Numpy

NumPy, which stands for Numerical Python, is a library consisting of multidimensional array objects and a collection of routines for processing those arrays. Using NumPy, mathematical and logical operations on arrays can be performed.

1. Arrays in NumPy: NumPy's main object is the homogeneous multidimensional array.

- It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers.
- In NumPy dimensions are called *axes*. The number of axes is *rank*.
- NumPy's array class is called **ndarray**. It is also known by the alias **array**.

Example :

```
[[ 1, 2, 3],
 [ 4, 2, 5]]
Here,
rank = 2 (as it is 2-dimensional or it has 2 axes)
first dimension(axis) length = 2, second dimension has length = 3
overall shape can be expressed as: (2, 3)
```

```
# Python program to demonstrate
# basic array characteristics
import numpy as np

# Creating array object
arr = np.array( [[ 1, 2, 3],
                 [ 4, 2, 5]] )

# Printing type of arr object
print("Array is of type: ", type(arr))

# Printing array dimensions (axes)
print("No. of dimensions: ", arr.ndim)

# Printing shape of array
print("Shape of array: ", arr.shape)

# Printing size (total number of elements) of array
print("Size of array: ", arr.size)

# Printing type of elements in array
print("Array stores elements of type: ", arr.dtype)
```

2. Array creation: There are various ways to create arrays in NumPy.

- For example, you can create an array from a regular Python **list** or **tuple** using the **array** function. The type of the resulting array is deduced from the type of the elements in the sequences.
- Often, the elements of an array are originally unknown, but its size is known. Hence, NumPy offers several functions to create arrays with **initial placeholder content**. These minimize the necessity of growing arrays, an expensive operation.
  For example: np.zeros, np.ones, np.full, np.empty, etc.
- To create sequences of numbers, NumPy provides a function analogous to range that returns arrays instead of lists.
- **arange:** returns evenly spaced values within a given interval. **step** size is specified.
- **linspace:** returns evenly spaced values within a given interval. **num** no. of elements are returned.
- **Reshaping array:** We can use **reshape** method to reshape an array. Consider an array with shape (a1, a2, a3, …, aN). We can reshape and convert it into another array with shape (b1, b2, b3, …, bM). The only required condition is:
  a1 x a2 x a3 … x aN = b1 x b2 x b3 … x bM . (i.e original size of array remains unchanged.)
- **Flatten array:** We can use **flatten** method to get a copy of array collapsed into **one dimension**. It accepts *order* argument. Default value is 'C' (for row-major order). Use 'F' for column major order.

Note: Type of array can be explicitly defined while creating array.

```
# Python program to demonstrate
# array creation techniques
import numpy as np

# Creating array from list with type float
a = np.array([[1, 2, 4], [5, 8, 7]], dtype = 'float')
print ("Array created using passed list:\n", a)

# Creating array from tuple
b = np.array((1 , 3, 2))
print ("\nArray created using passed tuple:\n", b)

# Creating a 3X4 array with all zeros
c = np.zeros((3, 4))
print ("\nAn array initialized with all zeros:\n", c)

# Create a constant value array of complex type
d = np.full((3, 3), 6, dtype = 'complex')
print ("\nAn array initialized with all 6s."
            "Array type is complex:\n", d)

# Create an array with random values
e = np.random.random((2, 2))
print ("\nA random array:\n", e)

# Create a sequence of integers
# from 0 to 30 with steps of 5
```

```
f = np.arange(0, 30, 5)
print ("\nA sequential array with steps of 5:\n", f)

# Create a sequence of 10 values in range 0 to 5
g = np.linspace(0, 5, 10)
print ("\nA sequential array with 10 values between"
                                        "0 and 5:\n", g)

# Reshaping 3X4 array to 2X2X3 array
arr = np.array([[1, 2, 3, 4],
                [5, 2, 4, 2],
                [1, 2, 0, 1]])

newarr = arr.reshape(2, 2, 3)

print ("\nOriginal array:\n", arr)
print ("Reshaped array:\n", newarr)

# Flatten array
arr = np.array([[1, 2, 3], [4, 5, 6]])
flarr = arr.flatten()

print ("\nOriginal array:\n", arr)
print ("Fattened array:\n", flarr)
```

3. Array Indexing: Knowing the basics of array indexing is important for analysing and manipulating the array object. NumPy offers many ways to do array indexing.

- **Slicing:** Just like lists in python, NumPy arrays can be sliced. As arrays can be multidimensional, you need to specify a slice for each dimension of the array.
- **Integer array indexing:** In this method, lists are passed for indexing for each dimension. One to one mapping of corresponding elements is done to construct a new arbitrary array.
- **Boolean array indexing:** This method is used when we want to pick elements from array which satisfy some condition.

```
# Python program to demonstrate
# indexing in numpy
import numpy as np

# An exemplar array
arr = np.array([[-1, 2, 0, 4],
                [4, -0.5, 6, 0],
                [2.6, 0, 7, 8],
                [3, -7, 4, 2.0]])

# Slicing array
temp = arr[:2, ::2]
print ("Array with first 2 rows and alternate"
                    "columns(0 and 2):\n", temp)

# Integer array indexing example
temp = arr[[0, 1, 2, 3], [3, 2, 1, 0]]
```

```
print ("\nElements at indices (0, 3), (1, 2), (2, 1),"
                                    "(3, 0):\n", temp)

# boolean array indexing example
cond = arr > 0 # cond is a boolean array
temp = arr[cond]
print ("\nElements greater than 0:\n", temp)
```

Basic operations: Plethora of built-in arithmetic functions are provided in NumPy.

Operations on single array: We can use overloaded arithmetic operators to do element-wise operation on array to create a new array. In case of +=, -=, *= operators, the exsisting array is modified.

```
# Python program to demonstrate
# basic operations on single array
import numpy as np

a = np.array([1, 2, 5, 3])

# add 1 to every element
print ("Adding 1 to every element:", a+1)

# subtract 3 from each element
print ("Subtracting 3 from each element:", a-3)

# multiply each element by 10
print ("Multiplying each element by 10:", a*10)

# square each element
print ("Squaring each element:", a**2)

# modify existing array
a *= 2
print ("Doubled each element of original array:", a)

# transpose of array
a = np.array([[1, 2, 3], [3, 4, 5], [9, 6, 0]])

print ("\nOriginal array:\n", a)
print ("Transpose of array:\n", a.T)
```

Unary operators: Many unary operations are provided as a method of **ndarray** class. This includes sum, min, max, etc. These functions can also be applied row-wise or column-wise by setting an axis parameter.

```
# unary operators in numpy
import numpy as np

arr = np.array([[1, 5, 6],
                [4, 7, 2],
                [3, 1, 9]])

# maximum element of array
print ("Largest element is:", arr.max())
```

```
print ("Row-wise maximum elements:",
                    arr.max(axis = 1))

# minimum element of array
print ("Column-wise minimum elements:",
                         arr.min(axis = 0))

# sum of array elements
print ("Sum of all array elements:",
                              arr.sum())

# cumulative sum along each row
print ("Cumulative sum along each row:\n",
                         arr.cumsum(axis = 1))
```

Binary operators: These operations apply on array elementwise and a new array is created. You can use all basic arithmetic operators like +, -, /, , *etc. In case of +=, -=,* = operators, the exsisting array is modified.

```
# Python program to demonstrate
# binary operators in Numpy
import numpy as np

a = np.array([[1, 2],
              [3, 4]])
b = np.array([[4, 3],
              [2, 1]])

# add arrays
print ("Array sum:\n", a + b)

# multiply arrays (elementwise multiplication)
print ("Array multiplication:\n", a*b)

# matrix multiplication
print ("Matrix multiplication:\n", a.dot(b))
```

- **Universal functions (ufunc):** NumPy provides familiar mathematical functions such as sin, cos, exp, etc. These functions also operate elementwise on an array, producing an array as output.

Note: All the operations we did above using overloaded operators can be done using ufuncs like np.add, np.subtract, np.multiply, np.divide, np.sum, etc.

```
# Python program to demonstrate
# universal functions in numpy
import numpy as np

# create an array of sine values
a = np.array([0, np.pi/2, np.pi])
print ("Sine values of array elements:", np.sin(a))
```

```
# exponential values
a = np.array([0, 1, 2, 3])
print ("Exponent of array elements:", np.exp(a))

# square root of array values
print ("Square root of array elements:", np.sqrt(a))
```

Sorting array: There is a simple **np.sort** method for sorting NumPy arrays. Let's explore it a bit.

```
# Python program to demonstrate sorting in numpy
import numpy as np

a = np.array([[1, 4, 2],
                [3, 4, 6],
              [0, -1, 5]])

# sorted array
print ("Array elements in sorted order:\n",
                    np.sort(a, axis = None))

# sort array row-wise
print ("Row-wise sorted array:\n",
                np.sort(a, axis = 1))

# specify sort algorithm
print ("Column wise sort by applying merge-sort:\n",
            np.sort(a, axis = 0, kind = 'mergesort'))

# Example to show sorting of structured array
# set alias names for dtypes
dtypes = [('name', 'S10'), ('grad_year', int), ('cgpa', float)]

# Values to be put in array
values = [('Hrithik', 2009, 8.5), ('Ajay', 2008, 8.7),
           ('Pankaj', 2008, 7.9), ('Aakash', 2009, 9.0)]

# Creating array
arr = np.array(values, dtype = dtypes)
print ("\nArray sorted by names:\n",
            np.sort(arr, order = 'name'))

print ("Array sorted by grauation year and then cgpa:\n",
                np.sort(arr, order = ['grad_year', 'cgpa']))
```

Pandas Data Structures

Pandas deal with the following three data structures

- Series
- Data Frame
- Panel

These data structures are built on top of Numpy array, which means they are fast.

Data Structure	Dimensions	Description
Series	1	1D labeled homogeneous array, size immutable.
Data Frames	2	General 2D labeled, size-mutable tabular structure with potentially heterogeneously typed columns.
Panel	3	General 3D labeled, size-mutable array.

a. Series: Series is a one-dimensional array like structure with homogeneous data. For example, the following series is a collection of integers 10, 23, 56, …

10	23	56	17	52	61	73	90	26	72

Key Points:

- Homogeneous data
- Size Immutable
- Values of Data Mutable

b. DataFrame: DataFrame is a two-dimensional array with heterogeneous data. For example,

Name	Age	Gender	Rating
Steve	32	Male	3.45
Lia	28	Female	4.6
Vin	45	Male	3.9
Katie	38	Female	2.78

The table represents the data of a sales team of an organization with their overall performance rating. The data is represented in rows and columns. Each column represents an attribute and each row represents a person.

Key Points:

- Heterogeneous data
- Size Mutable
- Data Mutable

c. **Panel:** Panel is a three-dimensional data structure with heterogeneous data. It is hard to represent the panel in graphical representation. But a panel can be illustrated as a container of Data Frame.

Key Points:

- Heterogeneous data
- Size Mutable
- Data Mutable

pandas.Series

Series is a one-dimensional labeled array capable of holding data of any type (integer, string, float, python objects, etc.) The axis labels are collectively called index.

- A pandas Series can be created using the following constructor −

```
pandas.Series(data, index, dtype, copy)
```

- The parameters of the constructor are as follows −

S.No	Parameter & Description
1	**Data -** data takes various forms like ndarray, list, constants and dicts
2	**Index -** Index values must be unique and hashable, same length as data. Default **np.arrange(n)** if no index is passed.
3	**Dtype -** dtype is for data type. If None, data type will be inferred
4	**Copy -** Copy data. Default False

A series can be created using various inputs like −

- Array
- Dict
- Scalar value or constant

a. Create an Empty Series: A basic series, which can be created is an Empty Series.

Example:

```
#import the pandas library and aliasing as pd
import pandas as pd
s = pd.Series()
print s
```

Its **output** is as follows −

```
Series([], dtype: float64)
```

b. Create a Series from ndarray

If data is an ndarray, then index passed must be of the same length. If no index is passed, then by default index will be **range(n)** where **n** is array length, i.e., [0,1,2,3…. **range(len(array))-1].**

Example 1

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
data = np.array(['a','b','c','d'])
s = pd.Series(data)
print s
```

Its **output** is as follows −

```
0  a
1  b
2  c
3  d
dtype: object
```

We did not pass any index, so by default, it assigned the indexes ranging from 0 to **len(data)-1**, i.e., 0 to 3.

Example 2

```
#import the pandas library and aliasing as pd

import pandas as pd

import numpy as np

data = np.array(['a','b','c','d'])

s = pd.Series(data,index=[100,101,102,103])

print s
```

Its **output** is as follows −

```
100  a
101  b
102  c
103  d
dtype: object
```

We passed the index values here. Now we can see the customized indexed values in the output.

c. Create a Series from dict

A **dict** can be passed as input and if no index is specified, then the dictionary keys are taken in a sorted order to construct index. If **index** is passed, the values in data corresponding to the labels in the index will be pulled out.

Example 1

```
#import the pandas library and aliasing as pd

import pandas as pd

import numpy as np

data = {'a' : 0., 'b' : 1., 'c' : 2.}

s = pd.Series(data)

print s
```

Its **output** is as follows −

```
a 0.0
b 1.0
c 2.0
dtype: float64
```

Observe − Dictionary keys are used to construct index.

Example 2

```
#import the pandas library and aliasing as pd

import pandas as pd

import numpy as np

data = {'a' : 0., 'b' : 1., 'c' : 2.}

s = pd.Series(data,index=['b','c','d','a'])

print s
```

Its **output** is as follows −

```
b 1.0
c 2.0
d NaN
a 0.0
dtype: float64
```

Observe − Index order is persisted and the missing element is filled with NaN (Not a Number).

d. Create a Series from Scalar

If data is a scalar value, an index must be provided. The value will be repeated to match the length of **index**

```
#import the pandas library and aliasing as pd

import pandas as pd

import numpy as np

s = pd.Series(5, index=[0, 1, 2, 3])

print s
```

Its **output** is as follows −

```
0  5
```

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

e. **Accessing Data from Series with Position:** Data in the series can be accessed similar to that in an **ndarray.**

Example 1

Retrieve the first element. As we already know, the counting starts from zero for the array, which means the first element is stored at zeroth position and so on.

```
import pandas as pd

s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve the first element

print s[0]
```

Its **output** is as follows −

```
1
```

Example 2

Retrieve the first three elements in the Series. If a : is inserted in front of it, all items from that index onwards will be extracted. If two parameters (with : between them) is used, items between the two indexes (not including the stop index)

```
import pandas as pd

s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])


#retrieve the first three element

print s[:3]
```

Its **output** is as follows −

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

Example 3

Retrieve the last three elements.

```
import pandas as pd

s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve the last three element

print s[-3:]
```

Its **output** is as follows −

```
c  3
d  4
e  5
dtype: int64
```

Retrieve Data Using Label (Index)

A Series is like a fixed-size **dict** in that you can get and set values by index label.

Example 1

Retrieve a single element using index label value.

```
import pandas as pd

s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve a single element

print s['a']
```

Its **output** is as follows −

```
1
```

Example 2

Retrieve multiple elements using a list of index label values.

```
import pandas as pd

s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve multiple elements
```

```
print s[['a','c','d']]
```

Its **output** is as follows −

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

Example 3

If a label is not contained, an exception is raised.

```
import pandas as pd

s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])


#retrieve multiple elements

print s['f']
```

Its **output** is as follows −

```
…
KeyError: 'f'
```

Pandas.DataFrames

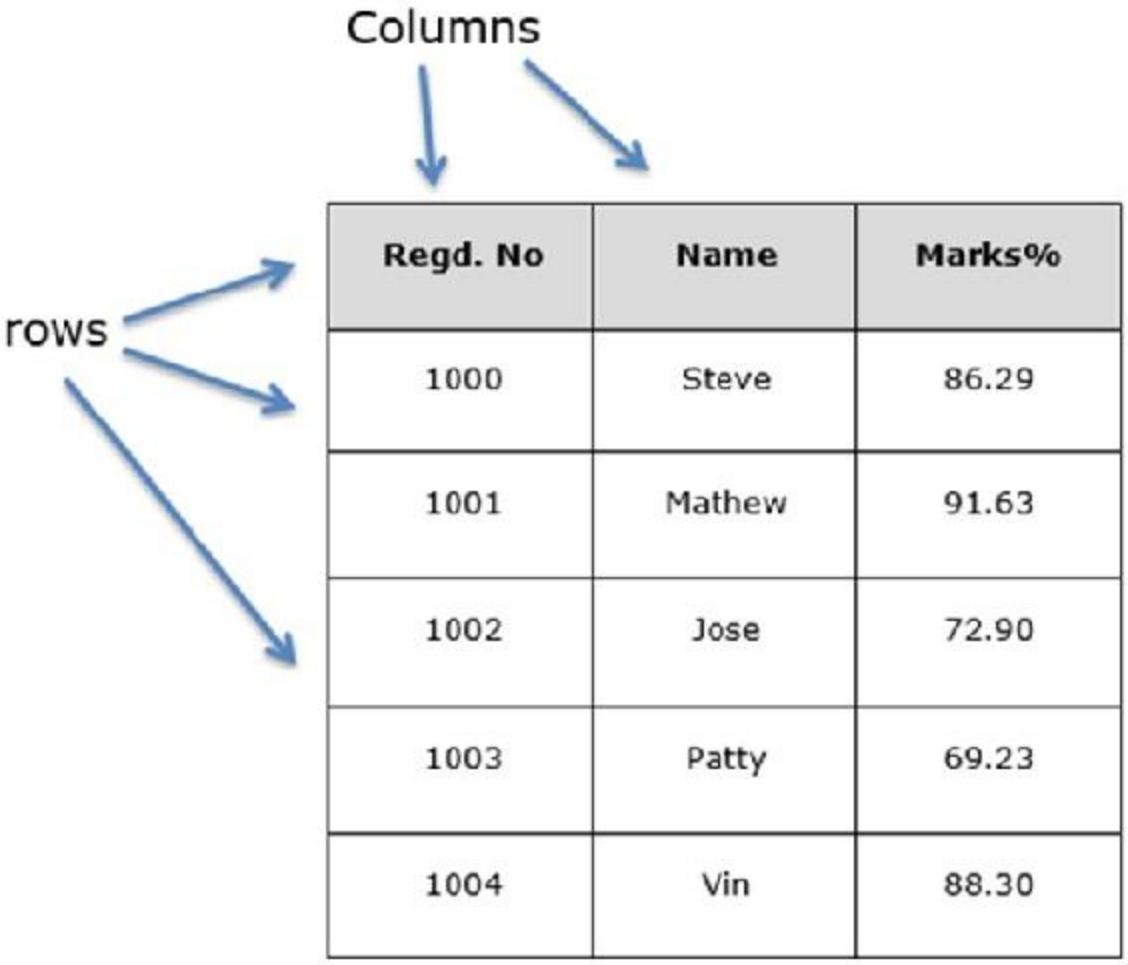
A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns.

Features of DataFrame

- Potentially columns are of different types
- Size – Mutable
- Labeled axes (rows and columns)
- Can Perform Arithmetic operations on rows and columns

Structure

Let us assume that we are creating a data frame with student's data.



You can think of it as an SQL table or a spreadsheet data representation.

pandas.DataFrame

A pandas DataFrame can be created using the following constructor –

```
pandas.DataFrame( data, index, columns, dtype, copy)
```

The parameters of the constructor are as follows –

S.No	Parameter & Description
1	**data** data takes various forms like ndarray, series, map, lists, dict, constants and also another DataFrame.

2	**index** For the row labels, the Index to be used for the resulting frame is Optional Default np.arrange(n) if no index is passed.
3	**columns** For column labels, the optional default syntax is - np.arrange(n). This is only true if no index is passed.
4	**dtype** Data type of each column.
4	**copy** This command (or whatever it is) is used for copying of data, if the default is False.

Create DataFrame

A pandas DataFrame can be created using various inputs like –

- Lists
- dict
- Series
- Numpy ndarrays
- Another DataFrame

In the subsequent sections of this chapter, we will see how to create a DataFrame using these inputs.

Create an Empty DataFrame

A basic DataFrame, which can be created is an Empty Dataframe.

Example

```
#import the pandas library and aliasing as pd

import pandas as pd
```

```
df = pd.DataFrame()

print df
```

Its **output** is as follows –

```
Empty DataFrame
Columns: []
Index: []
```

Create a DataFrame from Lists

The DataFrame can be created using a single list or a list of lists.

Example 1

```
import pandas as pd

data = [1,2,3,4,5]

df = pd.DataFrame(data)

print df
```

Its **output** is as follows –

```
   0
0  1
1  2
2  3
3  4
4  5
```

Example 2

```
import pandas as pd

data = [['Alex',10],['Bob',12],['Clarke',13]]

df = pd.DataFrame(data,columns=['Name','Age'])

print df
```

Its **output** is as follows –

```
      Name      Age
0     Alex      10
1     Bob       12
2     Clarke    13
```

Example 3

```
import pandas as pd

data = [['Alex',10],['Bob',12],['Clarke',13]]

df = pd.DataFrame(data,columns=['Name','Age'],dtype=float)

print df
```

Its **output** is as follows –

```
      Name     Age
0     Alex     10.0
1     Bob      12.0
2     Clarke   13.0
```

Note – Observe, the **dtype** parameter changes the type of Age column to floating point.

Create a DataFrame from Dict of ndarrays / Lists

All the **ndarrays** must be of same length. If index is passed, then the length of the index should equal to the length of the arrays.

If no index is passed, then by default, index will be range(n), where **n** is the array length.

Example 1

```
import pandas as pd

data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}

df = pd.DataFrame(data)

print df
```

Its **output** is as follows –

```
      Age      Name
0     28        Tom
1     34       Jack
2     29      Steve
3     42      Ricky
```

Note – Observe the values 0,1,2,3. They are the default index assigned to each using the function range(n).

Example 2

Let us now create an indexed DataFrame using arrays.

```
import pandas as pd

data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}

df = pd.DataFrame(data, index=['rank1','rank2','rank3','rank4'])

print df
```

Its **output** is as follows –

```
         Age    Name
rank1    28      Tom
rank2    34     Jack
rank3    29    Steve
rank4    42    Ricky
```

Note – Observe, the **index** parameter assigns an index to each row.

Create a DataFrame from List of Dicts

List of Dictionaries can be passed as input data to create a DataFrame. The dictionary keys are by default taken as column names.

Example 1

The following example shows how to create a DataFrame by passing a list of dictionaries.

```
import pandas as pd

data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]

df = pd.DataFrame(data)

print df
```

Its **output** is as follows –

```
   a   b     c
0  1   2   NaN
1  5   10  20.0
```

Note – Observe, NaN (Not a Number) is appended in missing areas.

Example 2

The following example shows how to create a DataFrame by passing a list of dictionaries and the row indices.

```
import pandas as pd

data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]

df = pd.DataFrame(data, index=['first', 'second'])

print df
```

Its **output** is as follows –

```
        a   b       c
first   1   2     NaN
second  5   10   20.0
```

Example 3

The following example shows how to create a DataFrame with a list of dictionaries, row indices, and column indices.

```
import pandas as pd

data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]

#With two column indices, values same as dictionary keys

df1 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b'])

#With two column indices with one index with other name

df2 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b1'])

print df1

print df2
```

Its **output** is as follows –

```
#df1 output
         a  b
first    1  2
second   5  10

#df2 output
         a  b1
first    1  NaN
second   5  NaN
```

Note – Observe, df2 DataFrame is created with a column index other than the dictionary key; thus, appended the NaN's in place. Whereas, df1 is created with column indices same as dictionary keys, so NaN's appended.

Create a DataFrame from Dict of Series

Dictionary of Series can be passed to form a DataFrame. The resultant index is the union of all the series indexes passed.

Example

```
import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),

      'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)

print df
```

Its **output** is as follows –

```
      one    two
a     1.0    1
b     2.0    2
c     3.0    3
d     NaN    4
```

Note – Observe, for the series one, there is no label **'d'** passed, but in the result, for the **d** label, NaN is appended with NaN.

Let us now understand **column selection, addition**, and **deletion** through examples.

Column Selection

We will understand this by selecting a column from the DataFrame.

Example

```
import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),

      'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)

print df ['one']
```

Its **output** is as follows –

```
a     1.0
b     2.0
c     3.0
```

```
d     NaN
Name: one, dtype: float64
```

Column Addition

We will understand this by adding a new column to an existing data frame.

Example

```
import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),

      'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)



# Adding a new column to an existing DataFrame object with column label by passing
new series

print ("Adding a new column by passing as Series:")

df['three']=pd.Series([10,20,30],index=['a','b','c'])

print df



print ("Adding a new column using the existing columns in DataFrame:")

df['four']=df['one']+df['three']

print df
```

Its **output** is as follows –

```
Adding a new column by passing as Series:
     one   two   three
a    1.0    1    10.0
b    2.0    2    20.0
c    3.0    3    30.0
d    NaN    4    NaN

Adding a new column using the existing columns in DataFrame:
      one   two   three    four
a     1.0    1    10.0     11.0
b     2.0    2    20.0     22.0
c     3.0    3    30.0     33.0
d     NaN    4     NaN     NaN
```

Column Deletion

Columns can be deleted or popped; let us take an example to understand how.

Example

```
# Using the previous DataFrame, we will delete a column

# using del function

import pandas as pd


d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),

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

     'three' : pd.Series([10,20,30], index=['a','b','c'])}


df = pd.DataFrame(d)

print ("Our dataframe is:")

print df


# using del function

print ("Deleting the first column using DEL function:")

del df['one']

print df


# using pop function

print ("Deleting another column using POP function:")

df.pop('two')

print df
```

Its **output** is as follows –

```
Our dataframe is:
      one   three  two
a     1.0    10.0   1
b     2.0    20.0   2
c     3.0    30.0   3
d     NaN     NaN   4

Deleting the first column using DEL function:
```

```
      three    two
a     10.0     1
b     20.0     2
c     30.0     3
d     NaN      4

Deleting another column using POP function:
   three
a  10.0
b  20.0
c  30.0
d  NaN
```

Row Selection, Addition, and Deletion

We will now understand row selection, addition and deletion through examples. Let us begin with the concept of selection.

Selection by Label

Rows can be selected by passing row label to a **loc** function.

```
import pandas as pd


d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),

     'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}


df = pd.DataFrame(d)

print df.loc['b']
```

Its **output** is as follows –

```
one 2.0
two 2.0
Name: b, dtype: float64
```

The result is a series with labels as column names of the DataFrame. And, the Name of the series is the label with which it is retrieved.

Selection by integer location

Rows can be selected by passing integer location to an **iloc** function.

```
import pandas as pd


d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
```

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


df = pd.DataFrame(d)

print df.iloc[2]
```

Its **output** is as follows –

```
one   3.0
two   3.0
Name: c, dtype: float64
```

Slice Rows

Multiple rows can be selected using ' : ' operator.

```
import pandas as pd


d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),

    'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}


df = pd.DataFrame(d)

print df[2:4]
```

Its **output** is as follows –

```
      one    two
c     3.0     3
d     NaN     4
```

Addition of Rows

Add new rows to a DataFrame using the **append** function. This function will append the rows at the end.

```
import pandas as pd


df = pd.DataFrame([[1, 2], [3, 4]], columns = ['a','b'])

df2 = pd.DataFrame([[5, 6], [7, 8]], columns = ['a','b'])
```

```
df = df.append(df2)

print df
```

Its **output** is as follows −

```
   a  b
0  1  2
1  3  4
0  5  6
1  7  8
```

Deletion of Rows

Use index label to delete or drop rows from a DataFrame. If label is duplicated, then multiple rows will be dropped.

If you observe, in the above example, the labels are duplicate. Let us drop a label and will see how many rows will get dropped.

```
import pandas as pd


df = pd.DataFrame([[1, 2], [3, 4]], columns = ['a','b'])

df2 = pd.DataFrame([[5, 6], [7, 8]], columns = ['a','b'])


df = df.append(df2)


# Drop rows with label 0

df = df.drop(0)


print df
```

Its **output** is as follows −

```
  a b
1 3 4
1 7 8
```

In the above example, two rows were dropped because those two contain the same label 0.

Pandas. A **panel** is a 3D container of data. The term **Panel data** is derived from econometrics and is partially responsible for the name pandas – **pan(el)-da(ta)**-s.

The names for the 3 axes are intended to give some semantic meaning to describing operations involving panel data. They are –

- **items** – axis 0, each item corresponds to a DataFrame contained inside.
- **major_axis** – axis 1, it is the index (rows) of each of the DataFrames.
- **minor_axis** – axis 2, it is the columns of each of the DataFrames.

pandas.Panel()

A Panel can be created using the following constructor –

```
pandas.Panel(data, items, major_axis, minor_axis, dtype, copy)
```

The parameters of the constructor are as follows –

Parameter	Description
data	Data takes various forms like ndarray, series, map, lists, dict, constants and also another DataFrame
items	axis=0
major_axis	axis=1
minor_axis	axis=2
dtype	Data type of each column
copy	Copy data. Default, **false**

Create Panel

A Panel can be created using multiple ways like –

- From ndarrays
- From dict of DataFrames

From 3D ndarray

```
# creating an empty panel

import pandas as pd

import numpy as np

data = np.random.rand(2,4,5)

p = pd.Panel(data)

print p
```

Its **output** is as follows –

```
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 4 (major_axis) x 5 (minor_axis)
Items axis: 0 to 1
Major_axis axis: 0 to 3
Minor_axis axis: 0 to 4
```

Note – Observe the dimensions of the empty panel and the above panel, all the objects are different.

From dict of DataFrame Objects

```
#creating an empty panel

import pandas as pd

import numpy as np



data = {'Item1' : pd.DataFrame(np.random.randn(4, 3)),

        'Item2' : pd.DataFrame(np.random.randn(4, 2))}

p = pd.Panel(data)

print p
```

Its **output** is as follows –

```
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 4 (major_axis) x 5 (minor_axis)
Items axis: 0 to 1
Major_axis axis: 0 to 3
```

```
Minor_axis axis: 0 to 4
```

Create an Empty Panel

An empty panel can be created using the Panel constructor as follows –

```
#creating an empty panel

import pandas as pd

p = pd.Panel()

print p
```

Its **output** is as follows –

```
<class 'pandas.core.panel.Panel'>
Dimensions: 0 (items) x 0 (major_axis) x 0 (minor_axis)
Items axis: None
Major_axis axis: None
Minor_axis axis: None
```

Selecting the Data from Panel

Select the data from the panel using –

- Items
- Major_axis
- Minor_axis

Using Items

```
# creating an empty panel

import pandas as pd

import numpy as np

data = {'Item1' : pd.DataFrame(np.random.randn(4, 3)),

        'Item2' : pd.DataFrame(np.random.randn(4, 2))}

p = pd.Panel(data)

print p['Item1']
```

Its **output** is as follows –

```
          0         1         2
0  0.488224 -0.128637  0.930817
1  0.417497  0.896681  0.576657
```

```
2   -2.775266   0.571668   0.290082
3   -0.400538  -0.144234   1.110535
```

We have two items, and we retrieved item1. The result is a DataFrame with 4 rows and 3 columns, which are the **Major_axis** and **Minor_axis** dimensions.

Using major_axis

Data can be accessed using the method **panel.major_axis(index)**.

```
# creating an empty panel

import pandas as pd

import numpy as np

data = {'Item1' : pd.DataFrame(np.random.randn(4, 3)),

        'Item2' : pd.DataFrame(np.random.randn(4, 2))}

p = pd.Panel(data)

print p.major_xs(1)
```

Its **output** is as follows –

```
      Item1       Item2
0   0.417497    0.748412
1   0.896681   -0.557322
2   0.576657       NaN
```

Using minor_axis

Data can be accessed using the method **panel.minor_axis(index).**

```
# creating an empty panel

import pandas as pd

import numpy as np

data = {'Item1' : pd.DataFrame(np.random.randn(4, 3)),

        'Item2' : pd.DataFrame(np.random.randn(4, 2))}

p = pd.Panel(data)

print p.minor_xs(1)
```

Its **output** is as follows –

```
       Item1       Item2
0   -0.128637   -1.047032
1    0.896681   -0.557322
```

```
2    0.571668    0.431953
3   -0.144234    1.302466
```

Note – Observe the changes in the dimensions.