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

Introduction to pandas Data Structures

To get started with pandas, you will need to get comfortable with its two workhorse data structures: **Series and DataFrame**. While they are not a universal solution for every problem, they provide a solid, easy-to-use basis for most applications.

Pandas Series Object

A Series is the primary building block of pandas.

Series represents a `one-dimensional` `labeled indexed array` based on the NumPy ndarray.

Like an array, a Series can hold zero or more values of any single data type

labelelled index array sai muraad hai ke hum apni marzi sai index desakte hain

Creating Series

A Series can be created and initialized by passing either a **scalar value, a NumPy ndarray, a Python list, or a Python Dict** as the data parameter of the Series constructor. This is the default parameter and does not need to be specified if it is the first item.

```
In [3]: # Create one item series
        # 2 is scalar so we have to understand the output 0 is the index of that value 2
        s1 = pd.Series(2)
        s1

Out[3]: 0    2
        dtype: int64
```

In [5]:

```
# Create a series of multiple items from a list
# ye labelled indexes hain joke default arhe hain series ke andar
s2 = pd.Series([1,2,3,4,5])
s2
```

Out[5]:

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

In [6]:

```
# Get the values in the series
# agar series mai sai sirf values uthaani hain to values ki property use karenge

s2.values
```

Out[6]:

```
array([1, 2, 3, 4, 5], dtype=int64)
```

In [7]:

```
# Get the index of the series
s2.index
```

Out[7]:

```
RangeIndex(start=0, stop=5, step=1)
```

In [9]:

```
# Explicitly create and index
# index is alpha, not integer
# jab ismain humne labelled indexes diye ismai to mojood ismai dono hote hain, yaani yahan par integer indexi
ng bhi chalegi..
# ..joke position based hai, or labelled indexing bhi chalegi, jo position based indexing hoti hai wo out nhi
hoti balke ..
# .. mojoood rehti hain
s3 = pd.Series([1,2,3], index=['a','b','c'])
s3
```

Out[9]:

```
a    1
b    2
c    3
dtype: int64
```

In [11]:

```
# Lookup by label value, not integer position
print(f"value by label 's3['c']' is {s3['c']} and value by index 's3[2]' is {s3[2]}")
# access both by label and index
```

```
value by label 's3['c']' is 3 and value by index 's3[2]' is 3
```

In [13]:

```
# Create a series from an existing index
# Scalar value will be copied at each index label
s4 = pd.Series(["A","B","C","D","E"], index=s2.index)
s4
```

Out[13]:

```
0    A
1    B
2    C
3    D
4    E
dtype: object
```

In [14]:

```
# Create sereis from dict
# to ismai keys hamare pass labelled indexes bangyi hain
s4 = pd.Series({
    'a': 1,
    'b': 2,
    'c': 3,
    'd': 4
})

s4
```

Out[14]:

```
a    1
b    2
c    3
d    4
dtype: int64
```

In [15]:

```
# NumPy array bhi pass karsakte hain

s5 = pd.Series(np.array([22,23,44,55,66]))
s5
```

Out[15]:

```
0    22
1    23
2    44
3    55
4    66
dtype: int32
```

Size, shape, uniqueness, and counts of values

In [19]:

```
# Example series, which also contains a NaN means empty value
s= pd.Series([0,1,1,2,3,4,4,5,6,7,np.nan])
s
```

Out[19]:

```
0     0.0
1     1.0
2     1.0
3     2.0
4     3.0
5     4.0
6     4.0
7     5.0
8     6.0
9     7.0
10    NaN
dtype: float64
```

In [20]:

```
print(len(s))
print(s.size)     # number of elements
print(s.shape)
print(s.count())   # count return not null values   >>>> not null values kitni hain
print(s.unique())
print(s.value_counts())   # kosni value kitni dafa hai
```

```
11
11
(11,)
10
[ 0.  1.  2.  3.  4.  5.  6.  7. nan]
4.0    2
1.0    2
7.0    1
6.0    1
5.0    1
3.0    1
2.0    1
0.0    1
dtype: int64
```

Peeking at data with heads, tails, and take

In [22]:

```
# First five (by default)
# jab hamare pass bohot bara data hota hai to hum sirf first 5 rows uthalete hain using head()
s.head()
```

Out[22]:

```
0    0.0
1    1.0
2    1.0
3    2.0
4    3.0
dtype: float64
```

In [24]:

```
# First three (we can also specify)

# s.head(3)     >>>>> same as below >>>>>>>
s.head(n = 3)
```

Out[24]:

```
0    0.0
1    1.0
2    1.0
dtype: float64
```

In [26]:

```
# Last five (by default)
s.tail()
```

Out[26]:

```
6     4.0
7     5.0
8     6.0
9     7.0
10    NaN
dtype: float64
```

In [27]:

```
# Last three (we can also specify)

# s.tail(3)    >>>>>>   same as below     >>>>>>>>>>
s.tail(n = 3)
```

Out[27]:

```
8     6.0
9     7.0
10    NaN
dtype: float64
```

In [28]:

```
#The .take() method will return the rows in a series that correspond to the zero-based positions:

# only take specific items      >>>>>    like fancy indexing   >>>>>>>

s.take([9,3,9])
```

Out[28]:

```
9    7.0
3    2.0
9    7.0
dtype: float64
```

Looking up values in Series

In [29]:

```
# Single item lookup

print(s3)
s3['a']
```

```
a    1
b    2
c    3
dtype: int64
```

Out[29]:

```
1
```

In [35]:

```
#Accessing this Series using an integer value will perform a zero-based position lookup of the value:

# lookup by position since the index is not an integer
s3[1]

# jab main apne labelled index provide karunga or wo bhi integer mai karunga to by default indexing wo nhi ka
rega mere labell
# ke accordance hee karega
```

Out[35]:

```
2
```

In [36]:

```
# Multiple items
s3[['c', 'a']]
```

Out[36]:

```
c    3
a    1
dtype: int64
```

In [37]:

```
# Series with an integer index, but not starting with 0
# ab yahan mai ne labelled indexing di hain to kia ye integer based indexing samjh rha hai ya labelled base ?
s5 = pd.Series([1,2,3], index=[2,3,4])
s5
```

Out[37]:

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

label-based lookup versus position-based lookup

In [38]:

```
s5[2]  # 2 is considered as label based look up
       # coz label also has 2 init
```

Out[38]:

```
1
```

In [39]:
```
s5[0]   # now see in this case we have integer label lookup,position lookup is not working
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-39-001a9f7426c3> in <module>
----> 1 s5[0]   # now see in this case we have integer label lookup,position lookup is not working

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\series.py in __getitem__(self, key)
    869         key = com.apply_if_callable(key, self)
    870         try:
--> 871             result = self.index.get_value(self, key)
    872 
    873             if not is_scalar(result):

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\indexes\base.py in get_value(self, series, key)
   4403         k = self._convert_scalar_indexer(k, kind="getitem")
   4404         try:
-> 4405             return self._engine.get_value(s, k, tz=getattr(series.dtype, "tz", None))
   4406         except KeyError as e1:
   4407             if len(self) > 0 and (self.holds_integer() or self.is_boolean()):

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

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

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

pandas\_libs\hashtable_class_helper.pxi in pandas._libs.hashtable.Int64HashTable.get_item()

pandas\_libs\hashtable_class_helper.pxi in pandas._libs.hashtable.Int64HashTable.get_item()

KeyError: 0
```

In [40]:
```
s5.loc[2]  # loc also works on label based look up
```

Out[40]:
```
1
```

In [41]:
```
# integer location lao mai forcefully keh rha hun
s5.iloc[2]  #iLoc forcefully works on position based look up even you dont specify position based index
```

Out[41]:
```
3
```

In [42]:

```
# Multiple items by label (loc)
s5.loc[[4,3]]
```

Out[42]:

```
4    3
3    2
dtype: int64
```

In [46]:

```
s5[[0,2]]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-46-80606c9cd4a5> in <module>
----> 1 s5[[0,2]]

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\series.py in __getitem__(self, key)
    908             key = check_bool_indexer(self.index, key)
    909 
--> 910         return self._get_with(key)
    911 
    912     def _get_with(self, key):

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\series.py in _get_with(self, key)
    941         if key_type == "integer":
    942             if self.index.is_integer() or self.index.is_floating():
--> 943                 return self.loc[key]
    944             else:
    945                 return self._get_values(key)

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\indexing.py in __getitem__(self, key)
   1766 
   1767             maybe_callable = com.apply_if_callable(key, self.obj)
-> 1768             return self._getitem_axis(maybe_callable, axis=axis)
   1769 
   1770     def _is_scalar_access(self, key: Tuple):

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\indexing.py in _getitem_axis(self, key, axis)
   1952                     raise ValueError("Cannot index with multidimensional key")
   1953 
-> 1954                 return self._getitem_iterable(key, axis=axis)
   1955 
   1956             # nested tuple slicing

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\indexing.py in _getitem_iterable(self, key, axis)
   1593         else:
   1594             # A collection of keys
-> 1595             keyarr, indexer = self._get_listlike_indexer(key, axis, raise_missing=False)
   1596             return self.obj._reindex_with_indexers(
   1597                 {axis: [keyarr, indexer]}, copy=True, allow_dups=True

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\indexing.py in _get_listlike_indexer(self, key, axis, raise_missing)
   1550             keyarr, indexer, new_indexer = ax._reindex_non_unique(keyarr)
   1551 
-> 1552         self._validate_read_indexer(
   1553             keyarr, indexer, o._get_axis_number(axis), raise_missing=raise_missing
   1554         )

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\indexing.py in _validate_read_indexer(self, key, index
er, axis, raise_missing)
   1652             # just raising
   1653             if not (ax.is_categorical() or ax.is_interval()):
-> 1654                 raise KeyError(
   1655                     "Passing list-likes to .loc or [] with any missing labels "
   1656                     "is no longer supported, see "

KeyError: 'Passing list-likes to .loc or [] with any missing labels is no longer supported, see https://panda
s.pydata.org/pandas-docs/stable/user_guide/indexing.html#deprecate-loc-reindex-listlike'
```

In [47]:

```
s5.iloc[[0,2]]
```

Out[47]:

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

```
In [48]: s5.iloc[[0,2,3]]   # integer location will throw an exception
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\indexing.py in _get_list_axis(self, key, axis)
   2110         try:
-> 2111             return self.obj._take_with_is_copy(key, axis=axis)
   2112         except IndexError:

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\series.py in _take_with_is_copy(self, indices, axis, *
*kwargs)
    841         """
--> 842         return self.take(indices=indices, axis=axis, **kwargs)
    843 

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\series.py in take(self, indices, axis, is_copy, **kwar
gs)
    817         indices = ensure_platform_int(indices)
--> 818         new_index = self.index.take(indices)
    819 

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\indexes\base.py in take(self, indices, axis, allow_fil
l, fill_value, **kwargs)
    762                 )
--> 763             taken = self.values.take(indices)
    764         return self._shallow_copy(taken)

IndexError: index 3 is out of bounds for size 3

During handling of the above exception, another exception occurred:

IndexError                                Traceback (most recent call last)
<ipython-input-48-cfa427db3ba6> in <module>
----> 1 s5.iloc[[0,2,3]]   # integer location will throw an exception

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\indexing.py in __getitem__(self, key)
   1766 
   1767             maybe_callable = com.apply_if_callable(key, self.obj)
-> 1768             return self._getitem_axis(maybe_callable, axis=axis)
   1769 
   1770     def _is_scalar_access(self, key: Tuple):

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\indexing.py in _getitem_axis(self, key, axis)
   2127         # a list of integers
   2128         elif is_list_like_indexer(key):
```

```
-> 2129                 return self._get_list_axis(key, axis=axis)
   2130
   2131             # a single integer

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\indexing.py in _get_list_axis(self, key, axis)
   2112             except IndexError:
   2113                 # re-raise with different error message
-> 2114                 raise IndexError("positional indexers are out-of-bounds")
   2115
   2116         def _getitem_axis(self, key, axis: int):

IndexError: positional indexers are out-of-bounds
```

Alignment via index labels

- alignment ka word jab use karte hain jab 2 cheezon ko barabar karte hain, eik doosre ke saath alignment karte hain

```
In [49]: s6 = pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])
         s6
```

```
Out[49]: a    1
         b    2
         c    3
         d    4
         dtype: int64
```

```
In [50]: s7 = pd.Series([4, 3, 2, 1], index=['d', 'c', 'b', 'a'])
         s7
```

```
Out[50]: d    4
         c    3
         b    2
         a    1
         dtype: int64
```

In [52]:
```
# Add them
# Ye apna same label dhoondega add hone keliye numPy ki tarah nhi add hoga lekin agar label naa ho ?
s6 + s7    # it first aligns the data as per label then perfroms operation
```

Out[52]:
```
a    2
b    4
c    6
d    8
dtype: int64
```

-Nan + number = NaN

(NaN added to a number results in NaN)

-number + NaN = Nan

(Number added to a Nan results in NaN)

In [53]:
```
s8 = pd.Series({
    'a': 1,
    'b': 2,
    'c': 3,
    'd': 5
})

s8
```

Out[53]:
```
a    1
b    2
c    3
d    5
dtype: int64
```

```
In [55]: s9 = pd.Series({
             'b': 6,
             'c': 7,
             'd': 9,
             'e': 10
         })

         s9
```

```
Out[55]: b     6
         c     7
         d     9
         e    10
         dtype: int64
```

Ab yahan a ko a nhi mil rha, b ko b mil gya, c ko c milgya, d ko d mil gya, or e ko e nhi mil rha

- Ab yahan alignment karne keliye wo dono taraf ke labels ko barabar karega yaani pehli series mai a nhi hai to pehle ye a create karega, phir ye aage move karega phir dekhega ke e nhi hai to ye phir e create karega, to dono taraf hojayega abcde, yaani dono lables align hgye, yaani har lablel dono ke pass agya to question ye hai ke isne a align kia hai uski value kia hogi? or secondly usne e align kia hai to e ki value kia hogi ? to ANS is `NaN` , jo labell missing add hoga uski value `NaN` hogi

```
In [56]: # NaN's result for a and e
         # demonstrates alignment
         s8 + s9
```

```
Out[56]: a     NaN
         b     8.0
         c    10.0
         d    14.0
         e     NaN
         dtype: float64
```

```
In [57]: s10 = pd.Series([1.0, 2.0, 3.0], index=['a', 'a', 'b'])
         s10
```

```
Out[57]: a    1.0
         a    2.0
         b    3.0
         dtype: float64
```

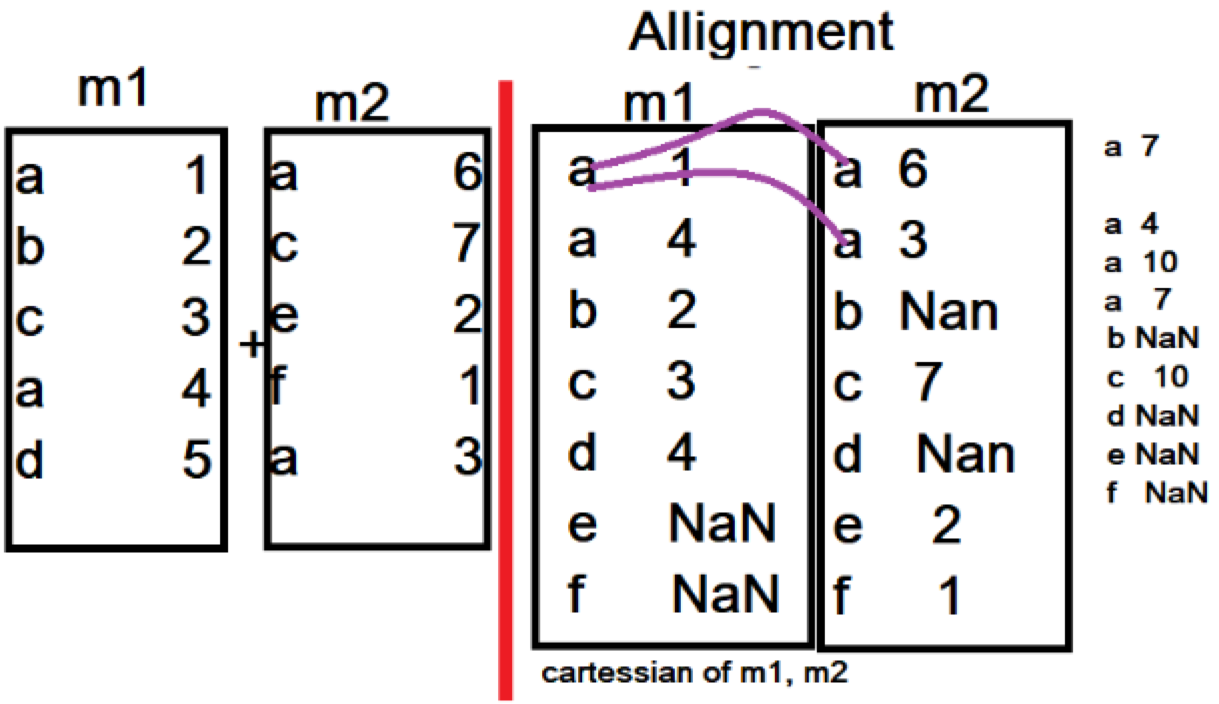
```
In [58]: s11 = pd.Series([4.0, 5.0, 6.0], index=['a', 'a', 'c'])
         s11
```

```
Out[58]: a    4.0
         a    5.0
         c    6.0
         dtype: float64
```

When two Series objects are added(or any other operation performed), the resulting series has four 'a' index labels.

```
In [60]: s10 + s11
```

```
Out[60]: a    5.0
         a    6.0
         a    6.0
         a    7.0
         b    NaN
         c    NaN
         dtype: float64
```



Allignment

m1

a	1
b	2
c	3
a	4
d	5

+

m2

a	6
c	7
e	2
f	1
a	3

m1

a	1
a	4
b	2
c	3
d	4
e	NaN
f	NaN

m2

a	6
a	3
b	Nan
c	7
d	Nan
e	2
f	1

cartesian of m1, m2

a	7
a	4
a	10
a	7
b	NaN
c	10
d	NaN
e	NaN
f	NaN

The Special Case of Not a Number (NaN)

```
In [61]: # Mean of numpy array values
         nda = np.array([1,2,3,4,5])
         nda.mean()

Out[61]: 3.0
```

```
In [65]: # Mean of numpy array values with NaN
         nda = np.array([1,2,3,4,np.NaN])     # lekin agar ap isi array sereies banalete hain to Pandas NaN ko ignore
          kardega, not count
         nda.mean()

Out[65]: nan
```

```
In [68]: s = pd.Series(nda)
         s.mean()       # 10 / 4 = 2.5   >>>>>> saari NaN values ko drop kardiya

Out[68]: 2.5
```

```
In [70]: # Handle NaN value like NumPy
         s.mean(skipna=False)  # >>>>> skip NaN values false hai matlab skip nhi karega

Out[70]: nan
```

Boolean Selection

In [4]:

```
# Which rows have values that are > 5 ?
s = pd.Series(np.arange(0,10))

s > 5
```

Out[4]:

```
0    False
1    False
2    False
3    False
4    False
5    False
6     True
7     True
8     True
9     True
dtype: bool
```

In [5]:

```
# Select rows where values are > 5
logicalResults = s > 5
s[logicalResults]
```

Out[5]:

```
6    6
7    7
8    8
9    9
dtype: int32
```

In [6]:

```
# a little shorter version
s[s > 5]
```

Out[6]:

```
6    6
7    7
8    8
9    9
dtype: int32
```

In [7]:
```
# commented as it throws an exception
# s[s > 5 and s < 8]

# correct syntax
s[(s > 5) & (s < 8)]
```

Out[7]:
```
6    6
7    7
dtype: int32
```

In [8]:
```
# dono functions True ko dhoondte hain false ko nhi dhoondte
print(pd.Series([True, False, False, True, True]).all())
print(pd.Series([True, False, False, True, True]).any())
```

```
False
True
```

In [9]:
```
np.array([True,False,True,True]).sum()
```

Out[9]:
```
3
```

In [10]:
```
# Are all items >= 0 ?
(s >= 0).all()
```

Out[10]:
```
True
```

In [11]:
```
s < 2
```

Out[11]:
```
0     True
1     True
2    False
3    False
4    False
5    False
6    False
7    False
8    False
9    False
dtype: bool
```

```
In [12]: # Any items < 2 ?
         s[s < 2].any()

Out[12]: True
```

```
In [13]: # How many values < 2 ?
         (s < 2).sum()

Out[13]: 2
```

Reindexing a Series

Reindexing in pandas is a process that makes the data in a Series or DataFrame match a given set of labels. This is core to the functionality of pandas as it enables label alignment across multiple objects, which may originally have different indexing schemes. This process of performing a reindex includes the following steps:

1. Reordering existing data to match a set of labels.
2. Inserting NaN markers where no data exists for a label.
3. Possibly, filling missing data for a label using some type of logic (defaulting to adding NaN values).

Jab hum 2 series ko add kar rhe they to alignment ki wajah sai NaN values create horhi thin joke masla tha uska solution ye hai ke ap series ko reindex karlen, reindexing kuch masle hal kardegi, reindex ka matlab hota hai ke index ko change kardena

```
In [5]: # sample series of five items
        s = pd.Series(np.random.randn(5))
        s

Out[5]: 0    1.521034
        1    0.571025
        2    0.866678
        3    1.761646
        4    1.401842
        dtype: float64
```

In [6]:

```
# change the index
s.index = ['a', 'b', 'c', 'd', 'e']
s
```

Out[6]:

```
a    1.521034
b    0.571025
c    0.866678
d    1.761646
e    1.401842
dtype: float64
```

let's examine a slightly more practical example. The following code concatenates two Series objects resulting in duplicate index labels, which may not be desired in the resulting Series:

In [7]:

```
# concat copies index values verbatim (as it is),
# potentially making duplicates since we have some or all label index same

# seed hua wa hai to har dafa random mai same number ayenge

np.random.seed(123456)

s1 = pd.Series(np.random.randn(3))  # default indexing >> 0,1,2
s2 = pd.Series(np.random.randn(3))  # default indexing >> 0,1,2

# jab humne concatenate kia to index mai redundancy agyi, yaani lets say ke index hamri keys hain data keys, to keys jo hain..
# ..repeat hogyin, yaani ab jis ke through record ko access karna hai, usmain doubling hogyi hai, to ab jab bhi combining mai..
# .. doubling hojaye to combined waali series ka index ko reset karen

combined = pd.concat([s1, s2])
combined
```

Out[7]:

```
0    0.469112
1   -0.282863
2   -1.509059
0   -1.135632
1    1.212112
2   -0.173215
dtype: float64
```

In [8]:

```python
# reset the index so that duplication of index may be removed
combined.index = np.arange(0, len(combined))
combined
```

Out[8]:

```
0    0.469112
1   -0.282863
2   -1.509059
3   -1.135632
4    1.212112
5   -0.173215
dtype: float64
```

Reindexing using the `.index` property in-place modifies the Series.

```
yaani original mai jakar change kardega, ab puraane wale index dobara wapis nhi askte
```

reindex() method

Greater flexibility in creating a new index is provided using the .reindex() method. An example of the flexibility of .reindex() over assigning the .index property directly is that the list provided to .reindex() can be of a different length than the number of rows in the Series:

```
reindex() ka method ziyada flexible hai or ye in-place nhi karta balke ye reindex karke return karta hai series k
o, nayi series banake, jiske baad original wali series save rehti hai
```

In [9]:

```
np.random.seed(123456)

s1 = pd.Series(np.random.randn(4), ['a', 'b', 'c', 'd'])
print(s1)

# reindex with different number of labels
# result in dropped rows and/or  NaN's

s2 = s1.reindex(['a', 'c', 'g'])
# a or c ki value mil jayegi isko s1 main lekin g mojood nhi hai to NaN create hojayegi or d kiunke isko humne call hee nhi..
# ..kia to ye dropped hojayega
s2
```

```
a    0.469112
b   -0.282863
c   -1.509059
d   -1.135632
dtype: float64
```

Out[9]:

```
a    0.469112
c   -1.509059
g         NaN
dtype: float64
```

Things to be noted:

1. reindex() donot re-index inplace, it will return a new series original will not be modified
2. if any index not matching the previous index will be assigned NaN
3. The index present in previous indexes, if not included in re-index then the row will not be added in new series.

```
In [10]: print(combined)
```

```
0    0.469112
1   -0.282863
2   -1.509059
3   -1.135632
4    1.212112
5   -0.173215
dtype: float64
```

```
In [11]: combined.reindex([9,5,3,4,0,1,2,6])  # not in-place
```

```
Out[11]: 9         NaN
         5   -0.173215
         3   -1.135632
         4    1.212112
         0    0.469112
         1   -0.282863
         2   -1.509059
         6         NaN
         dtype: float64
```

```
In [12]: combined   # last indexing is still there.
```

```
Out[12]: 0    0.469112
         1   -0.282863
         2   -1.509059
         3   -1.135632
         4    1.212112
         5   -0.173215
         dtype: float64
```

Reindexing is also useful when you want to align two Series to perform an operation on matching elements from each series; however, for some reason, the two Series had index labels that will not initially align. The following example demonstrates this, where the first Series has indexes as sequential integers, but the second has a string representation of what would be the same values:

In [13]:

```
# different types for the same values of labels
# cause big trouble

s1 = pd.Series([0, 1, 2], index=[0, 1, 2])
s2 = pd.Series([3, 4, 5], index=['0', '1', '2'])

# s2 mai index integer nhi hain a, b, c bhi quotation marks mai nhi hote
s1 + s2
```

Out[13]:

```
0    NaN
1    NaN
2    NaN
0    NaN
1    NaN
2    NaN
dtype: float64
```

you can easily guess what had happened here.

all values are NaN because the operation tries to add the item in the first series with the integer label 0, which has a value of 0, but can't find the item in the other series and therefore, the result is NaN (and this fails six times in this case).

Once this situation is identified:
it becomes a fairly `trivial` situation to fix by reindexing the second series:

```
trivial means maamooli, haqeer
```

In [14]:
```
# reindex by casting the label types and we will get the desired result

s2.index = s2.index.values.astype(int)

s1 + s2
```

Out[14]:
```
0    3
1    5
2    7
dtype: int64
```

Overriding the default action of inserting **NaN** as a missing value during reindexing can be changed by using the **fill_value** parameter of the method.

```
hamara to data hee chala jayega jab sab jaga NaN ajayega, to mai isko forcefully kehsakta hun ke NaN mat daalo bal
ke 0 ya koi or value daaldo
```

In [15]:
```
# fill with 0 instead of NaN
s2 = s.copy()
s2
```

Out[15]:
```
a    1.521034
b    0.571025
c    0.866678
d    1.761646
e    1.401842
dtype: float64
```

In [16]:
```
s2_reindexed = s2.reindex(['a', 'f'], fill_value=0)
s2_reindexed
```

Out[16]:
```
a    1.521034
f    0.000000
dtype: float64
```

ffill, bfill, & nearest

In [17]:
```
# create example to demonstrate fills
s3 = pd.Series(['red', 'green', 'blue'], index=[0,8,10])
s3
```

Out[17]:
```
0       red
8     green
10     blue
dtype: object
```

In [18]:
```
# forward fill example
s3.reindex(np.arange(0,15), method='ffill')
```

Out[18]:
```
0       red
1       red
2       red
3       red
4       red
5       red
6       red
7       red
8     green
9     green
10     blue
11     blue
12     blue
13     blue
14     blue
dtype: object
```

In [19]:

```
# backward fill example
s3.reindex(np.arange(0,15), method='bfill')
```

Out[19]:

```
0       red
1     green
2     green
3     green
4     green
5     green
6     green
7     green
8     green
9      blue
10     blue
11      NaN
12      NaN
13      NaN
14      NaN
dtype: object
```

In [20]:

```
# nearest fill example
s3.reindex(np.arange(0,15), method='nearest')  # nearest: use nearest valid observation to fill gap
```

Out[20]:

```
0       red
1       red
2       red
3       red
4     green
5     green
6     green
7     green
8     green
9      blue
10     blue
11     blue
12     blue
13     blue
14     blue
dtype: object
```

Slicing a Series

In [21]:

```
# a series to use for slicing
# using index labels not starting at 0 to demonstrate
# position based slicing

s = pd.Series(np.arange(100,110), index=np.arange(10,20))

s
```

Out[21]:

```
10    100
11    101
12    102
13    103
14    104
15    105
16    106
17    107
18    108
19    109
dtype: int32
```

In [22]:

```
# items at position 0, 2, 4
s[0:6:2]  # [startofrow:endofrow:step]

# # equivalent to
# s.iloc[[0, 2, 4]]
```

Out[22]:

```
10    100
12    102
14    104
dtype: int32
```

```
In [23]: # first five by slicing, same as .head(5)
         s[:5]
```

```
Out[23]: 10    100
         11    101
         12    102
         13    103
         14    104
         dtype: int32
```

Missing Data in Series

```
NaN values represent data is missing in the series
```

```
In [25]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
         obj3 = pd.Series(sdata)
         obj3
```

```
Out[25]: Ohio      35000
         Texas     71000
         Oregon    16000
         Utah       5000
         dtype: int64
```

```
In [26]: states = ['California', 'Ohio', 'Oregon', 'Texas']
         obj4 = pd.Series(sdata, index=states)
         obj4
```

```
Out[26]: California        NaN
         Ohio          35000.0
         Oregon        16000.0
         Texas         71000.0
         dtype: float64
```

In [27]:

```
pd.isnull(obj4)     # obj4.isnull()
```

Out[27]:

```
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

In [28]:

```
pd.notnull(obj4)   # obj4.notnull()
```

Out[28]:

```
California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
```

In [29]:

```
# Hum apne series ka naam rakhsakte hain or index ka bhi naam rakhsakte hain

obj4.name = "population"
obj4.index.name = "state"

obj4
```

Out[29]:

```
state
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
Name: population, dtype: float64
```

In [ ]:

```
```