1.4.3 Series Object Attributes

To see series attributes in action

Scan QR Code

When you create a Series type object, all information related to it is available through attributes. You can use these attributes in the following format to get information about the Series object.

```
<Series object>.<attribute name>
```

Some common attributes of Series object are listed in the table below. The code examples for the usage of these attributes follow the Table 1.2.

Table 1.2 *Common attributes of Series objects*

Attribute	Description
<Series object>.index	The index (axis labels) of the Series.
<Series object>.values	Return Series as ndarray or ndarray-like depending on the dtype
<Series object>.dtype	return the dtype object of the underlying data
<Series object>.shape	return a tuple of the shape of the underlying data
<Series object>.nbytes	return the number of bytes in the underlying data
<Series object>.ndim	return the number of dimensions of the underlying data
<Series object>.size	return the number of elements in the underlying data
<Series object>.itemsize	return the size of the dtype of the item of the underlying data
<Series object>.hasnans	return True if there are any NaN values ; otherwise return False
<Series object>.empty	return True if the Series object is empty, false otherwise

Following examples show how to view various attributes of Series objects.

(a) Retrieving Index Array (index attribute) & Data Array (values attribute) of a Series Object

You can access the **index array** and **data values' array** of an existing Series object **obj5** as shown below :

```
In [51]: obj5.index
Out[51]: Index(['Feb', 'Jan', 'Mar'], dtype='object')

In [52]: obj5.values
Out[52]: array([28, 31, 31], dtype=int64)

In [53]: obj6.index
Out[53]: RangeIndex(start=0, stop=5, step=1)

In [54]: obj7.index
Out[54]: Int64Index([9, 10, 11, 12], dtype='int64')
```

See, <object>.index and <object>.values returned the index array and data values' array respectively

obj6 was created as :
obj6 = pd.Series(data = np.arange(11, 25, 3))

obj7 was created as :
obj7 = pd.Series(index = a, data = a * 2)
where a is a NumPy array ([9,10,11,12])

For the rest of the attributes (covered in points below), we shall be using following two objects (given as **Reference 1.2**)

```
>>>obj2 = pd.Series( [ 3.5, 5., 6.5, 8.] )
>>>obj2
0  3.5
1  5.0
2  6.5
3  8.0
dtype: float64
```

```
>>>obj3 = pd.Series( [6.5, np.NaN, 2.34] )
>>>obj3
0  6.50
1   NaN
2  2.34
dtype: float64
```

Reference 1.2

(b) Retrieving Data Type (dtype) and Size of Type (itemsize)

To retrieve the data type of individual elements of a series object, use <objectname>.dtype.

To know about the type of *Series object* itself, you can use **type( )** of Python. You can use **itemsize** attribute to know the number of bytes allocated to each data item *e.g.*,

```
In [29]: obj2.dtype
Out[29]: dtype('float64')

In [30]: obj2.itemsize
Out[30]: 8

In [31]: type(obj2)
Out[31]: pandas.core.series.Series
```

The **dtype** attribute return the type of data values stored in Series object

The **itemsize** returns the size in bytes for each item

The **type()**, however, tells the type of the object passed to it

(c) Retrieving Shape

The *shape of a Series object (shape attribute) tells how big it is, i.e., how many elements it contains including missing or empty values (NaNs).* Since there is only one axis in Series object, it is shown as (<n>,) where *n* is the number of elements in the object *e.g.*,

NOTE

The shape of a series object tell how big it is, *i.e.*, how many elements it contains including missing or empty values (NaNs).

```
>>> print(obj2.shape , obj3.shape)
(4,) (3,)
```

*See the number of elements in objects **obj2** and **obj3** are shown.*

(d) Retrieving Dimension (number of axis : ndim attribute), Size (size attribute) and Number of Bytes (nbytes attribute)

To know about the dimension (number of axis), use <objectname>.ndim.

To know about the number of elements in the Series object, use <objectname>.size.

To know total number of bytes taken by Series object data, use <objectname>.nbytes (*nbytes* is equal to the **size * itemsize**)

```
>>> obj2.ndim
1
```

Series object is 1-dimesinal object.

```
>>> print (obj2.size, obj3.size)
4 3
```

obj2 has 4 elements and obj3 has 3 elements

```
>>> print (obj2.nbytes, obj3.nbytes)
32 24
```

***obj2** has 4 elements, hence **4*8 = 32** bytes and **obj3** has 3 elements, hence 3 * 8 = 24 bytes*

(e) Checking Emptiness (empty attribute) and Presence of NaNs (hasnans attribute)

See we have created an empty Series object namely **obj1** also and then checked emptiness **obj1** along with emptiness of objects **obj2** and **obj3** of Reference 1.2. (given on previous page).

```
In [37]: obj1=pd.Series()

In [38]: obj1.empty
Out[38]: True
```

The object **obj1 is** empty but **obj2** and **obj3** (ref 10.2) are not empty as they contain some data

```
In [39]: obj2.empty
Out[39]: False
```

- Similarly, to check if a Series object contains some NaN value or not, you can use **hasnans** attribute as shown below (using objects of *reference 2.2*)
- You can use **len( )** to get total number of elements and **<series>.count( )** method with Series object to get the count of non-NaN values in a series object, *e.g.,* :

NOTE

If you use **len( )** on a Series object, then it returns total elements in it including NaNs but **<series>.count( )** returns only the count of non-NaN values in a Series object.

```
In [41]: obj2.hasnans
Out[41]: False

In [42]: obj3.hasnans
Out[42]: True
```

hasnans attrin\bute tells the presence of NaN values and count() returns count of non_NaN values

```
In [43]: obj2.count()
Out[43]: 4

In [44]: obj3.count()
Out[44]: 2
```

```
In [11]: len(obj3)
Out[11]: 3
```

len() gives total no. of elements

EXAMPLE 13 *Consider the two series objects* **s11** *and* **s12** *that you created in examples 11 and 12 respectively. Print the attributes of both these objects in a report form as shown below :*

```
Attribute name        Object s11        Object s12
Data type
Shape
No. of bytes
No. of dimensions
Item size
Has NaNs ?
Empty ?
```

SOLUTION

```
import pandas as pd
:    # statements here to create objects s11 and s12 from previous examples
print("Attribute name   \t\t Object s11 \t Object s12 ")
print("————————         \t \t———— \t ———— ")
print("Data type(.dtype)          :\t", s11.dtype, '\t\t', s12.dtype)
print("Shape   (.shape)           :\t", s11.shape, '\t\t', s12.shape)
print("No. of bytes (.nbytes)     :\t", s11.nbytes, '\t\t', s12.nbytes)
print("No. of dimensions(.ndim)   :\t", s11.ndim, '\t\t', s12.ndim)
print("Item size (.itemsize)      :\t", s11.itemsize, '\t\t', s12.itemsize)
print("Has NaNs? (.hasnans)       :\t", s11.hasnans, '\t\t', s12.hasnans)
print("Empty?   (.empty)          :\t", s11.empty, '\t\t', s12.empty)
```

Output

```
Attribute name                  Object s11        Object s12
————————                        ——————————        ——————————
Data type(.dtype)          :    int64             float32
Shape     (.shape)         :    (4,)              (5,)
No. of bytes (.nbytes)     :    32                20
No. of dimensions(.ndim)   :    1                 1
Item size (.itemsize)      :    8                 4
Has NaNs? (.hasnans)       :    False             True
Empty?    (.empty)         :    False             False
```

1.4.4 Accessing a Series Object and its Elements

Once you have created *Series type object*, you can access it in many ways. You can access its indexes separately, its data separately and even access individual elements and slices. Let us see how. Consider following Series objects (**obj5**, **obj6**, **obj7** and **obj8**) (Fig. 1.3) :

```
In [35]: obj5
Out[35]:
Feb    28
Jan    31
Mar    31
dtype: int64
```

```
In [37]: obj6
Out[37]:
0    11
1    14
2    17
3    20
4    23
dtype: int32
```

```
In [28]: obj7
Out[28]:
9     18
10    20
11    22
12    24
dtype: int32
```

```
In [29]: obj8
Out[29]:
9      81
10    100
11    121
12    144
dtype: int32
```

Figure 1.3 Some sample panda Series Objects.

For all the following operations we shall be using the sample objects shown in Fig. 1.3.

1. Accessing Individual Elements

To access individual elements of a Series object, you can give its index in square brackets along with its name, *i.e.*, as :

```
<Series Object name>[<valid index>]
```

Following figure shows you elements accessed from the objects ***obj5***, ***obj6***, ***obj7*** and ***obj8***.

```
In [39]: obj6[3]
Out[39]: 20
```

```
In [32]: obj7[9]
Out[32]: 18
```

```
In [33]: obj8[11]
Out[33]: 121
```

```
In [38]: obj5['Feb']
Out[38]: 28
```

See all these objects' legal indexes are used to access individual elements of these objects.

As you see in above figure, we have use only valid or legal indexes (*i.e.*, which exist in series object) to access an element.

If the Series object has duplicate indexes, then giving an index with the Series object will return all the entries with that index, *e.g.*, see below :

```
In [15]: ob3
Out[15]:
a     2.75
b    12.50
a    22.25
a    32.00
b    41.75
dtype: float64
```

```
In [16]: ob3['b']
Out[16]:
b    12.50
b    41.75
dtype: float64
```

With duplicate indexes in a Series object, all entries with the same index are returned.

BUT if you try to give an index which is not a legal index for a **Series object**, it will give you an error.

See adjacent figure.

2 is not a legal index for **obj7** (its legal indexes are 9 to 12 refer above), hence the error.

```
In [31]: obj7[2]
Traceback (most recent call last):

  File "<ipython-input-31-67505774af56>", line 1, in <module>
    obj7[2]

  File "pandas/_libs/hashtable_class_helper.pxi", line 817, in
pandas._libs.hashtable.Int64HashTable.get_item

KeyError: 2
```

2. Extracting Slices from Series Object

Like other sequences, you can extract slice too from a Series object to retrieve subsets. Let us see how you can extract slices from Series objects. Here, you need to understand **an important thing about slicing**, which is that :

Slicing takes place position wise and not the index wise in a series object.

To understand this, let us consider the same *Series objects* as given in Fig. 1.3. Internally there is a position associated with element – first element gets the position as 0, second element gets the position as 1 and so on. Irrespective of their indexes, positions always start with 0 and go on like 1, 2, 3 and so on. (*see* Fig. 1.4)

obj5

Position	Index	Data
0	Feb	28
1	Jan	31
2	Mar	31

obj6

Position	Index	Data
0	0	11
1	1	14
2	2	17
3	3	20
4	4	23

obj7

Position	Index	Data
0	9	18
1	10	20
2	11	22
3	12	24

All individual elements have position numbers starting from 0 onwards i.e., *0* for first element, *1* for 2nd element and so on

Figure 1.4 Position number associated with each element of Series object

When you have to extract slices, then you need to specify slices as [start : end : step] like you do for other sequences, but the ***start*** and ***stop*** signify the positions of elements not the indexes. Consider following examples :

```
In [67]: obj5[1: ]
Out[67]:
Jan    31
Mar    31
dtype: int64
```

```
In [68]: obj6[2:5]
Out[68]:
2    17
3    20
4    23
dtype: int32
```

```
In [70]: obj7[0: : 2]
Out[70]:
9     18
11    22
dtype: int32
```

Irrespective of the indexes, the slices have been extracted *position wise.*

All other rules of slices apply, *i.e.*, you can specify steps, you can reverse the elements, the range of slice can be outside the range of positions etc.

NOTE

A **slice object** is created from Series object using a syntax of <Object>[start : end : step], but the ***start*** and ***stop*** signify the positions of elements not the indexes. The **slice object** of a Series object is also a **panda Series type object**.

Even though **obj7** has indexes 10, 11, 12 but the slice object is empty, because slice is extracted *position wise* not *index wise*

```
In [71]: obj7[10:15]
Out[71]: Series([], dtype: int32)
```

```
In [72]: obj7[: : -1 ]
Out[72]:
12    24
11    22
10    20
9     18
dtype: int32
```

Slice object with values reversed

EXAMPLE 14 *Consider a Series object s8 that stores the number of students in each section of class 12 (as shown below).*

```
A   39
B   41
C   42
D   44
```

First two sections have been given a task of selling tickets @ 100/- per ticket as part of a social experiment. Write code to display how much they have collected.

SOLUTION

```
import pandas as pd
print("Tickets amount:")
print(s8[ :2] *100)
```

Output

```
Tickets amount:
A    3900
B    4100
dtype: int64
```

1.4.5 Operations on Series Object

Let us now talk about how you can perform various types of operations on Pandas Series objects.

1. Modifying Elements of Series Object

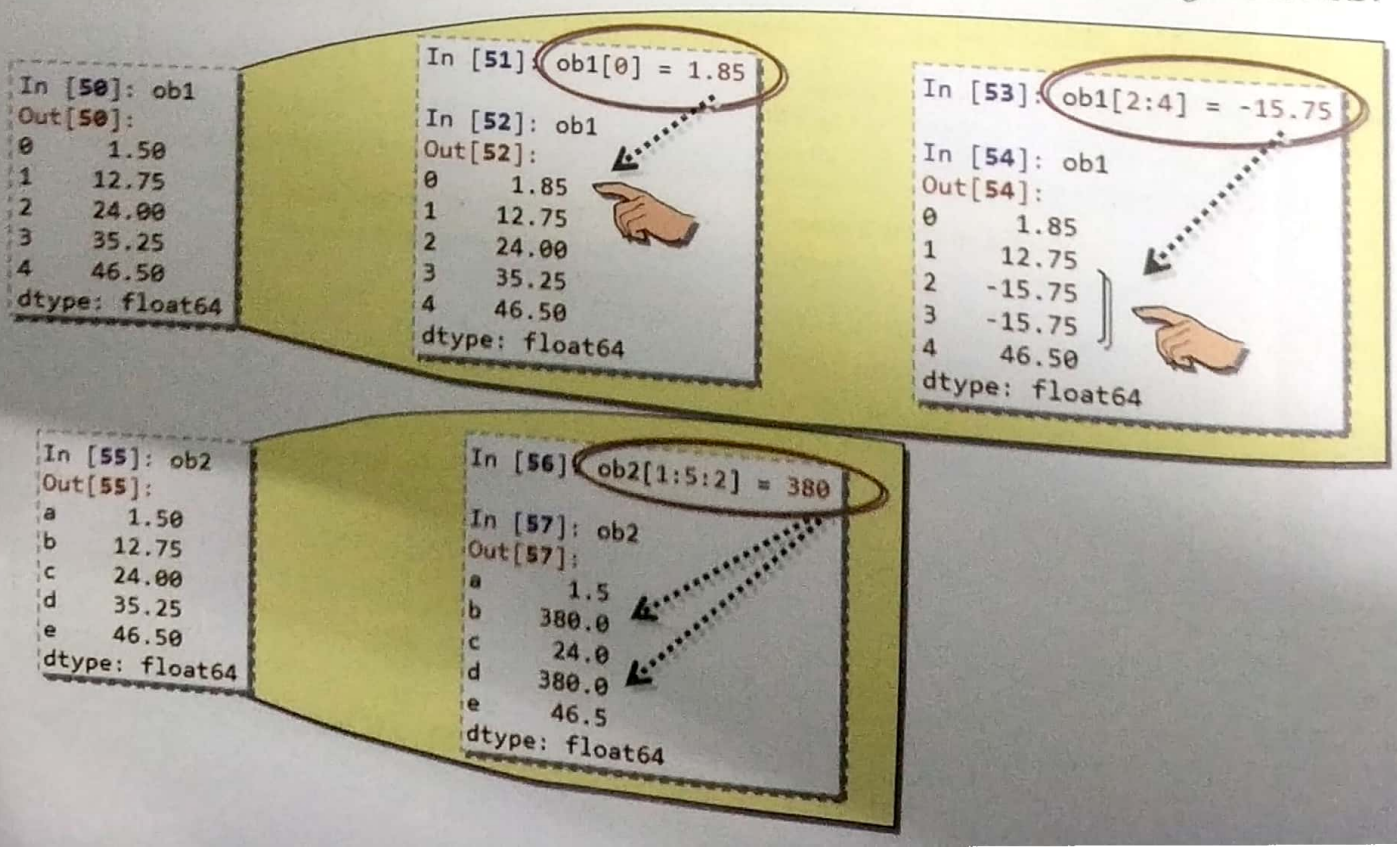
The data values of a Series object can be easily modified through item assignment, *i.e.*,

```
<SeriesObject>[<index> ] = <new_data_value>
```

Above assignment will change the data value of the given index in the Series object.

```
<SeriesObject>[start : stop] = <new_data_value>
```

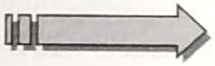
Above assignment will replace all the values falling in given slice. Consider following screenshots :



Renaming Indexes

You can even change or rename indexes of a Series object by assigning new index array to its index attribute, *i.e.*, : <Object>.index = <new index array>, *e.g.*, see below :

```
In [55]: ob2
Out[55]:
a     1.50
b    12.75
c    24.00
d    35.25
e    46.50
dtype: float64
```



```
In [58]: ob2.index= ['v', 'w', 'x', 'y', 'z' ]

In [59]: ob2
Out[59]:
v     1.5
w   380.0
x    24.0
y   380.0
z    46.5
dtype: float64
```

See how the new indexes get assigned in the order as given in the new index array

NOTE

Please note that Series object's values can be modified but size cannot. So you can say that **Series objects are *value-mutable* but *size-immutable* objects.**

The size of new index array must match with existing index array's size. In other words, you cannot change the size of a Series object by assigning more or less number of indexes (see below) :

```
In [59]: ob2
Out[59]:
v     1.5
w   380.0
x    24.0
y   380.0
z    46.5
dtype: float64
```

```
In [60]: ob2.index= ['v', 'w', 'x', 'y', 'z', '1', '2' ]
Traceback (most recent call last):
  File "C:\ProgramData\Anaconda3\lib\site-packages\pandas\core
\internals.py", line 3074, in set_axis
    (old_len, new_len))

ValueError: Length mismatch: Expected axis has 5 elements, new values have 7
elements
```

EXAMPLE 15 *Consider the Series object* **s13** *that stores the contribution of each section, as shown below :*

```
A    6700
B    5600
C    5000
D    5200
```

Write code to modify the amount of section 'A' as 7600 and for sections 'C' and 'D' as 7000. Print the changed object.

Output

```
Series object after modifying amounts:
A 7600
B 5600
C 7000
D 7000
```

SOLUTION

```
import pandas as pd
s13[0] = 7600                    # to modify section 'A''s amount
s13[2: ] = 7000                  # to modify sections 'C' and 'D''s amount
print("Series object after modifying amounts:")
print(s13)
```

2. The head( ) and tail( ) Functions

The ***head( )*** function is used to fetch first *n* rows from a Pandas object and ***tail( )*** function returns last *n* rows from a Pandas object. The syntax to use these functions is :

```
<pandas object>.head([n])
Or      <pandas object>.tail([n])
```

If you do not provide any value for *n*, then ***head( )*** and ***tail( )*** will return first 5 and last 5 rows respectively of a Pandas object. Consider below given screenshots :

```
In [42]: ob7
Out[42]:
0     2.75
1     7.60
2    12.45
3    17.30
4    22.15
5    27.00
6    31.85
7    36.70
8    41.55
9    46.40
10   51.25
dtype: float64
```

```
In [44]: ob7.head()
Out[44]:
0    2.75
1    7.60
2   12.45
3   17.30
4   22.15
dtype: float64
```

First 5 rows

```
In [45]: ob7.head(7)
Out[45]:
0    2.75
1    7.60
2   12.45
3   17.30
4   22.15
5   27.00
6   31.85
dtype: float64
```

First 7 rows

```
In [46]: ob7.tail()
Out[46]:
6    31.85
7    36.70
8    41.55
9    46.40
10   51.25
dtype: float64
```

Last 5 rows

```
In [47]: ob7.tail(7)
Out[47]:
4    22.15
5    27.00
6    31.85
7    36.70
8    41.55
9    46.40
10   51.25
dtype: float64
```

Last 7 rows

EXAMPLE 16 *A Series object* ***trdata*** *consists of around 2500 rows of data. Write a program to print the following details :*

(i) First 100 rows of data (ii) Last 5 rows of data

SOLUTION

```
import pandas as pd
;           # trdata object's creation or loading happens here
print(trdata.head(100))
print(trdata.tail())
```

3. Vector Operations on Series Object

Vector operations mean that if you apply a function or expression then it is individually applied on each item of the object. Since **Series objects** are built upon NumPy arrays (***ndarrays***), they also support vectorized operations, just like ***ndarrays***.

Following examples will make it clear to you.

Suppose we have a Pandas Series object **ob2** as shown here :

```
In [67]: ob2
Out[67]:
a     1.50
b    12.75
c    24.00
d    35.25
e    46.50
dtype: float64
```

Then following all are legal operations :

```
ob2 + 2,   ob2 * 3,   ob8 = ob2 **2,   ob2 > 15   etc.
```

To see vector operations in action

Scan QR Code

In all the above expressions, the given operation will be carried out in vectorized way, *i.e.*, will be applied to each item of the Series object.

Consider following examples :

```
In [68]: ob2 + 2
Out[68]:
a     3.50
b    14.75
c    26.00
d    37.25
e    48.50
dtype: float64
```

```
In [69]: ob2 * 3
Out[69]:
a     4.50
b    38.25
c    72.00
d   105.75
e   139.50
dtype: float64
```

```
In [72]: ob2 > 15
Out[72]:
a    False
b    False
c     True
d     True
e     True
dtype: bool
```

```
In [73]: ob8 = ob2 ** 2

In [74]: ob8
Out[74]:
a       2.2500
b     162.5625
c     576.0000
d    1242.5625
e    2162.2500
dtype: float64
```

See each of the expression though applied on the Series type object but is carried out on each individual item of the Series object – **Vector Operations.**

Figure 1.5 Vector operations on Series objects

4. Arithmetic on Series Objects

You can perform arithmetic like *addition, subtraction, division* etc. with two Series objects and it will calculate result on two corresponding items of the two objects given in expression BUT it has a caveat – **the operation is performed only on the matching indexes**, *e.g.*, if first object has indexes 0,1,2 then it will perform arithmetic only with objects having 0,1,2 indexes ; for all other indexes, it will produce **NaN** (not a number).

Also, if the data items of the two matching indexes are not compatible for the operation, it will return NaN (***Not a Number***) as the result of those operations.

To understand this, consider below given (Fig. 1.6) *five* Series objects **ob1**, **ob2**, **ob3**, **ob4** and **ob5** (**ob1** and **ob3** have matching indexes ; **ob2** and **ob5** have matching indexes; **ob4** has some indexes matching with **ob1** and **ob3**) :

```
In [88]: ob1
Out[88]:
0    1.85
1   12.75
2  -15.75
3  -15.75
4   46.50
dtype: float64
```

```
In [75]: ob2
Out[75]:
a    1.50
b   12.75
c   24.00
d   35.25
e   46.50
dtype: float64
```

```
In [91]: ob3
Out[91]:
0    2.75
1   12.50
2   22.25
3   32.00
4   41.75
dtype: float64
```

```
In [98]: ob4
Out[98]:
0    1.255
1    5.530
2    9.805
3   14.080
4   18.355
5   22.630
6   26.905
7   31.180
dtype: float64
```

```
In [101]: ob5
Out[101]:
a    1.255
b    5.530
c    9.805
d   14.080
e   18.355
dtype: float64
```

Figure 1.6 (*a*) Some sample Series objects with matching and non-matching indexes

To see arithmetic in series in action

Scan QR Code

Now carefully look at the statements carrying out arithmetic operations on objects w matching indexes (see below) :

```
In [102]: ob1 * ob3
Out[102]:
0       5.0875
1     159.3750
2    -350.4375
3    -504.0000
4    1941.3750
dtype: float64
```

```
In [103]: ob1 / ob3
Out[103]:
0    0.672727
1    1.020000
2   -0.707865
3   -0.492188
4    1.113772
dtype: float64
```

```
In [104]: ob2 + ob5
Out[104]:
a     2.755
b    18.280
c    33.805
d    49.330
e    64.855
dtype: float64
```

```
In [104]: ob2 + ob5
Out[104]:
a     2.755
b    18.280
c    33.805
d    49.330
e    64.855
dtype: float64
```

Since objects **ob1** and **ob3** have matching indexes (both have indexes in the range 0 to 4), it successfully carries out given arithmetic operation on corresponding items of matching indexes, *i.e.*, items with index 0 of both the objects are performed the given operation and result given for the index 0, similarly corresponding items having index 1 of both the objects are performed the given operation and result given for index 1, and so on.

Same thing is applied for expression **ob2 + ob5**, *i.e.*, corresponding values of index 'a' are added, similarly corresponding values of index 'a' are added, and so on.

But if you try to perform operation on objects that have some or all non-matching indexes, ther it will add values of matching indexes, if any and for non-matching indexes of both the objects it will return the result as *Not a Number i.e.*, NaN. **NaN represents missing data**. (see below)

```
In [105]: ob1 + ob4
Out[105]:
0     3.105
1    18.280
2    -5.945
3    -1.670
4    64.855
5       NaN
6       NaN
7       NaN
dtype: float64
```

Computed the given operation for matching indexes (**0, 1, 2, 3, 4** in both objects) and **NaN** for non-matching indexes (**5, 6, 7** of *ob4*)

NaN (*Not a Number*) represents missing data

```
In [92]: ob1 +ob2
Out[92]:
0   NaN
1   NaN
2   NaN
3   NaN
4   NaN
a   NaN
b   NaN
c   NaN
d   NaN
e   NaN
dtype: float64
```

None of the indexes matched, because (**0, 1, 2, 3, 4**) of *ob1* do not match with indexes ('a', 'b', 'c', 'd', 'e') of *ob2*, hence **NaN** for all the indexes of both the objects

Figure 1.6 (*b*)

NOTE

When you perform arithmetic operations on two **Series** type objects , the data is aligned on the basis of matching indexes (this is called **Data Alignment in Pandas objects**) and then performed arithmetic; for non-overlapping indexes, the arithmetic operations result as a NaN (Not a Number).

You can store the result of object arithmetic in another object, which will also be a Series object, *i.e.*, if you give :

```
>>> ob6 = ob1 + ob3
```

Then **ob6** will also be a Series object (if **ob1** and **ob3** are Pandas Series objects).

EXAMPLE **17** *Number of students in classes 11 and 12 in three streams ('Science', 'Commerce' and 'Humanities') are stored in two Series objects c11 and 12. Write code to find total number of students in classes 11 and 12 , stream wise.*

SOLUTION

```
import pandas as pd
:  # creating Series objects
c11 = pd.Series(data = [30, 40, 50], index = ['Science','Commerce','Humanities'])
c12 = pd.Series(data = [37, 44, 45], index = ['Science','Commerce','Humanities'])
# adding two objects to get total no. of students
print("Total no. of students")
print(c11+c12) # series objects arithmetic
```

Output

```
Total no. of students
Science      67
Commerce     84
Humanities   95
dtype: int64
```

EXAMPLE **18** *Object1* ***Population*** *stores the details of population in four metro cities of India and Object2* ***AvgIncome*** *stores the total average income reported in previous year in each of these metros. Calculate income per capita for each of these metro cities.*

SOLUTION

Statement continuation mark. Do not type it while typing code in a .py file, rather type the whole statement in single line

```
import pandas as pd
Population = pd.Series([10927986, 12691836, 4631392, 4328063 ], \
index = ['Delhi', 'Mumbai', 'Kolkata', 'Chennai'])
AvgIncome = pd.Series([72167810927986, 85087812691836, 4226784631392, 5261784328063 ], \
index = ['Delhi', 'Mumbai', 'Kolkata', 'Chennai'])
perCapita = AvgIncome / Population
print("Population in four metro cities ")
print(Population)
print("Avg. Income in four metro cities")
print(AvgIncome)
print("Per Capita Income in four metro cities ")
print(perCapita)
```

Expression to calculate per capita income

The output produced by above file is as shown on the right.

See how easy it has become to calculate per capita income if we have huge data stored about all the cities and average income etc.

```
In [113]: runfile('E:/Informatics Practi
perCapita.py', wdir='E:/Informatics Prac
Population in four metro cities
Delhi      10927986
Mumbai     12691836
Kolkata     4631392
Chennai     4328063
dtype: int64
Avg. Income in four metro cities
Delhi      72167810927986
Mumbai     85087812691836
Kolkata     4226784631392
Chennai     5261784328063
dtype: int64
Per Capita Income in four metro cities
Delhi      6.603944e+06
Mumbai     6.704137e+06
Kolkata    9.126381e+05
Chennai    1.215737e+06
dtype: float64
```

5. Filtering Entries

You can filter out entries from a Series object using expressions that are of Boolean type, (*i.e.*, the expressions that yield a Boolean value) as per following syntax :

```
<Series Object>[ [<Boolean expression on Series Object>]
```

Consider following examples (which are based on objects **ob1**, **ob2** and **ob3** from Fig. 1.6) :

Applying comparison operator directly on Series object works in vectorized way i.e., applies this check on each element and then returns True /False for each element

```
In [20]: ob1>5
Out[20]:
0    False
1     True
2    False
3    False
4     True
dtype: bool
```

(a)

```
In [21]: ob1[ob1>5]
Out[21]:
1    12.75
4    46.50
dtype: float64
```

(b)

```
In [23]: ob2>10
Out[23]:
a    False
b     True
c     True
d     True
e     True
dtype: bool
```

(c)

```
In [24]: ob2[ob2>10]
Out[24]:
b    12.75
c    24.00
d    35.25
e    46.50
dtype: float64
```

(d)

*But when this check is applied in the form <Series object> [<Boolean expression>], then it returns **filtered result** containing only the elements that return True for the given Boolean expression.*

When you apply a comparison operator directly on a Pandas Series object, then it works like vectorized operation and applies this check on each individual element of Series object (as you can see in (*a*) and (*c*) above) BUT when you apply this check with the Series object inside [ ] as per syntax given above then, you will find that it returns filtered result containing only the values that return ***True*** for the given Boolean expression (as you can see in (*b*) and (*d*) above).

EXAMPLE 19 *What will be the output produced by the following program ?*

```
import pandas as pd
info = pd.Series(data = [31, 41, 51])
print(info)
print(info > 40)
print(info[info > 40])
```

SOLUTION

```
0    31
1    41
2    51
dtype: int64
```

← *Series object*

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

← *Vectorized operation result*

```
1    41
2    51
dtype: int64
```

← *Filtered result*

EXAMPLE 20 *Series object s11 stores the charity contribution made by each section (see below) :*

```
A   6700
B   5600
C   5000
D   5200
```

Write a program to display which sections made a contribution more than ₹5500/-

SOLUTION

```
import pandas as pd
 :              # s11 created or loaded here
print("Contribution > 5500 by :")
print(s11 [s11 > 5500])
```

Output

```
Contribution > 5500 by :
A   6700
B   5600
dtype: int64
```

6. Sorting Series Values

You can sort the values of a Series object on the basis of values and indexes.

Sorting on the Basis of Values

To sort a Series object on the basis of values, you may use **sort_values( )** function as per the following index :

Optional argument

```
<Series object>.sort_values([ascending =True|False])
```

The argument **ascending** is optional and if skipped, it takes the value ***True*** by default.

It means, the **sort_values( )** arranges the values in a Series object in ascending order by default.

For example consider the Series object **s11**,

```
>>> s11
A  6700
B  5600
C  5000
D  5200
dtype: int64
```

```
>>> s11.sort_values()
C  5000
D  5200
B  5600
A  6700
dtype: int64
```

Values sorted in ascending order (default setting)

```
>>> s11.sort_values(ascending = False)
A  6700
B  5600
D  5200
C  5000
dtype: int64
```

This time the values are sorted in descending order

To sort values in descending order, you may write **<series>.sort_values(ascending = False)**

Sorting on the Basis of Indexes

To sort a Series object on the basis of indexes, you may use **sort_index( )** function as per the following index :

Optional argument

```
<Series object>.sort_index([ascending = True|False])
```

The argument **ascending** is optional and if skipped, it takes the value ***True*** by default.

For example :

```
>>> s11.sort_index(ascending = False)
D  5200
C  5000
B  5600
A  6700
dtype: int64
```

Series sorted in descending order of indexes

```
>>> s11.sort_index()
A  6700
B  5600
C  5000
D  5200
dtype: int64
```

Series sorted in ascending order of indexes

Solved problems 13, 14 are based on sorting a Series object.

1.4.6 Difference between NumPy Arrays and Series Objects

The major differences in *ndarrays* and *Series objects* are listed below :

(*i*) In case of ndarrays, you can perform vectorized operations only if the **shapes** of two ndarrays match, otherwise it returns an error. But with Series objects, in case of vectorized operations, the data of two *Series objects* is aligned as per matching indexes and operation is performed on them and for non-matching indexes, **NaN** is returned. You have already read about this functionality in *section 1.4.5, point 4.*

(*ii*) In ndarrays, the indexes are always numeric starting from 0 onwards, BUT Series objects can have any type of indexes, including numbers (not necessarily starting from 0), letters, labels, strings etc.

```
In [88]: a1
Out[88]: array([1, 2, 3, 4])

In [89]: a3
Out[89]: array([11, 12, 13, 14, 15, 16, 17])

In [90]: a1+a3
Traceback (most recent call last):

  File "<ipython-input-90-f3952d2e37d2>", line 1, in <module>
    a1+a3

ValueError: operands could not be broadcast together with shapes (4,) (7,)
```

Ndarrays cannot perform vectorized operations on arrays with different shapes

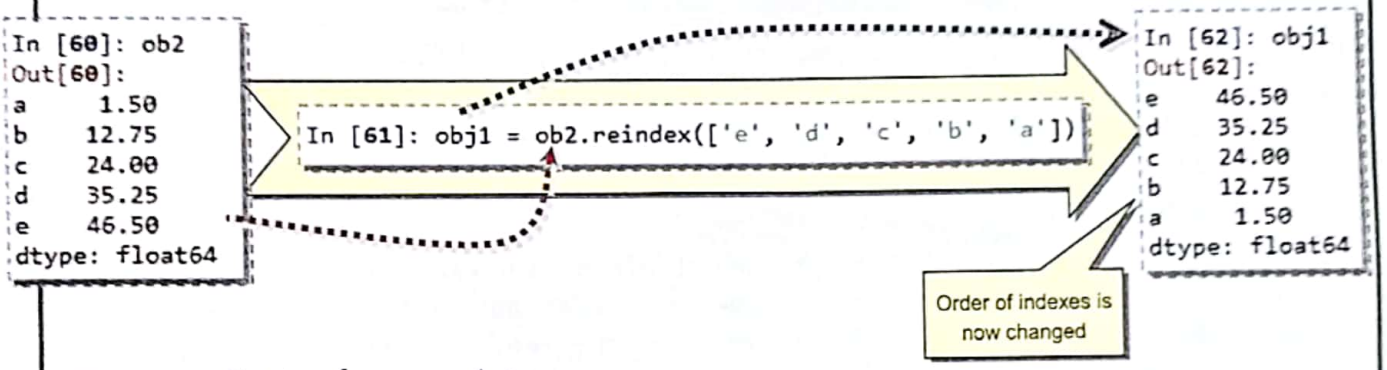
Some Additional Operations on Series Objects

Reindexing

Sometimes you need to create a similar object but with a **different order of same** indexes. You can use reindexing for this purpose as per this syntax :

```
<Series Object > = <Object>.reindex(<sequence with new order of indexes>)
```

With this, the same datavalues and their indexes will be stored in the new object as per the defined order of index in the **reindex( )**. See below :

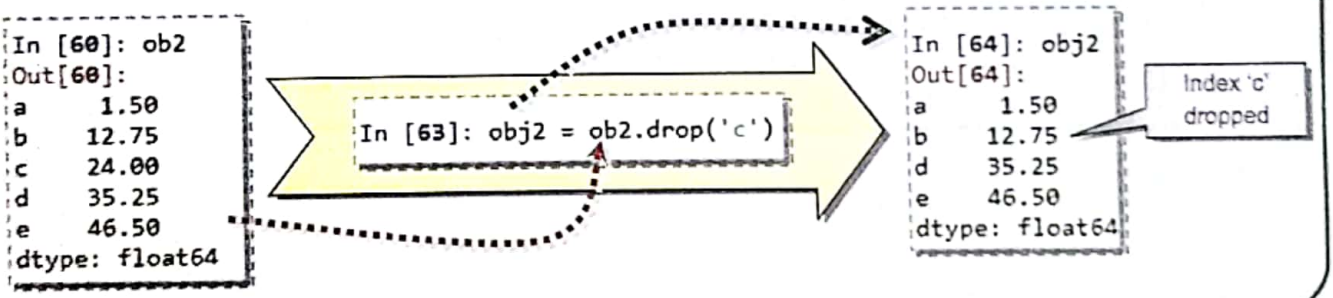


Dropping Entries from an Axis

Sometime, you do not need a data value at a particular index. You can remove that entry from series object using drop( ) as per this syntax :

```
<Series Object>.drop(<index to be removed>)
```

For example, see below :



1.5 DataFrame Data Structure

A **DataFrame** in another Pandas structure, which stores data in two-dimensional way. It is actually a **two-dimensional** (tabular and spreadsheet like) **labeled array**, which is actually an **ordered collection of columns** where columns may store different types of data, *e.g., numeric* or *string* or *floating point* or *Boolean* type etc.

Since DataFrame data structure is like a two-dimensional array, let us first understand what a two-dimensional array is like.

DATAFRAME

" A **DataFrame** is a two-dimensional labeled array like Pandas data structure that stores an ordered collection columns that can store data of different types. "