Pandas

In [1]:
```
# Importing the pandas
import pandas as pd
```

In [2]:
```
# Let's see the version of the Pandas using '__version__' mtehod
pd.__version__
```

Out[2]:
```
'0.23.4'
```

In [3]:
```
# to display all the contents of the pandas namespace, you can type this: pd.<TAB>
#pd.<delete this including'<>' and press tab>
#pd.
```

In [4]:
```
# To see the simple ducumentation use '?'
pd?
```

In [5]:
```
# To see the documentation in more detail we can use '??'
pd??
```

In [6]:
```
# We can also use 'help()' method to see the documentation of any package
help(pd)
```

```
Help on package pandas:

NAME
    pandas

DESCRIPTION
    pandas - a powerful data analysis and manipulation library for Python
    =====================================================================

    **pandas** is a Python package providing fast, flexible, and expressive data
    structures designed to make working with "relational" or "labeled" data both
    easy and intuitive. It aims to be the fundamental high-level building block for
    doing practical, **real world** data analysis in Python. Additionally, it has
    the broader goal of becoming **the most powerful and flexible open source data
    analysis / manipulation tool available in any language**. It is already well on
    its way toward this goal.

    Main Features
    -------------
    Here are just a few of the things that pandas does well:

      - Easy handling of missing data in floating point as well as non-floating
        point data.
      - Size mutability: columns can be inserted and deleted from DataFrame and
        higher dimensional objects
      - Automatic and explicit data alignment: objects can be explicitly aligned
        to a set of labels, or the user can simply ignore the labels and let
        `Series`, `DataFrame`, etc. automatically align the data for you in
        computations.
      - Powerful, flexible group by functionality to perform split-apply-combine
        operations on data sets, for both aggregating and transforming data.
      - Make it easy to convert ragged, differently-indexed data in other Python
        and NumPy data structures into DataFrame objects.
      - Intelligent label-based slicing, fancy indexing, and subsetting of large
        data sets.
      - Intuitive merging and joining data sets.
      - Flexible reshaping and pivoting of data sets.
      - Hierarchical labeling of axes (possible to have multiple labels per tick).
      - Robust IO tools for loading data from flat files (CSV and delimited),
        Excel files, databases, and saving/loading data from the ultrafast HDF5
        format.
```

```
      - Time series-specific functionality: date range generation and frequency
        conversion, moving window statistics, moving window linear regressions,
        date shifting and lagging, etc.

PACKAGE CONTENTS
    _libs (package)
    _version
    api (package)
    compat (package)
    computation (package)
    conftest
    core (package)
    errors (package)
    formats (package)
    io (package)
    json
    lib
    parser
    plotting (package)
    testing
    tests (package)
    tools (package)
    tseries (package)
    tslib
    types (package)
    util (package)

SUBMODULES
    _hashtable
    _lib
    _tslib
    offsets

DATA
    IndexSlice = <pandas.core.indexing._IndexSlice object>
    NaT = NaT
    __docformat__ = 'restructuredtext'
    datetools = <module 'pandas.core.datetools' from 'C:\\Users\...\lib\\s...
    describe_option = <pandas.core.config.CallableDynamicDoc object>
    get_option = <pandas.core.config.CallableDynamicDoc object>
    json = <module 'pandas.json' from 'C:\\Users\\user\\Anaconda3\\lib\\si...
    lib = <module 'pandas.lib' from 'C:\\Users\\user\\Anaconda3\\lib\\site...
```

```
    options = <pandas.core.config.DictWrapper object>
    parser = <module 'pandas.parser' from 'C:\\Users\\user\\Anaconda3\\lib...
    plot_params = {'xaxis.compat': False}
    reset_option = <pandas.core.config.CallableDynamicDoc object>
    set_option = <pandas.core.config.CallableDynamicDoc object>
    tslib = <module 'pandas.tslib' from 'C:\\Users\\user\\Anaconda3\\lib\\...

VERSION
    0.23.4

FILE
    c:\users\user\anaconda3\lib\site-packages\pandas\__init__.py
```

In [7]:
```
# To see all the available options with any package we can use 'dir()' method.
dir(pd)
```

Out[7]:
```
['Categorical',
 'CategoricalIndex',
 'DataFrame',
 'DateOffset',
 'DatetimeIndex',
 'ExcelFile',
 'ExcelWriter',
 'Expr',
 'Float64Index',
 'Grouper',
 'HDFStore',
 'Index',
 'IndexSlice',
 'Int64Index',
 'Interval',
 'IntervalIndex',
 'MultiIndex',
 'NaT',
 'Panel',
 'Period',
 'PeriodIndex',
 'RangeIndex',
 'Series',
 'SparseArray',
 'SparseDataFrame',
 'SparseSeries',
 'Term',
 'TimeGrouper',
 'Timedelta',
 'TimedeltaIndex',
 'Timestamp',
 'UInt64Index',
 'WidePanel',
 '_DeprecatedModule',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__docformat__',
 '__file__',
 '__loader__',
 '__name__',
```

```
 '__package__',
 '__path__',
 '__spec__',
 '__version__',
 '_hashtable',
 '_lib',
 '_libs',
 '_np_version_under1p10',
 '_np_version_under1p11',
 '_np_version_under1p12',
 '_np_version_under1p13',
 '_np_version_under1p14',
 '_np_version_under1p15',
 '_tslib',
 '_version',
 'api',
 'bdate_range',
 'compat',
 'concat',
 'core',
 'crosstab',
 'cut',
 'date_range',
 'datetime',
 'datetools',
 'describe_option',
 'errors',
 'eval',
 'factorize',
 'get_dummies',
 'get_option',
 'get_store',
 'groupby',
 'infer_freq',
 'interval_range',
 'io',
 'isna',
 'isnull',
 'json',
 'lib',
 'lreshape',
 'match',
```

```
 'melt',
 'merge',
 'merge_asof',
 'merge_ordered',
 'notna',
 'notnull',
 'np',
 'offsets',
 'option_context',
 'options',
 'pandas',
 'parser',
 'period_range',
 'pivot',
 'pivot_table',
 'plot_params',
 'plotting',
 'pnow',
 'qcut',
 'read_clipboard',
 'read_csv',
 'read_excel',
 'read_feather',
 'read_fwf',
 'read_gbq',
 'read_hdf',
 'read_html',
 'read_json',
 'read_msgpack',
 'read_parquet',
 'read_pickle',
 'read_sas',
 'read_sql',
 'read_sql_query',
 'read_sql_table',
 'read_stata',
 'read_table',
 'reset_option',
 'scatter_matrix',
 'set_eng_float_format',
 'set_option',
 'show_versions',
```

```
 'test',
 'testing',
 'timedelta_range',
 'to_datetime',
 'to_msgpack',
 'to_numeric',
 'to_pickle',
 'to_timedelta',
 'tools',
 'tseries',
 'tslib',
 'unique',
 'util',
 'value_counts',
 'wide_to_long']
```

In [ ]:

Introducing Pandas Objects

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. Three fundamental Pandas data structures:

- *The Series,*
- *DataFrame, and*
- *Index. More Details on pandas can be found at [Pandas Details (http://pandas.pydata.org/)](http://pandas.pydata.org/)*

Series Object

In [8]:
```
import numpy as np
import pandas as pd
```

In [9]:
```
# Let's look at the signature of the pandas Series
pd.Series?
```

In [10]:
```
# A Pandas Series is a one-dimensional array of indexed data
data = pd.Series([1.0, 2.0, 0.3, 0.4, 0.5, 0.6])
data
```

Out[10]:
```
0    1.0
1    2.0
2    0.3
3    0.4
4    0.5
5    0.6
dtype: float64
```

In [11]:
```
# A Pandas Series is a one-dimensional array of indexed data: Let's name the series using 'name' parameter
data = pd.Series([1.0, 2.0, 0.3, 0.4, 0.5, 0.6], name='raju')
data
```

Out[11]:
```
0    1.0
1    2.0
2    0.3
3    0.4
4    0.5
5    0.6
Name: raju, dtype: float64
```

In [12]:
```
# Or equivalently it can be created first by creating the list as follows:
list = [1.0, 2.0, 0.3, 0.4, 0.5, 0.6]
data = pd.Series(list)
data
```

Out[12]:
```
0    1.0
1    2.0
2    0.3
3    0.4
4    0.5
5    0.6
dtype: float64
```

In [13]:
```
# We can access the values of pandas series using 'values' and sequence of indices using 'index' attributes
```

In [14]:
```
print(data.values)
data.values
```

```
[1.  2.  0.3 0.4 0.5 0.6]
```

Out[14]:
```
array([1. , 2. , 0.3, 0.4, 0.5, 0.6])
```

In [15]:
```
print(data.index)
data.index
```

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

Out[15]:
```
RangeIndex(start=0, stop=6, step=1)
```

In [16]:
```
print(data)
```

```
0    1.0
1    2.0
2    0.3
3    0.4
4    0.5
5    0.6
dtype: float64
```

In [17]:
```
# Data can be accessed using index method 'data[index]' or 'data[range_index]'
print(data[1])
```

```
2.0
```

In [18]:
```
print(data[1:4])
```

```
1    2.0
2    0.3
3    0.4
dtype: float64
```

In [19]:
```
print(data[1:])
```

```
1    2.0
2    0.3
3    0.4
4    0.5
5    0.6
dtype: float64
```

In [20]:
```
print(data[:4])
```

```
0    1.0
1    2.0
2    0.3
3    0.4
dtype: float64
```

In [21]:
```
print(data[:])
```

```
0    1.0
1    2.0
2    0.3
3    0.4
4    0.5
5    0.6
dtype: float64
```

In [22]:
```
# Pandas provides 'explicit index' notation: That is, we can use our own index to access the values
data_ei = pd.Series([1.0, 2.0, 0.3, 0.4, 0.5, 0.6], index=['a', 'b','c','d', 'e', 'f'])
data_ei
```

Out[22]:
```
a    1.0
b    2.0
c    0.3
d    0.4
e    0.5
f    0.6
dtype: float64
```

In [23]:
```
# Now we can access the values using explicitely defined index.
```

In [24]:
```
data_ei['d']
```

Out[24]:
```
0.4
```

In [25]:
```
data_ei['b':'e']
```

Out[25]:
```
b    2.0
c    0.3
d    0.4
e    0.5
dtype: float64
```

In [26]:
```
# The index in pandas series can be nonsequential or noncontiguous:
data_nsi= pd.Series([1.0, 2.0, 0.3, 0.4, 0.5, 0.6], index=[10, 20,'c',30, 'e', 40])
data_nsi
```

Out[26]:
```
10    1.0
20    2.0
c     0.3
30    0.4
e     0.5
40    0.6
dtype: float64
```

In [27]:
```
data_nsi['c']
```

Out[27]:
```
0.3
```

In [28]:
```
data_nsi[30]
```

Out[28]:
```
0.4
```

In [29]:
```
# From the above explnation it is clear that pandas series works just like a dictionary with 'keys' as indices and 'va
lues'
# as series of data elements
```

In [30]:
```
# So we can use dictionary to construct a pandas series type

dic_ages = {'Pruthvi1':20, 'Pruthvi2':30, 'Pruthvi3':40, 'Pruthvi4':30, 'Pruthvi5':20, 'Pruthvi6':70}
data_ages =pd.Series(dic_ages)
data_ages
```

Out[30]:
```
Pruthvi1    20
Pruthvi2    30
Pruthvi3    40
Pruthvi4    30
Pruthvi5    20
Pruthvi6    70
dtype: int64
```

In [31]:
```
# Another example : some states and their capitals as a dictionary to create a Series object
States_Capitals = {'Karnataka':'Bangalore', 'Andrapradesh':'Hyderabad',
                   'Tamilnadu':'Chennai', 'Keral':'Thiruvanathapuram', 'Maharastra':'Mumbai', 'India':'Dehli'}
data_capitals = pd.Series(States_Capitals)
data_capitals
```

Out[31]:
```
Karnataka              Bangalore
Andrapradesh           Hyderabad
Tamilnadu                Chennai
Keral          Thiruvanathapuram
Maharastra                Mumbai
India                      Dehli
dtype: object
```

In [32]:
```
# Now we can use "keys" to access the values of the pandas series
# Ex.:
data_capitals['Karnataka']
```

Out[32]:
```
'Bangalore'
```

In [33]:
```
data_capitals['Andrapradesh']
```

Out[33]:
```
'Hyderabad'
```

CONSTRUCTING A SERIES OBJECT IN PANDAS

In [34]:
```
# ---->:  pd.Series(data, index=index)
# where index is an optional argument, and data can be one of any type
```

In [35]:
```
# constructing series objetcs with Numpy array elements or list of elements
pd.Series([2, 4, 6])
```

Out[35]:
```
0    2
1    4
2    6
dtype: int64
```

In [36]:
```
list = ['A', 'B', 'C', 'D']
pd.Series(list)
```

Out[36]:
```
0    A
1    B
2    C
3    D
dtype: object
```

In [37]:
```
# Data can be a scalar : which can repeat to fullfil the indices
pd.Series(5, index=[1, 2, 3, 4, 5])
```

Out[37]:
```
1    5
2    5
3    5
4    5
5    5
dtype: int64
```

In [38]:
```
# Data can be a dictionary
data_dic = pd.Series({5:'a', 2:'e', 3:'f', 1:'w', 4:'s', 6:'d'})
data_dic
```

Out[38]:
```
5    a
2    e
3    f
1    w
4    s
6    d
dtype: object
```

In [39]:
```
pd.Series({2:'a', 1:'b', 3:'c'})
```

Out[39]:
```
2    a
1    b
3    c
dtype: object
```

In [40]:
```
pd.Series({2:'a', 1:'b', 3:'c'})
```

Out[40]:
```
2    a
1    b
3    c
dtype: object
```

In [41]:
```
# Index can be explicitly set to get different result even after using the dictionary as the series elements:
data_ei = pd.Series({5:'a', 2:'e', 3:'f', 1:'w', 4:'s', 6:'d'}, index =[2, 3, 4])
data_ei
```

Out[41]:
```
2    e
3    f
4    s
dtype: object
```

Data Frame Object

DataFrame can be thought of either as a generalizationof a NumPy array, or as a specialization of a Python dictionary. A DataFrame is an analog of a two-dimensional array with both flexible row indices and flexible column names.

pd.DataFrame()

In [42]:
```
import pandas as pd
import numpy as np
```

In [43]:
```
# Let's look at the signature of the pandas DataFrame first
pd.DataFrame?
```

In [44]:
```
states_capitals = {'Karnataka':'Bangalore', 'Andrapradesh':'Hyderabad',
                   'Tamilnadu':'Chennai', 'Kerala':'Thiruvanathapuram', 'Maharastra':'Mumbai', 'India':'Dehli'}
data_capitals = pd.Series(states_capitals)
data_capitals
```

Out[44]:
```
Karnataka              Bangalore
Andrapradesh           Hyderabad
Tamilnadu                Chennai
Kerala         Thiruvanathapuram
Maharastra                Mumbai
India                      Dehli
dtype: object
```

In [45]:
```
states_lang = {'Karnataka':'Kannada', 'Andrapradesh':'Telugu',
'Tamilnadu':'Tamil', 'Kerala':'Malayalam', 'Maharastra':'hindi', 'India':'Hindi'}
sl_df = pd.Series(states_lang)
sl_df
```

Out[45]:
```
Karnataka         Kannada
Andrapradesh       Telugu
Tamilnadu           Tamil
Kerala          Malayalam
Maharastra          hindi
India               Hindi
dtype: object
```

In [46]:
```
# Now we can use a dictionary to construct a single two-dimensional object with pandas dataframe object
# Here the 'keys' remains same but the values are now having two columns.
states_CL = pd.DataFrame({'Capitals': data_capitals, 'Language': sl_df})
states_CL
```

Out[46]:

	Capitals	Language
Karnataka	*Bangalore*	*Kannada*
Andrapradesh	*Hyderabad*	*Telugu*
Tamilnadu	*Chennai*	*Tamil*
Kerala	*Thiruvanathapuram*	*Malayalam*
Maharastra	*Mumbai*	*hindi*
India	*Dehli*	*Hindi*

In [47]:
```
print(states_CL)
```

```
                    Capitals   Language
Karnataka            Bangalore    Kannada
Andrapradesh          Hyderabad     Telugu
Tamilnadu               Chennai      Tamil
Kerala        Thiruvanathapuram  Malayalam
Maharastra              Mumbai      hindi
India                   Dehli       Hindi
```

In [48]:
```
# We can use 'index' attribute to access the row indices
states_CL.index
```

Out[48]:
```
Index(['Karnataka', 'Andrapradesh', 'Tamilnadu', 'Kerala', 'Maharastra',
       'India'],
      dtype='object')
```

In [49]:
```
# We can use 'column' attribute to access the column indices
states_CL.columns
```

Out[49]:
```
Index(['Capitals', 'Language'], dtype='object')
```

In [50]:
```
# We can access the values using rows and column index
# DataFrame maps a column name to a Series of column data
# Ex.:
states_CL['Capitals']
```

Out[50]:
```
Karnataka               Bangalore
Andrapradesh            Hyderabad
Tamilnadu                 Chennai
Kerala          Thiruvanathapuram
Maharastra                Mumbai
India                      Dehli
Name: Capitals, dtype: object
```

In [51]:
```
print(type(states_CL['Capitals']))
print(type(states_CL))
```

```
<class 'pandas.core.series.Series'>
<class 'pandas.core.frame.DataFrame'>
```

In [52]:
```
states_CL['Language']
```

Out[52]:
```
Karnataka         Kannada
Andrapradesh       Telugu
Tamilnadu           Tamil
Kerala          Malayalam
Maharastra          hindi
India               Hindi
Name: Language, dtype: object
```

In [53]:
```
# We can access the particular row index value using the 'dot' operation or '[index]' method
# Ex.:
states_CL['Language'].Kerala
```

Out[53]:
```
'Malayalam'
```

In [54]:
```
states_CL['Language']['Kerala']
```

Out[54]:
```
'Malayalam'
```

Constrcuting DataFrame Object

In [55]:
```
import pandas as pd
import numpy as np
```

In [56]:
```
# A DataFrame is a collection of Series objects, and a single column DataFrame can be constructed from a single Series:

states_lang = {'Karnataka':'Kannada', 'Andrapradesh':'Telugu', 'Tamilnadu':'Tamil', 'Kerala':'Malayalam',
               'Maharastra':'hindi', 'India':'Hindi'}
sl_df = pd.Series(states_lang)
sl_df
pd.DataFrame(sl_df)                                  # , columns = ['states_lang']
```

Out[56]:

	0
Karnataka	*Kannada*
Andrapradesh	*Telugu*
Tamilnadu	*Tamil*
Kerala	*Malayalam*
Maharastra	*hindi*
India	*Hindi*

In [57]:
```
pd.DataFrame(sl_df, columns = ['states_lang'])
```

Out[57]:

	states_lang
Karnataka	*Kannada*
Andrapradesh	*Telugu*
Tamilnadu	*Tamil*
Kerala	*Malayalam*
Maharastra	*hindi*
India	*Hindi*

In [58]:
```
# From a list of dictionary with values as a list of array-like objects

d_list = {'a':[0,1,2,3,4], 'b':[0,2,4,6,8], 'c': [0,3,6,9,12]}
d_list
```

Out[58]:
```
{'a': [0, 1, 2, 3, 4], 'b': [0, 2, 4, 6, 8], 'c': [0, 3, 6, 9, 12]}
```

In [59]:
```
df_dic = pd.DataFrame(d_list)
df_dic
```

Out[59]:

	a	b	c
0	*0*	*0*	*0*
1	*1*	*2*	*3*
2	*2*	*4*	*6*
3	*3*	*6*	*9*
4	*4*	*8*	*12*

In [60]:
```
# Or by using list comprehension

data = [{'a': i, 'b': 2 * i, 'c': 3*i} for i in range(5)]
df_lc = pd.DataFrame(data)
df_lc
```

Out[60]:

	a	b	c
0	*0*	*0*	*0*
1	*1*	*2*	*3*
2	*2*	*4*	*6*
3	*3*	*6*	*9*
4	*4*	*8*	*12*

In [61]:
```
# From a dictionary of series objects
states_CL = pd.DataFrame({'Capitals': data_capitals, 'Language': sl_df})
states_CL
```

Out[61]:

	Capitals	Language
Karnataka	*Bangalore*	*Kannada*
Andrapradesh	*Hyderabad*	*Telugu*
Tamilnadu	*Chennai*	*Tamil*
Kerala	*Thiruvanathapuram*	*Malayalam*
Maharastra	*Mumbai*	*hindi*
India	*Dehli*	*Hindi*

In [62]:
```
# From a two dimentional numpy array:

# print(np.random.rand(6, 3))
pd.DataFrame(np.random.rand(6, 3), index=['A','B','C','D', 'E','F'], columns=['a', 'b', 'c'])
```

Out[62]:

	a	b	c
A	*0.147959*	*0.244359*	*0.059290*
B	*0.946948*	*0.793141*	*0.042128*
C	*0.602496*	*0.972126*	*0.113256*
D	*0.818788*	*0.542978*	*0.670613*
E	*0.566262*	*0.967541*	*0.359025*
F	*0.808923*	*0.695645*	*0.663341*

In [65]:
```
#pd.DataFrame(np.random.rand(6, 3), index=['A','B','C','D', 'E','F'], columns=['a', 'b'])    # you will get error if y
ou run this cell
```

In [66]:
```
# From a numpy structured array

A = np.zeros(5, dtype=[('A', 'i'), ('B', 'f')])
A
```

Out[66]:
```
array([(0, 0.), (0, 0.), (0, 0.), (0, 0.), (0, 0.)],
      dtype=[('A', '<i4'), ('B', '<f4')])
```

In [67]:
```
data_A = pd.DataFrame(A)
data_A
```

Out[67]:

	A	B
0	*0*	*0.0*
1	*0*	*0.0*
2	*0*	*0.0*
3	*0*	*0.0*
4	*0*	*0.0*

Pandas Index Object

In [68]:
```
# Inedx object can be thought of as an 'immutable array' or 'ordered set'.
```

In [69]:
```
import pandas as pd
```

In [70]:
```
pd.Index?
```

In [71]:
```
# Let's construct a index from a list of obects

ind= pd.Index([1, 2, 3, 4, 5, 6, 7, 7])
ind
```

Out[71]:
```
Int64Index([1, 2, 3, 4, 5, 6, 7, 7], dtype='int64')
```

In [72]:
```
ind= pd.Index([1, 2, 3, 4, 5, 6, 7, 7], name='integers')
ind
```

Out[72]:
```
Int64Index([1, 2, 3, 4, 5, 6, 7, 7], dtype='int64', name='integers')
```

In [73]:
```
# We can use slicing on index objects : standard Python indexing notation
```

In [74]:
```
ind[2]
```

Out[74]:
```
3
```

In [75]:
```
ind[:3]
```

Out[75]:
```
Int64Index([1, 2, 3], dtype='int64', name='integers')
```

In [76]:
```
ind[::2]
```

Out[76]:
```
Int64Index([1, 3, 5, 7], dtype='int64', name='integers')
```

In [77]:
```
# We can use different attributes to get the details of index object
print(ind.dtype, ind.shape, ind.size, ind.ndim)
```

```
int64 (8,) 8 1
```

In [78]:
```
# Index cannot be mutable : It throughs an error
ind[2]= 3
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-78-6365c82699b5> in <module>()
      1 # Index cannot be mutable : It throughs an error
----> 2 ind[2]= 3

~\Anaconda3\lib\site-packages\pandas\core\indexes\base.py in __setitem__(self, key, value)
   2063
   2064     def __setitem__(self, key, value):
-> 2065         raise TypeError("Index does not support mutable operations")
   2066
   2067     def __getitem__(self, key):

TypeError: Index does not support mutable operations
```

In [79]:
```
# Index as ordered set
```

In [80]:
```
indA = pd.Index([1, 3, 5, 7, 8, 9])
indB = pd.Index([2, 3, 5, 7, 10, 11])
```

In [81]:
```
indA & indB  # intersection
```

Out[81]:
```
Int64Index([3, 5, 7], dtype='int64')
```

In [82]:
```
indA | indB # union
```

Out[82]:
```
Int64Index([1, 2, 3, 5, 7, 8, 9, 10, 11], dtype='int64')
```

In [83]:
```
indA.difference(indB) # differnce
```

Out[83]:
```
Int64Index([1, 8, 9], dtype='int64')
```

In [84]:
```
indA ^ indB # symmtric difference
```

Out[84]:
```
Int64Index([1, 2, 8, 9, 10, 11], dtype='int64')
```

In [85]:
```
indA.intersection(indB)
```

Out[85]:
```
Int64Index([3, 5, 7], dtype='int64')
```

In [86]:
```
indA.union(indB)
```

Out[86]:
```
Int64Index([1, 2, 3, 5, 7, 8, 9, 10, 11], dtype='int64')
```

In [87]:
```
indA.symmetric_difference(indB)
```

Out[87]:
```
Int64Index([1, 2, 8, 9, 10, 11], dtype='int64')
```

In [ ]:

In [ ]: