```
#ser1 = pd.Series([4, 7, -5, 3],index = [1,2,3,4])
ser1 = pd.Series([4, 7, -5, 3])

ser1
#for i in ser1.index:
#    if i == 3:
#        ser1[i] = 50
#print(ser1)
0    4
1    7
2   -5
3    3
dtype: int64
```

Getting Started with Pandas:

At the very basic level, Pandas objects can be thought of as enhanced versions of NumPy structured arrays in which the rows and columns are identified with labels rather than simple integer indices

It contains high-level data structures and manipulation tools designed to make data analysis fast and easy in Python. pandas is built on top of NumPy and makes it easy to use in NumPy-centric applications.

To get started with pandas, you will need to get comfortable with its three data structures:

```
1. Series
2. DataFrame
3. Index
```

Series

A Series is a one-dimensional array-like object containing an array of data and an associated array of data labels, called its index. The simplest Series is formed from only an array of data:

```
import pandas as pd
ser = pd.Series([4, 7, -5, 3])
print(ser)
0    4
1    7
2   -5
3    3
dtype: int64
```

Above output shows the index on the left and the values on the right. Since we did not specify an index for the data, a default one consisting of the integers 0 through N - 1 (where N is the length of the data) is created. You can get the array representation and index object of the Series via its values and index attributes, respectively.

the Series wraps both a sequence of values and a sequence of indices, which we can access with the values and index attributes:

```
#The values are simply a familiar NumPy array:
```

```
print("The Values of Series are: \n",ser.values)


#The index is an array-like object of type pd.Index
print("The index in Series are: \n",ser.index)
The Values of Series are:
 [ 4  7 -5  3]
The index in Series are:
 RangeIndex(start=0, stop=4, step=1)
#Like with a NumPy array, data can be accessed by the associated
#index via the familiar Python square-bracket notation:

print(ser[1])

print(ser[1:3])
7
1    7
2   -5
dtype: int64
```

From what we've seen so far, it may look like the Series object is basically interchangeable with a one-dimensional NumPy array. The essential difference is the presence of the index: while the Numpy Array has an implicitly defined integer index used to access the values, the Pandas Series has an explicitly defined index associated with the values.

This explicit index definition gives the Series object additional capabilities. For example, the index need not be an integer, but can consist of values of any desired type. For example, if we wish, we can use strings as an index:

Often it will be desirable to create a Series with an index identifying each data point:

```
import pandas as pd
ser1 = pd.Series([4, 7, -5, 3],index = ['Bob','Joe','Will','Sam'])
#ser1 = pd.Series([4, 7, -5, 3],index = [1,2,3,4])
print("Customized index example: \n",ser1)

print("The index in Series are: \n",ser1.index)
Customized index example:
 Bob     4
Joe     7
Will   -5
Sam     3
dtype: int64
The index in Series are:
 Index(['Bob', 'Joe', 'Will', 'Sam'], dtype='object')
print(ser1['Bob'])
print(ser1[0])
4
4
#Compared with a regular NumPy array,
#you can use values in the index when selecting single values
#or a set of values:

print("Selecting single value from Series :",ser1['Bob'])
print("Selecting single value from Series :",ser1[0])
```

```
print("Selecting multiple values from Series
:\n",ser1[['Bob','Joe','Will']])
Selecting single value from Series : 4
Selecting single value from Series : 4
Selecting multiple values from Series :
 Bob     4
Joe     7
Will   -5
dtype: int64
#NumPy array operations, such as filtering with a boolean array, scalar
multiplication,
#or applying math functions, will preserve the index-value link:

print("Fetch all positive values from Series: \n",ser1[ser1>0])

print("Scalar multiplication will preserve index :\n",ser1*2)
Fetch all positive values from Series:
 Bob    4
Joe    7
Sam    3
dtype: int64
Scalar multiplication will preserve index :
 Bob      8
Joe     14
Will   -10
Sam      6
dtype: int64
#Another way to think about a Series is as a fixed-length, ordered dict, as
it is a mapping
#of index values to data values. It can be substituted into many functions
that expect a dict:

print("Is Bob present in Series : ",'Bob' in ser1)
print("Is Bob1 present in Series : ",'Bob1' in ser1)
Is Bob present in Series :  True
Is Bob1 present in Series :  False
print("keys are: ",ser1.keys())
print("Index are: ", ser1.index)
keys are:  Index(['Bob', 'Joe', 'Will', 'Sam'], dtype='object')
Index are:  Index(['Bob', 'Joe', 'Will', 'Sam'], dtype='object')
ser.values
array([ 4,  7, -5,  3], dtype=int64)
list(ser1.items())
[('Bob', 4), ('Joe', 7), ('Will', -5), ('Sam', 3)]
#sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000,
#         'Utah': 5000,'Ohio': 50000}
#sdata
{'Ohio': 50000, 'Oregon': 16000, 'Texas': 71000, 'Utah': 5000}
#pd.Series([1,2,3,4],index = ['a','a','a','b'])
a    1
a    2
a    3
b    4
dtype: int64
#you can create a Series from it by passing the dict:

sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000,
         'Utah': 5000}
#sdata = {1: 35000, 2: 71000, 3: 16000, 5: 5000}

print("Type of sdata: ", type(sdata))
```

```
ser3 = pd.Series(sdata)

print("Series is : \n",ser3)
print("Type if series :",type(ser3))
Type of sdata:  <class 'dict'>
Series is :
 Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
dtype: int64
Type if series : <class 'pandas.core.series.Series'>
pd.Series(sdata,index = ['Ohio','Texas','Oregon','Utah'])
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
#By default, a Series will be created where the index is drawn from
#the sorted keys.
#From here, typical dictionary-style item access can be performed:

ser3['Ohio']
35000
#Unlike a dictionary, though, the Series also supports array-style
#operations such as slicing:
print(ser3)
print(ser3['Ohio':'Texas'])
print(ser3[0:3])
Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
dtype: int64
Ohio      35000
Oregon    16000
Texas     71000
dtype: int64
Ohio      35000
Oregon    16000
Texas     71000
dtype: int64
sdata['Ohio':'Texas']
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-7-b1ef054ad0c3> in <module>()
----> 1 sdata['Ohio':'Texas']

TypeError: unhashable type: 'slice'
#When only passing a dict, the index in the resulting Series
#will have the dict’s keys in sorted order.
#where index is an optional argument

states = ['California', 'Ohio', 'Oregon', 'Texas']
ser4 = pd.Series(sdata, index=states)
print("Original Dict: \n", sdata)
print("New Series:\n",ser4)
Original Dict:
 {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
New Series:
 California        NaN
```

```
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

In this case, 3 values found in sdata were placed in the appropriate locations, but since no value for 'California' was found, it appears as NaN (not a number) which is considered in pandas to mark missing or NA values.The isnull and notnull functions in pandas should be used to detect missing data:

```
print("Use isnull() :\n",pd.isnull(ser4))

print("Use notnull() :\n",pd.notnull(ser4))

#Series also has these as instance method:
print("Use isnull() as instance method :\n",ser4.isnull())
print("Use notnull() as instance method :\n",ser4.notnull())
Use isnull() :
 California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
Use notnull() :
 California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
Use isnull() as instance method :
 California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
Use notnull() as instance method :
 California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
pd.Series([1,1.2,3])
0    1.0
1    1.2
2    3.0
dtype: float64
#A critical Series feature for many applications is that it automatically
aligns differently indexed
#data in arithmetic operations:

print("First Series: \n",ser3)
print("Second Series: \n",ser4)

print("Arithmetic Operation - Addition:\n",ser3+ser4)
First Series:
 Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
dtype: int64
```

```
Second Series:
 California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
Arithmetic Operation - Addition:
 California         NaN
Ohio           70000.0
Oregon         32000.0
Texas         142000.0
Utah               NaN
dtype: float64
#Both the Series object itself and its index have a name attribute, which
integrates with
#other key areas of pandas functionality:

ser4.name = 'population'
ser4.index.name = 'state'

print(ser4)
state
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
Name: population, dtype: float64
ser4.index[1]
'Ohio'
ser4.index[1] = ['Ohio1']
print(ser4)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-68-c5b78d377d77> in <module>()
----> 1 ser4.index[1] = ['Ohio1']
      2 print(ser4)

C:\Users\manish.khati\AppData\Local\Continuum\Anaconda3\lib\site-
packages\pandas\core\indexes\base.py in __setitem__(self, key, value)
   1618
   1619     def __setitem__(self, key, value):
-> 1620         raise TypeError("Index does not support mutable
operations")
   1621
   1622     def __getitem__(self, key):

TypeError: Index does not support mutable operations
ser4.index = ['California','Ohio1','Oregon','Texas']
print(ser4)
California        NaN
Ohio1         35000.0
Oregon        16000.0
Texas         71000.0
Name: population, dtype: float64
```

Dataframe

DataFrames are essentially multidimensional arrays with attached row and column labels, and often with heterogeneous types and/or missing data. As well as offering a convenient storage interface for labeled data, Pandas implements a number of powerful data operations familiar to users of both database frameworks and spreadsheet programs.

A DataFrame represents a tabular, spreadsheet-like data structure containing 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.

There are numerous ways to construct a DataFrame, though one of the most common is from a dict of equal-length lists or NumPy arrays

```
import pandas as pd

data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
'year': [2000, 2001, 2002, 2001, 2002],
'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}
frame = pd.DataFrame(data)
print(frame)

#The resulting DataFrame will have its index assigned automatically as with
Series,
#and the columns are placed in sorted order:
   pop   state  year
0  1.5    Ohio  2000
1  1.7    Ohio  2001
2  3.6    Ohio  2002
3  2.4  Nevada  2001
4  2.9  Nevada  2002
#If you specify a sequence of columns, the DataFrame’s columns will be
exactly what you pass:

print(pd.DataFrame(data, columns=['year', 'state', 'pop']))
   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
#Like the Series object, the DataFrame has an index attribute that gives
access to the index labels:

print(frame.index)
RangeIndex(start=0, stop=5, step=1)
#Additionally, the DataFrame has a columns attribute, which is an Index
object holding the column labels:
frame.columns
Index(['pop', 'state', 'year'], dtype='object')
#As with Series, if you pass a column that isn’t contained in data, it will
appear with NA
#values in the result:

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

print(frame2)
       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
#A column in a DataFrame can be retrieved as a Series either by dict-like
notation or by attribute:

print(frame2['state'])
print("Another Way of retrieving: \n",frame2.state)
one        Ohio
two        Ohio
three      Ohio
four     Nevada
five     Nevada
Name: state, dtype: object
Another Way of retrieving:
 one        Ohio
two        Ohio
three      Ohio
four     Nevada
five     Nevada
Name: state, dtype: object
```

Though this is a useful shorthand, keep in mind that it does not work for all cases! For example, if the column names are not strings, or if the column names conflict with methods of the DataFrame, this attribute-style access is not possible. For example, the DataFrame has a pop() method, so data.pop will point to this rather than the "pop" column.

Though this is a useful shorthand, keep in mind that it does not work for all cases! For example, if the column names are not strings, or if the column names conflict with methods of the DataFrame, this attribute-style access is not possible. For example, the DataFrame has a pop() method, so data.pop will point to this rather than the "pop" column

```
print(frame2['pop'])
one      1.5
two      1.7
three    3.6
four     2.4
five     2.9
Name: pop, dtype: float64
#DataFrame has a pop() method, so data.pop will point to this rather than
the "pop" column.
print("Another Way of retrieving: \n",frame2.pop)
Another Way of retrieving:
 <bound method NDFrame.pop of        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>
frame2.index
Index(['one', 'two', 'three', 'four', 'five'], dtype='object')
#Rows can also be retrieved by position or name by a couple of methods,
such as the
#loc indexing field:

print("Dataframe: \n",frame2)

print("Fetch the third element from the DF :\n",frame2.loc['three'])

print("Fetch the third element from the DF :\n",frame2.iloc[2])
Dataframe:
```

```
        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
Fetch the third element from the DF :
 year     2002
state    Ohio
pop       3.6
debt      NaN
Name: three, dtype: object
Fetch the third element from the DF :
 year     2002
state    Ohio
pop       3.6
debt      NaN
Name: three, dtype: object
```

The Pandas Index Object

We have seen here that both the Series and DataFrame objects contain an explicit index that lets you reference and modify data. This Index object is an interesting structure in itself, and it can be thought of either as an immutable array or as an ordered set (technically a multi-set, as Index objects may contain repeated values). Those views have some interesting consequences in the operations available on Index objects. As a simple example, let's construct an Index from a list of integers:

Index Object

pandas’s Index objects are responsible for holding the axis labels and other metadata(like the axis name or names).

```
import pandas as pd
ind = pd.Index([2, 3, 5, 7, 11])
ind
Int64Index([2, 3, 5, 7, 11], dtype='int64')
obj = pd.Series(range(3), index=['a', 'b', 'c'])

index1 = obj.index
print(index1)
Index(['a', 'b', 'c'], dtype='object')
#Index as immutable array

#The Index in many ways operates like an array. For example, we can use
standard Python
#indexing notation to retrieve values or slices:

print(ind[1])
print(ind[::2])
3
Int64Index([2, 5, 11], dtype='int64')
#Index objects also have many of the attributes familiar from NumPy arrays:
print(ind.size, ind.shape, ind.ndim, ind.dtype)
5 (5,) 1 int64
```

```
#One difference between Index objects and NumPy arrays is that indices are
immutable-that is,
#they cannot be modified via the normal means:
ind[1] = 0
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-24-8c2d7a24abd9> in <module>()
      1 #One difference between Index objects and NumPy arrays is that
indices are immutable-that is,
      2 #they cannot be modified via the normal means:
----> 3 ind[1] = 0

C:\Users\manish.khati\AppData\Local\Continuum\Anaconda3\lib\site-
packages\pandas\core\indexes\base.py in __setitem__(self, key, value)
   1618
   1619     def __setitem__(self, key, value):
-> 1620         raise TypeError("Index does not support mutable
operations")
   1621
   1622     def __getitem__(self, key):

TypeError: Index does not support mutable operations
import numpy as np
#Columns can be modified by assignment. For example, the empty 'debt'
column could
#be assigned a scalar value or an array of values:
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
'year': [2000, 2001, 2002, 2001, 2002],
'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}

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

print(frame2)

frame2['debt'] = 16.5
print(frame2)

frame2['debt'] = np.arange(5)
print(frame2)
       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
       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
       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
#When assigning lists or arrays to a column, the value’s length must match
the length
```

```
#of the DataFrame. If you assign a Series, it will be instead conformed exactly to the
#DataFrame’s index, inserting missing values in any holes:

val = pd.Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])
#val1 = pd.Series([-1.2, -1.5, -1.7])

frame2['debt'] = val
print(frame2)
       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
#Assigning a column that doesn’t exist will create a new column. The del keyword will
#delete columns as with a dict:

frame2['eastern'] = frame2.state == 'Ohio'
print(frame2)

#del (frame2.eastern) chk
del frame2['eastern']
print(frame2)
       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
       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
##to check if "debt" column is present in DF
print("Is \"debt\" present in frame2 Dataframe: ",
      'debt' in frame2.columns)
print("Is \"debt1\" present in frame2 Dataframe: ",
      'debt1' in frame2.columns)

##to check if "two" row index is present in DF
print("Is \"two\" present in frame2 Dataframe: ",
      'two' in frame2.index)
Is "debt" present in frame2 Dataframe:  True
Is "debt1" present in frame2 Dataframe:  False
Is "two" present in frame2 Dataframe:  True
```

Operating on Data in Pandas

One of the essential pieces of NumPy is the ability to perform quick element-wise operations, both with basic arithmetic (addition, subtraction, multiplication, etc.) and with more sophisticated operations (trigonometric functions, exponential and logarithmic functions, etc.). Pandas inherits much of this functionality from NumPy, and the ufuncs

```
#Index Preservation
```

```
#Because Pandas is designed to work with NumPy, any NumPy ufunc will work
on Pandas Series and DataFrame objects.

import pandas as pd
import numpy as np

rng = np.random.RandomState(42)
ser = pd.Series(rng.randint(0, 10, 4))
ser
0    6
1    3
2    7
3    4
dtype: int32
df = pd.DataFrame(rng.randint(0, 10, (3, 4)),
                  columns=['A', 'B', 'C', 'D'])
df
```

	A	B	C	D
0	6	9	2	6
1	7	4	3	7
2	7	2	5	4

If we apply a NumPy ufunc on either of these objects, the result will be another Pandas object with the indices preserved:

```
np.exp(ser)
0     403.428793
1      20.085537
2    1096.633158
3      54.598150
dtype: float64
np.sin(df * np.pi / 4)
```

	A	B	C	D
0	-1.000000	7.071068e-01	1.000000	-1.000000e+00
1	-0.707107	1.224647e-16	0.707107	-7.071068e-01
2	-0.707107	1.000000e+00	-0.707107	1.224647e-16

UFuncs: Index Alignment

For binary operations on two Series or DataFrame objects, Pandas will align indices in the process of performing the operation. This is very convenient when working with incomplete data

```
#Index alignment in Series

area = pd.Series({'Alaska': 1723337, 'Texas': 695662,
```

```
                  'California': 423967})
population = pd.Series({'California': 38332521, 'Texas': 26448193,
                        'New York': 19651127})

print("First Series: \n",area)
print("Second Series: \n",population)
First Series:
 Alaska        1723337
California     423967
Texas          695662
dtype: int64
Second Series:
 California    38332521
New York      19651127
Texas         26448193
dtype: int64
#Let's see what happens when we divide these to compute the population
density:

print("population density :\n",population/area)
population density :
 Alaska              NaN
California    90.413926
New York            NaN
Texas         38.018740
dtype: float64
```

Essential Functionality:

```
1. Reindexing
2. Indexing, selection, Filtering
#1. Reindexing

obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])
print("Original Series : \n",obj)

##Calling reindex on this Series rearranges the data according to the new
index, introducing
##missing values if any index values were not already present:
obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])

print("Series After Reindexing: \n",obj2)
Original Series :
 d    4.5
b    7.2
a   -5.3
c    3.6
dtype: float64
Series After Reindexing:
 a   -5.3
b    7.2
c    3.6
d    4.5
e    NaN
dtype: float64
#Fill the missing value
obj3 = obj.reindex(['a', 'b', 'c', 'd', 'e'], fill_value=0)
print("Series After Reindexing and refilling: \n",obj3)
Series After Reindexing and refilling:
 a   -5.3
```

```
b    7.2
c    3.6
d    4.5
e    0.0
dtype: float64
#For ordered data like time series, it may be desirable to do some
interpolation or filling
#of values when reindexing. The method option allows us to do this, using a
method such
#as ffill which forward fills the values:

obj4 = pd.Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])
print("Original Series: \n",obj4)
print("FOrward fill: \n",obj4.reindex(range(6)))

print("FOrward fill: \n",obj4.reindex(range(6), method='ffill'))
print("Backward fill: \n",obj4.reindex(range(6), method='bfill'))
Original Series:
 0      blue
2    purple
4    yellow
dtype: object
FOrward fill:
 0      blue
1       NaN
2    purple
3       NaN
4    yellow
5       NaN
dtype: object
FOrward fill:
 0      blue
1      blue
2    purple
3    purple
4    yellow
5    yellow
dtype: object
Backward fill:
 0      blue
1    purple
2    purple
3    yellow
4    yellow
5       NaN
dtype: object
#With DataFrame, reindex can alter either the (row) index, columns, or
both. When
#passed just a sequence, the rows are reindexed in the result:

frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
                     index=['a', 'c', 'd'],
                  columns=['Ohio', 'Texas', 'California'])

print("Original Dataframe :\n",frame)

frame2 = frame.reindex(['a', 'b', 'c', 'd'])
print("Row Re-Index: \n",frame2)

states = ['Texas', 'Utah', 'California']
frame3 = frame.reindex(columns=states)
```

```
print("Column Re-Index: \n",frame3)
Original Dataframe :
    Ohio  Texas  California
a     0      1           2
c     3      4           5
d     6      7           8
Row Re-Index:
    Ohio  Texas  California
a   0.0    1.0         2.0
b   NaN    NaN         NaN
c   3.0    4.0         5.0
d   6.0    7.0         8.0
Column Re-Index:
    Texas  Utah  California
a      1   NaN           2
c      4   NaN           5
d      7   NaN           8
#Both can be reindexed in one shot, though interpolation will only apply
row-wise (axis0):
frame
frame4 =frame.reindex(index=['a', 'b', 'c', 'd'],columns=states)
print("REindexing on rows and column in one shot :\n",frame4)
frame5 = frame.reindex(index=['a', 'b', 'c', 'd'],
                       method = 'ffill')
print("REindexing with row interpolation :\n",frame5)
REindexing on rows and column in one shot :
    Texas  Utah  California
a    1.0   NaN         2.0
b    NaN   NaN         NaN
c    4.0   NaN         5.0
d    7.0   NaN         8.0
REindexing with row interpolation :
    Ohio  Texas  California
a     0      1           2
b     0      1           2
c     3      4           5
d     6      7           8
#Dropping entries from an axis

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

print("Original Series :\n",obj)
new_obj = obj.drop('c')
print("Series after drop :\n",new_obj)

#You can drop multiple index together
new_obj1 = obj.drop(['c','d'])
print("Series after  multiple drops :\n",new_obj1)
Original Series :
 a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64
Series after drop :
 a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

```
Series after  multiple drops :
 a    0.0
b    1.0
e    4.0
dtype: float64
#With DataFrame, index values can be deleted from either axis:
import pandas as pd
import numpy as np
data = pd.DataFrame(np.arange(16).reshape((4, 4)),
                 index=['Ohio', 'Colorado', 'Utah', 'New York'],
                 columns=['one', 'two', 'three', 'four'])

print("Original DataFrame :\n",data)

data1 = data.drop(['Colorado', 'Ohio'])
print("DataFrame after drop :\n",data1)

data2 = data.drop('two',axis=1)
print("DataFrame after drop :\n",data2)

data3 = data.drop(['two','three'], axis=1)
print("DataFrame after drop :\n",data3)
Original DataFrame :
           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
DataFrame after drop :
           one  two  three  four
Utah        8    9     10    11
New York   12   13     14    15
DataFrame after drop :
           one  three  four
Ohio        0      2     3
Colorado    4      6     7
Utah        8     10    11
New York   12     14    15
DataFrame after drop :
           one  four
Ohio        0     3
Colorado    4     7
Utah        8    11
New York   12    15
data2 = data.drop('Ohio',axis=1)
print("DataFrame after drop :\n",data2)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-19-58c8648f7994> in <module>()
----> 1 data2 = data.drop('Ohio',axis=1)
      2 print("DataFrame after drop :\n",data2)

C:\Users\manish.khati\AppData\Local\Continuum\Anaconda3\lib\site-packages\pandas\core\generic.py in drop(self, labels, axis, level, inplace, errors)
   2048                 new_axis = axis.drop(labels, level=level, errors=errors)
   2049             else:
-> 2050                 new_axis = axis.drop(labels, errors=errors)
   2051             dropped = self.reindex(**{axis_name: new_axis})
   2052             try:
```

```
C:\Users\manish.khati\AppData\Local\Continuum\Anaconda3\lib\site-
packages\pandas\core\indexes\base.py in drop(self, labels, errors)
   3573             if errors != 'ignore':
   3574                 raise ValueError('labels %s not contained in axis'
%
-> 3575                                  labels[mask])
   3576             indexer = indexer[~mask]
   3577         return self.delete(indexer)

ValueError: labels ['Ohio'] not contained in axis
#Drop a row if it contains a certain value

print("Original Data: \n",data)
data4 = data[(data.one != 0) & (data.two != 5)]
print("Dropped row with one = 0 : \n",data4)
Original Data:
           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
Dropped row with one = 0 :
           one  two  three  four
Utah         8    9     10    11
New York    12   13     14    15
#Drop a row by row number (index)
print("Original Data: \n",data)

data5 = data.drop(data.index[2])
print("Drop index 2: \n",data5)
Original Data:
           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
Utah
Drop index 2:
           one  two  three  four
Ohio         0    1      2     3
Colorado     4    5      6     7
New York    12   13     14    15
#can be extended to dropping a range

#Drop a row by row number (index)
print("Original Data: \n",data)

data6 = data.drop(data.index[[2,3]])
print("Drop index 2: \n",data6)
Original Data:
           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
Drop index 2:
           one  two  three  four
Ohio         0    1      2     3
Colorado     4    5      6     7
#dropping relative to the end of the DF.
```

```
print("Original Data: \n",data)

data7 = data.drop(data.index[-2])
print("Drop index 2: \n",data7)
Original Data:
           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
Drop index 2:
           one  two  three  four
Ohio         0    1      2     3
Colorado     4    5      6     7
New York    12   13     14    15
#select ranges relative to the top.

print("Original Data: \n",data)

data7 = data[:2]
print("Keep top 2: \n",data7)
Original Data:
           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
Keep top 2:
           one  two  three  four
Ohio         0    1      2     3
Colorado     4    5      6     7
Drop bottom 2:
           one  two  three  four
Ohio         0    1      2     3
Colorado     4    5      6     7
#2. Indexing, selection, Filtering

#Series indexing (obj[...]) works analogously to NumPy array indexing,
except you can
#use the Series’s index values instead of only integers.

obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])

print("Using Series Index: \n",obj['b'])
print("Using Integer Index: \n",obj[1])
Using Series Index:
 1.0
Using Integer Index:
 1.0
#Slicing with labels behaves differently than normal Python slicing in that
the endpoint
#is inclusive:

print("End Point is Inclusive in Series Index: \n",obj['b':'c'])
print("End Point Exclusive in integer Index: \n",obj[1:2])
End Point is Inclusive in Series Index:
 b    1.0
c    2.0
dtype: float64
End Point Exclusive in integer Index:
```

```
 b    1.0
dtype: float64
obj['b':'c'] = 5
print(obj)
a    0.0
b    5.0
c    5.0
d    3.0
dtype: float64
data = pd.DataFrame(np.arange(16).reshape((4, 4)),
                 index=['Ohio', 'Colorado', 'Utah', 'New York'],
                 columns=['one', 'two', 'three', 'four'])
print(data)

print(data['two'])

print(data[['three', 'one']])
          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
Ohio         1
Colorado     5
Utah         9
New York    13
Name: two, dtype: int32
          three  one
Ohio          2    0
Colorado      6    4
Utah         10    8
New York     14   12
#First selecting rows by slicing or a boolean array:

data[data['three'] > 5]
```

	one	two	three	four
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

```
data[(data['three'] > 5) & (data['one'] > 5)]
```

	one	two	three	four
Utah	8	9	10	11
New York	12	13	14	15

```
print(data.loc[['Colorado','Utah'], ['two', 'three']])
          two  three
Colorado    5      6
Utah        9     10
print(data.loc[data.three > 5,['two','three']])
          two  three
Colorado    5      6
Utah        9     10
New York   13     14
#Arithmetic and data alignment

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

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

print("s1 is: \n",s1)
print("s2 is: \n",s2)
print("s1 + s1 is: \n", s1+s2)
s1 is:
 a    7.3
c   -2.5
d    3.4
e    1.5
dtype: float64
s2 is:
 a   -2.1
c    3.6
e   -1.5
f    4.0
g    3.1
dtype: float64
s1 + s1 is:
 a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
list('bcd')
['b', 'c', 'd']
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 :\n",df1)
print("df2 :\n",df2)

print("df1+df2 :\n",df1+df2)
df3 = df1+df2
print("Df3: \n",df3)
df1 :
             b    c    d
Ohio      0.0  1.0  2.0
Texas     3.0  4.0  5.0
Colorado  6.0  7.0  8.0
df2 :
           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
df1+df2 :
             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
```

```
Df3:
             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
#Operations between DataFrame and Series

frame = pd.DataFrame(np.arange(12.).reshape((4, 3)),
                     columns=list('bde'),
                  index=['Utah', 'Ohio', 'Texas', 'Oregon'])

print("frmae:\n",frame)
series = frame.iloc[0]

print("Series:\n",series)
frmae:
           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
Series:
 b    0.0
d    1.0
e    2.0
Name: Utah, dtype: float64
#By default, arithmetic between DataFrame and Series matches the index of
the Series
#on the DataFrame's columns, broadcasting down the rows:

print("SUbtraction :\n",frame - series)
SUbtraction :
           b    d    e
Utah    0.0  0.0  0.0
Ohio    3.0  3.0  3.0
Texas   6.0  6.0  6.0
Oregon  9.0  9.0  9.0
#If an index value is not found in either the DataFrame's columns or the
Series's index,
#the objects will be reindexed to form the union:

series2 = pd.Series(range(3), index=['b', 'e', 'f'])

print("Addition :\n",frame + series2)
Addition :
           b   d     e   f
Utah    0.0 NaN   3.0 NaN
Ohio    3.0 NaN   6.0 NaN
Texas   6.0 NaN   9.0 NaN
Oregon  9.0 NaN  12.0 NaN
#Sorting and ranking

#Sorting a data set by some criterion is another important built-in
operation. To sort
#lexicographically by row or column index, use the sort_index method, which
returns
#a new, sorted object:

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

```
print("Sorted Object: \n",obj.sort_index())
Sorted Object:
 a    1
b    2
c    3
d    0
dtype: int32
#With a DataFrame, you can sort by index on either axis:

frame = pd.DataFrame(np.arange(8).reshape((2, 4)),
                     index=['three', 'one'],
                  columns=['d', 'a', 'b', 'c'])

print("Original Data Frame: \n",frame)

print("Sort :\n",frame.sort_index())

print("Sort axis = 1:\n",frame.sort_index(axis = 1))
Original Data Frame:
        d  a  b  c
three  0  1  2  3
one    4  5  6  7
Sort :
        d  a  b  c
one    4  5  6  7
three  0  1  2  3
Sort axis = 1:
        a  b  c  d
three  1  2  3  0
one    5  6  7  4
#The data is sorted in ascending order by default, but can be sorted in
descending order,too:

frame.sort_index(axis=1, ascending=False)
```

	d	c	b	a
three	0	3	2	1
one	4	7	6	5

```
#To sort a Series by its values, use its sort_values() method:

obj = pd.Series([4, 7, -3, 2])
print(obj)
print(obj.sort_values())
0    4
1    7
2   -3
3    2
dtype: int64
2   -3
3    2
0    4
1    7
dtype: int64
#Any missing values are sorted to the end of the Series by default:

obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])
print(obj.sort_values())
4   -3.0
5    2.0
```

```
0    4.0
2    7.0
1    NaN
3    NaN
dtype: float64
#On DataFrame, you may want to sort by the values in one or more columns.
To do so,
#pass one or more column names to the by option:

import pandas as pd
frame = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})

print("Frame :\n",frame)

print("Sorted Frame\n",frame.sort_values(by='b'))

print("Sorted Frame by multiple values\n",
      frame.sort_values(by=['a','b']))

frame2 = frame.sort_values(by=['a','b'])
print("Sorted New DF\n",frame2)
Frame :
    a  b
0  0  4
1  1  7
2  0 -3
3  1  2
Sorted Frame
    a  b
2  0 -3
3  1  2
0  0  4
1  1  7
Sorted Frame by multiple values
    a  b
2  0 -3
0  0  4
3  1  2
1  1  7
Sorted New DF
    a  b
2  0 -3
0  0  4
3  1  2
1  1  7
```