 'functools' module.

For using the reduce() function, we have to import the functools module first:

In [126]:
```
import functools

functools.reduce(lambda a,b: a+b, df['income'])
```

Out[126]: 25440

Summarizing and Computing Descriptive Statistics

In [127]:
```
#do your self
```

Correlation and Covariance

study link:https://machinelearningmastery.com/how-to-use-correlation-to-understand-the-relationship-between-variables/ (https://machinelearningmastery.com/how-to-use-correlation-to-understand-the-relationship-between-variables/)

Example:

In [ ]:
```
# In here we have installed pandas module datareader  by using !pip install pandas_datareader
```

In [130]:
```
import pandas_datareader.data as web
```

In [131]:
```
# dictionary comprehension

all_data = {ticker: web.get_data_yahoo(ticker) for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG']}
```

In [133]:
```
all_data['AAPL']
```

Out[133]:

Date	High	Low	Open	Close	Volume	Adj Close
2016-01-28	23.629999	23.097500	23.447500	23.522499	222715200.0	21.792925
2016-01-29	24.334999	23.587500	23.697500	24.334999	257666000.0	22.545681
2016-02-01	24.177500	23.850000	24.117500	24.107500	163774000.0	22.334911
2016-02-02	24.010000	23.570000	23.855000	23.620001	149428800.0	21.883259
2016-02-03	24.209999	23.520000	23.750000	24.087500	183857200.0	22.316381
...	...	...	...	...	...	...
2021-01-20	132.490005	128.550003	128.660004	132.029999	104319500.0	132.029999
2021-01-21	139.669998	133.589996	133.800003	136.869995	120529500.0	136.869995
2021-01-22	139.850006	135.020004	136.279999	139.070007	113907200.0	139.070007
2021-01-25	145.089996	136.539993	143.070007	142.919998	157611700.0	142.919998
2021-01-26	144.300003	141.369995	143.600006	142.989899	76091575.0	142.989899

1258 rows × 6 columns

In [132]:
```
frame
```

Out[132]:

	b	d	e
Utah	0.727765	1.314090	0.130133
Ohio	0.078893	0.157854	0.437879
Texas	2.351619	0.956398	1.405248
Oregon	0.197710	0.006349	0.678122

In [134]:
```
framing =frame.rename(columns={'b':"butterfly", 'd':"dog",'c':"Cat"},
                      index = {"Utah":22,"Ohio":33,"Texas":44,"Oregon":55})
framing
```

Out[134]:

	butterfly	dog	e
22	0.727765	1.314090	0.130133
33	0.078893	0.157854	0.437879
44	2.351619	0.956398	1.405248
55	0.197710	0.006349	0.678122

In [135]:
```
framing.insert(1,'for', np.abs(framing["e"]))
```

In [136]:
```
framing
```

Out[136]:

	butterfly	for	dog	e
22	0.727765	0.130133	1.314090	0.130133
33	0.078893	0.437879	0.157854	0.437879
44	2.351619	1.405248	0.956398	1.405248
55	0.197710	0.678122	0.006349	0.678122

In [ ]:

In [ ]: