Class XII - Chapter 1 – Working With Numpy

Introducing Python Pandas

- ✓ Python ***Panda is Python's library*** for **data analysis**.
- ✓ Panda – " **Pan**el **Da**ta **A**nalysis"

What is Data Analysis?

It refers to process of **evaluating big data** sets using analytical & statistical tools so as to discover useful information and conclusion **to support business decision making**.

Python pandas & Data Analysis

- ✓ Python pandas provide various tools for data analysis and makes it a simple and easy process.
- ✓ Author of Pandas is **Wes Mckinney**.

Using Pandas

- ✓ Pandas is an **opens source library** built for **Python programming language**, which provides high performance data analysis tools.
- ✓ In order to work with pandas in Python, you need to **import pandas library** in your python environment.
- ✓ **Benefits of using Panda for Data Analysis**
  1. It can **read or write** in many different data formats(integer,float,double,etc.)
  2. It can **calculate in all ways** data is organized, i.e., across rows and down columns.
  3. It can **easily select subsets** of data from bulky data sets and even **combine multiple datasets** together.
  4. It has functionality to **find and fill** missing data.
  5. It supports **advanced time-series functionality**(Time series forecasting is the use of a model to predict future values based on previously observed values)

*****Pandas is best at handling huge tabular data sets comprising different data formats.***

NumPy Arrays

- ✓ **NumPy('Numerical Python' or 'Numeric Python')** is an open source module of Python that offers functions and routines for fast mathematical computation on array and matrices.
- ✓ In order to use Numpy, you must import in your module by using a statement like:

```
import numpy as np
```

You can use any identifier name in place of np

- ✓ The above statement has given **np as alias name for numpy module**. Once imported you can use both names i.e. numpy or np for functions, **e.g.** numpy.array( ) is same as np.array( ).

Array

- ✓ It refers to a named **group of homogenous** (of same type) elements. E.g. **students array** containing 5 entries as [34, 37, 36, 41, 40] then students is an array.

Types of Numpy array

- ✓ A ***NumPy array is simply a grid that contains values of the same/homogenous type***. NumPy Arrays come in two forms:
  - 1-D(one dimensional) arrays known as **Vectors**(having single row/column only)
  - Multidimensional arrays known as **Matrices**(can have multiple rows and columns)

Example 1: (Creating a 1-D Numpy array)

```
import numpy as np
list = [1,2,3,4]
a1=np.array(list)
print(a1)
```

It will create a NumPy array from the given list

Output : [1 , 2 , 3 , 4]

*****Individual elements of above array can be accessed just like you access a list's i.e. arrayname [index]***

Example 2: (Creating a 2-D Numpy array)

```
import numpy as np
a7 = np.array([ [10,11,12,13] , [21,22,23,24] ])
print(a7[1,3])
print(a7[1][3])
print(a7)
```

This is a 2-D array having rank 2

You can access elements of multi-dimension arrays as

<array>[row][col]

or as

<array>[row, col]

Output:

```
24
24
[[10 11 12 13]
 [21 22 23 24]]
```

Storage of 2D Arrays in Memory

Elements of arrays are stored in ***contiguous memory locations***. Therefore, 2D arrays are linearized for storage purpose in one of these two alternatives.

(i) Row-major or row wise
(ii) Column-major or column-wise

Row Major Implementation of 2D Arrays

This linearization technique stores firstly the first row of the array, then the second row of the array, then the third row, and so forth.

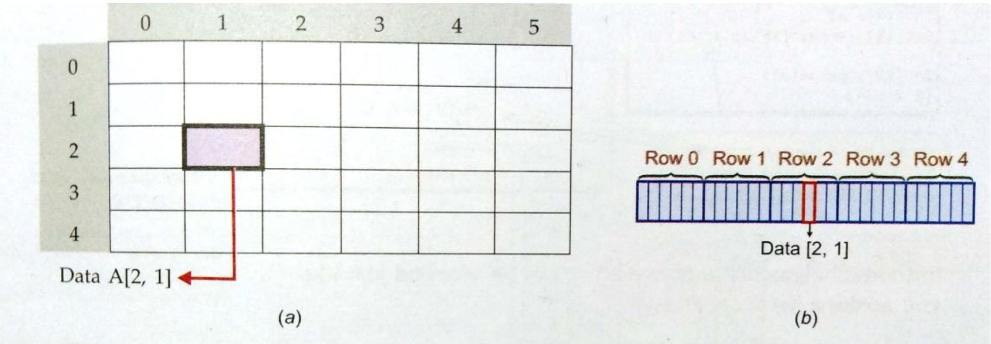


Figure 1.1 (a) Logical implementation (b) Physical Row-major implementation

Column Major Implementation of 2D Arrays

This linearization technique stores firstly the first column of the array, then the second column of the array, then the third column, and so forth.

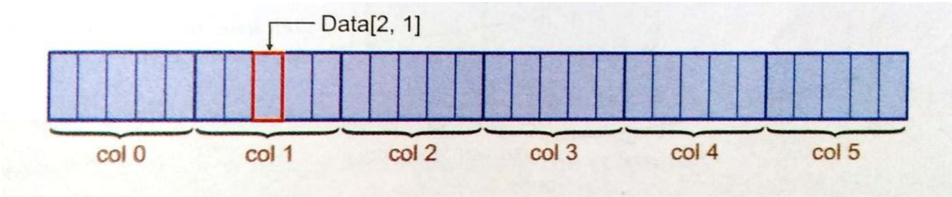
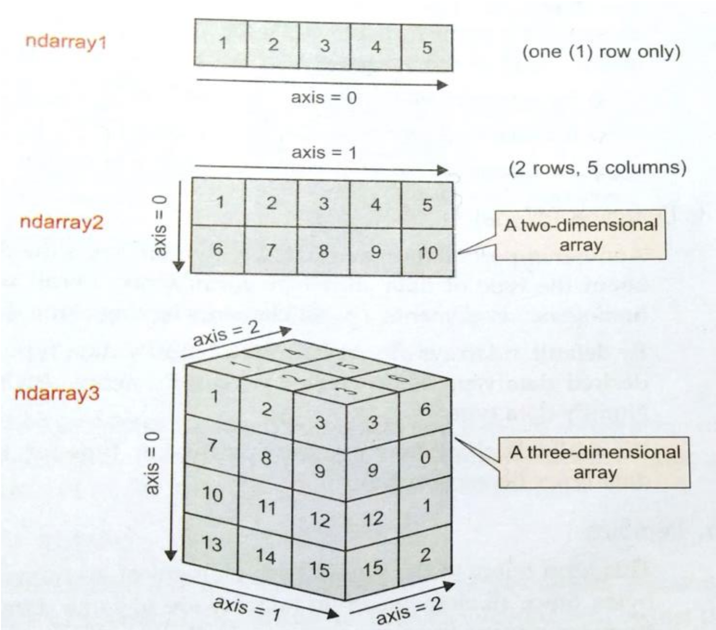


Figure 1.2 Column-major implementation.

Terms associated with Numpy Arrays

1. **Axes**
   - ✓ Numpy refers to the dimensions of its arrays as **axes.** The **axes** of an ndarray also describe the order of indexing in multi-dimensional ndarrays.



   - ✓ Axes are always numbered 0 onwards for ndarrays.
2. **Rank**
   - ✓ The number of axes in an ndarray is called its **rank.**
3. **Shape**
   - ✓ The shape of an ndarray tells about the **number of elements along each axis of it**.
4. **Datatype(dtype)**
   - ✓ It tells about the type of data stored in the ndarray.
   - ✓ By default, ndarrays have the datatype as float.

5. **Itemsize**
   - ✓ This term refers to the **size of each element** of an ndarray **in bytes**.
   - ✓ The datatype and itemsize are related. The itemsize is as per the datatype e.g., for data type int16(16 bit integer), the itemsize is 2 bytes(equal to 16 bits).

6. **type( ) function in NumPy**
   - ✓ It is used to **check the type of objects** in Python.

Example:

```
import numpy as np
list=[1,2,3,4]
a1=np.array(list)
a2 = np.array([ [10,11,12,13] , [21,22,23,24] ])
print(type(a1))
print(type(a2))
print(a1.shape)
print(a2.shape)
print(a2.itemsize)
```

The **shape** attribute gives the dimensions of a NumPy array.

The **itemsize** attribute returns the length of each element of array in bytes.

Output:

```
<class 'numpy.ndarray'>
<class 'numpy.ndarray'>
(4,)
(2, 4)
4
```

Difference between NumPy and List

S.No.	NumPy	List
1.	Once a Numpy array is created, you cannot change its size.	Size can be changed.
2.	Every NumPy array contain elements of homogenous types, i.e. all its elements have one and only one data type.	List can contain elements of different data type.
3.	NumPy arrays support vectorized operations, i.e. if you apply a function, it is performed on every item in the array.	It does not support vectorized.

NumPy Data Types

The NumPy arrays can have elements in data types supported by NumPy. Following table are the data types supported by NumPy:

S.No.	Data Type	Description	Size
1.	bool_	Boolean data type (stores *True* or *False*)	1 byte
2.	int_	Default type to store integers in ***int32*** or ***int64***	4 or 8 bytes
3.	int8	Stores signed integers in range –128 to 127	1 byte
4.	int16	Stores signed integers in range –32768 to 32767	2 bytes
5.	int32	Stores signed integers in range -2^{16} to $2^{16}-1$	4 bytes
6.	int64	Stores signed integers in range -2^{32} to $2^{32}-1$	8 bytes
7.	uint8	Stores unsigned integers in range 0 to 255	1 byte
8.	uint16	Stores integers in range 0 to $2^{16}-1$	2 bytes
9.	uint32	Stores integers in range 0 to $2^{32}-1$	4 bytes
10.	uint64	Stores integers in range 0 to $2^{64}-1$	8 bytes
11.	float_	Default type to store floating point (*float64*)	8 bytes
12.	float16	Stores **half precision floating point values** (*5 bits* exponent, *10 bit* mantissa)	2 bytes
13.	float32	Stores **single precision floating point values** (*8 bits* exponent, *23 bit* mantissa)	4 bytes

S.No.	Data Type	Description	Size
14.	float64	Stores **double precision floating point values** (*11 bits* exponent, *52 bit* mantissa)	8 bytes
15.	complex_	Default type to store complex numbers *(complex128)*	16 bytes
16.	complex64	Complex numbers represented by *two float32 numbers* for *real* and *imaginary* value components.	8 bytes
17.	complex128	Complex numbers represented by *two float64numbers* for *real* and *imaginary* value components.	16 bytes
18.	string_	Fixed-length string type.	1 byte per character
19.	unicode_	Fixed-length Unicode type.	number of bytes platform specific

Creating Numpy Arrays

1. **Using array( ) function**

   The array( ) is useful for creating ndarrays **from existing lists and tuples**. (see example given on pg.no.2)

2. **Using fromiter**

   - To create ndarrays from sequence of all types (numeric sequence, or string sequence or dictionaries etc.), you can use fromiter( ) function.

   - The syntax to use fromiter( ) function is :

numpy.fromiter(<iterable sequence >, dtype=<datatype>, [count=<number of elements to be read>])

If skipped, then all the elements are read.

ndarray from a dictionary

adict = { 1 : 'A' , 2 : 'B' , 3 : 'C' , 4 : 'D' , 5 : 'E' }
ar5 = np.fromiter(adict , dtype=np.int32)

The above statement will create an ndarray ***from the keys of dictionary*** adict having numpy datatype int32 (i.e., 32 bits or 4 bytes long).

ndarray from a String

Each element of ndarray can have length of 2 unicode characters.

astr = "thisIsTrue"
ar6 = np.fromiter(astr, dtype="U2")
print(ar6)
print(ar6[0] , ar6[4])

picking a smaller set of elements from a sequence using fromiter( )

astr = "thisIsTrue"
ar7 = np.fromiter(astr, dtype="U1" , count=3)
print(ar7)

count=3 means only first 3 characters will be picked from the string astr for the ndarray.

3. **Creating arrays with a numerical range using arange( )**

arange( ) creates a NumPy array with evenly spaced values within a specified numerical range. It is used as:

```
<arrayname> = numpy.arange([start,] stop [, step ] [, dtype ])
```

- The **start**, **stop** and **step** attribute provide the values for starting value stopping value and step value for a numerical range. **Start** and **step** values are optional. When only stop value is given , the numerical range is generated from *zero* to *stop value* with *step* 1.
- The **dtype** specifies the datatype for the NumPy array.

Example:

```
import numpy as np
arr1 = np.arange(7)
print(arr1)
arr2=np.arange(1,7,2,np.float32)
print(arr2)
```

Output:

```
[0 1 2 3 4 5 6]
[1. 3. 5.]
```

4. Creating arrays with a numerical range using linspace( )

linspace( ) is used to generate evenly spaced elements between two given limits.

```
<arrayname> = numpy.linspace(<start>, <stop>, <number of values to be generated>)
```

Example:

```
import numpy as np
arr1 = np.linspace(2,10,3)
print(arr1)
```

Output:

```
[ 2.  6. 10.]
```

5. Creating a 2-dimensional ndarrays using array( )

Refer example 2 on page no. 2.

6. Creating 2D ndarray using arange( )

Two steps: 1. Create an ndarray using **arange( )**

2. Reshape the ndarray created in previous step using reshape( ) as per syntax:

<ndarray>.reshape(<rows, columns>)

Consider following examples :

ar = np.arange(10)

```
In [3]: ar
Out[3]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

ar1 = ar.reshape(5,2)

```
In [5]: ar1
Out[5]:
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]])
```

Ndarray namely **ar1** created from **ar**. **ar1** has same number of elements but different shape (5 rows × 2 columns)

ar2 = ar.reshape(2,5)

```
In [7]: ar2
Out[7]:
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

Ndarray namely **ar2** created from **ar**. **ar2** has same number of elements but different shape (2 rows × 5 columns)

*** The no. of elements in the originally created ndarray must be the same as that of new 2D array being created through reshape( ).*

You can also combine arange( ) and reshape( ) in single statement as shown below:

```
ary = np.arange(8.0) . reshape(2, 4)
print(ary)
```

7. <u>Creating empty arrays using empty( )</u>

Sometimes you need to create empty arrays or an uninitialized array of specified shape and dtype, in which you can store actual data as and when required. For this you can use empty( ) function as:

```
numpy.empty(shape, [dtype = <Python's datatype or NumPy datatype>,] [ order = 'C' or 'F'])
```

(In place of **numpy**, you can also use **np** as you have given alternate name for ***numpy*** as ***np*** in the import statement)

- **shape** specifies the dimensions and is given as list *e.g.*, [*row*, *cols*]
- **order** as 'C' arranges array elements **row-wise** in memory that is, first row's elements then the second row's elements and so on. ('C' means 'C' – like)
- **order** as 'F' arranges array elements **row-wise** in memory that is, first row's elements then the second row's elements and so on. ('F' means 'Fortran' – like)

Both **dtype** and **order** are optional. By default **dtype** is taken as **float**, *i.e.*, when you do not specify any **dtype**. Similarly default order is **'C'**.

** After creating empty array, if you display the contents of the array, it will display any random contents, which are ***uninitialized garbage values.***

<u>Example:</u>

```
import numpy as np
arr1 = np.empty([3,2])
arr2 = np.empty([3,4] , dtype=np.int8)
print(arr1.dtype , arr2.dtype)
print(arr1)
```

No dtype specified

dtype specified as int8

empty( ) creates array with any random garbage values

Output:

```
float64 int8
[[2.67276450e+185 1.69506143e+190]
 [1.75184137e+190 9.48819320e+077]
 [1.63730399e-306 0.00000000e+000]]
```

8. **Creating arrays filled with zero using zeros( )**

The function zeros( ) takes same attributes as empty( ), and creates an array with specifies size and type but filled with zeros.

`numpy.zeros(shape, [dtype = <Python's datatype or NumPy datatype>,] [ order = 'C' or 'F'])`

(In place of **numpy**, you can also use **np** as you have given alternate name for ***numpy*** as ***np*** in the import statement)

- **shape** and **order** attributes work in identical way as in **empty( )** (refer to syntax details of empty( ) function above)

Example:

```
import numpy as np
arr1 = np.zeros([3,2],dtype=np.int64)
print(arr1)
```

Output:

```
[[0 0]
 [0 0]
 [0 0]]
```

9. **Creating arrays filled with 1's using ones( )**

The function ones( ) takes same attributes as empty( ), and creates an array with specified size and type but filled with ones.

`numpy.ones(shape, [dtype = <Python's datatype or NumPy datatype>,] [ order = 'C' or 'F'])`

(In place of **numpy**, you can also use **np** as you have given alternate name for **numpy** as **np** in the import statement)

- **shape** and **order** attributes work in identical way as in **empty( )** (refer to syntax details of empty( ) function above)

Example:

```
import numpy as np
arr1 = np.ones([3,2],dtype=np.int64)
print(arr1)
```

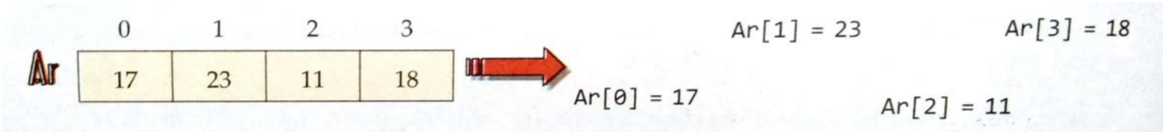
Output:

```
[[1 1]
 [1 1]
 [1 1]]
```

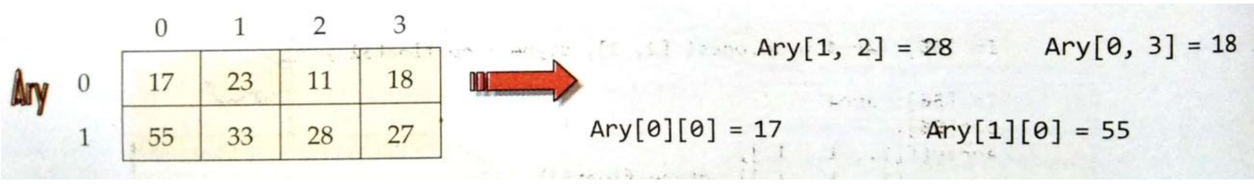
** There are three more functions **empty_like( ), zeros_like( ) and ones_like( )** that you can use to create an array similar to another existing array.

Accessing Individual Elements using Array Indexing

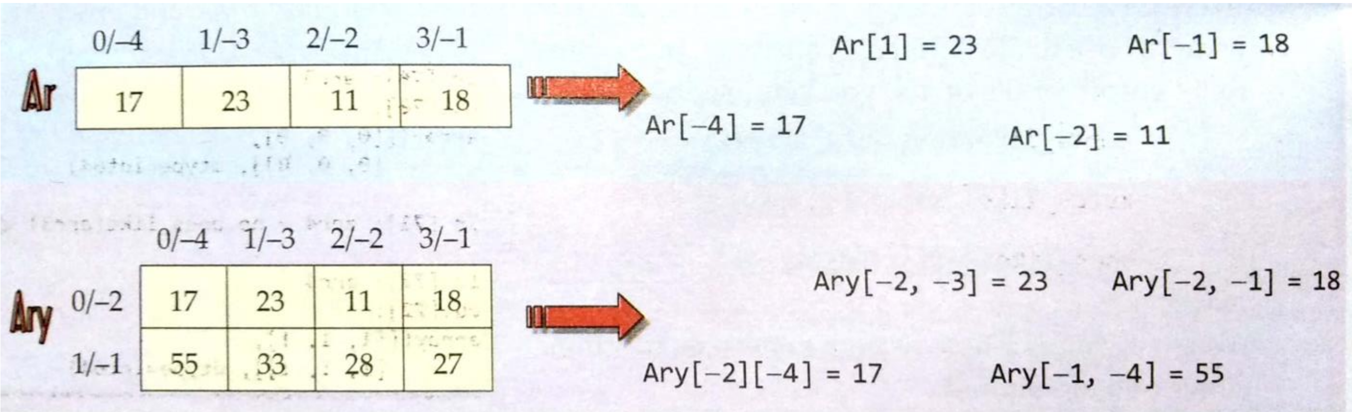
1. **For 1D arrays** - Syntax : <1D array>[<index>]



2. **For 2D arrays** – Syntax : (i) <2D array> [<rowindex>, <column index>]
   (ii)<2D array>[<rowindex>][<columnindex>]



**Negative indexes are also valid like in lists or strings,



Array Slices

- It refers to the process of **extracting a subset of elements from an existing array** and returning the result as another array, possibly in a different dimension from the original.

  Syntax for performing slicing : <Arrayname>[<start>: <stop> : <step>]

- When <start> , <stop> or <step> values are not specified then Python will assume their default values as :

  start = 0
  stop = dimension size
  step = 1

1D Array Slices

Given NumPy Array `Ar = = np.array([2, 4, 6, 8, 10, 12, 14, 16])`

1D array slice	Description	Example
`Ar[n:m]`	Extract 1D slice from n to m–1	`>>> Ar[3:7]` `array([ 8, 10, 12, 14])`
`Ar[:m]`	Extract 1D slice from 0 to m–1	`>>> Ar[:5]` `array([ 2, 4, 6, 8, 10])`
`Ar[n:]`	Extract 1D slice from n to the end	`>>> Ar[4:]` `array([10, 12, 14, 16])`
`Ar[n:-1]`	Extract 1D slice from n to end –1	`>>> Ar[:-1]` `array([2, 4, 6, 8, 10, 12, 14])`
`Ar[n:-2]`	Extract 1D slice from n to end –2	`>>> Ar[:-2]` `array([2, 4, 6, 8, 10, 12])`
`Ar[n:-2]`	Extract 1D slice from n to end –3	`>>> Ar[:-3]` `array([2, 4, 6, 8, 10])`
`Ar[n:m:k]`	Extract 1D slice from n to m–1 picking every *k*th element	`>>> Ar[2:7:2]` `array([6, 10, 14])`

2D Array Slices

- For extracting a slice from a 2D array, you need to specify syntax as:

row slicing parameters *column slicing parameters*

```
<array> [<start>:<stop>:<step>, <start>, <stop>:<step>]
```

- Like 1D array slices, when not specified , <start> takes default value 0, <stop> takes dimension size and <step> takes default value of 1.

- 2D array slice is computed as :
  (i) Extract rows as per row slice specified.
  (ii) On the extracted rows, apply column slice to get the desired 2D array slice.

Ary

	0/−5	1/−4	2/−3	3/−2	4/−1
0/−5	2	4	6	8	10
1/−4	12	14	16	18	20
2/−3	22	24	26	28	30
3/−2	32	34	36	38	40

A 5×5 array
[4 rows × 5 columns]

Example 1 Slice **Ary[:3, 3:]**

row slice = **:3**

$\Rightarrow$ start = 0, stop = 3, step = 1
i.e., **all row indexes : row-index < 3**

column slice = **3:**

$\Rightarrow$ start = 3, stop = 5, step = 1
i.e., all column indexes : **3 ≤ col-index** < **5**

Thus 2D slice will have
rows with index < 3
columns with **3 ≤ col-index < 5**

i.e.,

	0	1	2	3	4
0	2	4	6	8	10
1	12	14	16	18	20
2	22	24	26	28	30
3	32	34	36	38	40

row index < 3

3 ≤ col-index < 5

This meets the criteria and hence is the resultant slice (see output)

```
In [35]: Ary
Out[35]:
array([[ 2,  4,  6,  8, 10],
       [12, 14, 16, 18, 20],
       [22, 24, 26, 28, 30],
       [32, 34, 36, 38, 40]])

In [36]: Ary[ :3, 3: ]
Out[36]:
array([[ 8, 10],
       [18, 20],
       [28, 30]])
```

Example 2. Slice **Ary[1 : : 2, : 3]**

row slice = 1 : : 2

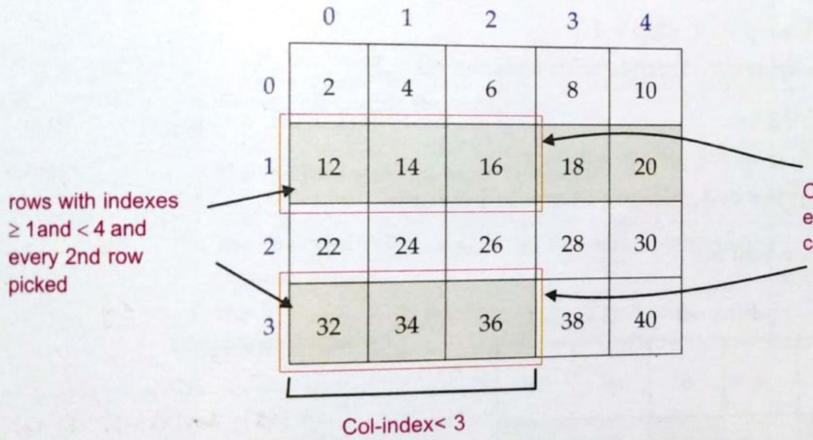
$\Rightarrow$ start = 1, stop = 4, step = 2

i.e., all row indexes $\geq$ 1 and < 4 and pick every 2nd row skipping in between

col slice = **: 3**

$\Rightarrow$ start = 0, stop = 3 *i.e.*, **col-index < 3**

Thus 2D slice will be



Thus the result will be as shown here.

```
In [38]: Ary[1: :2, :3 ]
Out[38]:
array([[12, 14, 16],
       [32, 34, 36]])
```

Example 3 **Ary[: : 3, : : 2]**

row slice = **: : 3**

$\Rightarrow$ start = 0, stop = 4, step = 3

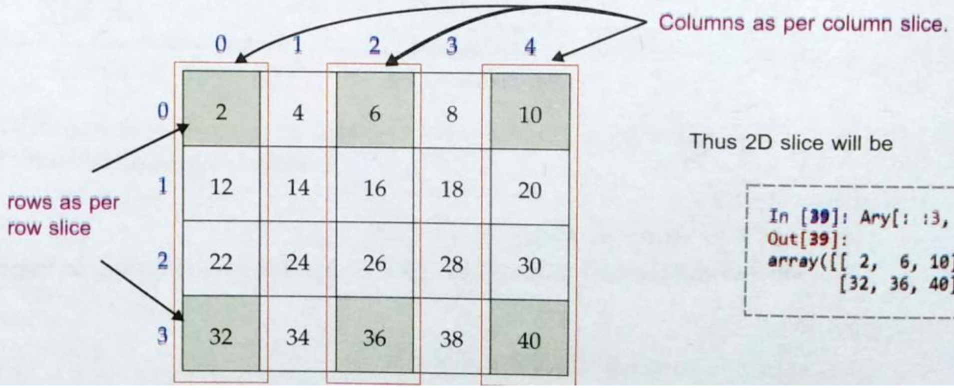
i.e., pick every 3rd row starting from 0th row such that row index remains < 4

column slice = **: : 2**

$\Rightarrow$ start = 0, stop = 5, step = 2

i.e., pick every 2nd column starting from 0th column such that col-index remains < 5.

Thus 2D slice will be



Thus 2D slice will be

```
In [39]: Ary[: :3, : :2]
Out[39]:
array([[ 2,  6, 10],
       [32, 36, 40]])
```

Example 4 **Ary[–3 : –1, –5 : : 2]**

row slice = **– 3 : –1**

⇒ start = –3, stop = –1, step = 1

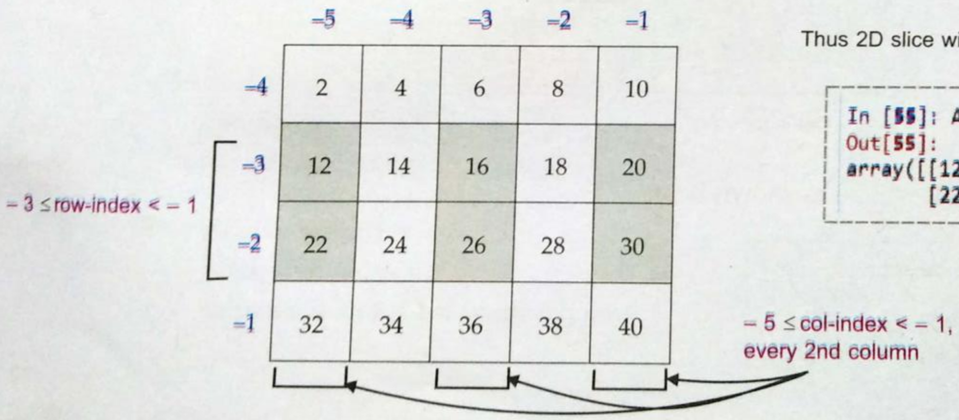
–3 ≤ row-index < –1, rows with indexes –3, –2

column slice = **– 5 : : 2**

⇒ start = – 5, stop = 4 or – 1, step = 2

– 5 ≤ col-index < –1, picking every 2nd column

Thus the extracted 2D slice will be



Thus 2D slice will be

```
In [55]: Ary[-3:-1, -5: :2]
Out[55]:
array([[12, 16, 20],
       [22, 26, 30]])
```

Some more examples of 2D array slicing are being given below.

> **NOTE**
> Giving dimensions as [:: −1, : : −1] reverses the entire 2D ndarray in both dimensions *i.e.*, horizontally as well as vertically. Solved problem 11 uses this.

Given NumPy Array Ary

```
array([[ 2,  4,  6,  8, 10],   [12, 14, 16, 18, 20],
       [22, 24, 26, 28, 30],   [32, 34, 36, 38, 40] ] )
```

2D array slice	Description	Example

`Ary[n:m,j:k]`	The 2D slice with rows from **n to m-1**, and columns from **j to k-1**	`>>> Ary[1:3, 3:5]`
		`array([[18, 20],`
		`       [28, 30]])`

`Ary[n:m,:]`	The 2D slice with rows from **0 to m-1**, all columns	`>>> Ary[1:3, ]`
		`array([[12, 14, 16, 18, 20],`
		`       [22, 24, 26, 28, 30]])`

`Ary[:,j:k]`	The 2D slice all rows, and columns from **j to k-1**	`>>> Ary[ : , 3:5]`
		`array([[ 8, 10],`
		`       [18, 20],`
		`       [28, 30],`
		`       [38, 40]])`

`Ary[n:m:p,j:k:l]`	The 2D slice with rows from **n to m-1** picking every p^{th} row, and columns from **j to k-1** picking every l^{th} column	`>>> Ary[1:4:2, 1:5:3]`
		`array([[14, 20],`
		`       [34, 40]])`

`Ary[n:-1,:]`
 	
`Ary[n:-2,:]`
	The 2D slice with rows from **n to end -1**, all columns
	The 2D slice with rows from **n to end -2**, all columns
		`>>> Ary[2:-1,]`
		`array([[22, 24, 26, 28, 30]])`
		`>>> Ary[1:-2,]`
		`array([[12, 14, 16, 18, 20]])`

`Ary[:,j:-2]`	The 2D slice all rows, columns **j to k-2**	`>>> Ary[: , 1 :-2 ]`
		`array([[ 4,  6],`
		`       [14, 16],`
		`       [24, 26],`
		`       [34, 36]])`

`Ary[n,:]`	The 2D slice with row *n*, all columns	`>>> Ary[3, ]` `array([32, 34, 36, 38, 40])`
`Ary[:,n]`	The 2D slice with all rows, column *n*	`>>> Ary[:, 2]` `array([ 6, 16, 26, 36])`
`Ary[3,::-1]`	The 2D slice with row 3, all columns; with every element reversed	`>>> Ary[3, ::-1]` `array([40, 38, 36, 34, 32])`

`Ary[:3,::-1]`	The 2D slice with all rows < 3, all columns, with reversed elements	`>>> Ary[:3, ::-1]`
		`array([[10,  8,  6,  4,  2],`
		`       [20, 18, 16, 14, 12],`
		`       [30, 28, 26, 24, 22]])`

`Ary[:3,::-2]`	The 2D slice with all rows < 3, from all columns pick every 2nd column in reversed order.	`>>> Ary[:3, ::-2]`
		`array([[10,  6,  2],`
		`       [20, 16, 12],`
		`       [30, 26, 22]])`

`Ary[-3:-1,-4::2]`	The 2D slice with rows as $-3 \leq$ **row** < -1 and from columns, **pick every 2nd column** with condition $-4 \leq$ **col**	`>>> Ary[-3:-1, -4: :2]`
		`array([[14, 18],`
		`       [24, 28]])`

Joining or Concatenating NumPy Arrays

1. Using hstack( ) and vstack( )
2. Using concatenate( )

1. Combining existing arrays horizontally or vertically

- Sometimes you want to create a 2D array from existing 1D or 2D arrays by stacking them next to one another, e.g.

- If you have two 1D arrays as :

1	4	9	3

6	5	7	2

Now, you may want to create a 2D array by stacking these two 1D arrays

horizontally as :

1	4	9	3	6	5	7	2

Syntax : numpy. hstack(<**tuple** containing names of 1D arrays to be stacked>)

or , vertically as :

1	4	9	3
6	5	7	2

Syntax : numpy. vstack(<**tuple** containing names of 1D arrays to be stacked>)

- Consider following examples. Suppose you have following sequences/arrays:

  lst1 = [1, 2, 3]
  lst2 = [4, 5, 6]
  lst3 = [[9, 8, 7] ,
          [6, 5, 4]]
  lst4 = [[4] ,
          [5]]

Now you can combine them vertically using vstack( ) as :

```
sar1 = np.vstack( ( lst1, lst2 ) )
```

Make sure to provide the names of existing arrays/lists/tuples etc. in a tuple

```
In [58]: sar1
Out[58]:
array([[1, 2, 3],
       [4, 5, 6]])

In [59]: sar1.shape
Out[59]: (2, 3)
```

```
sar2 = np.vstack( (lst2, lst3) )
```

Vertically stacked lst2 and lst3

```
In [61]: sar2
Out[61]:
array([[4, 5, 6],
       [9, 8, 7],
       [6, 5, 4]])

In [62]: sar2.shape
Out[62]: (3, 3)
```

```
sar3 = np.hstack( (lst3, lst4) )
```

Horizontally stacked lst3 and lst4

```
In [64]: sar3
Out[64]:
array([[9, 8, 7, 4],
       [6, 5, 4, 5]])

In [65]: sar3.shape
Out[65]: (2, 4)
```

** for ***hstack( ) to work***, the arrays being joined must match in their vertical size (rows) and for **vstack( ) to work**, the arrays being joined must match in their horizontal size (columns).

Joining 2D arrays using hstack( ) and vstack( )

```
>>> Arr1 = np. array([[0, 1, 2],
     [3, 4, 5],
     [6, 7, 8]])
>>> Arr2 = np.array([[10, 11, 12],
     [13, 14, 15],
     [16, 17, 18]])
>>> Arr3 = np.vstack((Arr1, Arr2))
>>> Arr3
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [10, 11, 12],
       [13, 14, 15],
       [16, 17, 18]])
```

See, the two arrays Arr1 and Arr2 got joined vertically

```
>>> Arr4 = np.hstack((Arr1, Arr2))
>>> Arr4
array([[ 0,  1,  2, 10, 11, 12],
       [ 3,  4,  5, 13, 14, 15],
       [ 6,  7,  8, 16, 17, 18]])
```

See, the two arrays Arr1 and Arr2 got joined horizontally

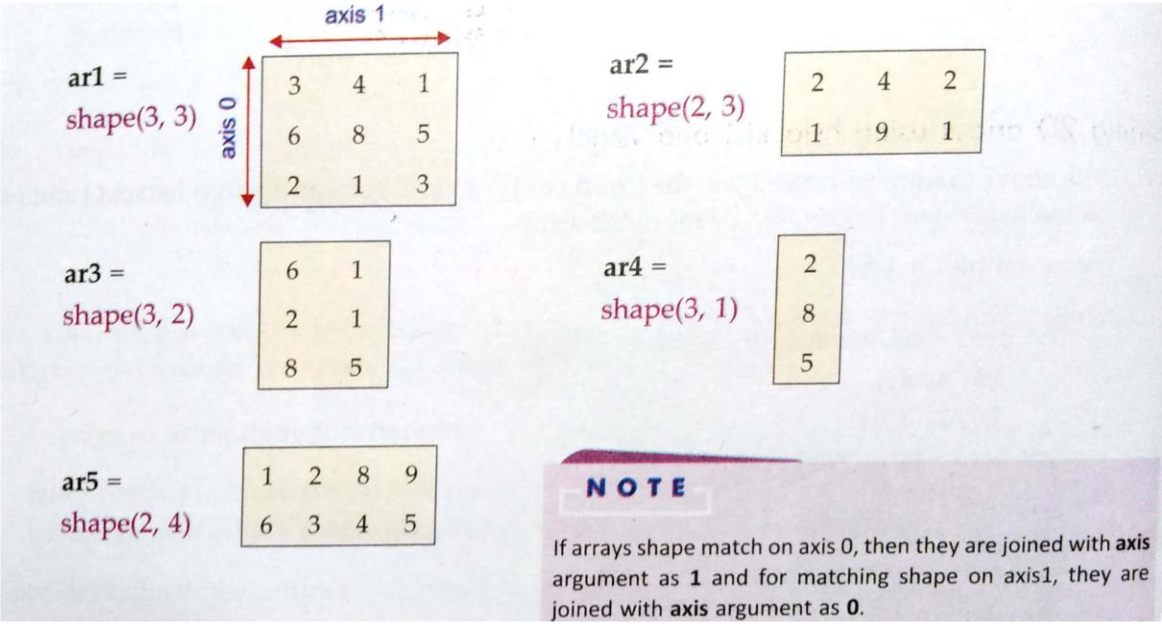
2. Combining existing arrays using concatenate( )

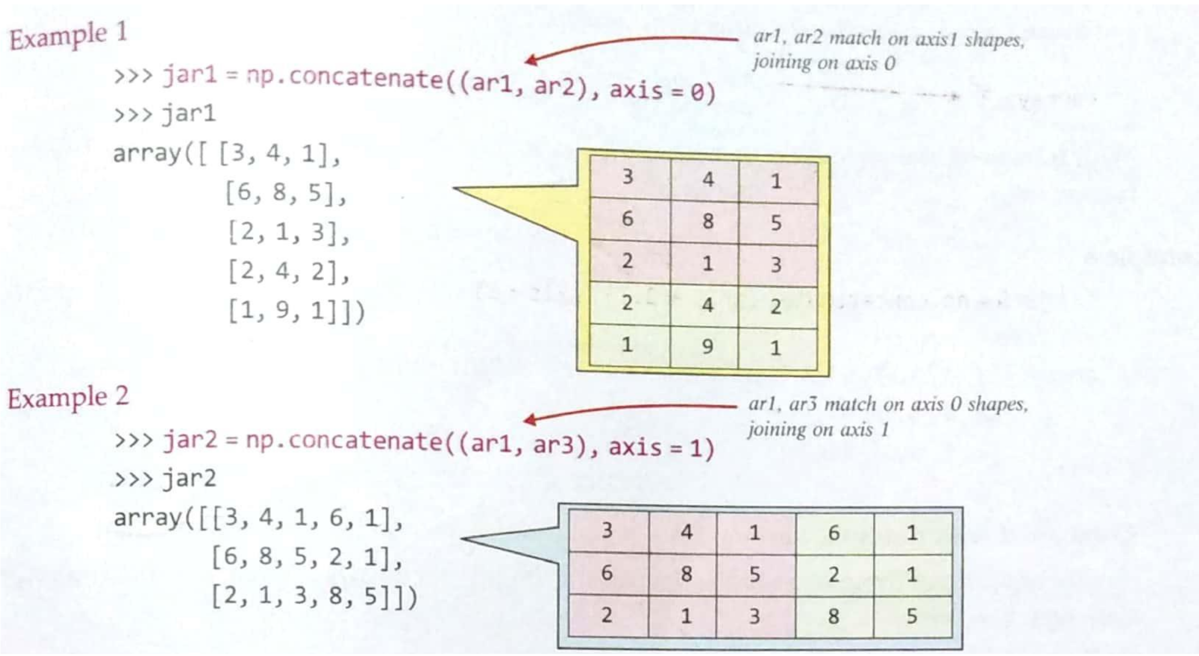
- The syntax for using concatenate( ) is :
  numpy. concatenate(<tuple of arrays to be joined>, [axis = <n>])

- The **axis** argument specifies the axis along which arrays are to be joined. **If skipped, axis is assumed as 0** (i.e., along the rows).
  If you specify axis = 1, then arrays are joined on axis 1, i.e., along the columns.

- If axis is 0, then the shape of the arrays being joined must match on column dimension.
  If axis is 1, then the shape of the arrays being joined must match on rows dimension.

Consider the following arrays:





Example 1

```
>>> jar1 = np.concatenate((ar1, ar2), axis = 0)
>>> jar1
array([ [3, 4, 1],
        [6, 8, 5],
        [2, 1, 3],
        [2, 4, 2],
        [1, 9, 1]])
```

ar1, ar2 match on axis1 shapes, joining on axis 0

3	4	1
6	8	5
2	1	3
2	4	2
1	9	1

Example 2

```
>>> jar2 = np.concatenate((ar1, ar3), axis = 1)
>>> jar2
array([[3, 4, 1, 6, 1],
       [6, 8, 5, 2, 1],
       [2, 1, 3, 8, 5]])
```

ar1, ar3 match on axis 0 shapes, joining on axis 1

3	4	1	6	1
6	8	5	2	1
2	1	3	8	5

Example 3

```
>>> jar3 = np.concatenate((ar1, ar4), axis = 1)
>>> jar3
array([[3, 4, 1, 2],
       [6, 8, 5, 8],
       [2, 1, 3, 5]])
```

3	4	1	2
6	8	5	8
2	1	3	5

Example 4

```
>>> jar4 = np.concatenate((ar2, ar1), axis = 0)
>>> jar4
array([[2, 4, 2],
       [1, 9, 1],
       [3, 4, 1],
       [6, 8, 5],
       [2, 1, 3]])
```

2	4	2
1	9	1
3	4	1
6	8	5
2	1	3

Example 5

```
>>> jar5 = np.concatenate((ar2, ar5), axis = 1)
>>> jar5
array([[2, 4, 2, 1, 2, 8, 9],
       [1, 9, 1, 6, 3, 4, 5]])
```

2	4	2	1	2	8	9
1	9	1	6	3	4	5

Transposing an array for concatenation

With transpose, the axes get swapped and you can join the arrays on non-matching axis. To get the transpose of an array, all you need to write is :

<array>.T

Example:

Example 6

joining ar1 and transpose of ar2(ar2.T) ar1 and ar2.1 having matching shapes on axis 0, thus joining on axis 1.

```
>>> jar6 = np.concatenate( (ar1, ar2.T), axis = 1)
>>> jar6
array([[3, 4, 1, 2, 1],
       [6, 8, 5, 4, 9],
       [2, 1, 3, 2, 1]])
```

3	4	1	2	1
6	8	5	4	9
2	1	3	2	1

** If you specify **axis = None**, then the arrays gets flattened. E.g.

Example 7

```
>>> jar7 = np.concatenate( (ar1, ar4), axis = None)
>>> jar7
array([3, 4, 1, 6, 8, 5, 2, 1, 3, 2, 8, 5])
```

With **axis = None**, the resultant array gets flattened

Splitting NumPy Arrays to Get Contiguous Subsets

1. **The hsplit( ) and vsplit( ) functions**
   - **hsplit( ) function** is used to extract the subsets of a Numpy array after **splitting it horizontally**. Similarly, you can use **vsplit( )** function to extract the subsets of a Numpy array after **splitting it vertically**.

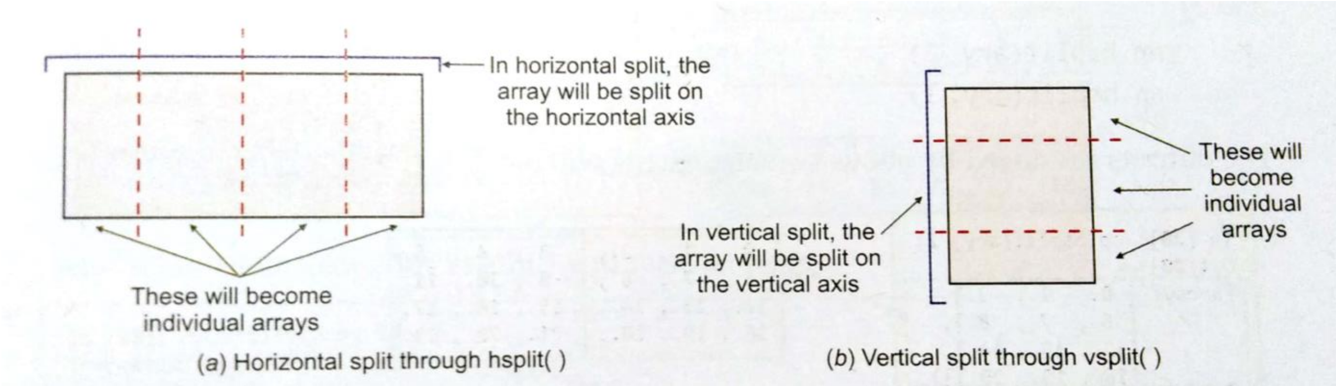


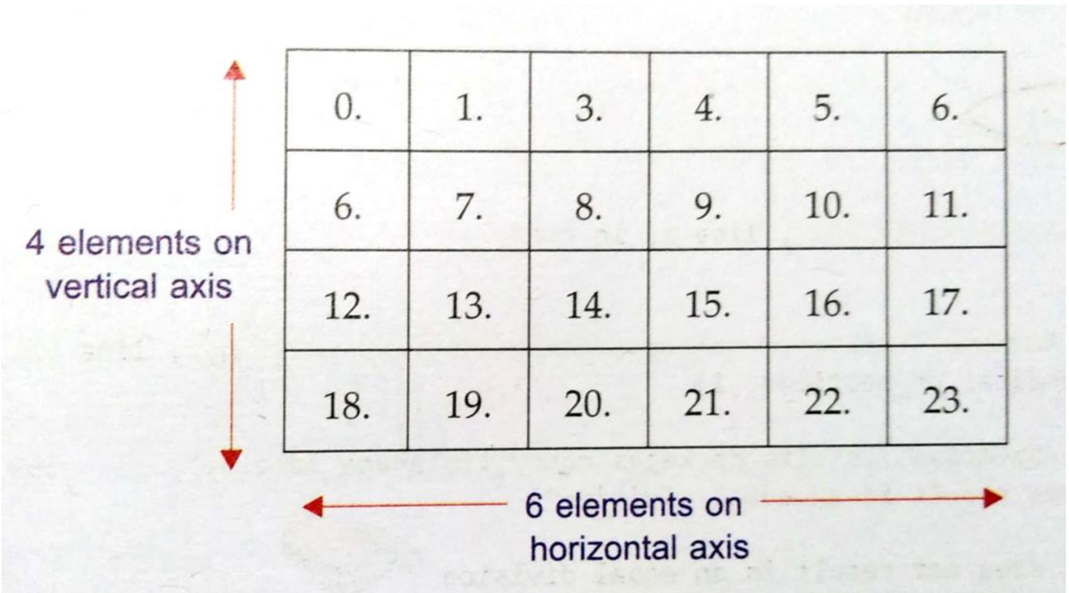
Figure 1.4 The working of hsplit( ) and vsplit( ).

   - The syntax of using hsplit( ) and vsplit( ) is similar, which is :

     numpy.hsplit(<array>, <n>)
     numpy.vsplit(<array>, <n>)

     where <array> is the NumPy array, and <n> is the no. of sections/subsets in which the array is to be divided.
     The <n> must be chosen so that it results in equal division of <array>, otherwise an error will be raised.

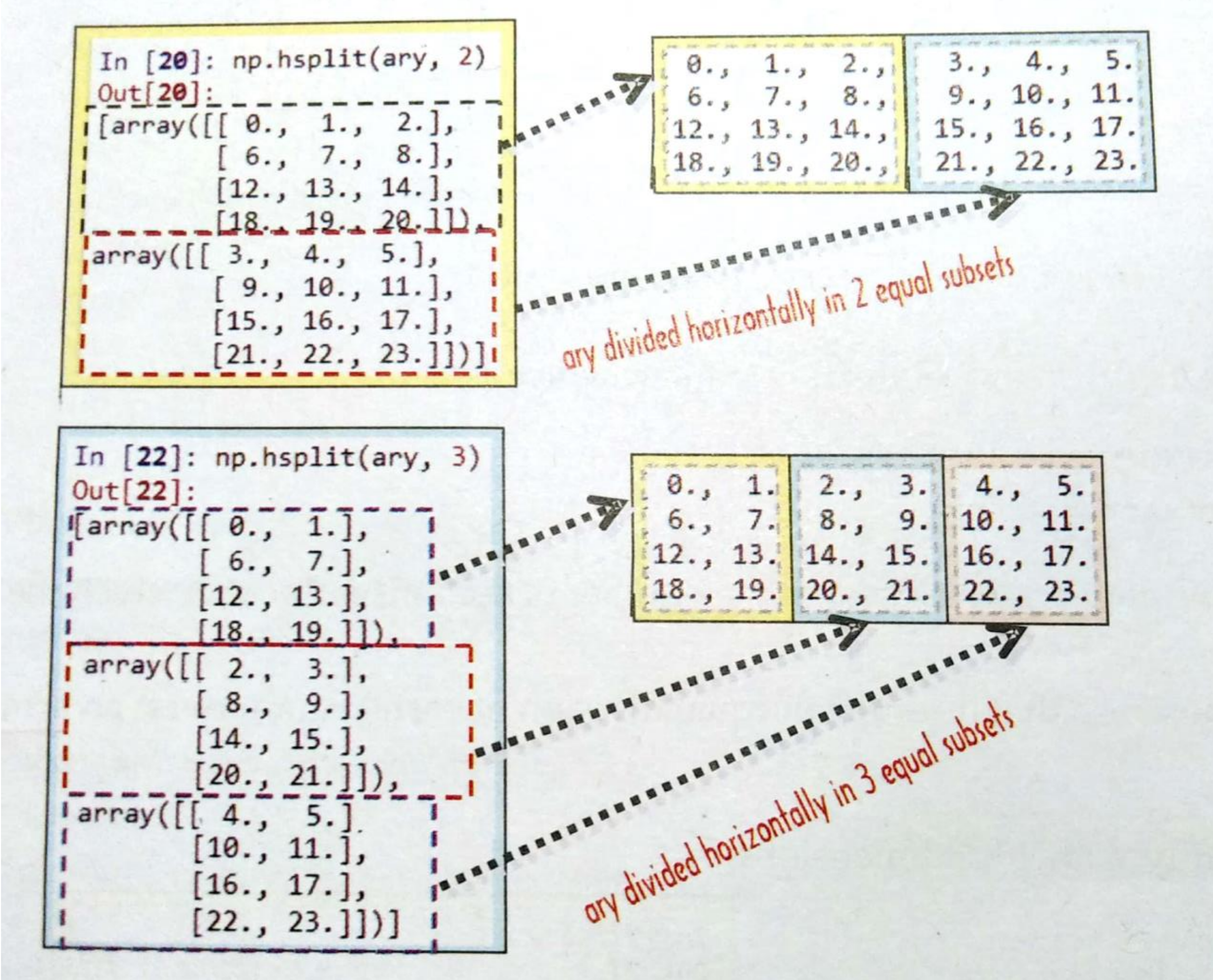
   - Consider following array with 4 x 6 dimensions, namely ary,



     So, horizontally we can split the arrays in 2 equal parts or 3 equal parts i.e, following two statements will yield equal subsets of array with horizontal split.

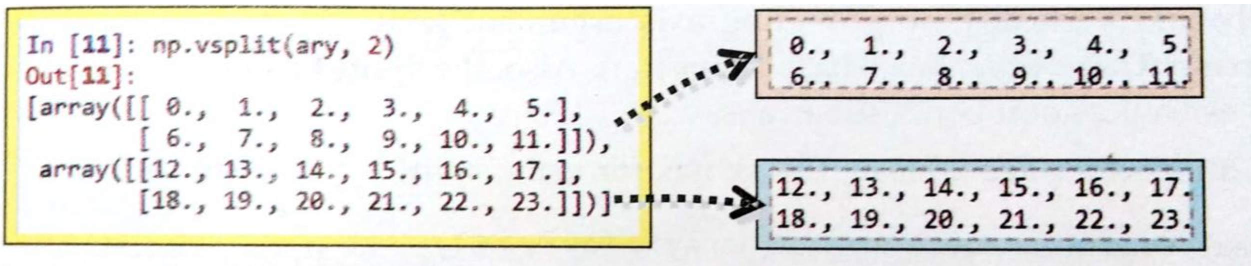
np. hsplit(ary , 2)
np. vsplit(ary, 3)

The O/P produced by above two statements will be :



But, **np.hsplit(ary , 4) will give error**, because the array **ary** cannot be equally divided in 4 or 5 subsets.

- Function vsplit( ) works identically as hsplit( ), but it divides the array subsets on vertical axis.



But, np.vsplit(ary , 3) will raise an error.

- You can assign these split subsets to individual array names and use them as per your convenience, e.g.

```
In[]: ar1, ar2 = np.vsplit(ary,2)
In[]: ar1
Out[14]: array([[ 0., 1., 2., 3., 4., 5.],
                [ 6., 7., 8., 9., 10., 11.]])
In[]: ar2
Out[15]: array([[12., 13., 14., 15., 16., 17.],
                [18., 19., 20., 21., 22., 23.]])
In[]: a1, a2, a3 = np.hsplit(ary,3)
In[]: a1
Out[17]: array([[ 0., 1.],
                [ 6., 7.],
                [12., 13.],
                [18., 19.]])
In[]: a2
Out[18]: array([[ 2., 3.],
                [ 8., 9.],
                [14., 15.],
                [20., 21.]])
In[]: a3
Out[19]: array([[ 4., 5.],
                [10., 11.],
                [16., 17.],
                [22., 23.]])
```

2. Using the split( ) function

- allows the splitting (horizontally or vertically) by providing axis argument.(axis = 0 for horizontal axis based division, axis =1 for vertical axis based division).

- split( ) ***allows you to divide array into equal as well as non-equal subarrays***.

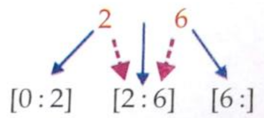
- The syntax for using split( ) is as given below:

numpy.split(<array>, <n>|<1D array> , [axis = 0])

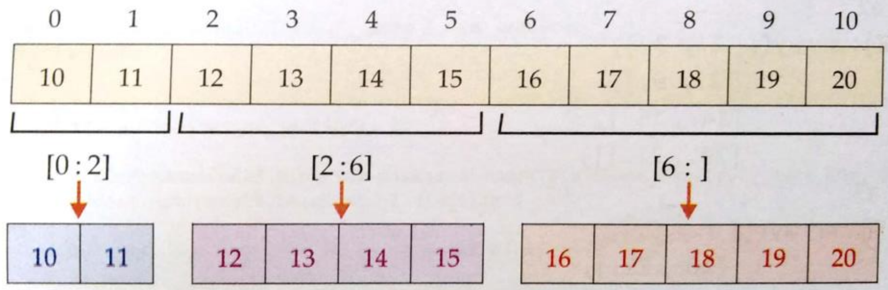
- <array> is the Numpy array to be split.
- With 2nd argument as <n>, for axis = 0, it behaves as vsplit( ) and for axis =1, it behaves as hsplit( ).
- If 2nd argument is given as 1D array then <array> is split in unqual subarrays as explained below.
- The axis argument is optional and if skipped, it takes the value 0 i.e., on horizontal axis. For axis = 1, the split happens on vertical axis.

e.g. (for 1D array)

```
ar1d = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
np.split(ar1d, [2, 6])
```



And then 1D array is sliced as per these slice ranges, *i.e.*,

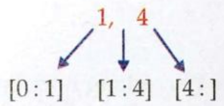


e.g.(for 2D array)- consider the 2D ndarray ary.

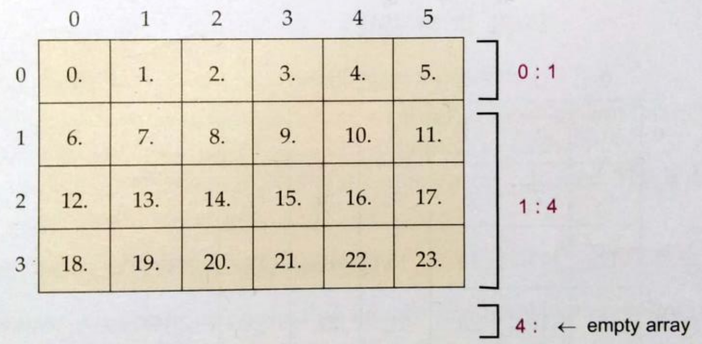
0.	1.	3.	4.	5.	6.
6.	7.	8.	9.	10.	11.
12.	13.	14.	15.	16.	17.
18.	19.	20.	21.	22.	23.

np. split(ary , [1, 4])

The given subset argument is [1, 4]



Since no axis is given, split will occur on vertical axis, *i.e.*, as



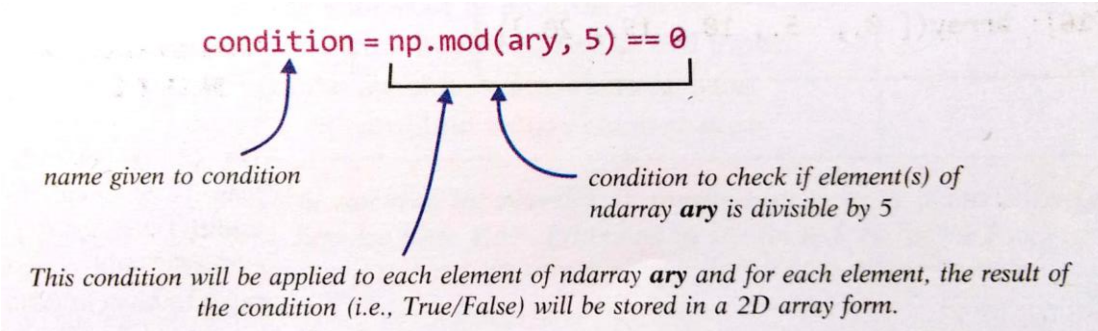
Extracting Condition based Non-contiguous Subsets

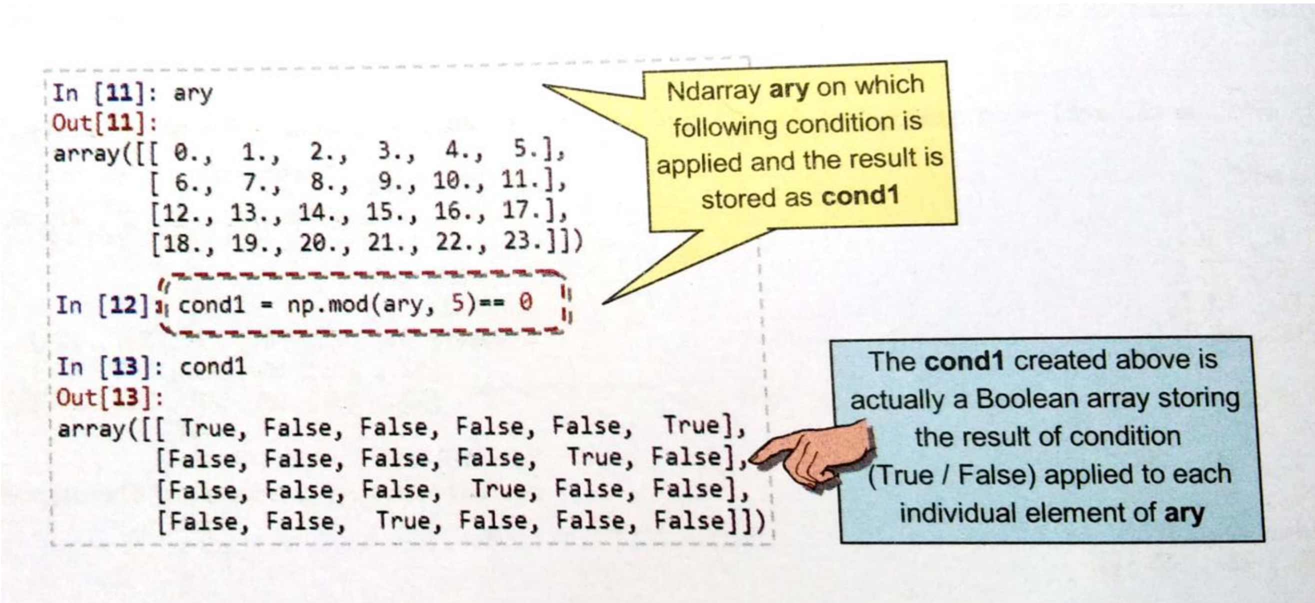
- You can extract non-contiguous subsets of a Numpy array **by applying condition on the NumPy array**. The specified condition will be applied to each element of the array and the elements meeting the criteria will be part of the subset array returned. This is done with the help of extract( ) as per following **syntax:**

numpy.extract(<condition>, <array>)

<condition> is a condition applied on an ndarray.
<array> is the ndarray on which the <condition> is applied.

Framing <condition> for extract ( )





Once you have saved the condition with a name, you can extract elements from the ndarray by using extract( ) as :

np.extract(cond1, ary)

And python will return a 1D array containing all the elements which satisfy the condition.

```
: np.extract(cond1, ary)
  array([ 0.,  5., 10., 15., 20.])
```

Arithmetic Operations on 2D Arrays

- Arithmetic operations (addition, subtraction, division, multiplication, remainder etc.)
- The arithmetic operations on 2D arrays can be performed in two ways:

(i) **Using Operators** – The syntax for using operators is :

<ndarray1> + <n> | <ndarray2>
<ndarray1> - <n> | <ndarray2>
<ndarray1> * <n> | <ndarray2>
<ndarray1> / <n> | <ndarray2>
<ndarray1> % <n> | <ndarray2>

The result of above operations is an ndarray.

(ii) **Using NumPy Functions** – add( ) , subtract( ) , multiply( ) , divide( ) , mod( ) or remainder( ).

The syntax of using the arithmetic functions is :

Numpy.add(<ndarray1>, <n>|<ndarray2>)
Numpy.subtract(<ndarray1>, <n>|<ndarray2>)
Numpy.multiply(<ndarray1>, <n>|<ndarray2>)
Numpy.divide(<ndarray1>, <n>|<ndarray2>)
Numpy.mod(<ndarray1>, <n>|<ndarray2>)
Numpy.remainder(<ndarray1>, <n>|<ndarray2>)

* <n> - scalar value

Arrays used in Examples

```
In [83]: ary
Out[83]:
array([[ 0.,  1.,  2.,  3.,  4.,  5.],
       [ 6.,  7.,  8.,  9., 10., 11.],
       [12., 13., 14., 15., 16., 17.],
       [18., 19., 20., 21., 22., 23.]])
```

```
In [84]: new
Out[84]:
array([[ 2.1,  3.1,  4.1,  5.1,  6.1,  7.1],
       [ 8.1,  9.1, 10.1, 11.1, 12.1, 13.1],
       [14.1, 15.1, 16.1, 17.1, 18.1, 19.1],
       [20.1, 21.1, 22.1, 23.1, 24.1, 25.1]])
```

```
In [107]: twos
Out[107]:
array([[2, 2, 2, 2, 2, 2],
       [2, 2, 2, 2, 2, 2],
       [2, 2, 2, 2, 2, 2],
       [2, 2, 2, 2, 2, 2]])
```

Arithmetic Operation	With Scalar Value	With Another ndarray

Add

With Scalar Value

```
In [73]: ary + .3
Out[73]:
array([[ 0.3,  1.3,  2.3,  3.3,  4.3,  5.3],
       [ 6.3,  7.3,  8.3,  9.3, 10.3, 11.3],
       [12.3, 13.3, 14.3, 15.3, 16.3, 17.3],
       [18.3, 19.3, 20.3, 21.3, 22.3, 23.3]])

In [74]: np.add(ary, .3)
Out[74]:
array([[ 0.3,  1.3,  2.3,  3.3,  4.3,  5.3],
       [ 6.3,  7.3,  8.3,  9.3, 10.3, 11.3],
       [12.3, 13.3, 14.3, 15.3, 16.3, 17.3],
       [18.3, 19.3, 20.3, 21.3, 22.3, 23.3]])
```

With Another ndarray

```
In [69]: ary + new
Out[69]:
array([[ 2.1,  4.1,  6.1,  8.1, 10.1, 12.1],
       [14.1, 16.1, 18.1, 20.1, 22.1, 24.1],
       [26.1, 28.1, 30.1, 32.1, 34.1, 36.1],
       [38.1, 40.1, 42.1, 44.1, 46.1, 48.1]])

In [70]: np.add(ary, new)
Out[70]:
array([[ 2.1,  4.1,  6.1,  8.1, 10.1, 12.1],
       [14.1, 16.1, 18.1, 20.1, 22.1, 24.1],
       [26.1, 28.1, 30.1, 32.1, 34.1, 36.1],
       [38.1, 40.1, 42.1, 44.1, 46.1, 48.1]])
```

Subtract

With Scalar Value

```
In [64]: ary  - 6
Out[64]:
array([[-6., -5., -4., -3., -2., -1.],
       [ 0.,  1.,  2.,  3.,  4.,  5.],
       [ 6.,  7.,  8.,  9., 10., 11.],
       [12., 13., 14., 15., 16., 17.]])

In [65]: np.subtract(ary, 6)
Out[65]:
array([[-6., -5., -4., -3., -2., -1.],
       [ 0.,  1.,  2.,  3.,  4.,  5.],
       [ 6.,  7.,  8.,  9., 10., 11.],
       [12., 13., 14., 15., 16., 17.]])
```

With Another ndarray

```
In [66]: new - ary
Out[66]:
array([[2.1, 2.1, 2.1, 2.1, 2.1, 2.1],
       [2.1, 2.1, 2.1, 2.1, 2.1, 2.1],
       [2.1, 2.1, 2.1, 2.1, 2.1, 2.1],
       [2.1, 2.1, 2.1, 2.1, 2.1, 2.1]])

In [67]: np.subtract( new, ary)
Out[67]:
array([[2.1, 2.1, 2.1, 2.1, 2.1, 2.1],
       [2.1, 2.1, 2.1, 2.1, 2.1, 2.1],
       [2.1, 2.1, 2.1, 2.1, 2.1, 2.1],
       [2.1, 2.1, 2.1, 2.1, 2.1, 2.1]])
```

Multiply

With Scalar Value

```
In [75]: ary * .3
Out[75]:
array([[0. , 0.3, 0.6, 0.9, 1.2, 1.5],
       [1.8, 2.1, 2.4, 2.7, 3. , 3.3],
       [3.6, 3.9, 4.2, 4.5, 4.8, 5.1],
       [5.4, 5.7, 6. , 6.3, 6.6, 6.9]])

In [76]: np.multiply(ary, .3)
Out[76]:
array([[0. , 0.3, 0.6, 0.9, 1.2, 1.5],
       [1.8, 2.1, 2.4, 2.7, 3. , 3.3],
       [3.6, 3.9, 4.2, 4.5, 4.8, 5.1],
       [5.4, 5.7, 6. , 6.3, 6.6, 6.9]])
```

With Another ndarray

```
In [109]: ary * twos
Out[109]:
array([[ 0.,  2.,  4.,  6.,  8., 10.],
       [12., 14., 16., 18., 20., 22.],
       [24., 26., 28., 30., 32., 34.],
       [36., 38., 40., 42., 44., 46.]])

In [110]: np.multiply(ary, twos)
Out[110]:
array([[ 0.,  2.,  4.,  6.,  8., 10.],
       [12., 14., 16., 18., 20., 22.],
       [24., 26., 28., 30., 32., 34.],
       [36., 38., 40., 42., 44., 46.]])
```

Arithmetic Operation	With Scalar Value	With Another ndarray
Divide	In [97]: ary/5 Out[97]: array([[0. , 0.2, 0.4, 0.6, 0.8, 1.], [1.2, 1.4, 1.6, 1.8, 2. , 2.2], [2.4, 2.6, 2.8, 3. , 3.2, 3.4], [3.6, 3.8, 4. , 4.2, 4.4, 4.6]]) In [98]: np.divide(ary, 5) Out[98]: array([[0. , 0.2, 0.4, 0.6, 0.8, 1.], [1.2, 1.4, 1.6, 1.8, 2. , 2.2], [2.4, 2.6, 2.8, 3. , 3.2, 3.4], [3.6, 3.8, 4. , 4.2, 4.4, 4.6]])	In [111]: ary / twos Out[111]: array([[0. , 0.5, 1. , 1.5, 2. , 2.5], [3. , 3.5, 4. , 4.5, 5. , 5.5], [6. , 6.5, 7. , 7.5, 8. , 8.5], [9. , 9.5, 10. , 10.5, 11. , 11.5]]) In [112]: np.divide(ary, twos) Out[112]: array([[0. , 0.5, 1. , 1.5, 2. , 2.5], [3. , 3.5, 4. , 4.5, 5. , 5.5], [6. , 6.5, 7. , 7.5, 8. , 8.5], [9. , 9.5, 10. , 10.5, 11. , 11.5]])
Remainder	In [59]: ary % 4 Out[59]: array([[0., 1., 2., 3., 0., 1.], [2., 3., 0., 1., 2., 3.], [0., 1., 2., 3., 0., 1.], [2., 3., 0., 1., 2., 3.]]) In [60]: np.remainder(ary, 4) Out[60]: array([[0., 1., 2., 3., 0., 1.], [2., 3., 0., 1., 2., 3.], [0., 1., 2., 3., 0., 1.], [2., 3., 0., 1., 2., 3.]])	In [62]: ary % new Out[62]: array([[0., 1., 2., 3., 4., 5.], [6., 7., 8., 9., 10., 11.], [12., 13., 14., 15., 16., 17.], [18., 19., 20., 21., 22., 23.]]) In [63]: np.mod(ary, new) Out[63]: array([[0., 1., 2., 3., 4., 5.], [6., 7., 8., 9., 10., 11.], [12., 13., 14., 15., 16., 17.], [18., 19., 20., 21., 22., 23.]])

<u>Applications of Numpy Arrays</u>

1. Covariance   2. Correlation   3. Linear regression

<u>Covariance</u>

- It is a tool in statistics in which we can **compare two different datasets**.
- The intuitive idea behind covariance is that it tells us how similar varying two datasets are. A **high positive covariance** between 2 datasets means that they are **strongly similar**. Similarly, a **high negative covariance** between 2 datasets means that they are **very dissimilar**.

<u>Calculating covariance using cov( )</u>

- Numpy provides a function namely cov( ) to calculate covariance, which can be used as:

numpy.cov(<arr1>,<arr2>)

where <arr1> and <arr2> are two sets of observations.
The result will be n x n matrix where n is the number of variables for which covariance is calculated.

e.g.

```
import numpy as np
a = np.array([1 , 2 , 3 , 4 , 5])
b = np.array([3 , 4 , 0 , -1 , -3] )
cov_mat = np.cov(a , b)
print(cov_mat)
```

Output:

Covariance (Negative values indicate they are not very similar)

```
[[ 2.5    ,   -4.25] ,
 [ -4.25   ,    8.3]]
```

The four values of **cov_mat** generated are like this:

```
cov_mat[0][0] = var(a)
cov_mat[0][1] = covariance(a, b)
cov_mat[1][0] = covariance(b,a) = covariance(a,b)
cov_mat[1][1] = var(b)
```

Correlation

- When you need to know only **whether two data sets are similar and different** and not how similar or different, you use correlation.
- It is basically normalised covariance.
- It **give two values**: 1 if the data sets have positive covariance and -1 if the datasets have negative covariance.
- To calculate correlation, you can use coeff( ) of numpy( ) as :

  numpy.corrcoef(<arr1> , <arr2>)

e.g.

```
import numpy as np
a = np.array([1 , 2 , 3 , 4 , 5])
b = np.array([3 , 4 , 0 , -1 , -3] )
correlation_mat = np.corrcoeff(a , b)
print(correlation_mat)
```

Output:

```
[ [1            ,   -0.93299621] ,
  [-0.93299621 ,    1           ] ]
```

Linear regression

- Suppose,we have a set of ordered pairs {(x_1, y_1) , (x_2 , y_2), ……, (x_n , y_n)} where all y_i are dependent on x_i. Our objective is to find their relation, how they are dependent on x. This is called **regression**. If relation between x and y is linear , that is y = ax + b , then it is called **linear regression**.
- So, linear regression is a method used to find a relationship between a dependent variable and a set of independent variables.
- For finding out linear regression, Numpy function polyfit( ) is used. The syntax of polyfit( ) is :

  numpy. polyfit(x , y , deg)

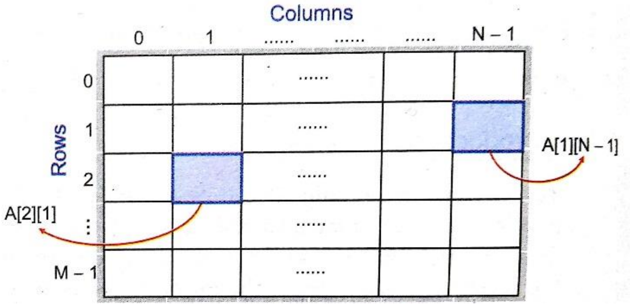
where

x is an array containing x-coordinates of the M sample points.
y is an array having same shape as x and contains y-coordinates of the sample points.
degree – specifies the degree of the polynomial.

****

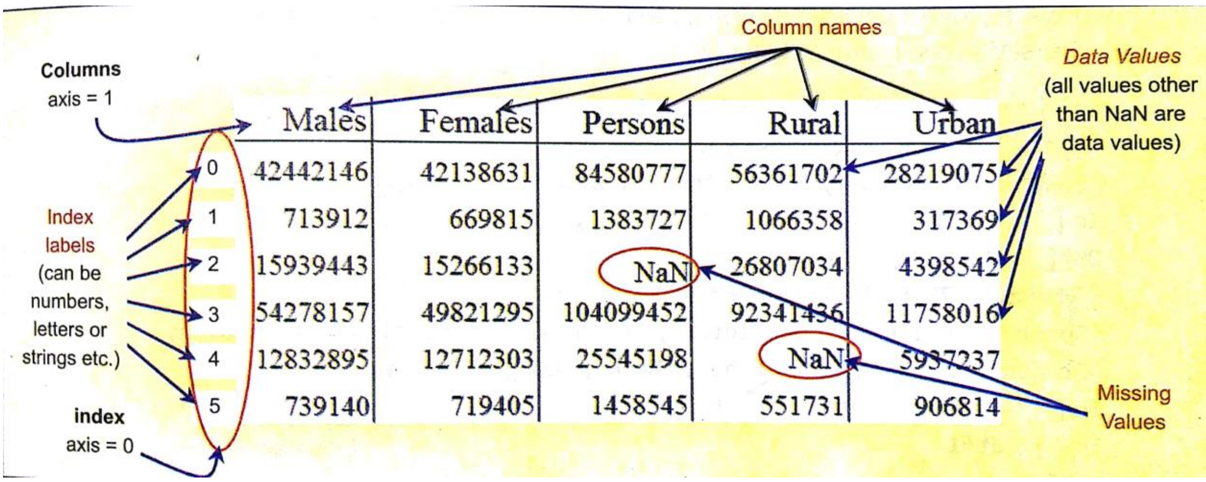
DataFrame Data Structure

✓ A DataFrame is another pandas data structure, which stores data in ***two-dimensional array***. It is actually a two dimensional labelled 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.



✓ A two dimensional array is an array in which each element is itself an array. For instance, an array A [m][n] is an *m* by *n* table with *m* rows and *n* columns containing *m x n* elements.

Characteristics



1. It has two indexes or we can say that two axes – ***a row index*** (axis=0) and ***column index*** (axis=1).
2. Each value is identifiable with the combination of *row index* and *column index.* The ***row index*** is known as *index* in general and the ***column index*** is called the *column-name*.
3. The indexes can of numbers or letters or strings.
4. There is no condition of having all data of same type across columns; its columns can have data of different types.
5. You can easily change its values, i.e., it is ***value-mutable.***
6. You can add or delete rows/columns in a DataFrame. In other words, it is ***size-mutable.***

Creating and Displaying a DataFrame

✓ A DataFrame object can be created by passing data in two-dimensional format. Like series data structure, before start working with DataFrame the following two libraries needs to be imported:

```
import pandas as pd
import numpy as np
```

✓ To create a DataFrame object, you can use syntax as :

```
<datFrameObject> = panda.DataFrame(<a 2D datastructure>,\
[columns = <column sequence> ], [index = <index sequence>])
```

*Both **D** and **F** are capital letters*

Command continuation mark

1. **Creating a DataFrame Object from a 2-D Dictionary**

✓ A two dimensional dictionary is a dictionary having items as (key: value) where value part is a data structure of any type : another dictionary, an ndarray, a Series object, a list etc. ***But here the value parts of all keys should have similar structure and equal lengths.***

(a) Creating a dataframe from a 2D dictionary having values as lists/ndarrays

e.g.

```
import numpy as np
import pandas as pd

dict1={'Students': ['Ruchika' , 'Neha', 'Mark' , 'Gurpreet' , 'Jamaal'] ,
        'Marks': [79.5 , 83.75 , 74 , 88.5 , 89] ,
        'Sport' : ['Cricket', 'Badminton', 'Football' , 'Athletics' , 'Kabaddi'] ,
      }

dtf1 = pd.DataFrame(dict1)
print(dtf1)
```

Output:

```
   Students  Marks      Sport
0   Ruchika  79.50    Cricket
1      Neha  83.75  Badminton
2      Mark  74.00   Football
3  Gurpreet  88.50  Athletics
4    Jamaal  89.00    Kabaddi
```

** As you can see that the DataFrame object created has its index assigned automatically (0 onwards) just as it happens with Series objects, and the columns are places in sorted order. **keys of the dictionary have become columns.**

** You can specify your own indexes too by specifying a sequence by the name **index** in the DataFrame( ) function, **e.g.**

```
dtf1 = pd.DataFrame(dict1, index=['I', 'II', 'III' , 'IV' , 'V'])
print(dtf1)
```

```
     Students  Marks      Sport
I     Ruchika  79.50    Cricket
II       Neha  83.75  Badminton
III      Mark  74.00   Football
IV   Gurpreet  88.50  Athletics
V      Jamaal  89.00    Kabaddi
```

(b) Creating a DataFrame from a 2D dictionary having values as dictionary objects:

e.g.

```
import numpy as np
import pandas as pd

yr2015 = { 'Qtr1' : 34500 , 'Qtr2' : 56000 , 'Qtr3' : 47000 , 'Qtr4': 49000}
yr2016 = {'Qtr1' : 44900, 'Qtr2' : 46100 , 'Qtr3' : 57000 , 'Qtr4': 59000}
yr2017 = { 'Qtr1' : 54500 , 'Qtr2' : 51000 , 'Qtr3' : 57000 , 'Qtr4': 58500}
diSales = { 2015 : yr2015 , 2016 : yr2016 , 2017 : yr2017}
```

```
df1 = pd.DataFrame(diSales)
print(df1)
```

Output:

```
      2015   2016   2017
Qtr1  34500  44900  54500
Qtr2  56000  46100  51000
Qtr3  47000  57000  57000
Qtr4  49000  59000  58500
```

- In this case, Python interprets the ***outer dict keys as the columns and the inner keys as the row indices.***
- As the keys of all inner dictionaries(yr2015 , yr2016 , yr2017) are exactly the same in number and names, the dataframe object df2 also has the same number of indexes. Since the inner keys have values in all the inner dictionaries, there is no missing value in the dataframe object.
- Now had there been a situation where inner dictionaries had non-matching keys, then in that case Python would have done following things:
  (i) There would have been **total number of indexes equal to sum of unique inner keys** in all the inner dictionaries.
  (ii) For a key that has no matching keys in other inner dictionaries, value **NaN** would be used to depict the missing values.

Example:

```
import numpy as np
import pandas as pd

yr2015 = { 'Qtr1' : 34500 , 'Qtr2' : 56000 , 'Qtr3' : 47000 , 'Qtr4': 49000}
yr2016 = {'Q1' : 44900, 'Q2' : 46100 , 'Q3' : 57000 , 'Q4': 59000}
yr2017 = { 'A' : 54500 , 'B' : 51000 , 'C' : 57000 }
diSales = { 2015 : yr2015 , 2016 : yr2016 , 2017 : yr2017}
df1 = pd.DataFrame(diSales)
print(df1)
```

Output:

```
         2015     2016     2017
A         NaN      NaN  54500.0
B         NaN      NaN  51000.0
C         NaN      NaN  57000.0
Q1        NaN  44900.0      NaN
Q2        NaN  46100.0      NaN
Q3        NaN  57000.0      NaN
Q4        NaN  59000.0      NaN
Qtr1  34500.0      NaN      NaN
Qtr2  56000.0      NaN      NaN
Qtr3  47000.0      NaN      NaN
Qtr4  49000.0      NaN      NaN
```

Keys **A**, **B**, **C** only have values for dictionary **yr2017** (**2017:yr2017**) hence **NaN** filled for other two dictionaries.

Keys **Q1**, **Q2**, **Q3**, **Q4** only have values for dictionary **yr2016** (**2016:yr2016**) hence **NaN** filled for other two dictionaries.

Keys **Qtr1**, **Qtr2**, **Qtr3**, **Qtr4** only have values for dictionary **yr2015** (**2015:yr2015**) hence **NaN** filled for other two dictionaries.

Total number of indexes are 11 (equal to sum of unique keys in inner dictionaries)

Example:

```
import numpy as np
import pandas as pd

yr2015 = { 'Qtr1' : 34500 , 'Qtr2' : 56000 , 'Qtr3' : 47000 , 'Qtr4': 49000}
yr2016 = {'Qtr1' : 44900, 'Qtr2' : 46100 , 'Q3' : 57000 , 'Q4': 59000}
```

```
yr2017 = { 'A' : 54500 , 'B' : 51000 , 'Qtr4' : 57000 }
diSales = { 2015 : yr2015 , 2016 : yr2016 , 2017 : yr2017}
df1 = pd.DataFrame(diSales)
print(df1)
```

Output:

```
          2015      2016      2017
A          NaN       NaN   54500.0
B          NaN       NaN   51000.0
Q3         NaN   57000.0       NaN
Q4         NaN   59000.0       NaN
Qtr1   34500.0   44900.0       NaN
Qtr2   56000.0   46100.0       NaN
Qtr3   47000.0       NaN       NaN
Qtr4   49000.0       NaN   57000.0
```

Total number of indexes are equal to total unique inner keys.

Like earlier example **NaN** fills the missing data

2. Creating a DataFrame Object from a 2-D ndarray

- ✓ You can also pass a two-dimensional NumPy array to DataFrame( ) to create a dataframe object.

Example:

```
import numpy as np
import pandas as pd

narr1=np.array([[40,43,53],[64,55,46]],np.int32)
dtf1 = pd.DataFrame(narr1)
print(dtf1)
```

Output:

```
    0   1   2
0  40  43  53
1  64  55  46
```

** As no **keys** are there, hence default names are given to indexes and columns, i.e. 0 onwards.

- ✓ You can however, specify your own column names and/or index names by giving a columns sequence and/or index sequence.

Example:

```
import numpy as np
import pandas as pd

narr1=np.array([[40,43,53],[64,55,46]],np.int32)
dtf1 = pd.DataFrame(narr1,columns=['First','Second','Three'], index=['A','B'])
print(dtf1)
```

Output:

```
   First  Second  Three
A     40      43     53
B     64      55     46
```

- ✓ If rows of ndarrays differ in length, i.e., if number of elements in each row differ, the Python will create just single column in the dataframe object and the type of column will be considered as **object.**

Example:

```
import numpy as np
import pandas as pd
```

```
narr1=np.array([[40,43],[64,55,46], [46.2,56.2]])
dtf1 = pd.DataFrame(narr1)
print(dtf1)
```

Output:

```
           0
0     [40, 43]
1  [64, 55, 46]
2  [46.2, 56.2]
```

Single column created this time because the lengths of rows of ndarray did not match.

3. **Creating a DataFrame object from a 2D dictionary with values as Series Object**

Example:

```
import numpy as np
import pandas as pd

population=pd.Series([10927986,12691836,4631392,4328063],\
                index=['Delhi', 'Mumbai','Kolkata','Chennai'])
AvgIncome = pd.Series([7216781092,8508781269,4226785362,5261784321],\
                index=['Delhi', 'Mumbai','Kolkata','Chennai'])
dict2 = {0 : population , 1 : AvgIncome}
dtf2 = pd.DataFrame(dict2)
print(dtf2)
```

Output:

```
                0           1
Delhi    10927986  7216781092
Mumbai   12691836  8508781269
Kolkata   4631392  4226785362
Chennai   4328063  5261784321
```

** Dataframe object created (dtf2) has **columns** assigned from the **keys** of the dictionary object and its **index** assigned from the **indexes of the series objects** which are the values of the dictionary object.

4. **Creating a DataFrame Object from another DataFrame Object**

Example:

```
import numpy as np
import pandas as pd

population=pd.Series([10927986,12691836,4631392,4328063],\
                index=['Delhi', 'Mumbai','Kolkata','Chennai'])
AvgIncome = pd.Series([7216781092,8508781269,4226785362,5261784321],\
                index=['Delhi', 'Mumbai','Kolkata','Chennai'])
dict2 = {0 : population , 1 : AvgIncome}
dtf2 = pd.DataFrame(dict2)
print(dtf2)
dtf3= pd.DataFrame(dtf2)
print(dtf3)
```

Output:

```
         0           1
Delhi    10927986  7216781092
Mumbai   12691836  8508781269
Kolkata   4631392  4226785362
Chennai   4328063  5261784321
         0           1
Delhi    10927986  7216781092
Mumbai   12691836  8508781269
Kolkata   4631392  4226785362
Chennai   4328063  5261784321
```

DataFrame Attributes

When you create a DataFrame object, all information related to it (such as its size, its datatype etc.) is available through attributes. You can use these attributes in the following format to get information about the dataframe object.

<DataFrame object>.<attribute name>

Attribute	Description
index	The index (row labels) of the DataFrame.
columns	The column labels of the DataFrame.
axes	Return a list representing both the axes (axis 0 *i.e.*, index and axis 1, *i.e.*, columns) of the DataFrame.
dtypes	Return the dtypes of data in the DataFrame.
size	Return an int representing the number of elements in this object.
shape	Return a tuple representing the dimensionality of the DataFrame.
values	Return a Numpy representation of the DataFrame.
empty	Indicator whether DataFrame is empty.
ndim	Return an int representing the number of axes/array dimensions.
T	Transpose index and columns.

(a) Retrieving index(axis 0), Columns(axis 1), axes' details and data type of columns

Example:

```
import numpy as np
import pandas as pd

population=pd.Series([10927986,12691836,4631392,4328063],\
                     index=['Delhi', 'Mumbai','Kolkata','Chennai'])
AvgIncome = pd.Series([7216781092,8508781269,4226785362,5261784321],\
                     index=['Delhi', 'Mumbai','Kolkata','Chennai'])
dict2 = {0 : population , 1 : AvgIncome}
dtf2 = pd.DataFrame(dict2)
print(dtf2)
print(dtf2.index)
print(dtf2.columns)
print(dtf2.axes)
print(dtf2.dtypes)
```

Output:

```
          0           1
Delhi    10927986  7216781092
Mumbai   12691836  8508781269
Kolkata  4631392  4226785362
Chennai  4328063  5261784321
Index(['Delhi', 'Mumbai', 'Kolkata', 'Chennai'], dtype='object')
Int64Index([0, 1], dtype='int64')
[Index(['Delhi', 'Mumbai', 'Kolkata', 'Chennai'], dtype='object'), Int64Index([0, 1], dtype='int64')]
0    int64
1    int64
dtype: object
```

(b) Retrieving size(number of elements), shape, number of dimensions

Use attributes size, shape and ndim to get number if elements, dimensionality and number of axes respectively of a dataframe object, e.g.

Example:

```
import numpy as np
import pandas as pd

population=pd.Series([10927986,12691836,4631392,4328063],\
                     index=['Delhi','Mumbai','Kolkata','Chennai'])
AvgIncome = pd.Series([7216781092,8508781269,4226785362,5261784321],\
                     index=['Delhi','Mumbai','Kolkata','Chennai'])
dict2 = {0 : population , 1 : AvgIncome}
dtf2 = pd.DataFrame(dict2)
print(dtf2)
print(dtf2.size)
print(dtf2.shape)
print(dtf2.ndim)
```

Output:

```
          0           1
Delhi    10927986  7216781092
Mumbai   12691836  8508781269
Kolkata  4631392  4226785362
Chennai  4328063  5261784321
8
(4, 2)
2
```

(c) Checking for emptiness of dataframe or presence of NaNs in dataframe

Use attribute empty to check for emptiness of a dataframe

e.g.

```
import numpy as np
import pandas as pd

population=pd.Series([10927986,12691836,4631392,4328063],\
                index=['Delhi','Mumbai','Kolkata','Chennai'])
AvgIncome = pd.Series([7216781092,8508781269,4226785362,5261784321],\
                index=['Delhi','Mumbai','Kolkata','Chennai'])
dict2 = {0 : population , 1 : AvgIncome}
dtf2 = pd.DataFrame(dict2)
```

```
print(dtf2)
print(dtf2.empty)
```

Output:

```
               0           1
Delhi    10927986  7216781092
Mumbai   12691836  8508781269
Kolkata   4631392  4226785362
Chennai   4328063  5261784321
False
```

(d) Getting number of rows in a dataframe

The **len(<DF Object>)** will return the number of rows in a dataframe.

(e) Getting count of non-NA values in dataframe

You can use count( ) with dataframe to get the count of Non-NaN values, but count( ) with dataframe is little elaborate:

I. If you do not pass any argument or pass 0 (default is 0 only), then it returns count of Non-NA values for each column.
II. If you pass argument as 1, then it returns count of non-NaN values for each row.

Example:

```
import numpy as np
import pandas as pd

population=pd.Series([10927986,12691836,4631392,4328063],\
                 index=['Delhi', 'Mumbai','Kolkata','Chennai'])
AvgIncome = pd.Series([7216781092,8508781269,4226785362,5261784321],\
                  index=['Delhi', 'Mumbai','Kolkata','Chennai'])
dict2 = {0 : population , 1 : AvgIncome}
dtf2 = pd.DataFrame(dict2)
print(dtf2)
print(len(dtf2))
print(dtf2.count(0))
print(dtf2.count(1))
```

Output:

```
               0           1
Delhi    10927986  7216781092
Mumbai   12691836  8508781269
Kolkata   4631392  4226785362
Chennai   4328063  5261784321
4
0    4
1    4
dtype: int64
Delhi      2
Mumbai     2
Kolkata    2
Chennai    2
dtype: int64
```

(f) Transposing a Dataframe

You can transpose a dataframe by swapping its indexes and columns by using attribute T ,

e.g.

```
import numpy as np
import pandas as pd

population=pd.Series([10927986,12691836,4631392,4328063],\
                  index=['Delhi', 'Mumbai','Kolkata','Chennai'])
AvgIncome = pd.Series([7216781092,8508781269,4226785362,5261784321],\
                   index=['Delhi', 'Mumbai','Kolkata','Chennai'])
dict2 = {0 : population , 1 : AvgIncome}
dtf2 = pd.DataFrame(dict2)
print(dtf2)
print(dtf2.T)
```

Output:

```
                0           1
Delhi    10927986  7216781092
Mumbai   12691836  8508781269
Kolkata   4631392  4226785362
Chennai   4328063  5261784321
        Delhi      Mumbai     Kolkata     Chennai
0    10927986    12691836     4631392     4328063
1  7216781092  8508781269  4226785362  5261784321
```

SELECTING OR ACCESSING DATA

1. Selecting/Accessing a Column

Single column at a time

<DataFrame object> [<Column name>]

Or

<DataFrame object>.<Column name>

Multiple columns at a time

<DataFrame object>[[columnname , columnname, ……….]]

Example:

```
import numpy as np
import pandas as pd

population=pd.Series([10927986,12691836,4631392,4328063],\
                        index=['Delhi', 'Mumbai','Kolkata','Chennai'])
AvgIncome = pd.Series([7216781092,8508781269,4226785362,5261784321],\
                         index=['Delhi', 'Mumbai','Kolkata','Chennai'])
dict2 = {'Population' : population , 'Avg. Income'  : AvgIncome}
dtf2 = pd.DataFrame(dict2)
print(dtf2)
print("========")
print(dtf2.Population)
print("========")
print(dtf2[['Population','Avg. Income']])
```

Output:

```
        Population  Avg. Income
Delhi     10927986   7216781092
Mumbai    12691836   8508781269
Kolkata    4631392   4226785362
Chennai    4328063   5261784321
========
Delhi      10927986
Mumbai     12691836
Kolkata     4631392
Chennai     4328063
Name: Population, dtype: int64
========
        Population  Avg. Income
Delhi     10927986   7216781092
Mumbai    12691836   8508781269
Kolkata    4631392   4226785362
Chennai    4328063   5261784321
```

2. **Selecting/Accessing a SubSet from a Dataframe using Row/Column Name**

For this purpose, you can use following syntax to select/access a subset from a dataframe object:

<DataFrameObject>.loc [<startrow>: <endrow>, <startcolumn> :<endcolumn>]

I. To access a row, just give the row name/label as this : **<DF Object>.loc[<row label> , :]**
Make sure not to miss the COLON AFTER COMMA.

II. To access multiple rows, use : **<DF object>.loc[<start row> :<endrow>, :]**
Make sure not to miss the COLON AFTER COMMA.

III. To access selective columns, use : **<DF object>.loc[: , <start column> , <end column>]**

IV. To access a range of columns from a range of rows, use:
<DF object>.loc [<startrow>: <endrow>, <startcolumn> :<endcolumn>]

Example:

```
import numpy as np
import pandas as pd

population=pd.Series([10927986,12691836,4631392,4328063],\
                    index=['Delhi', 'Mumbai','Kolkata','Chennai'])
AvgIncome = pd.Series([7216781092,8508781269,4226785362,5261784321],\
                    index=['Delhi', 'Mumbai','Kolkata','Chennai'])
dict2 = {'Population' : population , 'Avg. Income'  : AvgIncome}
dtf2 = pd.DataFrame(dict2)
print(dtf2)
print("==Accessing Single row==")
print(dtf2.loc['Delhi' , :])
print(dtf2.loc['Kolkata' ,:])
print("==Accessing Multiple rows==")
print(dtf2.loc['Mumbai' : 'Chennai' , :])
print("==Accessing Columns==")
print(dtf2.loc[ : , 'Population'])
print("==Accessing range of columns and rows==")
print(dtf2.loc['Delhi' : 'Mumbai' , 'Population' : 'Avg. Income'])
```

Output:

```
        Population  Avg. Income
Delhi     10927986   7216781092
Mumbai    12691836   8508781269
Kolkata   4631392   4226785362
Chennai   4328063   5261784321
==Accessing Single row==
Population     10927986
Avg. Income    7216781092
Name: Delhi, dtype: int64
Population      4631392
Avg. Income    4226785362
Name: Kolkata, dtype: int64
==Accessing Multiple rows==
        Population  Avg. Income
Mumbai    12691836   8508781269
Kolkata   4631392   4226785362
Chennai   4328063   5261784321
==Accessing Columns==
Delhi     10927986
Mumbai    12691836
Kolkata   4631392
Chennai   4328063
Name: Population, dtype: int64
==Accessing range of columns and rows==
       Population  Avg. Income
Delhi    10927986   7216781092
Mumbai   12691836   8508781269
```

3. **Obtaining a Subset/Slice from a Dataframe using Row/Column Numeric Index/Position**

Sometimes your dataframe object does not contain row or column labels or even you may not remember them. In such cases, you can extract subset from dataframe using the row and column ***numeric index/position,*** but this time you will use **iloc** instead of loc. **iloc** means ***integer location.***

<DF object>.iloc[<startrow index>: <endrow index>, <startcolumn index> :<endcolumn index>]

***** endindex is excluded here.***

Example:

```
import numpy as np
import pandas as pd

population=pd.Series([10927986,12691836,4631392,4328063],\
                     index=['Delhi','Mumbai','Kolkata','Chennai'])
AvgIncome = pd.Series([7216781092,8508781269,4226785362,5261784321],\
                     index=['Delhi','Mumbai','Kolkata','Chennai'])
dict2 = {'Population' : population , 'Avg. Income'  : AvgIncome}
dtf2 = pd.DataFrame(dict2)
print(dtf2)
print(dtf2.iloc[0:2,0:1])
```

Output:

```
        Population  Avg. Income
Delhi     10927986   7216781092
Mumbai    12691836   8508781269
Kolkata   4631392   4226785362
Chennai   4328063   5261784321
       Population
Delhi    10927986
Mumbai   12691836
```

4. **Selecting/Accessing Individual Value**

(i) Either give name of row or numeric index in square brackets with, i.e., as this :

<DF object>.<column>[<row name or row numeric index>]

(ii) You can use **at** or **iat** attribute with DF object as shown below:

<DF object>.**at** [<row name>, <column name>]

Or

<DF object>. **iat** [<numeric row index>, <numeric column index>]

Example:

```
import numpy as np
import pandas as pd

population=pd.Series([10927986,12691836,4631392,4328063],\
                    index=['Delhi', 'Mumbai','Kolkata','Chennai'])
AvgIncome = pd.Series([7216781092,8508781269,4226785362,5261784321],\
                     index=['Delhi', 'Mumbai','Kolkata','Chennai'])
dict2 = {'Population' : population , 'Avg. Income'  : AvgIncome}
dtf2 = pd.DataFrame(dict2)
print(dtf2)
print(dtf2.Population['Delhi'])
print(dtf2.at['Delhi', 'Population'])
print(dtf2.iat[0,0])
```

Output:

```
         Population  Avg. Income
Delhi      10927986  7216781092
Mumbai     12691836  8508781269
Kolkata     4631392  4226785362
Chennai     4328063  5261784321
10927986
10927986
10927986
```

5. Assigning/Modifying Data Values in Dataframe

(a) To change or modify a single data value, use syntax :

<DF>.<columnname>[<row name/label>] = <value>

Example:

```
import numpy as np
import pandas as pd

population=pd.Series([10927986,12691836,4631392,4328063],\
                    index=['Delhi', 'Mumbai','Kolkata','Chennai'])
AvgIncome = pd.Series([7216781092,8508781269,4226785362,5261784321],\
                     index=['Delhi', 'Mumbai','Kolkata','Chennai'])
dict2 = {'Population' : population , 'Avg. Income'  : AvgIncome}
dtf2 = pd.DataFrame(dict2)
print(dtf2)
dtf2.Population['Mumbai'] = 63819621
print(dtf2)
```

Output:

```
       Population  Avg. Income
Delhi      10927986  7216781092
Mumbai     12691836  8508781269
Kolkata     4631392  4226785362
Chennai     4328063  5261784321
       Population  Avg. Income
Delhi      10927986  7216781092
Mumbai     63819621  8508781269
Kolkata     4631392  4226785362
Chennai     4328063  5261784321
```

6. **<u>Adding Columns , rows and Deleting Columns in DataFrames</u>**

(a) To change or add a column, use syntax :

<DF object>[< column name >] = <new value>

If the given column name does not exist in dataframe then a new column with this name is added. ***But the rows of this new column have the same given value.***

Other ways of adding a column to a dataframe :

<DF object> . at [: , <columnname>] = <values for column>

Or

<DF Object> . loc [: , <columnname>] = < values for column >

(b) Similarly, to change or add a row, use syntax:

<DF object> . at [<rowname> , :] = <new value>

Or

<DF Object> . loc [<row name> , :] = <new value>

Likewise, if there is no row with such row label , then Python adds new row with this *row label* and assigns given values to all its columns. ***But the columns of this new row have the same given value.***

(c) If you want to add a new column that has different values for all its rows, then you can assign the data values for each row of the column in form of a list, e.g.

<DF Object>[<column name>] = [<value>, <value>, ……]

<u>Example:</u>

```
import numpy as np
import pandas as pd

population=pd.Series([10927986,12691836,4631392,4328063],\
                    index=['Delhi', 'Mumbai','Kolkata','Chennai'])
AvgIncome = pd.Series([7216781092,8508781269,4226785362,5261784321],\
                    index=['Delhi', 'Mumbai','Kolkata','Chennai'])
dict2 = {'Population' : population , 'Avg. Income'  : AvgIncome}
dtf2 = pd.DataFrame(dict2)
print(dtf2)
print("==Adding Column==")
dtf2['density']=1219
print(dtf2)
print("==Adding Row==")
dtf2.at['Bangalore', : ] = 1200
print(dtf2)
print("==Adding Column with different values==")
dtf2['density']= [1500, 1219 , 1630, 1050, 1100]
print(dtf2)
```

<u>Output:</u>

```
        Population  Avg. Income
Delhi     10927986   7216781092
Mumbai    12691836   8508781269
Kolkata    4631392   4226785362
Chennai    4328063   5261784321
==Adding Column==
        Population  Avg. Income  density
Delhi     10927986   7216781092     1219
Mumbai    12691836   8508781269     1219
Kolkata    4631392   4226785362     1219
Chennai    4328063   5261784321     1219
==Adding Row==
           Population   Avg. Income  density
Delhi      10927986.0  7.216781e+09   1219.0
Mumbai     12691836.0  8.508781e+09   1219.0
Kolkata     4631392.0  4.226785e+09   1219.0
Chennai     4328063.0  5.261784e+09   1219.0
Bangalore      1200.0  1.200000e+03   1200.0
==Adding Column with different values==
           Population   Avg. Income  density
Delhi      10927986.0  7.216781e+09     1500
Mumbai     12691836.0  8.508781e+09     1219
Kolkata     4631392.0  4.226785e+09     1630
Chennai     4328063.0  5.261784e+09     1050
Bangalore      1200.0  1.200000e+03     1100
```

7. **Deleting Columns and rows**

To delete a column, you use **del** statement as this :

<Df object>

del <Df object>[<column name>]

To delete rows from a dataframe, you can use :

<DF>.drop(<DF object>.index[[index value(s)]])

e.g.

```
import numpy as np
import pandas as pd

population=pd.Series([10927986,12691836,4631392,4328063],\
                     index=['Delhi', 'Mumbai','Kolkata','Chennai'])
AvgIncome = pd.Series([7216781092,8508781269,4226785362,5261784321],\
                     index=['Delhi', 'Mumbai','Kolkata','Chennai'])
dict2 = {'Population' : population , 'Avg. Income'  : AvgIncome}
dtf2 = pd.DataFrame(dict2)
print("Dataframe before deletion of column")
dtf2['density']= [1500, 1219 , 1630, 1050]
print(dtf2)
print("Dataframe after deletion of column")
del dtf2['density']
print(dtf2)
print("Dataframe after deletion of first and third row")
print(dtf2.drop(dtf2.index[[0,2]]))
```

Descriptive Statistics with Pandas

sal_df

	2016	2017	2018	2019
Qtr1	34500	44900	54500	61000.0
Qtr2	56000	46100	51000	NaN
Qtr3	47000	57000	57000	NaN
Qtr4	49000	59000	58500	NaN

1. Functions **min( )** and **max( )**
   - The min( ) and max( ) functions find out the minimum or maximum value respectively.
   - The syntax for using min( ) and max( ) is :

     <dataframe>.min(axis=0 or 1 , skipna = True or False , numeric_only = True or False)

<dataframe>.max(axis=0 or 1 , skipna = True or False , numeric_only = True or False)

axis = 0 (default) minimum calculated along the columns.
axis = 1 minimum calculated along the rows.
skipna = (True or False) Exclude NA/null values when computing result
numeric_only = (True or False) Include only float, int , boolean columns. If None, will attempt to use everything, then use only numeric data.

e.g.1

```
In [38]: sal_df.min()
Out[38]:
2016     34500.0
2017     44900.0
2018     51000.0
2019     61000.0
dtype: float64
```

By default, the calculation is done on index/rows , *i.e.*, **axis = 0**) and for each column the calculated result is displayed

```
In [40]: sal_df.max()
Out[40]:
2016     56000.0
2017     59000.0
2018     58500.0
2019     61000.0
dtype: float64
```

```
In [39]: sal_df.min(axis = 1)
Out[39]:
Qtr1     34500.0
Qtr2     46100.0
Qtr3     47000.0
Qtr4     49000.0
dtype: float64
```

When axis=1 argument is specified then calculation is done along the columns and for each row, the calculated result is displayed

```
In [41]: sal_df.max(axis = 1)
Out[41]:
Qtr1     61000.0
Qtr2     56000.0
Qtr3     57000.0
Qtr4     59000.0
dtype: float64
```

e.g. 2. sal_df.min(axis=1, skipna=False)

```
Qtr1    34500.0
Qtr2        NaN
Qtr3        NaN
Qtr4        NaN
```

e.g. 3. sal_df.max(axis=0, skipna=False)

```
2016    56000.0
2017    59000.0
2018    58500.0
2019        NaN
```

2. <u>Functions **mode( ) , mean( ) , median( )**</u>

<u>Mode( )</u>

- It returns the mode value (i.e., the value that appears most often) from a set of values.
- The Syntax( ) for using mode( ) is:

<dataframe>.mode(axis=0 , numeric_only=False)

- The mode( ) gets the mode(s) of each element along the axis selected.

<u>Mean( )</u>

- It returns the computed mean(average) from a set of values.
- The syntax( ) for using mean( ) is :
  <dataframe>.mean(axis=0 or 1 , skipna = True or False , numeric_only = True or False)

Median( )

- It returns the middle number from a set of numbers.
- The syntax( ) for using mean( ) is :
  <dataframe>.median(axis=0 or 1 , skipna = True or False , numeric_only = True or False)

e.g.1.

mode( )
Returns the mode (the value appearing the most)

```
In [44]: sal_df.mode()
Out[44]:
    2016   2017   2018     2019
0  34500  44900  51000  61000.0
1  47000  46100  54500      NaN
2  49000  57000  57000      NaN
3  56000  59000  58500      NaN
```

```
In [45]: sal_df.mode(axis = 1 )
Out[45]:
            0        1        2        3
Qtr1  34500.0  44900.0  54500.0  61000.0
Qtr2  46100.0  51000.0  56000.0      NaN
Qtr3  57000.0      NaN      NaN      NaN
Qtr4  49000.0  58500.0  59000.0      NaN
```

median( )
Returns the middle value

```
In [46]: sal_df.median()
Out[46]:
2016    48000.0
2017    51550.0
2018    55750.0
2019    61000.0
dtype: float64
```

```
In [47]: sal_df.median(axis = 1)
Out[47]:
Qtr1    49700.0
Qtr2    51000.0
Qtr3    57000.0
Qtr4    58500.0
dtype: float64
```

mean( )
Returns the mean/average value

```
In [48]: sal_df.mean()
Out[48]:
2016    46625.0
2017    51750.0
2018    55250.0
2019    61000.0
dtype: float64
```

```
In [49]: sal_df.mean(axis = 1)
Out[49]:
Qtr1    48725.000000
Qtr2    51033.333333
Qtr3    53666.666667
Qtr4    55500.000000
dtype: float64
```

e.g.2. sal_df.mean(axis=1, skipna=False)

```
Qtr1    48725.0
Qtr2        NaN
Qtr3        NaN
Qtr4        NaN
```

3. Functions **count( )** and **sum( )**

count( )

- It **counts** the **non-NA entries** for each row or column.
- The Syntax for using count( ) is :

  <dataframe>.count(axis=0 or 1 , numeric_only=True or False)

sum( )

- It returns the **sum of the values for the requested axis**.
- The Syntax for using sum( ) is:

  <dataframe>.sum(axis=0 or 1 , skipna = True or False , numeric_only = True or False, min_count=0)

  min_count – the required number of valid values to perform the operation , default value is 0.

e.g.

count( )
Returns count of non NA values for each row/column

```
In [50]: sal_df.count()
Out[50]:
2016    4
2017    4
2018    4
2019    1
dtype: int64
```

```
In [51]: sal_df.count(axis = 1)
Out[51]:
Qtr1    4
Qtr2    3
Qtr3    3
Qtr4    3
dtype: int64
```

sum( )
Returns sum of values for given *axis*.

```
In [52]: sal_df.sum()
Out[52]:
2016     186500.0
2017     207000.0
2018     221000.0
2019      61000.0
dtype: float64
```

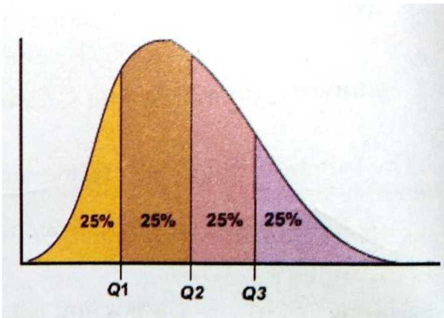
```
In [53]: sal_df.sum(axis = 1)
Out[53]:
Qtr1    194900.0
Qtr2    153100.0
Qtr3    161000.0
Qtr4    166500.0
dtype: float64
```

5. Functions **quantile( )** and **var( )**
   - The qunatile( ) function **returns the values at the given quantiles** over requested axis(axis 0 or 1).

Quantile

- These are points in a distribution that relate to the rank order of values in that distribution.
- The quantile of a value is the fraction of observations less than or equal to the value.

Quartiles:



- Lower Quartile (Q1) has one-fourth of data values at or below it(middle of smaller half)
- Upper Quartile (Q3) has three-fourth of data values at or below it(middle of larger half)
- Interquartile range(IQR) = Q3 – Q2
- The only **2-quantile** is called the **median**.
- The **3-quantiles** are called **tertiles or terciles**.
- The **4-quantiles** are called **quartiles**.

- The Syntax of quantile( ) function

  <dataframe>.quantile(q=0.5 , axis = 0 or 1 , numeric_only=True or False)

  Parameters:

  q – float or array like, default 0.5 (50% quantile). 0<=q<=1, the quantile(s) to compute.

  - **If q is an array**, **a DataFrame** will be returned where the index is q, the columns are the columns of self, and the values are the quantiles.
  - **If q is a float** , **a Series** will be returned where the index is the columns of self and the values are the quantiles.

  e.g.

```
In [55]: sal_df.quantile(q = [0.25, 0.5, 0.75, 1.0])
Out[55]:
         2016     2017     2018     2019
0.25  43875.0  45800.0  53625.0  61000.0
0.50  48000.0  51550.0  55750.0  61000.0
0.75  50750.0  57500.0  57375.0  61000.0
1.00  56000.0  59000.0  58500.0  61000.0
```

Quantiles columnwise

```
In [56]: sal_df.quantile(q = [0.25, 0.5, 0.75, 1.0], axis = 1)
Out[56]:
         Qtr1     Qtr2     Qtr3     Qtr4
0.25  42300.0  48550.0  52000.0  53750.0
0.50  49700.0  51000.0  57000.0  58500.0
0.75  56125.0  53500.0  57000.0  58750.0
1.00  61000.0  56000.0  57000.0  59000.0
```

Quantiles rowwise

var( ) function

- It computes variance and returns unbiased variance over requested axis.
- The syntax for using the var( ) function is:

  <dataframe>.var(axis=0 or 1 , skipna =True or False, numeric_only=True or False)

e.g.

```
In [57]: sal_df.var()
Out[57]:
2016     8.022917e+07
2017     5.299000e+07
2018     1.075000e+07
2019              NaN
dtype: float64
```

```
In [58]: sal_df.var(axis = 1)
Out[58]:
Qtr1     1.336692e+08
Qtr2     2.450333e+07
Qtr3     3.333333e+07
Qtr4     3.175000e+07
dtype: float64
```

Applying Functions on a Subset of Dataframe

Sometimes, you need to apply a function on a selective column or a row or a subset of the data frame.

- Applying Functions **on a column** of a DataFrame

  To apply a function on a column, you need to use following in place of dataframe name

  ```
  <dataframe>[<column name>]
  ```

  And then apply the function on it (*see* examples below)

  ```
  In [17]: sal_df[2018].min()
  Out[17]: 51000
  ```

  Applying functions on individual column of a dataframe

  ```
  In [19]: sal_df[2019].count()
  Out[19]: 1
  ```

- Applying Functions **on Multiple Columns** of a Dataframe

  To apply a function on multiple columns, you need to use following in place of dataframe name :

  ```
  <dataframe>[ [<column name>, <columnname>, ...] ]
  ```

  group of column names given in a list within [ ] of dataframe. Notice double [[ ]]

  And then apply the function on it (*see* examples below)

  ```
  In [20]: sal_df[[2018, 2019]].count()
  Out[20]:
  2018    4
  2019    1
  dtype: int64
  ```

  Applying functions on multiple columns of a dataframe

  ```
  In [21]: sal_df[[2018, 2019]].max()
  Out[21]:
  2018    58500.0
  2019    61000.0
  dtype: float64
  ```

- Applying Functions **on a Row** of a Dataframe

  To apply a function on a row, you need to use following in place of dataframe name :

  ```
  <dataframe>.loc[<row index>, :]
  ```

  And then apply the function on it (*see* examples below)

  ```
  In [22]: sal_df.loc['Qtr2', :].max()
  Out[22]: 56000.0
  ```

  Applying functions on individual row of a dataframe

  ```
  In [23]: sal_df.loc['Qtr2', :].count()
  Out[23]: 3
  ```

- Applying Functions **on a Range of Rows** of a Dataframe

To apply a function on multiple rows, you need to use following in place of dataframe name:

```
<dataframe>.loc[ <start row>: <end row>, : ]
```

And then apply the function on it (*see* examples below)

```
In [28]: sal_df.loc['Qtr3':'Qtr4' , :].count()
Out[28]:
2016    2
2017    2
2018    2
2019    0
dtype: int64
```

```
In [29]: sal_df.loc['Qtr3':'Qtr4' , :].max()
Out[29]:
2016    49000.0
2017    59000.0
2018    58500.0
2019        NaN
dtype: float64
```

- Applying functions to a **subset** of the Dataframes

To apply a function on a subset of dataframe, you need to use following in place of dataframe name :

```
<dataframe>.loc[ <start row> : <end row>, : <start column> : <end column>]
```

And then apply the function on it (*see* examples below)

```
In [30]: sal_df.loc['Qtr3':'Qtr4' , 2018:2019].max()
Out[30]:
2018    58500.0
2019        NaN
dtype: float64
```

```
In [31]: sal_df.loc['Qtr3':'Qtr4' , 2018:2019].count()
Out[31]:
2018    2
2019    0
dtype: int64
```

<u>Advanced Operations on Dataframe</u>

1. Pivoting 2. Sorting 3. Aggregation

<u>Pivoting</u>

- Pivoting technique **rearranges the data from rows and columns**, by possibly **aggregating data** from multiple sources, in a report form (with rows transferred to columns) so that data can be viewed in a different perspective.

- In simplest term, the pivoting means **summarising the data in a way to make understanding of descriptive data easier.** For example, consider the following data:

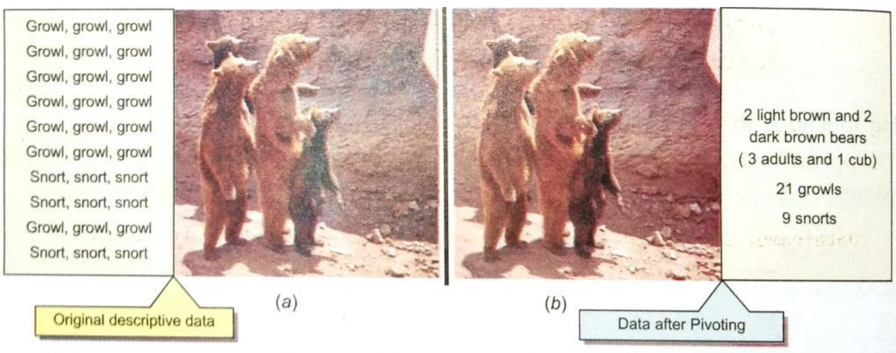


Figure 2.2 Impact of Pivoting : (*a*) Original, descriptive dataset (*b*) Summarised data by pivoting.

<u>Using pivot( ) function</u>

(i) First of all, represent data in a Dataframe datastructure of pandas :

```
import pandas as pd
d1 = {    'Tutor' : ['Tahira', 'Gurjyot', 'Anusha', 'Jacob', 'Venkat'],
          'Classes': [28, 36, 41, 32, 40],
          'Country' :['USA', 'UK', 'Japan', 'USA', 'Brazil']
     }
dfd = pd.DataFrame(d1)
```

```
In [13]: dfd
Out[13]:
   Classes Country   Tutor
0       28     USA  Tahira
1       36      UK Gurjyot
2       41   Japan  Anusha
3       32     USA   Jacob
4       40  Brazil  Venkat
```

(ii) Once you have represented your data in the form of a dataframe, you can pivot it using **function pivot( )** as per following syntax :

```
<dataframe>.pivot(index = <columnname>, columns = <columnname>,
values = <columnname>)
```

e.g.,

```
dfd.pivot(index = 'country', columns = 'Tutor', values = 'classes')
pivot(index = , columns = , values = )
```

Specify here the column which is to be treated as index (i.e., as rows)

Specify here the column, whose values will become columns

Specify here the column, whose values are to be spread across the dataframe created as per specified ***index*** *and* ***columns.***

Tutor	Classes	Country
Tahira	28	USA
Gurjyot	36	UK
Anusha	41	Japan
Jacob	32	USA
Venkat	40	Brazil

Tutor / Country	Anusha	Gurjyot	Jacob	Tahira	Venkat
Brazil	NaN	NaN	NaN	NaN	40.0
Japan	41.0	NaN	NaN	NaN	NaN
UK	NaN	36.0	NaN	NaN	NaN
USA	NaN	NaN	32.0	28.0	NaN

```
dfd.pivot(index = 'Country', columns='Tutor', values='Classes')
```

You can skip the values argument, and if you skip the values argument, it will consider the rest of the columns(not mentioned in **index** and **columns** arguments) for values automatically. E.g.

```
In [21]: dfd.pivot(index = 'Tutor', columns='Country')
Out[21]:
        Classes
Country  Brazil Japan    UK   USA
Tutor
Anusha      NaN  41.0   NaN   NaN
Gurjyot     NaN   NaN  36.0   NaN
Jacob       NaN   NaN   NaN  32.0
Tahira      NaN   NaN   NaN  28.0
Venkat     40.0   NaN   NaN   NaN
```

<u>Error while using pivot( )</u>

- Consider the following DataFrame df1:

	Classes	Country	Quarter	Tutor
0	28	USA	1	Tahira
1	36	UK	1	Gurjyot
2	41	Japan	1	Anusha
3	32	USA	1	Jacob
4	40	Brazil	1	Venkat
5	36	USA	2	Tahira
6	40	USA	2	Gurjyot
7	36	Japan	2	Anusha
8	40	Brazil	2	Jacob
9	46	USA	2	Venkat
10	24	Brazil	3	Tahira
11	30	USA	3	Gurjyot
12	44	UK	3	Anusha
13	40	Brazil	3	Jacob
14	32	USA	3	Venkat
15	36	Japan	4	Tahira
16	32	Japan	4	Gurjyot
17	36	Brazil	4	Anusha
18	42	UK	4	Jacob
19	38	USA	4	Venkat

- If we try to use pivot( ) for the above data frame:

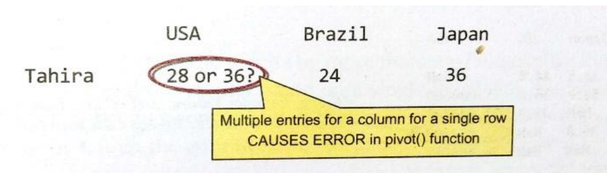
  df1.pivot(index= "Tutor", Columns = "Country")

  it will give error as ***"Index contains duplicate entries, cannot reshape".***

- **E.g.** Let us consider one Tutor say Tahira's entries.

Classes	Country	Quarter	Tutor
28	USA	1	Tahira
36	USA	2	Tahira
24	Brazil	3	Tahira
36	Japan	4	Tahira

If you try to create a row for the tutor Tahira from above data with columns as Country:



Therefore, ***with pivot( ), if there are multiple entries for a columns value for the same value for index(row), it leads to error. Hence,before you use pivot( ), you should ensure that the data does not have rows with duplicate values for the specified columns.***

<u>Using pivot_table( ) Function</u>

- *For data having multiple values for same row and column combination, you can use another pivoting function – the **pivot-table( ) function**.*
- It is different from the pivot( ) function in following ways:
  1. It **does not raise error for multiple entries** of a row, column combination.
  2. It **aggregates the multiple entries present** for a row-column combination; you need to specify what type of aggregation you want(sum, mean, etc.)
- **Syntax:**

```
pandas.pivot_table(<dataframe>, values=None, index=None, columns=None, aggfunc='mean')
```

or

```
(<dataframe>.pivot_table( values = None, index = None, columns = None, aggfunc = 'mean')
```

where

the **index** argument contains the column name for rows.

the **columns** argument contains the column name for columns.

the **values** argument contains the column names for data of the pivoted table.

the **aggfunc** argument contains the function as per which data is to be aggregated, if skipped, it, **by default will compute the mean** of the multiple entries for the same row-column combination.

- **E.g.**

```
Country  Japan  Brazil  Japan   UK      USA
Tutor
Anusha     NaN    36.0   38.5  44.0       NaN
Gurjot    32.0     NaN    NaN  36.0  35.000000
Jacob      NaN    40.0    NaN  42.0  32.000000
Tahira     NaN    24.0   36.0   NaN  32.000000
Venkat     NaN    40.0    NaN   NaN  38.666667
```

Notice, for index **Tahira** and column **USA**, the mean of 2 values (28 , 36) has been given here.

You can use any aggregate function for **aggfun argument (i.e. , min , max , mode , median , mean , count etc.)*

E.g.2. Considering Dataframe df1, compute total classes per tutor.

```
In [60]: df1.pivot_table(index = 'Tutor', values = 'Classes', aggfunc = 'sum')
Out[60]:
        Classes
Tutor
Anusha      157
Gurjyot     138
Jacob       154
Tahira      124
Venkat      156
```

E.g.3. Considering Dataframe df1, computer number of countries (count) per tutor.

```
In [61]: df1.pivot_table(index = 'Tutor', values = 'Country', aggfunc = 'count')
Out[61]:
        Country
Tutor
Anusha        4
Gurjyot       4
Jacob         4
Tahira        4
Venkat        4
```

E.g.4. Considering Dataframe df1, compute total classes by country.

```
In [62]: df1.pivot_table(index = 'Country', values = 'Classes', aggfunc = 'sum')
Out[62]:
          Classes
Country
Brazil        144
Brazil         36
Japan         145
UK            122
USA           282
```

E.g.5. Considering Dataframe df1, compute total classes on two field, Tutor and country wise.

```
In [64]: df1.pivot_table(index=['Tutor','Country'], values=['Classes'], aggfunc='sum')
Out[64]:
                  Classes
Tutor   Country
Anusha  Brazil         36
        Japan          77
        UK             44
Gurjyot Japan          32
        UK             36
        USA            70
Jacob   Brazil         80
        UK             42
        USA            32
Tahira  Brazil         24
        Japan          36
        USA            64
Venkat  Brazil         40
        USA           116
```

Sorting

- It refers to **arranging values** in a particular order.
- The values can be sorted on the basis of a specific column or columns and can be ascending or descending order.
- **Syntax:**

  <dataframe>.sort_values(by , axis = 0 or 1 , ascending = True , inplace = False , na_position = 'first' or 'last')

  Parameters:

  by - Name or list of names to sort by.

  ascending – default True , if False, then sorting in descending order.

  inplace – bool , default False ; if True, perform operation in-place.

  na_position – first or last , default last ; first puts NaNs at the beginning, last puts NaNs at the end.

```
In [66]: df1.sort_values('Country')
Out[66]:
    Classes Country  Quarter    Tutor
4        40  Brazil        1   Venkat
8        40  Brazil        2    Jacob
10       24  Brazil        3   Tahira
13       40  Brazil        3    Jacob
17       36  Brazil        4   Anusha
2        41   Japan        1   Anusha
16       32   Japan        4  Gurjyot
15       36   Japan        4   Tahira
7        36   Japan        2   Anusha
1        36      UK        1  Gurjyot
18       42      UK        4    Jacob
12       44      UK        3   Anusha
0        28     USA        1   Tahira
14       32     USA        3   Venkat
9        46     USA        2   Venkat
6        40     USA        2  Gurjyot
5        36     USA        2   Tahira
3        32     USA        1    Jacob
11       30     USA        3  Gurjyot
19       38     USA        4   Venkat
```

```
In [71]: df1.sort_values('Tutor')
Out[71]:
    Classes Country  Quarter    Tutor
2        41   Japan        1   Anusha
17       36  Brazil        4   Anusha
7        36   Japan        2   Anusha
12       44      UK        3   Anusha
1        36      UK        1  Gurjyot
6        40     USA        2  Gurjyot
11       30     USA        3  Gurjyot
16       32   Japan        4  Gurjyot
3        32     USA        1    Jacob
8        40  Brazil        2    Jacob
18       42      UK        4    Jacob
13       40  Brazil        3    Jacob
0        28     USA        1   Tahira
5        36     USA        2   Tahira
10       24  Brazil        3   Tahira
15       36   Japan        4   Tahira
9        46     USA        2   Venkat
4        40  Brazil        1   Venkat
14       32     USA        3   Venkat
19       38     USA        4   Venkat
```

```
In [67]: df1.sort_values(['Country', 'Tutor'])
Out[67]:
    Classes Country  Quarter    Tutor
8        40  Brazil        2    Jacob
13       40  Brazil        3    Jacob
10       24  Brazil        3   Tahira
4        40  Brazil        1   Venkat
17       36  Brazil        4   Anusha
2        41   Japan        1   Anusha
7        36   Japan        2   Anusha
16       32   Japan        4  Gurjyot
15       36   Japan        4   Tahira
12       44      UK        3   Anusha
1        36      UK        1  Gurjyot
18       42      UK        4    Jacob
6        40     USA        2  Gurjyot
11       30     USA        3  Gurjyot
3        32     USA        1    Jacob
0        28     USA        1   Tahira
5        36     USA        2   Tahira
9        46     USA        2   Venkat
14       32     USA        3   Venkat
19       38     USA        4   Venkat
```

Values sorted Country wise and within Country, Tutor-wise

```
In [68]: df1.sort_values(by =['Tutor', 'Country'])
Out[68]:
    Classes Country  Quarter    Tutor
17       36  Brazil        4   Anusha
2        41   Japan        1   Anusha
7        36   Japan        2   Anusha
12       44      UK        3   Anusha
16       32   Japan        4  Gurjyot
1        36      UK        1  Gurjyot
6        40     USA        2  Gurjyot
11       30     USA        3  Gurjyot
8        40  Brazil        2    Jacob
13       40  Brazil        3    Jacob
18       42      UK        4    Jacob
3        32     USA        1    Jacob
10       24  Brazil        3   Tahira
15       36   Japan        4   Tahira
0        28     USA        1   Tahira
5        36     USA        2   Tahira
4        40  Brazil        1   Venkat
9        46     USA        2   Venkat
14       32     USA        3   Venkat
19       38     USA        4   Venkat
```

Values sorted Tutor wise and within Tutor, country wise

```
In [72]: df1.sort_values(by =['Tutor', 'Country'], ascending = False)
Out[72]:
    Classes Country  Quarter    Tutor
9        46     USA        2   Venkat
14       32     USA        3   Venkat
19       38     USA        4   Venkat
4        40  Brazil        1   Venkat
0        28     USA        1   Tahira
5        36     USA        2   Tahira
15       36   Japan        4   Tahira
10       24  Brazil        3   Tahira
3        32     USA        1    Jacob
18       42      UK        4    Jacob
8        40  Brazil        2    Jacob
13       40  Brazil        3    Jacob
6        40     USA        2  Gurjyot
11       30     USA        3  Gurjyot
1        36      UK        1  Gurjyot
16       32   Japan        4  Gurjyot
12       44      UK        3   Anusha
2        41   Japan        1   Anusha
7        36   Japan        2   Anusha
17       36  Brazil        4   Anusha
```

Values sorted in descending order

Aggregation

- With large amount of data, most often we need to aggregate data so as to analyse it effectively.
- Pandas offers many aggregate functions, using which you can aggregate data and get summary statistics of the data.

S.No.	Aggregation	Description
1.	count( )	Total number of items
2.	sum( )	Sum of all items
3.	mean( ), median( )	Mean and median
4.	min( ), max( )	Minimum and maximum
5.	std( ), var( )	Standard deviation and variance
6.	mad( )	Mean absolute deviation

1. **The mad( ) function**
   - It is used to calculate the **mean absolute deviation** of the values for the requested axis.
   - The Mean Absolute Deviation (MAD) of a set of data is the average distance between each data value and the mean.
   - **Syntax:**

     <dataframe>.mad(axis=None , skipna = True or False)

     Parameters :

     axis = 0(along columns) or 1(along axis)

     skipna = default True ; Exclude NA/null values.

   - **E.g.** sal_df.mad(axis =1) – finding MAD along the rows.

     sal_df.mad( ) - finding MAD along the columns.

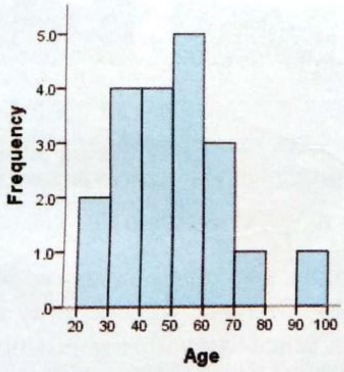
2. **The std( ) function**
   - It calculates the **standard deviation** of a given set of numbers.
   - **E.g.** sal_df.std( ) ,

     sal_df.std(axis = 1)

<u>Creating Histogram</u>

- A Histogram is a plot that lets you discover, and, show, the underlying frequency distribution (shape) of a set of continuous data.
- Consider the following histogram that has been computed using the following dataset containing ages of 20 people.

37	28	38	44	53	69	74	53	35	38	66	46	24	45	92	48	51	62	58	57



Bin	Frequency	Scores Included in Bin
20 – 30	2	28, 24
30 – 40	4	37, 38, 35, 38
40 – 50	4	44, 46, 45, 48
50 – 60	5	53, 53, 51, 58, 57
60 – 70	3	69, 66, 62
70 – 80	1	74
80 – 90	0	–
90 – 100	1	92

- Unlike a bar chart, there are no “gaps” between the bars(although some bars might be “absent” reflecting no frequencies). This is because a histogram represents a continuous data set, and as such, there are no gaps in the data.

- To create a histogram from a dataframe, you can use **hist( )** function of dataframe, which draws one histogram of the DataFrame’s columns.

- **Syntax:**

  Dataframe.hist(column=None, by=None , grid = True , bins = 10)

  Parameters:

  column – string or sequence ; if passed will be used to limit data to a subset of columns.

by – used to form histograms for separate groups.
grid – default True ; whether to show axis grid lines.
bins – default 10 ; Number of histogram bins to be used.

- **E.g.** df1.hist( ) -- by default creates histogram for all numeric columns.
  df1.hist(column='Classes') – Argument 'column' specifies the column for which histogram is to be created.

<u>Function Application</u>

- It means that a function(a library function or user defined function) may be applied on a dataframe in multiple ways:
  (a) on the whole dataframe.
  (b) row-wise or column wise
  (c) on individual elements, i.e., element-wise

- For above mentioned three types of function application, Pandas offers following three functions:
  (a) **pipe( )** – dataframe wise function application
  (b) **apply( )** – row-wise/column wise function application
  (c) **applymap( )** – individual element wise function application

<u>The pipe( ) function</u>

- A pipe is a technique for **passing information from one program process to another** where one command or function's output/result is taken as input for another command/function.

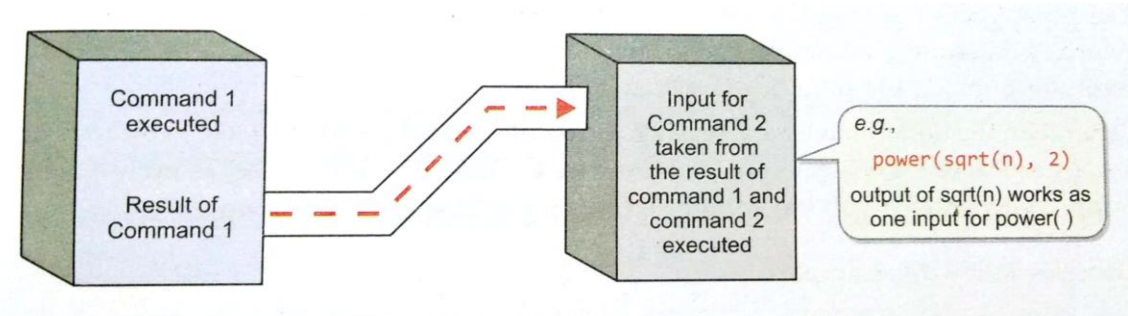
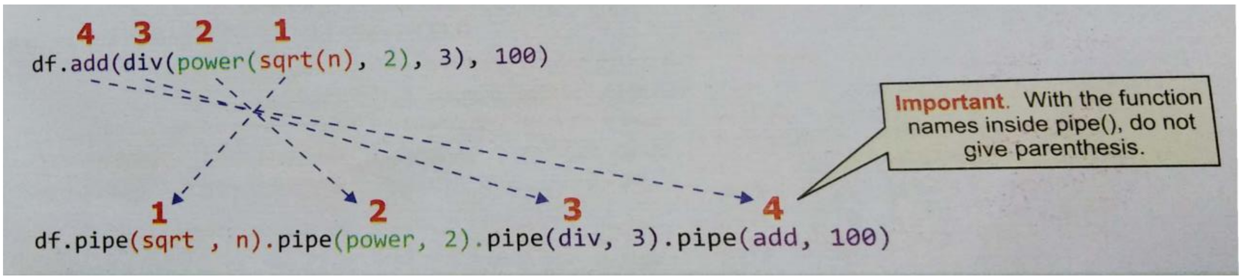


Figure 2.4 Meaning of pipe in computer processing terms

- The pipe() function of pandas does the same. General form of doing this is the **sandwich style** of invoking functions.

  e.g. power(sqrt(n) * 2)

- The piping of functions through pipe( ) basically means the chaining of function in the order they are executed. The pipe( ) works like this:



- **Syntax for using pipe( ) function:**

  <dataframe>.pipe(func , *args)

  Parameters:

  func – function name to be applied on the dataframe with the provided args.
  args – optional,positional arguments passed into **func.**

- When pipe( ) function applied on a dataframe, it will return a DataFrame and when applied on numbers, it will return numbers. Consider following examples:

pipe( ) Example 1 Function add( ) followed by multiply( ) applied on a dataframe.

(**Note.** Both numpy and pandas libraries are imported.)

```
In [15]: sal_df
Out[15]:
       2016   2017   2018     2019
Qtr1  34500  44900  54500  61000.0
Qtr2  56000  46100  51000      NaN
Qtr3  47000  57000  57000      NaN
Qtr4  49000  59000  58500      NaN
```

```
In [18]: np.multiply( sal_df.add(30) , 3)
Out[18]:
          2016      2017      2018      2019
Qtr1  103590.0  134790.0  163590.0  183090.0
Qtr2  168090.0  138390.0  153090.0       NaN
Qtr3  141090.0  171090.0  171090.0       NaN
Qtr4  147090.0  177090.0  175590.0       NaN
```

```
In [20]: sal_df.pipe(np.add, 30).pipe(np.multiply,3)
Out[20]:
          2016      2017      2018      2019
Qtr1  103590.0  134790.0  163590.0  183090.0
Qtr2  168090.0  138390.0  153090.0       NaN
Qtr3  141090.0  171090.0  171090.0       NaN
Qtr4  147090.0  177090.0  175590.0       NaN
```

See, both these commands produced same results

pipe( ) Example 2 Function add( ) followed by multiply( ), followed by sqrt( ) and floor( ) applied on a dataframe.

(**Note.** Both numpy and pandas libraries are imported)

```
In [22]: sal_df.pipe(np.add, 30).pipe(np.multiply,3).pipe(np.sqrt, ).pipe(np.floor, )
Out[22]:
       2016   2017   2018   2019
Qtr1  321.0  367.0  404.0  427.0
Qtr2  409.0  372.0  391.0    NaN
Qtr3  375.0  413.0  413.0    NaN
Qtr4  383.0  420.0  419.0    NaN
```

```
In [25]: np.floor(np.sqrt(np.multiply(sal_df.add(30), 3)))
Out[25]:
       2016   2017   2018   2019
Qtr1  321.0  367.0  404.0  427.0
Qtr2  409.0  372.0  391.0    NaN
Qtr3  375.0  413.0  413.0    NaN
Qtr4  383.0  420.0  419.0    NaN
```

Compare the two commands and their respective results

The apply( ) and applymap( ) functions

1. **apply( )** is a **series function**, so it applies the given function to one row or one column of the dataframe.
2. **applymap( )** is an **element function**, so it applies the given function to each individual element, separately – without taking into account other elements.

- The syntax for using **apply( )** is :

```
<dataframe>.apply(<funcname>, axis = 0)
```

Parameters

<funcname>	the function to be applied on the series inside the dataframes *i.e.*, on rows and columns. It should be a function that works with series and similar objects.
axis	0 or 1 default 0 ; axis along with the function is applied. If **axis** is **0 or 'index'** : function is applied on each column If **axis** is **1 or 'columns'** : function is applied on each row.

- The syntax for using **applymap( )** is :

```
<dataframe>.applymap(<funcname>)
```

where

<funcname>	is the function to be called and it should be a function that works on a single value and returns a single value.

- **e.g.**

np.mean([333, 666, 444]) would yield **481.0**

mean( ) worked with multiple values provided in a list object.

and

np.mean(333) would yield **333.0**

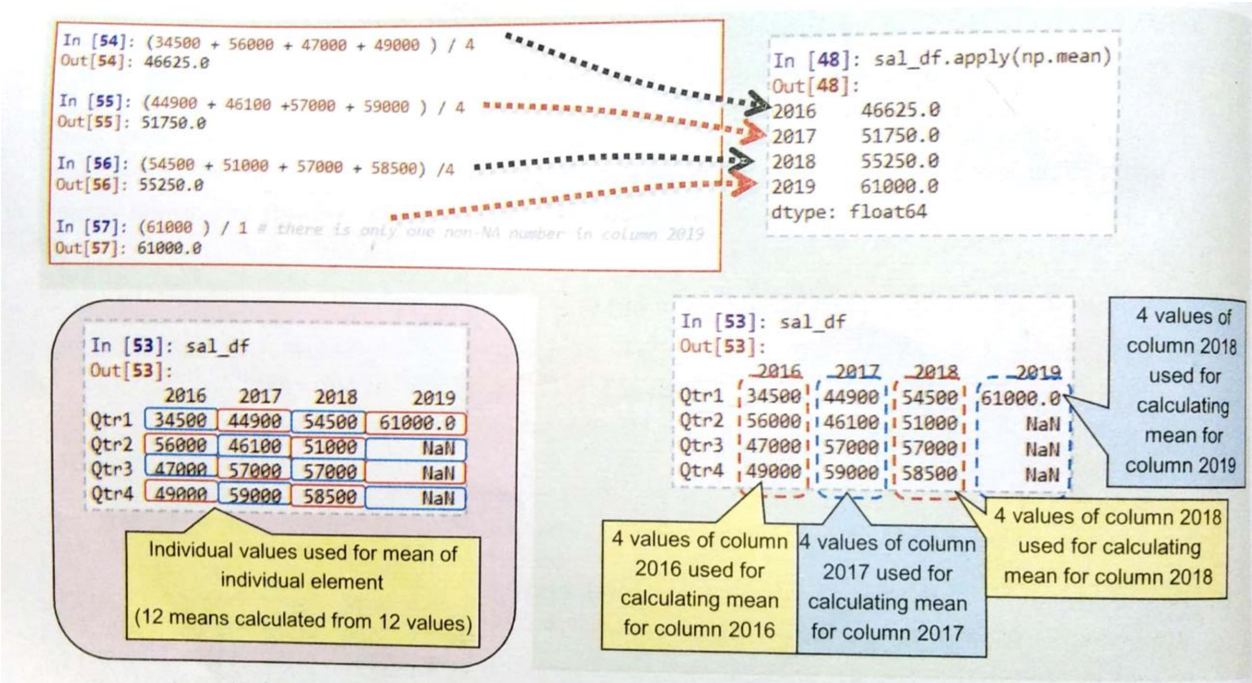
***mean( )** worked with a single value*

```
In [48]: sal_df.apply(np.mean)
Out[48]:
2016    46625.0
2017    51750.0
2018    55250.0
2019    61000.0
dtype: float64
```

For the same function **np.mean**, the **apply()** returned single value per column

While for the same function **np.mean**, the applymap() returned single value per element

```
In [47]: sal_df.applymap(np.mean)
Out[47]:
         2016     2017     2018     2019
Qtr1  34500.0  44900.0  54500.0  61000.0
Qtr2  56000.0  46100.0  51000.0      NaN
Qtr3  47000.0  57000.0  57000.0      NaN
Qtr4  49000.0  59000.0  58500.0      NaN
```



(*b*) (*a*)

Figure 2.5 (*a*) For apply( ), function is applied on series (a row or a column)
(*b*) For applymap( ), function is applied on individual elements

- For apply( ), be default, the axis is 0, i.e., the function is applied on individual columns. To apply the function row-wise, you may write:

```
<dataframe>.apply(<func>, axis = 1)
```

e.g.,

```
In [58]: sal_df.apply(np.mean, axis = 1)
Out[58]:
Qtr1    48725.000000
Qtr2    51033.333333
Qtr3    53666.666667
Qtr4    55500.000000
dtype: float64
```

See, this time, the mean has been calculated row-wise (axis 1)

- **e.g.2.** numpy.cumsum( ), the cumulative sum function which works like this : sum of elements so far, i.e., for a column:

	Column 0	Column 1
Row 0	Elem 0, 0	Elem 0, 1
Row 1	Elem 0, 0 + Elem 1, 0	Elem 0, 1 + Elem 1, 1
Row 2	Elem 0, 0 + Elem 1, 0 + Elem 2, 0	Elem 0, 1 + Elem 1, 1 + Elem 2, 1
Row 3	Elem 0, 0 + Elem 1, 0 + Elem 2, 0 + Elem 3, 0	Elem 0, 1 + Elem 1, 1 + Elem 2, 1 + Elem 3, 1

when the series function **numpy.cumsum** is used with **apply( )** and **applymap( ):**

```
In [44]: sal_df.apply(np.cumsum)
Out[44]:
        2016    2017    2018     2019
Qtr1   34500   44900   54500  61000.0
Qtr2   90500   91000  105500      NaN
Qtr3  137500  148000  162500      NaN
Qtr4  186500  207000  221000      NaN
```

np.cumsum() applied column-wise here (column treated as Series) because of apply( )

```
In [45]: sal_df.applymap(np.cumsum)
Out[45]:
         2016     2017     2018       2019
Qtr1  [34500]  [44900]  [54500]  [61000.0]
Qtr2  [56000]  [46100]  [51000]      [nan]
Qtr3  [47000]  [57000]  [57000]      [nan]
Qtr4  [49000]  [59000]  [58500]      [nan]
```

np.cumsum( ) applied on individual elements because of applymap( )

- for apply( ) the function name should be a Series or array function, i.e., a function that works with Series type objects. If you give name of a single element function as argument (e.g. srqt), then the function will be applied to all elements individually and not to a row or a column and the result will be same as that of a the applymap( ).

```
In [59]: sal_df.apply(np.sqrt)
Out[59]:
            2016        2017        2018        2019
Qtr1  185.741756  211.896201  233.452351  246.981781
Qtr2  236.643191  214.709106  225.831796         NaN
Qtr3  216.794834  238.746728  238.746728         NaN
Qtr4  221.359436  242.899156  241.867732         NaN
```

```
In [60]: sal_df.applymap(np.sqrt)
Out[60]:
            2016        2017        2018        2019
Qtr1  185.741756  211.896201  233.452351  246.981781
Qtr2  236.643191  214.709106  225.831796         NaN
Qtr3  216.794834  238.746728  238.746728         NaN
Qtr4  221.359436  242.899156  241.867732         NaN
```

The result of both **apply()** and **applymap()** is same in above case, because the function name passed to apply is not a Series function, rather it is a single value function. Hence for apply() also, this applied to individual values

<u>Function groupby( )</u>

- Within a dataframe, based on a field's values, we can group the data. In simple words, the ***duplicate values in the same field are grouped together to form groups***. **E.g.** from dataframe df1 (on page no. 20), we can for creating Tutor wise groups:

    - All the rows having **Tutor as Tahira** will be clubbed to form Tahira group.
    - All the rows having **Tutor as Anusha** will be clubbed to form Anusha group.
    - All the rows having **Tutor as Gurjyot** will be clubbed to form Gurjyot group and so on.

- The **syntax of groupby( )** is :

  <dataframe>.groupby(by=None , axis = 0)

  by – labels or list of labels to be used for grouping.
  axis – 0 (for columns) , 1 (for rows)

- The ***groupby( ) creates the groups internally and does not display the grouped data by default***, **e.g.**

```
In [75]: df1.groupby('Tutor')
Out[75]: <pandas.core.groupby.DataFrameGroupBy object at 0x085A4490>
```

Python created groups based on Tutor column's values but did not display grouped data

- You can store the GroupBy object in a variable name and then use **following attributes and functions to get information about groups or to display groups:**

`<GroupByObject>.groups`	lists the groups created
`<GroupByObject>.get_group(<value>)`	lists the group created for the passed value
`<GroupByObject>.size()`	lists the size of the groups created
`<GroupByObject>.count()`	lists the count of non-NA values for each column in the groups created
`<GroupByObject>.[<columnname>].head()`	lists the specified column from the grouped object created

- **Example:**

```
In [78]: gdf = df1.groupby('Tutor')

In [79]: gdf.groups
Out[79]:
{'Anusha': Int64Index([2, 7, 12, 17], dtype='int64'),
 'Gurjyot': Int64Index([1, 6, 11, 16], dtype='int64'),
 'Jacob': Int64Index([3, 8, 13, 18], dtype='int64'),
 'Tahira': Int64Index([0, 5, 10, 15], dtype='int64'),
 'Venkat': Int64Index([4, 9, 14, 19], dtype='int64')}

In [80]: gdf.get_group('Venkat')
Out[80]:
    Classes Country  Quarter   Tutor
4        40  Brazil        1  Venkat
9        46     USA        2  Venkat
14       32     USA        3  Venkat
19       38     USA        4  Venkat

In [81]: gdf.get_group('Gurjyot')
Out[81]:
    Classes Country  Quarter    Tutor
1        36      UK        1  Gurjyot
6        40     USA        2  Gurjyot
11       30     USA        3  Gurjyot
16       32   Japan        4  Gurjyot
```

```
In [82]: gdf.size()
Out[82]:
Tutor
Anusha     4
Gurjyot    4
Jacob      4
Tahira     4
Venkat     4
dtype: int64

In [83]: gdf.count()
Out[83]:
         Classes  Country  Quarter
Tutor
Anusha         4        4        4
Gurjyot        4        4        4
Jacob          4        4        4
Tahira         4        4        4
Venkat         4        4        4
```

```
In [111]: gdf2['Classes'].head()
Out[111]:
0     28
1     36
2     41
3     32
4     40
5     36
6     40
7     36
8     40
9     46
10    24
11    30
12    44
13    40
14    32
15    36
16    32
17    36
18    42
19    38
Name: Classes, dtype: int64
```

First of all, we created the groupby object based on field 'Tutor' and stored it in object namely **gdf**. All other attributes and functions are then applied to this object **gdf**

Grouping on Multiple columns

- For instance, you want to create groups for Tutors and for each tutor group, a country-wise subgroup, so you should write groupby( ) as:

  gdf2=df1. groupby(['Tutor', 'Country'])
- Now you can apply all the group attributes and functions on the groupby object gdf2 :

```
In [89]: gdf2.groups
Out[89]:
{('Anusha', 'Brazil '): Int64Index([17], dtype='int64'),
 ('Anusha', 'Japan'): Int64Index([2, 7], dtype='int64'),
 ('Anusha', 'UK'): Int64Index([12], dtype='int64'),
 ('Gurjyot', 'Japan'): Int64Index([16], dtype='int64'),
 ('Gurjyot', 'UK'): Int64Index([1], dtype='int64'),
 ('Gurjyot', 'USA'): Int64Index([6, 11], dtype='int64'),
 ('Jacob', 'Brazil'): Int64Index([8, 13], dtype='int64'),
 ('Jacob', 'UK'): Int64Index([18], dtype='int64'),
 ('Jacob', 'USA'): Int64Index([3], dtype='int64'),
 ('Tahira', 'Brazil'): Int64Index([10], dtype='int64'),
 ('Tahira', 'Japan'): Int64Index([15], dtype='int64'),
 ('Tahira', 'USA'): Int64Index([0, 5], dtype='int64'),
 ('Venkat', 'Brazil'): Int64Index([4], dtype='int64'),
 ('Venkat', 'USA'): Int64Index([9, 14, 19], dtype='int64')}
```

```
In [96]: gdf2.size()
Out[96]:
Tutor    Country
Anusha   Brazil     1
         Japan      2
         UK         1
Gurjyot  Japan      1
         UK         1
         USA        2
Jacob    Brazil     2
         UK         1
         USA        1
Tahira   Brazil     1
         Japan      1
         USA        2
Venkat   Brazil     1
         USA        3
dtype: int64
```

- But ***while using get_group( ), you need to pass all the values of group-columns in a tuple***. The passed values based group must exist in the groupby object , otherwise Python will give error.

To get a group having tutor name as 'Anusha' and Country as 'UK', pass a sequence containing both these values

```
In [95]: gdf2.get_group(('Anusha',"UK" ))
Out[95]:
    Classes Country  Quarter   Tutor
12       44      UK        3  Anusha
```

```
In [94]: gdf2.get_group(('Anusha',"USA" ))
Traceback (most recent call last):

  File "<ipython-input-94-d0f452dfd705>", line 1, in <module>
    gdf2.get_group(('Anusha',"USA" ))

  File "C:\ProgramData\Anaconda3\lib\si
\groupby.py", line 765, in get_group
    raise KeyError(name)

KeyError: ('Anusha', 'USA')
```

If the passed values do not have a group, Python raises KeyError (No group for '*Anusha*' & '*USA*' 'combination')

Aggregation via groupby( )

- The agg( ) method aggregates the data of the dataframe using one or more operations over the specified axis. The syntax for using agg( ) is :

  <dataframe>.agg(func , axis =0)

  func – function, str or list
  axis - 0 or 1

- E.g.

```
In [86]: gdf.agg([np.mean, np.median, np.sum])
Out[86]:
        Classes              Quarter
           mean median  sum     mean median sum
Tutor
Anusha    39.25   38.5  157      2.5    2.5  10
Gurjyot   34.50   34.0  138      2.5    2.5  10
Jacob     38.50   40.0  154      2.5    2.5  10
Tahira    31.00   32.0  124      2.5    2.5  10
Venkat    39.00   39.0  156      2.5    2.5  10
```

Three aggregate functions (mean, median and sum) applied to groups created via groupby () above

You may combine the **groupby( )** and **agg( )** in single command:

```
In [87]: df1.groupby('Tutor').agg([np.mean, np.median, np.sum])
Out[87]:
        Classes              Quarter
           mean median  sum     mean median sum
Tutor
Anusha    39.25   38.5  157      2.5    2.5  10
Gurjyot   34.50   34.0  138      2.5    2.5  10
Jacob     38.50   40.0  154      2.5    2.5  10
Tahira    31.00   32.0  124      2.5    2.5  10
Venkat    39.00   39.0  156      2.5    2.5  10
```

groupby() and **agg()** combined in single statement

<u>The transform( ) function</u>

- This function transforms the aggregate data by repeating the summary result for each row of the group and makes the result have the same shape as original data and thus the result of transform can be combined with the dataframe easily.**E.g.**

```
In [104]: df1.groupby('Tutor').agg(np.mean)
Out[104]:
         Classes  Quarter
Tutor
Anusha     39.25      2.5
Gurjyot    34.50      2.5
Jacob      38.50      2.5
Tahira     31.00      2.5
Venkat     39.00      2.5
```

See, **agg()** created one row per group with aggregate function result

```
In [106]: df1
Out[106]:
    Classes Country  Quarter    Tutor
0        28     USA        1   Tahira
1        36      UK        1  Gurjyot
2        41   Japan        1   Anusha
3        32     USA        1    Jacob
4        40  Brazil        1   Venkat
5        36     USA        2   Tahira
6        40     USA        2  Gurjyot
7        36   Japan        2   Anusha
8        40  Brazil        2    Jacob
9        46     USA        2   Venkat
10       24  Brazil        3   Tahira
11       30     USA        3  Gurjyot
12       44      UK        3   Anusha
13       40  Brazil        3    Jacob
14       32     USA        3   Venkat
15       36   Japan        4   Tahira
16       32   Japan        4  Gurjyot
17       36  Brazil        4   Anusha
18       42      UK        4    Jacob
19       38     USA        4   Venkat
```

The **transform()** also calculated the same aggregate function but repeated the calculated result for every row of the group, e.g., for 'Venkat' group, for every row of ***venkat***tutor (rows 4, 9, 14, 19) , you will find same aggregated result 39.00 for ***Classes*** and 2.5 for Qu

```
In [105]: df1.groupby('Tutor').transform(np.mean)
Out[105]:
    Classes  Quarter
0     31.00      2.5
1     34.50      2.5
2     39.25      2.5
3     38.50      2.5
4     39.00      2.5
5     31.00      2.5
6     34.50      2.5
7     39.25      2.5
8     38.50      2.5
9     39.00      2.5
10    31.00      2.5
11    34.50      2.5
12    39.25      2.5
13    38.50      2.5
14    39.00      2.5
15    31.00      2.5
16    34.50      2.5
17    39.25      2.5
18    38.50      2.5
19    39.00      2.5
```

- The transform( ) function's output can now be added as columns to the dataframe. To add one column, you need to first use transform for one column at a time, i.e. as shown below:

```
df1.groupby('Tutor')['Classes'].transform(np.mean)
```

By specifying the column name in square bracket with groupby object

- Now you can save the transformed result in a new column.

```
df1['ClassesMean'] = df1.groupby('Tutor')['Classes'].transform(np.mean)
```

```
In [108]: df1['ClassesMean'] = df1.groupby('Tutor')['Classes'].transform(np.mean)

In [109]: df1
Out[109]:
    Classes Country  Quarter    Tutor  ClassesMean
0        28     USA        1   Tahira        31.00
1        36      UK        1  Gurjyot        34.50
2        41   Japan        1   Anusha        39.25
3        32     USA        1    Jacob        38.50
4        40  Brazil        1   Venkat        39.00
5        36     USA        2   Tahira        31.00
6        40     USA        2  Gurjyot        34.50
7        36   Japan        2   Anusha        39.25
8        40  Brazil        2    Jacob        38.50
9        46     USA        2   Venkat        39.00
10       24  Brazil        3   Tahira        31.00
11       30     USA        3  Gurjyot        34.50
12       44      UK        3   Anusha        39.25
13       40  Brazil        3    Jacob        38.50
14       32     USA        3   Venkat        39.00
15       36   Japan        4   Tahira        31.00
16       32   Japan        4  Gurjyot        34.50
17       36  Brazil        4   Anusha        39.25
18       42      UK        4    Jacob        38.50
19       38     USA        4   Venkat        39.00
```

<u>Reindexing and Altering Labels</u>

- Index refers to lables of axis 0 , i.e., row labels and columns refers to the labels of axis 1 i.e., column labels.
- There are methods to rearrange and rename indexes or column labels :
  1. rename( ) – A method that simply ***renames the index and/or column labels*** in a dataframe.
  2. reindex( ) – A method that can specify the ***new order of existing indexes and column labels***, and/or also create new indexes/column labels.
  3. reindex_like( ) – A method for ***creating indexes/column-labels*** based on other dataframe object.

1. <u>The rename( ) method</u>

- This function ***renames the existing indexes/column-labels in a dataframe***.
- The old and new index/column labels are to be provided in the form of a dictionary where ***keys are the old indexes/row labels, and the values are the new names*** for the same, e.g.

{'Qtr1' : 1 , 'Qtr2' : 2 , ……. }

The above dictionary implies that old index/column-label namely 'Qtr1' should be now renamed as 1, 'Qtr2' should be renamed as 2 , and so on.

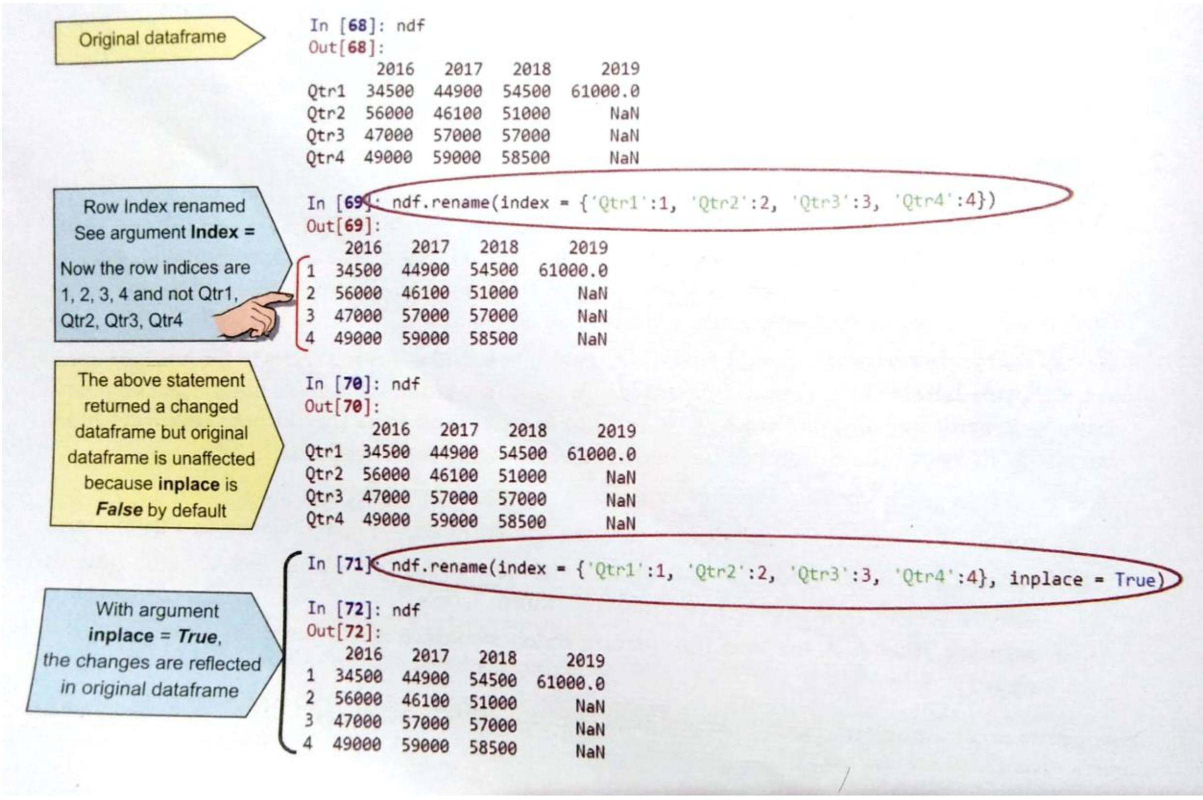
- **Syntax :**

<dataframe>.rename(index=None , columns = None , inplace=False)

or

<dataframe>.rename({dictionary with old and new labels}, axis = 0 or 1)

- **E.g.**




```
In [68]: ndf
Out[68]:
       2016   2017   2018     2019
Qtr1  34500  44900  54500  61000.0
Qtr2  56000  46100  51000      NaN
Qtr3  47000  57000  57000      NaN
Qtr4  49000  59000  58500      NaN

In [69]: ndf.rename(index = {'Qtr1':1, 'Qtr2':2, 'Qtr3':3, 'Qtr4':4})
Out[69]:
    2016   2017   2018     2019
1  34500  44900  54500  61000.0
2  56000  46100  51000      NaN
3  47000  57000  57000      NaN
4  49000  59000  58500      NaN

In [70]: ndf
Out[70]:
       2016   2017   2018     2019
Qtr1  34500  44900  54500  61000.0
Qtr2  56000  46100  51000      NaN
Qtr3  47000  57000  57000      NaN
Qtr4  49000  59000  58500      NaN

In [71]: ndf.rename(index = {'Qtr1':1, 'Qtr2':2, 'Qtr3':3, 'Qtr4':4}, inplace = True)

In [72]: ndf
Out[72]:
    2016   2017   2018     2019
1  34500  44900  54500  61000.0
2  56000  46100  51000      NaN
3  47000  57000  57000      NaN
4  49000  59000  58500      NaN
```

```
In [75]: ndf.rename( {2016:16, 2017:17, 2018:18, 2019:19}, axis = 1)
Out[75]:
         16     17     18       19
Qtr1  34500  44900  54500  61000.0
Qtr2  56000  46100  51000      NaN
Qtr3  47000  57000  57000      NaN
Qtr4  49000  59000  58500      NaN
```

```
In [76]: ndf.rename( columns = {2016:16, 2017:17, 2018:18, 2019:19} )
Out[76]:
         16     17     18       19
Qtr1  34500  44900  54500  61000.0
Qtr2  56000  46100  51000      NaN
Qtr3  47000  57000  57000      NaN
Qtr4  49000  59000  58500      NaN
```

2. **<u>The reindex( ) method</u>**

- This function is used to ***change the order or existing indices/labels***.
- Syntax:

Dataframe.reindex(index=None, columns=None , fill_value=nan)

Or

Dataframe.reindex([list of rearranged index/column labels] , axis = 0 or 1)

- **e.g.**

```
ndf.reindex(['Qtr4', 'Qtr1', 'Qtr3', 'Qtr2'])
ndf.reindex(['Qtr4', 'Qtr1', 'Qtr3', 'Qtr2'] , axis = 0)
```

See the new order of row- indices is as per the order of indices mentioned in reindex()

(compare it with original **ndf** listed earlier)

```
In [78]: ndf.reindex(['Qtr4', 'Qtr1', 'Qtr3', 'Qtr2'])
Out[78]:
       2016   2017   2018     2019
Qtr4  49000  59000  58500      NaN
Qtr1  34500  44900  54500  61000.0
Qtr3  47000  57000  57000      NaN
Qtr2  56000  46100  51000      NaN
```

An alternate command for the above result will be:

```
ndf.reindex(index = ['Qtr4', 'Qtr1', 'Qtr3', 'Qtr2'])
```

Reordering as well as adding/deleting indexes/labels

- Existing row-indices/column-labels are reordered as per given order and non-existing row-indexes/column-labels create new rows/columns and by default NaN values are filled in them.
- **e.g.**

```
In [88]: ndf.reindex([2019, 2018, 2017, 2016, 2015, 2014], axis = 1)
Out[88]:
         2019   2018   2017   2016  2015  2014
Qtr1  61000.0  54500  44900  34500   NaN   NaN
Qtr2      NaN  51000  46100  56000   NaN   NaN
Qtr3      NaN  57000  57000  47000   NaN   NaN
Qtr4      NaN  58500  59000  49000   NaN   NaN
```

See, the column labels are as per mentioned order (existing as well as non-existing)

For non-existing labels, new columns with NaN values have been created.

Newly added columns (by default filled with NaN)

The new dataframe generated by **reindex( )** contains only the row-indices/column-labels as per the given mapper sequence (see below).

```
In [89]: ndf.reindex(['Qtr4', 'Qtr1', 'QtNil'])
Out[89]:
          2016     2017     2018     2019
Qtr4   49000.0  59000.0  58500.0      NaN
Qtr1   34500.0  44900.0  54500.0  61000.0
QtNil      NaN      NaN      NaN      NaN
```

See, only 3 row indices aer there as mentioned in the given mapper sequence

Existing ones remain and new ones added, BUT if an existing index/label is not mentioned in the mapper list, it will not be a part of the new dataframe

Specifying fill values for new rows/columns

- By using argument **fill_value,** you can specify which will be filled in the newly added row/column. In the absence of **fill_value** argument, the new row/column is filled with NaN.
- **E.g.**

```
In [91]: ndf.reindex(['Qtr4', 'Qtr1', 'QtNil'], fill_value = 1000)
Out[91]:
        2016   2017   2018     2019
Qtr4   49000  59000  58500      NaN
Qtr1   34500  44900  54500  61000.0
QtNil   1000   1000   1000   1000.0
```

```
In [92]: ndf.reindex(columns = [2019, 2017, 2015], fill_value = 5000)
Out[92]:
         2019   2017  2015
Qtr1  61000.0  44900  5000
Qtr2      NaN  46100  5000
Qtr3      NaN  57000  5000
Qtr4      NaN  59000  5000
```

3. <u>The reindex_like( ) method</u>

- This function rearrange the row/column labels as per the row/ column labels of some other dataframe.
- This function does the following things:

(a) If the current dataframe has some **matching row-indexes/column-labels** as the passed dataframe, then **retain the index/label and its data.**

(b) If the current dataframe has some **row-indexes/column-labels** in it, which are **not in the passed dataframe, drop them.**

(c) If the current dataframe does not have some row-indexes/column-labels which are in the passed dataframe, then **add them to current dataframe with value as NaN.**

(d) The **reindex_like( ) ensure that the current dataframe object conforms to the same indexes/labels on all axes.**

- **Syntax:**

  <dataframe>.reindex_like(other dataframe)

- **E.g.** consider the two dataframes:

```
In [110]: ndf2
Out[110]:
         2019     2017    2015    2013    2011
Qtr1  61000.0  44900.0  5000.0  5000.0  5000.0
Qtr3      NaN  57000.0  5000.0  5000.0  5000.0
Qtr4      NaN  59000.0  5000.0  5000.0  5000.0
Qtn       NaN      NaN     NaN     NaN     NaN
```

Notice, **ndf2** has 2 columns 2019 and 2017 same as sal_df and 3 rows (*Qtr1, Qtr3, Qtr4*) same as **sal_df**

sal_df has extra columns as 2016 , 2018 and extra row as **Qtr2**

```
In [104]: sal_df
Out[104]:
       2016   2017   2018     2019
Qtr1  34500  44900  54500  61000.0
Qtr2  56000  46100  51000      NaN
Qtr3  47000  57000  57000      NaN
Qtr4  49000  59000  58500      NaN
```

If we issue command as:

ndf2.reindex_like(sal_df)

output will be:

```
In [112]: ndf2.reindex_like(sal_df)
Out[112]:
      2016     2017  2018     2019
Qtr1   NaN  44900.0   NaN  61000.0
Qtr2   NaN      NaN   NaN      NaN
Qtr3   NaN  57000.0   NaN      NaN
Qtr4   NaN  59000.0   NaN      NaN
```

See **ndf2** has same indexes and labels on both axes same as passed dataframe **sal_df**

ndf2 has retained columns 2017 and 2019 for the rows Qtr1, Qtr3 and Qtr4

It has added a ***new row Qtr2*** as per **sal_df** with NaN values and dropped row Qtn which is not in sal_df

It has added columns 2016 , 2018 as per **sal_df** with NaN values and dropped columns 2015, 2013, 2011 which are not in **sal_df**