

UNIT-I

NumPy I: THE ABSOLUTE BASICS FOR BEGINNERS

Installing NumPy, Import NumPy, Difference between Python lists and NumPy Array, what is an Array, How to Create Basic Arrays, Adding Removing and Sorting Elements, How do You Know the Size and Shape of an Array, Can you Reshape An Array, 1D to 2D Arrays (How do you add new Axis to an Array), Indexing and Slicing, How to Create an array with an Existing Data, Basic Array Operations, More Useful Array Operations

UNIT-II

NumPy II: Creating Matrices, Generating Random Numbers, How to get Unique Items and Counts, Transposing and Reshaping Matrices, Reverse an Array, Reshaping and Flattening Multi-Dimensional Arrays, How to Accessing the Doc-string for more information, Working with Mathematical Formulas, How to save and load NumPy Objects.

UNIT-III

Pandas: User Guide: Object Creation, Viewing Data, Selection, Missing Data, Operations, Merge, Grouping, Reshaping, Time Series, Categorical, Getting Data In/ Out

Introduction to Data Structures: Series, Data Frame.

UNIT-IV

Matplotlib: Features of Matplotlib, anatomy and customization of a Matplotlib Plot.

Plotting and Plot Customization: Creating a plot and figure, Axes, Subplots, Changing Figure sizes.

Customizing Plots: Plot Titles, Labels and Legends, Text, Ticks, Layouts. Changing Colour of Elements, Visualization Examples.

UNIT-V

Scikit-Learn: Introduction to Machine Learning with Scikit-Learn: Machine Learning: The Problem Setting, Loading an Example Datasets, Learning and Predicting, Model Persistence, Conventions.

A Tutorial on Statistical-Learning for Scientific Data Processing: Statistical Learning, Supervised Learning, Model Selection

UNIT-VI

Scipy: Basic Functions, Special Functions, Compressed Sparse Graph Routines, Spatial Data Structures and Algorithms, Statistics, Building Specific Distributions.

UNIT-I

NumPy-I: THE ABSOLUTE BASICS FOR BEGINNERS

Installing NumPy, Import NumPy, Difference between Python lists and NumPy Array, what is an Array, How to Create Basic Arrays, Adding Removing and Sorting Elements, How do You Know the Size and Shape of an Array, Can you Reshape An Array, 1D to 2D Arrays (How do you add new Axis to an Array), Indexing and Slicing, How to Create an array with an Existing Data, Basic Array Operations, More Useful Array Operations

1.1 NumPy-I: THE ABSOLUTE BASICS FOR BEGINNERS

Introduction

NumPy (**Numerical Python**) is an open source Python library that's used in almost every field of science and engineering. It's the universal standard for working with numerical data in Python, and it's at the core of the scientific Python and PyData ecosystems. NumPy users include everyone from beginning coders to experienced researchers doing state-of-the-art scientific and industrial research and development. The NumPy API is used extensively in Pandas, SciPy, Matplotlib, scikit-learn scikit-image and most other data science and scientific Python packages.

The NumPy library contains multidimensional array and matrix data structures (you'll find more information about this in later sections). It provides **ndarray**, a homogeneous n-dimensional array object, with methods to efficiently operate on it. NumPy can be used to perform a wide variety of mathematical operations on arrays. It adds powerful data structures to Python that guarantee efficient calculations with arrays and matrices and it supplies an enormous library of

high-level mathematical functions that operate on these arrays and matrices.

1.2 What is NumPy?

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

Travis Oliphant created NumPy package in 2005 by injecting the features of the ancestor module Numeric into another module Numarray.

It is an extension module of Python which is mostly written in C. It provides various functions which are capable of performing the numeric computations with a high speed. NumPy provides various powerful data structures, implementing multi-dimensional arrays and matrices. These data structures are used for the optimal computations regarding arrays and matrices.

There are several important differences between NumPy arrays and the standard Python sequences:

- NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an ndarray will create a new array and delete the original.
- The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.
- NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences.
- A growing plethora of scientific and mathematical Python-based packages are using NumPy arrays; though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing, and they often output NumPy arrays. In other words, in order to efficiently use much (perhaps even most) of today's scientific/mathematical Python-

based software, just knowing how to use Python's built-in sequence types is insufficient - one also needs to know how to use NumPy arrays.

1.3 Why is NumPy Fast?

Vectorization describes the absence of any explicit looping, indexing, etc., in the code - these things are taking place. NumPy is a Python fundamental package used for efficient manipulations and operations on High-level mathematical functions, Multi-dimensional arrays, Linear algebra, Fourier Transformations, Random Number Capabilities, etc. It provides tools for integrating C, C++, and Fortran code in Python. NumPy is mostly used in Python for scientific computing.

Let us look at the below program which compares NumPy Arrays and Lists in Python in terms of execution time.

```

# importing required packages
import numpy
import time
# size of arrays and lists
size = 1000000
# declaring lists
list1 = range(size)
list2 = range(size)
# declaring arrays
array1 = numpy.arange(size)
array2 = numpy.arange(size)
# list
initialTime = time.time()
resultantList = [(a * b) for a, b in zip(list1, list2)]
# calculating execution time
print("Time taken by Lists :", (time.time() - initialTime),"seconds")
# NumPy array
initialTime = time.time()
resultantArray = array1 * array2
# calculating execution time
print("Time taken by NumPy Arrays :", (time.time() - initialTime),"seconds")

```

Output

```

Time taken by Lists : 0.10578775405883789 seconds
Time taken by NumPy Arrays : 0.004675388336181641 seconds

```

From the output of the above program, we see that the NumPy Arrays execute very much faster than the Lists in Python. There is a big difference between the execution time of arrays and lists.

NumPy Arrays are faster than Python Lists because of the following reasons:

- An array is a collection of homogeneous data-types that are stored in contiguous memory locations. On the other hand, a list in Python is a collection of

heterogeneous data types stored in non-contiguous memory locations.

- The NumPy package breaks down a task into multiple fragments and then processes all the fragments parallelly.
- The NumPy package integrates C, C++, and Fortran codes in Python. These programming languages have very little execution time compared to Python.

1.4 Who uses NumPy?

The NumPy API is used extensively in Pandas, SciPy, Matplotlib, scikit-learn, scikit-image and most other data science and scientific Python packages. The NumPy library contains multidimensional array and matrix data structures

1.5 NumPy Environment Setup (Installing NumPy)

NumPy doesn't come bundled with Python. We have to install it using the python pip installer. Execute the following command.

```
$ pip install numpy
```

Windows

On the Windows operating system, The SciPy stack is provided by the Anaconda which is a free distribution of the Python SciPy package.

It can be downloaded from the official website: <https://www.anaconda.com/>. It is also available for Linux and Mac.

The Canopy also comes with the full SciPy stack which is available as free as well as commercial license. We can download it by visiting the link: <https://www.enthought.com/products/canopy/>.

Enthought Canopy is a comprehensive Python-based analysis environment for scientists, engineers and analysts. It provides easy installation of the core analytic and scientific Python packages for rapid data collection, manipulation, analysis and visualization, algorithm design, and application development.

The Python (x, y) is also available free with the full SciPy distribution. Download it by visiting the link: <https://python-xy.github.io/>

Installing Numpy on Windows:

For Conda Users:

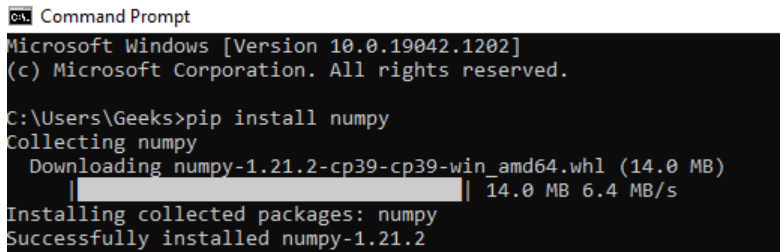
If you want the installation to be done through conda, you can use the below command:

```
conda install -c anaconda numpy
```

For PIP Users:

Users who prefer to use pip can use the below command to install NumPy:

```
$ pip install numpy
```



```
Command Prompt
Microsoft Windows [Version 10.0.19042.1202]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Geeks>pip install numpy
Collecting numpy
  Downloading numpy-1.21.2-cp39-cp39-win_amd64.whl (14.0 MB)
    |████████████████████████████████████████| 14.0 MB 6.4 MB/s
Installing collected packages: numpy
Successfully installed numpy-1.21.2
```

pip vs. conda

pip installs python packages in any environment.

conda installs any package in conda environments.

Linux

In Linux, the different package managers are used to install the SciPy stack. The package managers are specific to the different distributions of Linux. Let's look at each one of them.

Installing Numpy on Linux using Conda:

If you want the installation to be done through conda, you can use the below command:

```
conda install -c anaconda numpy
```

Type in “y” for yes when prompted.

Make sure you follow the best practices for installation using conda as:

- Use an environment for installation rather than in the base environment using the below command:

```
conda create -n my-env  
conda activate my-env
```

Verifying Numpy Installation on Linux using Conda:

Use the below command to verify if the above package has successfully installed:

```
conda list numpy
```

You will get a similar message as shown below if the installation has been successful:

```
(my-env) root@gfg-VirtualBox:~# conda list numpy
# packages in environment at /root/anaconda3/envs/my-env:
#
# Name                    Version           Build    Channel
numpy                     1.19.1           py37hbc911f0_0  anaconda
numpy-base               1.19.1           py37hfa32c7d_0  anaconda
(my-env) root@gfg-VirtualBox:~#
```

Installing Numpy on Linux using PIP:

Users who prefer to use pip can use the below command to install NumPy:

```
$ pip install numpy
```

You will get a similar message once the installation is complete:

```
(my-env) gfg@gfg-VirtualBox:~$ pip install numpy
Collecting numpy
  Downloading numpy-1.21.2-cp38-cp38-manylinux_2_12_x86_64.manylinux2010_x86_64.whl (15.8 MB)
    | 15.8 MB 209 kB/s
Installing collected packages: numpy
Successfully installed numpy-1.21.2
(my-env) gfg@gfg-VirtualBox:~$
```

Ubuntu

Execute the following command on the terminal.

```
$ sudo apt-get install python-numpy  
  
$ python-scipy python-matplotlibpythonipythonnotebook python-pandas  
  
$ python-sympy python-nose
```

Redhat

On Redhat, the following commands are executed to install the Python SciPy package stack.

```
$ sudo yum install numpyscipy python-matplotlibpython  
  
$ python-pandas sympy python-nose atlas-devel
```

To verify the installation, open the Python prompt by executing python command on the terminal (cmd in the case of windows) and try to import the module NumPy as shown in the below image. If it doesn't give the error, then it is installed successfully.

1.6 Difference between Python lists and NumPy Array

Numpy Array	Python List
It is the core Library of python which is used for scientific computing.	The core library of python provides list.
It can contain similar data types.	It Contains different types of datatypes.
We need to Numpy Library to access Numpy Arrays.	It is built-in function of python.
It is Homogeneous.	It is both homogeneous and heterogeneous.
In this Element wise operation is possible.	Element wise operation is not possible on the list.
By using <code>numpy.array()</code> we can create N-Dimensional array.	It is by default 1-dimensional. In some cases, we can create an N-Dimensional list. But it is a long process.
It requires smaller memory consumption as compared to Python List.	It requires more memory as compared to Numpy Array.

In this each item is stored in a sequential manner.	It stores item in random location of the memory.
It is faster as compared to list.	It is slow as compared to NumPy Array.
It also have some optimism function	It does not have some optimism function.

1.7 What is an Array?

An array is a central data structure of the NumPy library. An array is a grid of values and it contains information about the raw data, how to locate an element, and how to interpret an element. It has a grid of elements that can be indexed in [various ways](#). The elements are all of the same type.

Advantages of using an Array

- Arrays can handle very large datasets efficiently
- Computationally-memory efficient
- Faster calculations and analysis than lists
- Diverse functionality (many functions in Python packages). With several Python packages that make trend modeling, statistics, and visualization easier.

1.8 How to Create Basic Arrays

In Python, you can create new data types, called arrays using the NumPy package. NumPy arrays are optimized for numerical analyses and contain only a single data type.

You first import NumPy and then use the `array()` function to create an array. The `array()` function takes a list as an input.

```
import numpy
my_array = numpy.array([0, 1, 2, 3, 4])
print(my_array)
```

Output

```
[0, 1, 2, 3, 4]
```

The type of `my_array` is a `numpy.ndarray`.

```
print(type(my_array))
```

Output

```
<class 'numpy.ndarray'>
```

Array Examples

a) Example of creating an Array

NumPy is used to work with arrays. The array object in NumPy is called `ndarray`. We can create a NumPy `ndarray` object by using the `array()` function.

```
import numpy as np

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

print(arr)

print(type(arr))
```

Output

```
[1 2 3 4 5]
<class 'numpy.ndarray'>
```

To create an ndarray, we can pass a list, tuple or any array-like object into the array() method, and it will be converted into an ndarray:

```
import numpy as np

arr = np.array((1, 2, 3, 4, 5))

print(arr)
```

Output

```
[1 2 3 4 5]
```

```
import numpy as np

a_list = [1, 2, 3, 4]
a_list
```

Output

```
[1, 2, 3, 4]
```

```
an_array = np.array(a_list)
an_array
```

Output

```
array([1, 2, 3, 4])
```

1.9 Dimensions in Arrays

A dimension in arrays is one level of array depth.

0-D Arrays

0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

```
import numpy as np

arr = np.array(42)

print(arr)
```

Output

```
42
```

1-D Arrays

An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array. These are the most common and basic arrays.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
```

Output

```
[1 2 3 4 5]
```

2-D Arrays

An array that has 1-D arrays as its elements is called a 2-D array. These are often used to represent matrix or 2nd order tensors.

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr)
```

Output

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

3-D arrays

An array that has 2-D arrays (matrices) as its elements is called 3-D array. These are often used to represent a 3rd order tensor.

```
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
print(arr)
```

Output

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

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

Check Number of Dimensions?

NumPy Arrays provides the `ndim` attribute that returns an integer that tells us how many dimensions the array have.

```
import numpy as np
a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

Output

```
0
1
2
3
```

Higher Dimensional Arrays

An array can have any number of dimensions. When the array is created, you can define the number of dimensions by using the `ndmin` argument.

```
import numpy as np
arr = np.array([1, 2, 3, 4], ndmin=5)
print(arr)
print('number of dimensions :', arr.ndim)
```

Output

```
[[[[[1 2 3 4]]]]]
number of dimensions : 5
```

1.10 NumPy Array Indexing

Access Array Elements

Array indexing is the same as accessing an array element. You can access an array element by referring to its index number. The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr[0])
```

Output

```
1
```

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

Output

2

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

Output

7

Example of Array indexing

You can select a specific index element of an array using indexing notation.

```
import numpy as np

months_array = np.array(['Jan', 'Feb', 'March', 'Apr', 'May'])
print(months_array[3])
```

Output

Apr

You can also slice a range of elements using the slicing notation specifying a range of indices.

```
print(months_array[2:5])
```

Output

['March', 'Apr', 'May']

Access 2-D Arrays

To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element. Think of 2-D arrays like a table with rows and columns, where the dimension represents the row and the index represents the column.

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('2nd element on 1st row: ', arr[0, 1])
```

Output

```
2nd element on 1st row: 2
```

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('5th element on 2nd row: ', arr[1, 4])
```

Output

```
5th element on 2nd row: 10
```

Access 3-D Arrays

To access elements from 3-D arrays we can use comma separated integers representing the dimensions and the index of the element.

```
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(arr[0, 1, 2])
```


Output

6

Example Explained

`arr[0, 1, 2]` prints the value 6.

And this is why:

The first number represents the first dimension, which contains two arrays:

```
[[1, 2, 3], [4, 5, 6]]
```

and:

```
[[7, 8, 9], [10, 11, 12]]
```

Since we selected 0, we are left with the first array:

```
[[1, 2, 3], [4, 5, 6]]
```

The second number represents the second dimension, which also contains two arrays:

```
[1, 2, 3]
```

and:

```
[4, 5, 6]
```

Since we selected 1, we are left with the second array:

```
[4, 5, 6]
```

The third number represents the third dimension, which contains three values:

```
4
```

```
5
```

```
6
```

Since we selected 2, we end up with the third value:

6

Negative Indexing

Use negative indexing to access an array from the end.

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('Last element from 2nd dim: ', arr[1, -1])
```

Output

```
Last element from 2nd dim: 10
```

we can access the last item in a NumPy array by using the index of -1, as shown below:

```
# Accessing the Last Item in a NumPy Array
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr[-1])
```

Output

```
5
```

```
# Using Negative Indexing with Two-Dimensions
import numpy as np
arr = np.array([
    [1, 2, 3],
    [4, 5, 6]
])
print(arr[0, -1])
```

Output

3

1.11 NumPy Array Slicing

Slicing in python means taking elements from one given index to another given index. We pass slice instead of index like this: [start:end]. We can also define the step, like this: [start:end:step].

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5])
```

Output

[2 3 4 5]

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[4:])
```

Output

[5 6 7]

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[:4])
```

Output

[1 2 3 4]

Negative Slicing

Use the minus operator to refer to an index from the end:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[-3:-1])
```

Output

```
[5 6]
```

STEP

Use the step value to determine the step of the slicing:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5:2])
```

Output

```
[2 4]
```

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[:,2])
```

Output

```
[1 3 5 7]
```

Slicing 2-D Arrays

```
import numpy as np
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arr[1, 1:4])
```

Output

```
[7 8 9]
```

From both elements, return index 2:

```
import numpy as np
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arr[0:2, 2])
```

Output

```
[3 8]
```

```
import numpy as np
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arr[0:2, 1:4])
```

Output

```
[[2 3 4]
 [7 8 9]]
```

1.12 Adding an Element

Append/ Add an element to Numpy Array in Python (3 Ways).

- Use `append()` to add an element to Numpy Array.
- Use `concatenate()` to add an element to Numpy Array.
- Use `insert()` to add an element to Numpy Array.

a) Add element to Numpy Array using append()

Numpy module in python, provides a function to `numpy.append()` to add an element in a numpy array. We can pass the numpy array and a single value as arguments to the `append()` function. It doesn't modify the existing array, but returns a copy of the passed array with given value added to it.

```
import numpy as np
# Create a Numpy Array of integers
arr = np.array([11, 2, 6, 7, 2])
# Add / Append an element at the end of a numpy array
new_arr = np.append(arr, 10)
print('New Array: ', new_arr)
print('Original Array: ', arr)
```

Output

```
New Array: [11  2  6  7  2 10]
Original Array: [11  2  6  7  2]
```

The `append()` function created a copy of the array, then added the value 10 at the end of it and finally returned it.

b) Add element to Numpy Array using concatenate()

Numpy module in python, provides a function `numpy.concatenate()` to join two or more arrays. We can use that to add a single element in a numpy array. But for that we need to encapsulate the single value in a sequence data structure like list and pass a tuple of array & list to the `concatenate()` function.

```
import numpy as np
# Create a Numpy Array of integers
arr = np.array([11, 2, 6, 7, 2])
# Add / Append an element at the end of a numpy array
new_arr = np.concatenate( (arr, [10] ) )
print('New Array: ', new_arr)
print('Original Array: ', arr)
```

Output

```
New Array: [11  2  6  7  2 10]
Original Array: [11  2  6  7  2]
```

It returned a new array containing values from both sequences i.e. array and list. It didn't modified the original array, but returned a new array containing all values from original numpy array and a single value added along with them in the end.

c) Add element to Numpy Array using insert()

Using `numpy.insert()` function in the NumPy module, we can also insert an element at the end of a numpy array

Syntax :

```
numpy.insert(array, object, values, axis = None)
```

```
import numpy as np
# Create a Numpy Array of integers
arr = np.array([11, 2, 6, 7, 2])
# Add / Append an element at the end of a numpy array
new_arr = np.insert(arr,1,10)
print('New Array: ', new_arr)
print('Original Array: ', arr)
```

Output

```
New Array: [11 10  2  6  7  2]
Original Array: [11  2  6  7  2]
```

1.13 Removing an Element

we will discuss how to remove elements from a 1D or 2D Numpy Array by index position using `numpy.delete()`. Then we will also see how to remove rows and columns from 2D numpy array.

Syntax: `numpy.delete(arr, obj, axis=None)`

Arguments:

arr : Numpy array from which elements needs to be deleted.

obj : Index position or list of index positions of items to be deleted from numpy array `arr`.

axis : Axis along which we want to delete.

If 1 then delete columns.

If 0 then delete rows.

If None then flatten the array and then apply delete on it.

a) Delete an element in 1D Numpy Array by Index position

Suppose we have a numpy array of numbers i.e.

```
# Create a Numpy array from list of numbers
arr = np.array([4,5,6,7,8,9,10,11])
# Delete element at index position 2
arr = np.delete(arr, 2)
print('Modified Numpy Array by deleting element at index position 2')
print(arr)
```

Output

```
Modified Numpy Array by deleting element at index position 2
[ 4  5  7  8  9 10 11]
```

b) Delete multiple elements in 1D Numpy Array by Index position

To delete multiple elements from a numpy array by index positions, pass the numpy array and list of index positions to be deleted to np.delete() i.e.

```
# Create a Numpy array from list of numbers
arr = np.array([4, 5, 6, 7, 8, 9, 10, 11])
# Delete element at index positions 1,2 and 3
arr = np.delete(arr, [1,2,3])
print('Modified Numpy Array by deleting element at index position 1, 2 & 3')
print(arr)
```

Output

```
Modified Numpy Array by deleting element at index position 1, 2 & 3
[ 4  8  9 10 11]
```

It deleted the elements at index position 1,2 and 3 from the numpy array. It returned a copy of the passed array by deleting multiple element at given indices. Then we assigned the new array back to the same reference variable and it gave an effect that we deleted the elements from numpy array in place.

c) Remove a Specific NumPy Array Element by Value in a 1D array

Removing 8 values from an array.

```
import numpy as np
arr_1D = np.array([1 ,2, 3, 4, 5, 6, 7, 8])
arr_1D = np.delete(arr_1D, np.where(arr_1D == 8))
print(arr_1D)
```

Output

```
[1 2 3 4 5 6 7]
```

d) Delete rows & columns from a 2D Numpy Array

Suppose we have a 2D numpy array i.e.

```
# Create a 2D numpy array from list of list
arr2D = np.array([[11 ,12, 13, 11],
                  [21, 22, 23, 24],
                  [31, 32, 33, 34]])
print(arr2D)
```

Delete a column in 2D Numpy Array by column number

To delete a column from a 2D numpy array using np.delete() we need to pass the axis=1 along with numpy array and index of column i.e.

```

# Create a 2D numpy array from list of list
arr2D = np.array([[11 ,12, 13, 11],
                  [21, 22, 23, 24],
                  [31, 32, 33, 34]])

print(arr2D)
# Delete column at index 1
arr2D = np.delete(arr2D, 1, axis=1)
print('Modified 2D Numpy Array by removing columns at index 1')
print(arr2D)

```

Output

```

[[11 12 13 11]
 [21 22 23 24]
 [31 32 33 34]]
Modified 2D Numpy Array by removing columns at index 1
[[11 13 11]
 [21 23 24]
 [31 33 34]]

```

Delete multiple columns in 2D Numpy Array by column number

Pass axis=1 and list of column numbers to be deleted along with numpy array to np.delete() i.e.

```

# Create a 2D numpy array from list of list
arr2D = np.array([[11 ,12, 13, 11],
                  [21, 22, 23, 24],
                  [31, 32, 33, 34]])
# Delete column at index 2 & 3
arr2D = np.delete(arr2D, [2,3], axis=1)
print('Modified 2D Numpy Array by removing columns at index 2 & 3')
print(arr2D)

```

Output

```
Modified 2D Numpy Array by removing columns at index 2 & 3
[[11 12]
 [21 22]
 [31 32]]
```

Delete a row in 2D Numpy Array by row number

To delete a row from a 2D numpy array using `np.delete()` we need to pass the `axis=0` along with numpy array and index of row i.e. row number

```
# Create a 2D numpy array from list of list
arr2D = np.array([[11 ,12, 13, 11],
                  [21, 22, 23, 24],
                  [31, 32, 33, 34]])
# Delete row at index 0 i.e. first row
arr2D = np.delete(arr2D, 0, axis=0)
print('Modified 2D Numpy Array by removing rows at index 0')
print(arr2D)
```

Output

```
Modified 2D Numpy Array by removing rows at index 0
[[21 22 23 24]
 [31 32 33 34]]
```

Delete multiple rows in 2D Numpy Array by row number

```
# Create a 2D numpy array from list of list
arr2D = np.array([[11 ,12, 13, 11],
                  [21, 22, 23, 24],
                  [31, 32, 33, 34]])
# Delete row at index 0 i.e. first row
arr2D = np.delete(arr2D, [0,1], axis=0)
print('Modified 2D Numpy Array by removing rows at index 0 & 1')
print(arr2D)
```

Output

```
Modified 2D Numpy Array by removing rows at index 0 & 1
[[31 32 33 34]]
```

Delete specific elements in 2D Numpy Array by index position

When we don't pass axis argument to `np.delete()` then it's default value is `None`, which means 2D numpy array will be flattened for deleting elements at given index position. Let's use `np.delete()` to delete element at row number 0 and column 2 from our 2D numpy array

```
# Create a 2D numpy array from list of list
arr2D = np.array([[11 ,12, 13, 11],
                  [21, 22, 23, 24],
                  [31, 32, 33, 34]])
# Delete element in row 0 and column 2 from 2D numpy array
modArr = np.delete(arr2D, 2)
print('Modified 2D Numpy Array by removing element at row 0 & column 2')
print(modArr)
```

Output

```
Modified 2D Numpy Array by removing element at row 0 & column 2
[11 12 11 21 22 23 24 31 32 33 34]
```

1.14 Sorting an Array

Sorting means putting elements in an ordered sequence. Ordered sequence is any sequence that has an order corresponding to elements, like numeric or alphabetical, ascending or descending.

The NumPy ndarray object has a function called `sort()`, that will sort a specified array.

```
import numpy as np
arr = np.array([3, 2, 0, 1])
print(np.sort(arr))
```

Output

```
[0 1 2 3]
```

You can also sort arrays of strings, or any other data type:

```
import numpy as np
arr = np.array(['banana', 'cherry', 'apple'])
print(np.sort(arr))
```

Output

```
['apple' 'banana' 'cherry']
```

Sort a boolean array:

```
import numpy as np
arr = np.array([True, False, True])
print(np.sort(arr))
```

Output

```
[False True True]
```

Sorting a 2-D Array

If you use the `sort()` method on a 2-D array, both arrays will be sorted:

```
import numpy as np
arr = np.array([[3, 2, 4], [5, 0, 1]])
print(np.sort(arr))
```

Output

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

1.15 How do You Know the Size and Shape of an Array

Create a NumPy Array

Import NumPy and create a NumPy array.

```
import numpy as np
arr = np.array([ [1, 2, 3], [4, 5, 6], [7,8,9], [10,11,12] ])
print(arr)
```

Output

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

a) Size of an Array

NumPy array has an attribute called size that tells you the total number of elements in the array. You can check the size of a NumPy array using the ‘.size’ attribute. This will return the total number of elements present in the array:

```
import numpy as np
arr = np.array([ [1, 2, 3], [4, 5, 6], [7,8,9], [10,11,12] ])
print(arr.size)
```

Output

12

b) Data Types of Array

You can check the type of data present in the NumPy array using the `‘.dtype’` attribute. The data type of array `‘arr’` is `‘int’` (integer). Here, `‘32’` is related to memory allocation.

```
import numpy as np
arr = np.array([ [1, 2, 3], [4, 5, 6], [7,8,9], [10,11,12] ])
print(arr.dtype)
```

Output

int64

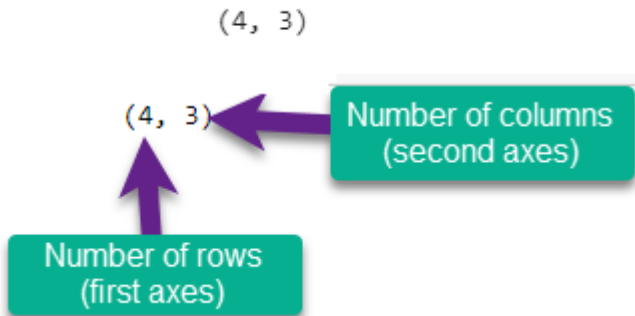
Note: If you have a 64-bit computer, dtype might be displayed as `int64`, and if you have 32-bit, it might be displayed as `int32`

c) Shape of an Array

An array has an attribute called `shape` that tells you the number of items along each axis. If we have an array with two axes, the `shape` attribute will tell you the total number of rows and columns in that array. The `shape` attribute of NumPy returns a tuple of integers that represents the number of items along each axis. Consider the given array:

```
import numpy as np
arr = np.array([ [1, 2, 3], [4, 5, 6], [7,8,9], [10,11,12] ])
print(arr.shape)
```


Output



You can check the shape of a NumPy array using the method `‘.shape’`. In array `‘arr’`, there are 4 rows and 3 columns.

d) Dimension of an Array

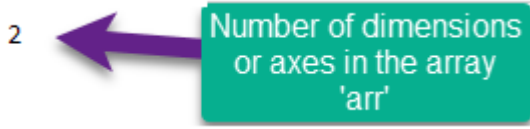
Consider the given array:

```
array([[ 5,  6,  7],
       [ 8,  9, 10],
       [11, 12, 13],
       [ 1,  2,  3]])
```

An array has an attribute called dimension that tells you the number of axes in the array. You can check the number of dimensions of a NumPy array using the `‘.ndim’` attribute. The array `‘arr’` is 2 dimensional (i.e., the array has two axes).

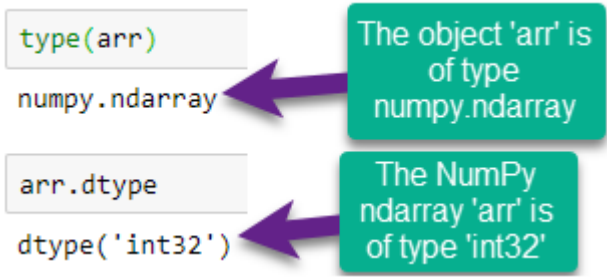
```
import numpy as np
arr = np.array([ [1, 2, 3], [4, 5, 6], [7,8,9], [10,11,12] ])
print(arr.ndim)
```

Output



e) `type()` function and `dtype` method

Everything in Python is an object. Suppose you have an object. If you want to know what type of object it is, you will use the `type()` function to learn about it. Now you know the object is a NumPy ndarray. If you want to know the type of data present in that array, you will use the attribute `dtype`. This is also called the type of NumPy array.



1.16 NumPy Array Reshaping

Reshaping arrays

Reshaping means changing the shape of an array. The shape of an array is the number of elements in each dimension. By reshaping we can add or remove dimensions or change number of elements in each dimension.

Reshape From 1-D to 2-D

Example

Convert the following 1-D array with 12 elements into a 2-D array. The outermost dimension will have 4 arrays, each with 3 elements:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newarr = arr.reshape(4, 3)
print(newarr)
```

Output

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

Reshape From 1-D to 3-D

Example

Convert the following 1-D array with 12 elements into a 3-D array. The outermost dimension will have 2 arrays that contain 3 arrays, each with 2 elements:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newarr = arr.reshape(2, 3, 2)
print(newarr)
```

Output

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

 [[ 7  8]
  [ 9 10]
 [11 12]]]
```

Can We Reshape Into any Shape?

Yes, as long as the elements required for reshaping are equal in both shapes.

Example

Try converting 1D array with 8 elements to a 2D array with 3 elements in each dimension (will raise an error):

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
newarr = arr.reshape(3, 3)
print(newarr)
```

Output

```
ValueError                                Traceback (most recent call last)
<ipython-input-54-79494d80387a> in <cell line: 5>()
      3 arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
      4
----> 5 newarr = arr.reshape(3, 3)
      6
      7 print(newarr)

ValueError: cannot reshape array of size 8 into shape (3,3)
```

Unknown Dimension

You are allowed to have one "unknown" dimension. Meaning that you do not have to specify an exact number for one of the dimensions in the reshape method. Pass -1 as the value, and NumPy will calculate this number for you.

Example

Convert 1D array with 8 elements to 3D array with 2x2 elements:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
newarr = arr.reshape(2, 2, -1)
print(newarr)
```

Output

```
[[[1 2]
   [3 4]]

 [[5 6]
   [7 8]]]
```

Flattening the arrays

Flattening array means converting a multidimensional array into a 1D array. We can use reshape(-1) to do this.

Example

Convert the array into a 1D array:

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6],[7,8,9]])
newarr = arr.reshape(-1)
print(newarr)
```

Output

```
[1 2 3 4 5 6 7 8 9]
```

1.17 Insert a new axis within a NumPy array

NumPy provides us with two different built-in functions to increase the dimension of an array i.e.,

1D array will become 2D array

2D array will become 3D array

3D array will become 4D array

4D array will become 5D array

Method 1: Using `numpy.newaxis()`

`newaxis` is also called as a pseudo-index that allows the temporary addition of an axis into a multiarray. `np.newaxis` uses the slicing operator to recreate the array while `numpy`.

```
>>> import numpy as np
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a.shape
(10,)
```

Output

```
(10, 1)
```

In the first example I use all elements from the first dimension and add a second dimension:

```
>>> a[:, np.newaxis]
array([[0],
       [1],
       [2],
       [3],
       [4],
       [5],
       [6],
       [7],
       [8],
       [9]])
>>> a[:, np.newaxis].shape
(10, 1)
```

Similarly you can use multiple `np.newaxis` to add multiple dimensions:

```
>>> a[np.newaxis, :, np.newaxis] # note the 3 [] pairs in the output
array([[[[0],
         [1],
         [2],
         [3],
         [4],
         [5],
         [6],
         [7],
         [8],
         [9]]]])
>>> a[np.newaxis, :, np.newaxis].shape
(1, 10, 1)
```

Method 2: Using `numpy.expand_dims()`

The `expand_dims()` function in NumPy is used to expand the shape of an input array that is passed to it.

Syntax

```
numpy.expand_dims(a, axis)
```

Parameter values

The `expand_dims()` function takes the following values:

a: This is the input array.

axis: This is the position in the resulting array where the new axis is to be positioned.

Example:

```
import numpy as np
# creating an input array
a = np.array([1, 2, 3, 4])

# getting the dimension of a
print(a.shape)

# expanding the axis of a
b = np.expand_dims(a, axis=1)

# getting the dimension of the new array
print(b.shape)
```


Output

```
(4,)  
(4, 1)
```

1.18 NumPy array from existing data(How to Create an array with an Existing Data)

NumPy provides us the way to create an array by using the existing data.

numpy.asarray

This routine is used to create an array by using the existing data in the form of lists, or tuples. This routine is useful in the scenario where we need to convert a python sequence into the numpy array object. The syntax to use the `asarray()` routine is given below.

```
numpy.asarray(sequence, dtype = None, order = None)
```

It accepts the following parameters.

1. **sequence:** It is the python sequence which is to be converted into the python array.
2. **dtype:** It is the data type of each item of the array.
3. **order:** It can be set to C or F. The default is C.

Example: Creating numpy array using the list

```
import numpy as np
l=[1,2,3,4,5,6,7]
a = np.asarray(l);
print(type(a))
print(a)
```

Output:

```
<class 'numpy.ndarray'>
[1 2 3 4 5 6 7]
```

Example: Creating a numpy array using Tuple

```
import numpy as np
l=(1,2,3,4,5,6,7)
a = np.asarray(l);
print(type(a))
print(a)
```

Output

```
<class 'numpy.ndarray'>
[1 2 3 4 5 6 7]
```

Example: creating a numpy array using more than one list

```
import numpy as np
l=[[1,2,3,4,5,6,7],[8,9]]
a = np.asarray(l);
print(type(a))
print(a)
```

Output

```
<class 'numpy.ndarray'>
[list([1, 2, 3, 4, 5, 6, 7]) list([8, 9])]
```

Numpy Arrays within the numerical range

Numpy.arrange

It creates an array by using the evenly spaced values over the given interval. The syntax to use the function is given below.

```
numpy.arange(start, stop, step, dtype)
```

It accepts the following parameters.

1. **start:** The starting of an interval. The default is 0.
2. **stop:** represents the value at which the interval ends excluding this value.
3. **step:** The number by which the interval values change.
4. **dtype:** the data type of the numpy array items.

Example

```
import numpy as np
arr = np.arange(0,10,2,float)
print(arr)
```

Output

```
[0.  2.  4.  6.  8.]
```

Example

```
import numpy as np
arr = np.arange(10,100,5,int)
print("The array over the given range is ",arr)
```

Output

```
The array over the given range is [10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85]
```

numpy.linspace

It is similar to the arrange function. However, it doesn't allow us to specify the step size in the syntax. Instead of that, it only returns evenly separated values over a specified period. The system implicitly calculates the step size. The syntax is given below.

```
numpy.linspace(start, stop, num, endpoint, retstep, dtype)
```

It accepts the following parameters.

- **start:** It represents the starting value of the interval.
- **stop:** It represents the stopping value of the interval.
- **num:** The amount of evenly spaced samples over the interval to be generated. The default is 50.
- **endpoint:** Its true value indicates that the stopping value is included in the interval.
- **rettstep:** This has to be a boolean value. Represents the steps and samples between the consecutive numbers.
- **dtype:** It represents the data type of the array items.

Example

```
import numpy as np
np.linspace(2.0, 3.0, num=5)
```

Output

```
array([2. , 2.25, 2.5 , 2.75, 3.  ])
```

Example

```
import numpy as np
arr = np.linspace(10, 20, 5, endpoint = False)
print("The array over the given range is ",arr)
```

Output

```
The array over the given range is [10. 12. 14. 16. 18.]
```

numpy.logspace

It creates an array by using the numbers that are evenly separated on a log scale. The syntax is given below.

```
numpy.logspace(start, stop, num, endpoint, base, dtype)
```

It accepts the following parameters.

1. **start:** It represents the starting value of the interval in the base.
2. **stop:** It represents the stopping value of the interval in the base.
3. **num:** The number of values between the range.
4. **endpoint:** It is a boolean type value. It makes the value represented by stop as the last value of the interval.
5. **base:** It represents the base of the log space.
6. **dtype:** It represents the data type of the array items.

Example

```
import numpy as np
arr = np.logspace(10, 20, num = 5, endpoint = True)
print("The array over the given range is ",arr)
```

Output

```
The array over the given range is [1.00000000e+10 3.16227766e+12 1.00000000e+15 3.1
1.00000000e+20]
```

1.19 Basic Array operation

There is a vast range of built-in operations that we can perform on these arrays.

1. **ndim** – It returns the dimensions of the array.
2. **itemsize** – It calculates the byte size of each element.
3. **dtype** – It can determine the data type of the element.
4. **reshape** – It provides a new view.
5. **slicing** – It extracts a particular set of elements.
6. **linspace** – Returns evenly spaced elements.
7. **max/min , sum, sqrt**
8. **ravel** – It converts the array into a single line.

There are also a few Special Operations like sine, cosine, tan, log, etc.

1. ndim

```
import numpy as np
a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

Output

```
0
1
2
3
```

2) itemsize

NumPy's itemsize property returns the memory size of the array's data-type in bytes.

```
a = np.array([1,2,3], dtype="int64")
a.itemsize
```

Output

8

Here, the contents in the array are irrelevant - only the data-type is important here. We set the datatype of the array as "int64", which means that each item in the array will be 8 bytes (i.e. 64 bits). This is exactly what the output is telling us

```
a = np.array([1,2,3], dtype="int32")
a.itemsize
```

Output

4

Here, 32-bit integer corresponds to 4 bytes, and so that's what is returned.

3. dtype –

The NumPy array object has a property called dtype that returns the data type of the array:


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

Output

int64

4. reshape-

Reshaping means changing the shape of an array. The shape of an array is the number of elements in each dimension. By reshaping we can add or remove dimensions or change number of elements in each dimension.

Reshape From 1-D to 2-D

Example

Convert the following 1-D array with 12 elements into a 2-D array.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newarr = arr.reshape(4, 3)
print(newarr)
```

Output

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

Reshape From 1-D to 3-D

Example

Convert the following 1-D array with 12 elements into a 3-D array. The outermost dimension will have 2 arrays that contains 3 arrays, each with 2 elements:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newarr = arr.reshape(2, 3, 2)
print(newarr)
```

Output

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

 [[ 7  8]
   [ 9 10]
   [11 12]]]
```

5. slicing-

Slicing in python means taking elements from one given index to another given index. We pass slice instead of index like this: [start:end]. We can also define the step, like this: [start:end:step].

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5])
```

Output

```
[2 3 4 5]
```

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[4:])
```

Output

```
[5 6 7]
```

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[:4])
```

Output

```
[1 2 3 4]
```

6. linspace –

The `numpy.linspace()` function is used to create an array of evenly spaced numbers within a specified range. The range is defined by the start and end points of the sequence, and the number of evenly spaced points to be generated between them.

```
import numpy as np
arr = np.linspace(10, 20, 5)
print("The array over the given range is ",arr)
```

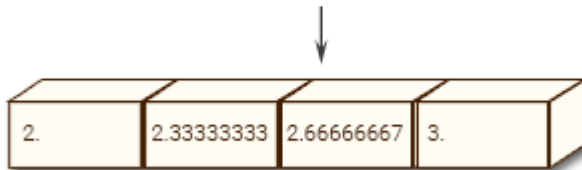
Output

The array over the given range is [10. 12.5 15. 17.5 20.]

Example

```
>>> np.linspace(2.0, 3.0, num=5)
array([2.   , 2.25, 2.5   , 2.75, 3.   ])
```

np.linspace(2.0, 3.0, num=4)



7. max/min , sum, sqrt

`numpy.sqrt(array[, out])` function is used to determine the positive square-root of an array, element-wise.

```
# Python program explaining
# numpy.sqrt() method

# importing numpy
import numpy as np
# applying sqrt() method on integer numbers
arr1 = np.sqrt([1, 4, 9, 16])
print("square-root of an array1 : ", arr1)
```

Output

square-root of an array1 : [1. 2. 3. 4.]

ndarray.max():

This function of ndarray.numpy returns the maximum value of ndarray object.

Syntax:

numpy.max(input_array)

```
# import numpy library
import numpy as np
# create numpy array
a = np.array([10,20,30,40,50])
# find the maximum element in the array
b = np.max(a)
print(b)
```

Output

50

ndarray.min():

This function of ndarray.numpy returns the minimum value of ndarray object.

Syntax:

numpy.min(input_array)

```
# import numpy library
import numpy as np
# create numpy array
a = np.array([10,20,30,40,50])
# find the maximum element in the array
b = np.min(a)
print(b)
```

Output

10

numpy.sum():

This function is used to return the sum of all elements of an array over specified axis. It is possible to add rows and column elements of an array.

Example

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

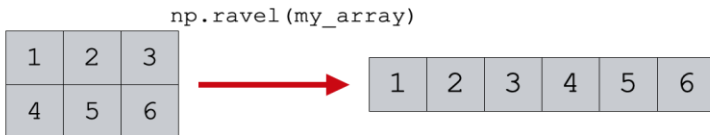
Output

15

8. numpy.ravel() in Python

The numpy module of Python provides a function called `numpy.ravel`, which is used to change a 2-dimensional array or a multi-dimensional array into a contiguous flattened array. The returned array has the same data type as the source array or input array. If the input array is a masked array, the returned array will also be a masked array.

NUMPY RAVEL CREATES A 1D ARRAY FROM A MULTI DIMENSIONAL ARRAY



Syntax:

`numpy.ravel(x, order='C')`

```
import numpy as np
x = np.array([[1, 3, 5], [11, 35, 56]])
y=np.ravel(x)
y
```

Output

```
array([ 1,  3,  5, 11, 35, 56])
```

```
import numpy as np
x = np.array([[1, 3, 5], [11, 35, 56]])
y = np.ravel(x, order='F')
z = np.ravel(x, order='C')
p = np.ravel(x, order='A')
q = np.ravel(x, order='K')
y
z
p
q
```

Output

```
array([ 1,  3,  5, 11, 35, 56])
```

More Useful Array Operations

There are also a few Special Operations like sine, cosine, tan, log, etc.

Trigonometric Functions

NumPy provides the ufuncs `sin()`, `cos()` and `tan()` that take values in radians and produce the corresponding sin, cos and tan values.

Example

Find sine value of $\pi/2$:

```
import numpy as np
x = np.sin(np.pi/2)
print(x)
```

Output

1.0

Example

Find sine values for all of the values in arr:

```
import numpy as np
arr = np.array([np.pi/2, np.pi/3, np.pi/4, np.pi/5])
x = np.sin(arr)
print(x)
```

Output

```
[1.          0.8660254  0.70710678 0.58778525]
```

Convert Degrees into Radians

By default all of the trigonometric functions take radians as parameters but we can convert radians to degrees and vice versa as well in NumPy.

Note: radians values are $\pi/180 * \text{degree_values}$.

Example

Convert all of the values in following array arr to radians:

```
import numpy as np
arr = np.array([90, 180, 270, 360])
x = np.deg2rad(arr)
print(x)
```

Output

```
[1.57079633 3.14159265 4.71238898 6.28318531]
```

Arithmetic Operations:

In the below example, you add two numpy arrays. The result is an element-wise sum of both the arrays.

```
import numpy as np

array_A = np.array([1, 2, 3])
array_B = np.array([4, 5, 6])

print(array_A + array_B)
```

Output

```
[5 7 9]
```

```
# Python program to perform arithmetic operations on the Numpy arrays

import numpy as np

# Initializing our array
array1 = np.arange(9, dtype = np.float_).reshape(3, 3)
print('First Array:')
print(array1)

print('\nSecond array:')
array2 = np.arange(11,20, dtype = np.float_).reshape(3, 3)
print(array2)

print('\nAdding two arrays:')
print(np.add(array1, array2))

print('\nSubtracting two arrays:')
print(np.subtract(array1, array2))

print('\nMultiplying two arrays:')
print(np.multiply(array1, array2))

print('\nDividing two arrays:')
print(np.divide(array1, array2))
```

Output

First Array:

```
[[0. 1. 2.]  
 [3. 4. 5.]  
 [6. 7. 8.]]
```

Second array:

```
[[11. 12. 13.]  
 [14. 15. 16.]  
 [17. 18. 19.]]
```

Adding two arrays:

```
[[11. 13. 15.]  
 [17. 19. 21.]  
 [23. 25. 27.]]
```

Subtracting two arrays:

```
[[ -11. -11. -11.]  
 [ -11. -11. -11.]  
 [ -11. -11. -11.]]
```

Multiplying two arrays:

```
[[ 0. 12. 26.]  
 [42. 60. 80.]  
 [102. 126. 152.]]
```

Dividing two arrays:

```
[[0.          0.08333333 0.15384615]  
 [0.21428571 0.26666667 0.3125    ]  
 [0.35294118 0.38888889 0.42105263]]
```

1.20 NumPy – Create 3D Array

To create a 3D (3 dimensional) array in Python using NumPy library, we can use any of the following methods.

`numpy.array()` – Creates array from given values.

`numpy.zeros()` – Creates array of zeros.

`numpy.ones()` – Creates array of ones.

`numpy.empty()` – Creates an empty array.

NumPy.empty

As the name specifies, the empty routine is used to create an uninitialized array of specified shape and data type.

The syntax is given below.

```
numpy.empty(shape, dtype = float, order = 'C')
```

It accepts the following parameters.

- **Shape:** The desired shape of the specified array.
- **dtype:** The data type of the array items. The default is the float.
- **Order:** The default order is the c-style row-major order. It can be set to F for FORTRAN-style column-major order.

Example

```
import numpy as np
arr = np.empty((3,2), dtype = int)
print(arr)
```

Output:

```
[ [ 140482883954664          36917984]
  [ 140482883954648    140482883954648]
  [6497921830368665435  172026472699604272]]
```

NumPy.Zeros

This routine is used to create the numpy array with the specified shape where each numpy array item is initialized to 0. The syntax is given below.

```
numpy.zeros(shape, dtype = float, order = 'C')
```

It accepts the following parameters.

- Shape: The desired shape of the specified array.
- dtype: The data type of the array items. The default is the float.
- Order: The default order is the c-style row-major order. It can be set to F for FORTRAN-style column-major order.

Example

```
import numpy as np
arr = np.zeros((3,2), dtype = int)
print(arr)
```

Output

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

NumPy.ones

This routine is used to create the numpy array with the specified shape where each numpy array item is initialized to 1. The syntax to use this module is given below.

```
numpy.ones(shape, dtype = none, order = 'C')
```

It accepts the following parameters.

- Shape: The desired shape of the specified array.
- dtype: The data type of the array items.
- Order: The default order is the c-style row-major order. It can be set to F for FORTRAN-style column-major order.

Example

```
import numpy as np
arr = np.ones((3,2), dtype = int)
print(arr)
```

Output

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

1.21 NumPy Data Types

Data Types in Python

By default Python have these data types:

strings - used to represent text data, the text is given under quote marks. e.g. "ABCD"

integer - used to represent integer numbers. e.g. -1, -2, -3

float - used to represent real numbers. e.g. 1.2, 42.42

boolean - used to represent True or False.

complex - used to represent complex numbers. e.g. $1.0 + 2.0j$, $1.5 + 2.5j$

Data Types in NumPy

NumPy has some extra data types, and refer to data types with one character, like i for integers, u for unsigned integers etc. Below is a list of all data types in NumPy and the characters used to represent them.

i - integer

b - boolean

u - unsigned integer

f - float

c - complex float

m - timedelta

M - datetime

O - object

S - string

U - unicode string

V - fixed chunk of memory for other type (void)

Checking the Data Type of an Array

The NumPy array object has a property called dtype that returns the data type of the array:

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

Output

int64

Create an array with data type string:

```
import numpy as np
arr = np.array([1, 2, 3, 4], dtype='S')
print(arr)
print(arr.dtype)
```

Output

```
[b'1' b'2' b'3' b'4']
|S1
```

Converting Data Type on Existing Arrays

The best way to change the data type of an existing array, is to make a copy of the array with the **astype() method**. The **astype()** function creates a copy of the array, and allows you to specify the data type as a parameter. The data type can be specified using a string, like 'f' for float, 'i' for integer etc. or you can use the data type directly like float for float and int for integer.

Example

Change data type from float to integer by using 'i' as parameter value:

```
import numpy as np

arr = np.array([1.1, 2.1, 3.1])

newarr = arr.astype('i')

print(newarr)
print(newarr.dtype)
```

Output

```
[1 2 3]
int32
```

UNIT-II

NumPy II: Creating Matrices, Generating Random Numbers, How to get Unique Items and Counts, Transporting and Reshaping Matrices, Reverse an Array, Reshaping and Flattering Multi-Dimensional Arrays, How to Accessing the Doc-string for more information, Working with Mathematical Formulas, How to save and load NumPy Objects.

2.1 Creating Matrices

We can create a matrix in Numpy using functions like `array()`, `ndarray()` or `matrix()`. Matrix function by default creates a specialized 2D array from the given input. The input should be in the form of a string or an array object-like. Let's demonstrate matrix creation using a `matrix()` with string as an input type.

```
import numpy as np
#creating matrix from string
A = np.matrix('1 2 3; 4 5 6')
print("Array created using string is :\n", A)
```

Output

```
Array created using string is :
[[1 2 3]
 [4 5 6]]
```

Now, let's demonstrate matrix creation using a `matrix()` with an array-like object as the input type.

```
import numpy as np
#creating matrix from array like object
B = np.matrix([[1, 2, 3], [4, 5, 6]])
print("Array created using array like object is :\n", B)
```

Output

```
Array created using array like object is :
[[1 2 3]
 [4 5 6]]
```

Matrix Functions in NumPy with Examples

Function	Description
matrix.T	Returns transpose of the input matrix
matrix.H	Returns complex conjugate transpose of the input matrix
matrix.I	Returns the multiplicative inverse of the matrix
matrix.A	Return the input matrix as a ndarray object.
matrix.all()	Checks whether all matrix elements along a given axis evaluate to True. It is similar to ndarray.all() function.
matrix.any()	Checks whether any of the matrix elements along a given axis evaluate to True.
matrix.argmax()	Returns the indices of the maximum values along an axis in the input matrix
matrix.argmin()	Returns the indices of the minimum values along an axis in the input matrix

matrix.argsort()	Returns the indices that would sort the matrix
matrix.astype()	Returns the copy of the input matrix after type change
matrix.choose()	Returns a new matrix with chosen indices from the input matrix
matrix.clip()	Returns a new matrix within chosen limits from the input matrix
matrix.compress()	Returns the selected slice of a matrix along the given axis
matrix.conj()	Returns the complex conjugate of the given matrix
matrix.cumprod()	Returns cumulative product of elements in the given matrix along a given axis
matrix.cumsum()	Returns the cumulative sum of elements in the given matrix along with the given axis
matrix.diagonal()	Returns diagonal elements of the matrix
matrix.dot()	The returns dot product of two matrices

Example #1 – Program to Find Transpose and Multiplicative Inverse of a Given Matrix

```
import numpy as np
A = np.matrix('1 2 3; 4 5 6')
print("Matrix is :\n", A)
#Transpose of matrix
print("The transpose of matrix A is :\n", A.getT())
#Complex conjugate transpose
print("Complex transpose of matrix A is :\n", A.getH())
#Multiplicative inverse
print("Multiplicative inverse of matrix A is :\n", A.getI())
```

Output

```
Matrix is :
[[1 2 3]
 [4 5 6]]
The transpose of matrix A is :
[[1 4]
 [2 5]
 [3 6]]
Complex transpose of matrix A is :
[[1 4]
 [2 5]
 [3 6]]
Multiplicative inverse of matrix A is :
[[-0.94444444  0.44444444]
 [-0.11111111  0.11111111]
 [ 0.72222222 -0.22222222]]
```

Example #2 – Program to Find the Maximum and Minimum Indices in A Given Matrix

```

import numpy as np
A = np.matrix('1 2 3; 4 5 6')
print("Matrix is :\n", A)
#maximum indices
print("Maximum indices in A :\n", A.argmax(0))
#minimum indices
print("Minimum indices in A :\n", A.argmin(0))

```

Output

```

Matrix is :
[[1 2 3]
 [4 5 6]]
Maximum indices in A :
[[1 1 1]]
Minimum indices in A :
[[0 0 0]]

```

Example #3 – Program to Illustrate Clipping in A Given Matrix

```

import numpy as np
A = np.matrix('1 2 3; 4 5 6')
print("Matrix is :\n", A)
#clipping matrix
print("Clipped matrix is :\n", A.clip(1,4))

```

Output

```

Matrix is :
[[1 2 3]
 [4 5 6]]
Clipped matrix is :
[[1 2 3]
 [4 4 4]]

```


Example #4 – Program to Find Cumulative Sum and Product of a Given Matrix

```
import numpy as np
A = np.matrix('1 2 3; 4 5 6; 7,8,9')
print("Matrix is :\n", A)
#cumulative product along axis = 0
print("Cumulative product of elements along axis 0 is : \n", A.cumprod(0))
#cumulative sum along axis = 0
print("Cumulative sum of elements along axis 0 is : \n", A.cumsum(0))
```

Output

```
Matrix is :
[[1 2 3]
 [4 5 6]
 [7 8 9]]
Cumulative product of elements along axis 0 is :
[[ 1  2  3]
 [ 4 10 18]
 [28 80 162]]
Cumulative sum of elements along axis 0 is :
[[ 1  2  3]
 [ 5  7  9]
 [12 15 18]]
```

Example #5 – Program to Find the Dot Product and Diagonal Values of a Given Matrix

In order to find the diagonal values of a given matrix, we can use a diagonal function with attributes such as offset, axis 1 and axis 2.

```
import numpy as np
A = np.matrix('1 2 3; 4 5 6; 7,8,9')
print("Matrix is :\n", A)
#diagonal values
print("Diagonal of matrix A :\n", A.diagonal(0,0,1))
#dot product
print("Dot product of matrix A with 2 :\n", A.dot(2))
```

Output

```
Matrix is :  
[[1 2 3]  
 [4 5 6]  
 [7 8 9]]  
Diagonal of matrix A :  
[[1 5 9]]  
Dot product of matrix A with 2 :  
[[ 2  4  6]  
 [ 8 10 12]  
 [14 16 18]]
```

Perform Matrix Multiplication in NumPy

We use the `np.dot()` function to perform multiplication between two matrices.

Example 6

```
import numpy as np  
  
# create two matrices  
matrix1 = np.array([[1, 3],  
                    [5, 7]])  
  
matrix2 = np.array([[2, 6],  
                    [4, 8]])  
  
# calculate the dot product of the two matrices  
result = np.dot(matrix1, matrix2)  
  
print("matrix1 x matrix2: \n",result)
```

Output

```
matrix1 x matrix2:  
[[14 30]  
 [38 86]]
```

Advantages of Matrix in NumPy

One of the main advantages of using the NumPy matrix is that they take less memory space and provide better runtime speed when compared with similar data structures in python (lists and tuples).

NumPy matrix support some specific scientific functions such as element-wise cumulative sum, cumulative product, conjugate transpose, and multiplicative inverse, etc. The python lists or strings fail to support these features.

2.2 Generating Random Numbers

a) What is a Random Number?

Random number does NOT mean a different number every time. Random means something that can not be predicted logically.

b) Pseudo Random and True Random.

Computers work on programs, and programs are definitive set of instructions. So it means there must be some algorithm to generate a random number as well. If there is a program to generate random number it can be predicted, thus it is not truly

random. Random numbers generated through a generation algorithm are called pseudo random.

c) Can we make truly random numbers?

Yes. In order to generate a truly random number on our computers we need to get the random data from some outside source. This outside source is generally our keystrokes, mouse movements, data on network etc.

We do not need truly random numbers, unless its related to security (e.g. encryption keys) or the basis of application is the randomness (e.g. Digital roulette wheels).

d) Generate Random Number

NumPy offers the random module to work with random numbers.

Example

Generate a random integer from 0 to 100:

```
from numpy import random
x = random.randint(100)
print(x)
```

Output

38

e) Generate Random Float

The random module's rand() method returns a random float between 0 and 1.

Example

Generate a random float from 0 to 1:

```
from numpy import random
x = random.rand()
print(x)
```

Output

0.6307042323508474

f) Generate Random Array

In NumPy we work with arrays, and you can use the two methods from the above examples to make random arrays.

Integers

The randint() method takes a size parameter where you can specify the shape of an array.

Example

Generate a 1-D array containing 5 random integers from 0 to 100:

```
from numpy import random
x=random.randint(100, size=(5))
print(x)
```

Output

[37 22 49 57 84]

Example

Generate a 2-D array with 3 rows, each row containing 5 random integers from 0 to 100:

```
from numpy import random
x = random.randint(100, size=(3, 5))
print(x)
```

Floats

The `rand()` method also allows you to specify the shape of the array.

Example

Generate a 1-D array containing 5 random floats:

```
from numpy import random
x = random.rand(5)
print(x)
```

Output

```
[0.02511319 0.61928452 0.7530847 0.0552907 0.98090184]
```

Example

Generate a 2-D array with 3 rows, each row containing 5 random numbers:

```
from numpy import random
x = random.rand(3, 5)
print(x)
```

Output

```
[[0.54226859 0.83730202 0.06992426 0.11426241 0.20466511]
 [0.17820341 0.61133874 0.45691592 0.40212779 0.42750675]
 [0.70737622 0.24798109 0.05378981 0.41590964 0.75470814]]
```

Generate Random Number From Array

The `choice()` method allows you to generate a random value based on an array of values.

Example

Return one of the values in an array:

```
from numpy import random
x = random.choice([3, 5, 7, 9])
print(x)
```

Output

3

The choice() method also allows you to return an array of values. Add a size parameter to specify the shape of the array.

Example

Generate a 2-D array that consists of the values in the array parameter (3, 5, 7, and 9):

```
from numpy import random
x = random.choice([3, 5, 7, 9], size=(3, 5))
print(x)
```

Output

```
[[5 9 7 9 5]
 [7 3 5 7 5]
 [5 9 5 7 9]]
```

2.3 How to get Unique Items and Counts

unique() to find the unique values in a NumPy array 'x'. The function is called with the optional argument

'return_inverse=True', which also returns an array of indices of the unique values in the original array.

The unique values are returned as an array 'u', which is sorted in ascending order.

```
import numpy as np
x = np.array([1,2,5,3,4,2,3,2,5,4])
u, indices = np.unique(x, return_inverse=True)
u
```

Output

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

In this case, the unique values in 'x' are [1, 2, 3, 4, 5]. The array 'indices' is also returned, which represents the original array 'x' in terms of the unique values in 'u'.

```
# import library
import numpy as np
# create 1d-array
arr = np.array([3, 3, 4,
                5, 6, 5,
                6, 4])

# find unique element
# from a array
rslt = np.unique(arr)
print(rslt)
```

Output

[3 4 5 6]

a) Unique values with their index in the array

Let's also get the indices in the input array that give the unique values. Pass True to the return_index parameter.

```
# create a 1d numpy array
ar = np.array([3, 2, 2, 1, 0, 1, 3, 3, 3])
# get unique values and indices
ar_unique, i = np.unique(ar, return_index=True)
# display the returned array
print("Unique values:", ar_unique)
# display the indices
print("Indices:", i)
```

Output

```
Unique values: [0 1 2 3]
Indices: [4 3 1 0]
```

b) Unique values with their counts in numpy array

You can also get the count for the number of times each unique value occurs in the input array. Pass True to the return_counts parameter.

```

# create a 1d numpy array
ar = np.array([3, 2, 2, 1, 0, 1, 3, 3, 3])
# get unique values and counts
ar_unique, i = np.unique(ar, return_counts=True)
# display the returned array
print("Unique values:", ar_unique)
# display the counts
print("Counts:", i)

```

Output

```

Unique values: [0 1 2 3]
Counts: [1 2 2 4]

```

c) Unique values with reverse index

To get the indices of values in the unique array so as to reconstruct the input array from the unique values array, pass True to the return_inverse parameter.

```

# create a 1d numpy array
ar = np.array([3, 2, 2, 1, 0, 1, 3, 3, 3])
# get unique values and inverse indices
ar_unique, inverse_i = np.unique(ar, return_inverse=True)
# display the returned array
print("Unique values:", ar_unique)
# display the inverse indices
print("Inverse indices:", i)

```

Output

```

Unique values: [0 1 2 3]

```

```

Inverse indices: [3 2 2 1 0 1 3 3 3]

```

d) Get unique rows of a 2D array

You can use the axis parameter to determine along which axis should the np.unique() function operate. By default, it flattens the input array and returns the unique values. For example, if you use it on a 2D array with the default value for the axis parameter –

```
# create a 2d numpy array
ar = np.array([[1, 1, 1],
               [0, 0, 0],
               [1, 1, 1]])
# get unique values in ar
ar_unique = np.unique(ar)
# display the returned array
print(ar_unique)
```

Output

```
[0 1]
```

e) Count Number of Unique Values

The following code shows how to count the total number of unique values in the NumPy array:

```
import numpy as np

#create NumPy array
my_array = np.array([1, 3, 3, 4, 4, 7, 8, 8])
#display unique values
np.unique(my_array)
#display total number of unique values
len(np.unique(my_array))
```

Output

5

2.4 Transporting and Reshaping Matrices

With the help of Numpy matrix.transpose() method, we can find the transpose of the matrix.

Syntax : matrix.transpose()

```
# import the important module in python
import numpy as np
# make matrix with numpy
mt= np.matrix('[4, 1; 12, 3]')
# applying matrix.transpose() method
b= mt.transpose()
print(b)
```

Output

```
[[ 4 12]
 [ 1  3]]
```

```
# import the important module in python
import numpy as np
# make matrix with numpy
mt= np.matrix('[4, 1, 9; 12, 3, 1; 4, 5, 6]')
# applying matrix.transpose() method
b= mt.transpose()
print(b)
```

Output

```
[[ 4 12  4]
 [ 1  3  5]
 [ 9  1  6]]
```

Reshaping Matrices (Reshaping arrays)

Reshaping means changing the shape of an array. The shape of an array is the number of elements in each dimension. By reshaping we can add or remove dimensions or change number of elements in each dimension.

Reshape From 1-D to 2-D

Example

Convert the following 1-D array with 12 elements into a 2-D array. The outermost dimension will have 4 arrays, each with 3 elements:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newarr = arr.reshape(4, 3)
print(newarr)
```

Output

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

Reshape From 1-D to 3-D

Example

Convert the following 1-D array with 12 elements into a 3-D array. The outermost dimension will have 2 arrays that contain 3 arrays, each with 2 elements:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newarr = arr.reshape(2, 3, 2)
print(newarr)
```

Output

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

 [[ 7  8]
 [ 9 10]
 [11 12]]]
```

Can We Reshape Into any Shape?

Yes, as long as the elements required for reshaping are equal in both shapes.

Example

Try converting 1D array with 8 elements to a 2D array with 3 elements in each dimension (will raise an error):

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
newarr = arr.reshape(3, 3)
print(newarr)
```

Output

```
ValueError                                Traceback (most recent call last)
<ipython-input-54-79494d80387a> in <cell line: 5>()
      3 arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
      4
----> 5 newarr = arr.reshape(3, 3)
      6
      7 print(newarr)

ValueError: cannot reshape array of size 8 into shape (3,3)
```

Unknown Dimension

You are allowed to have one "unknown" dimension. Meaning that you do not have to specify an exact number for one of the dimensions in the reshape method. Pass -1 as the value, and NumPy will calculate this number for you.

Example

Convert 1D array with 8 elements to 3D array with 2x2 elements:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
newarr = arr.reshape(2, 2, -1)
print(newarr)
```

Output

```
[[[1 2]
   [3 4]]

 [[5 6]
   [7 8]]]
```

2.5 Reverse an Array

In Python, the reverse of a NumPy array indicates changing the order. The first element will become the last element, the second last, the last the first, and so on. There are several methods for reversing a numpy array. Some of the most frequent methods for reversing a numpy array are as follows:

a) Using `flip()` function to Reverse a Numpy array

The `numpy.flip()` function reverses the order of array elements along the specified axis, preserving the shape of the array.

```
import numpy as np
# initialising numpy array
ini_array = np.array([1, 2, 3, 6, 4, 5])
# using shortcut method to reverse
res = np.flip(ini_array)
# printing result
print("final array", str(res))
```

Output

```
final array [5 4 6 3 2 1]
```


b) Using the list slicing method to reverse a Numpy array

```
import numpy as np
arr = np.array([2, 5, 1, 6, 7, 2, 4])
reve_arr = arr[::-1]
print(reve_arr)
```

Output

```
[4 2 7 6 1 5 2]
```

c) Using flipud function to Reverse a Numpy array

The numpy.flipud() function flips the array(entries in each column) in up-down direction, shape preserved.

```
import numpy as np
# initialising numpy array
ini_array = np.array([1, 2, 3, 6, 4, 5])
# using flipud method to reverse
res = np.flipud(ini_array)
# printing result
print("final array", str(res))
```

Output

```
final array [5 4 6 3 2 1]
```

d) Using the flip() Function

The syntax of the numpy.flip() function is as follows.

Syntax:

```
numpy.flip(arr, axis=None)
```

```
import numpy as np
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
reversed_arr = np.flip(arr)
print('Reversed Array: ')
print(reversed_arr)
```

Output

```
Reversed Array:
[[12 11 10  9]
 [ 8  7  6  5]
 [ 4  3  2  1]]
```

e) Using the reverse() function

The reverse() function is a built-in Python technique that allows us to immediately reverse a list.

```
import array
arr=array.array('i',[1,2,6,8,4,2])
print(arr)
arr.reverse()
print("Reversed Array:",arr)
```

Output

```
array('i', [1, 2, 6, 8, 4, 2])
Reversed Array: array('i', [2, 4, 8, 6, 2, 1])
```

f) Using the flipr() Function

We may quickly reverse an array using the numpy.flipr() method. The np.flipr() function flips the array from left to right.

The `numpy.fliplr()` method always takes an array as an argument and returns the same array with a left-right flip.

```
import numpy as np
a=[[1,23],
   [34,6]]
print(np.fliplr(a))
print("Original array:")
print(a)
```

Output

```
[[23  1]
 [ 6 34]]
Original array:
[[1, 23], [34, 6]]
```

2.6 Flattening Multi-Dimensional Arrays

Flattening array means converting a multidimensional array into a 1D array. We can use `reshape(-1)` to do this.

Example

Convert the array into a 1D array:

```
import numpy as np
arr = np.array([[[[1, 2, 3], [4, 5, 6],[7,8,9]]]])
newarr = arr.reshape(-1)
print(newarr)
```

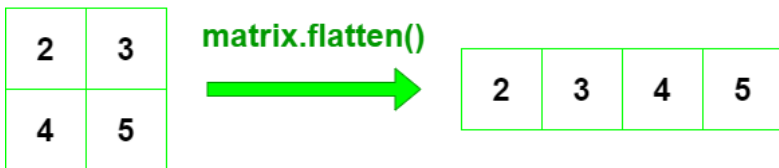
Output

```
[1 2 3 4 5 6 7 8 9]
```

numpy.ndarray.flatten() in Python

By using `ndarray.flatten()` function we can flatten a matrix to one dimension in python.

Syntax: `ndarray.flatten(order='C')`



Parameters:

`order: {'C', 'F', 'A', 'K'}(optional)`

If we set the order parameter to 'C', it means that the array gets flattened in row-major order. If 'F' is set, the array gets flattened in column-major order. The array is flattened in column-major order only when 'a' is Fortran contiguous in memory, and when we set the order parameter to 'A'. The last order is 'K', which flatten the array in same order in which the elements occurred in the memory. By default, this parameter is set to 'C'.

Example

```
import numpy as np
a = np.array([[1,4,7], [2,5,8],[3,6,9]])
b=a.flatten()
b
```

Output

```
array([1, 4, 7, 2, 5, 8, 3, 6, 9])
```

```
# importing numpy as np
import numpy as np

# declare matrix with np
gfg = np.array([[2, 3], [4, 5]])

# using array.flatten() method
flat_gfg = gfg.flatten()
print(flat_gfg)
```

Output

```
[2 3 4 5]
```

2.7 How to Accessing the Docstring

Python documentation strings (or docstrings) provide a convenient way of associating documentation with Python modules, functions, classes, and methods. It's specified in source code that is used, like a comment, to document a specific segment of code. Unlike conventional source code comments, the docstring should describe what the function does, not how.

What should a docstring look like?

- The doc string line should begin with a capital letter and end with a period.
- The first line should be a short description.

- If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description.
- The following lines should be one or more paragraphs describing the object's calling conventions, its side effects, etc.

Declaring Docstrings: The docstrings are declared using '''triple single quotes''' or """triple double quotes""" just below the class, method or function declaration. All functions should have a docstring.

Accessing Docstrings: The docstrings can be accessed using the `__doc__` method of the object or using the help function.

The below examples demonstrates how to declare and access a docstring.

Example 1: Using triple single quotes

```
def my_function():
    '''Demonstrates triple double quotes
    docstrings and does nothing really.'''

    return None

print("Using __doc__:")
print(my_function.__doc__)

print("Using help:")
help(my_function)
```

Output

```
Using __doc__:
Demonstrates triple double quotes
    docstrings and does nothing really.
Using help:
Help on function my_function in module __main__:

my_function()
    Demonstrates triple double quotes
    docstrings and does nothing really.
```

Example 2: Using triple double quotes

```
def my_function():
    """Demonstrates triple double quotes
    docstrings and does nothing really."""

    return None

print("Using __doc__:")
print(my_function.__doc__)

print("Using help:")
help(my_function)
```

Output

```
Using __doc__:
Demonstrates triple double quotes
    docstrings and does nothing really.
Using help:
Help on function my_function in module __main__:

my_function()
    Demonstrates triple double quotes
    docstrings and does nothing really.
```

Python Comments vs Docstrings

Python Comments

Comments are descriptions that help programmers better understand the intent and functionality of the program. They are completely ignored by the Python interpreter.

In Python, we use the hash symbol # to write a single-line comment. For example,

```
# Program to print "Hello World"
print("Hello World")
```

Output

```
Hello World
```

Python Comments Using Strings

If we do not assign strings to any variable, they act as comments.

For example,


```
"I am a single-line comment"  
  
...  
I am a  
multi-line comment!  
...  
  
print("Hello World")
```

Output

```
Hello World
```

Python docstrings

As mentioned above, Python docstrings are strings used right after the definition of a function, method, class, or module. They are used to document our code.

We can access these docstrings using the `__doc__` attribute.

Python `__doc__` attribute

Whenever string literals are present just after the definition of a function, module, class or method, they are associated with the object as their `__doc__` attribute. We can later use this attribute to retrieve this docstring.

```
def square(n):  
    '''Takes in a number n, returns the square of n'''  
    return n**2  
  
print(square.__doc__)
```

Output

Takes in a number n, returns the square of n

2.8 Working with Mathematical Formulas

Numpy contains a large number of mathematical functions which can be used to perform various mathematical operations. The mathematical functions include trigonometric functions, arithmetic functions, and functions for handling complex numbers. Let's discuss the mathematical functions.

Trigonometric functions

Numpy contains the trigonometric functions which are used to calculate the sine, cosine, and tangent of the different angles in radian. The sin, cos, and tan functions return the trigonometric ratio for the specified angles. Consider the following example.

```
import numpy as np
arr = np.array([0, 30, 60, 90, 120, 150, 180])
print("\nThe sin value of the angles",end = " ")
print(np.sin(arr * np.pi/180))
print("\nThe cosine value of the angles",end = " ")
print(np.cos(arr * np.pi/180))
print("\nThe tangent value of the angles",end = " ")
print(np.tan(arr * np.pi/180))
```

Output

```
The sin value of the angles [0.00000000e+00 5.00000000e-01 8.66025404e-01 1.00000000e+00
8.66025404e-01 5.00000000e-01 1.22464680e-16]

The cosine value of the angles [ 1.00000000e+00 8.66025404e-01 5.00000000e-01 6.12323400e-17
-5.00000000e-01 -8.66025404e-01 -1.00000000e+00]

The tangent value of the angles [ 0.00000000e+00 5.77350269e-01 1.73205081e+00 1.63312394e+16
-1.73205081e+00 -5.77350269e-01 -1.22464680e-16]
```

The `numpy.floor()` function

This function is used to return the floor value of the input data which is the largest integer not greater than the input value. Consider the following example.

```
import numpy as np
arr = np.array([12.202, 90.23120, 123.020, 23.202])
print(np.floor(arr))
```

Output

```
[ 12.  90. 123.  23.]
```

The `numpy.ceil()` function

This function is used to return the ceiling value of the array values which is the smallest integer value greater than the array element. Consider the following example.

```
import numpy as np
arr = np.array([12.202, 90.23120, 123.020, 23.202])
print(np.ceil(arr))
```

Output

```
[ 13.  91. 124.  24.]
```

Function: 1 | Sigmoid

A Sigmoid function is a mathematical function that has a characteristic S-shaped curve. The Sigmoid function is normally

used to refer specifically to the logistic function, also called the logistic sigmoid function.

$$\text{Sigmoid}(x) = \frac{e^x}{1+e^x}$$

```
import numpy as np

def sigmoid(x):
    sigmoid = 1 / (1 + np.exp(-x))
    return sigmoid

print('Sigmoid of 5 is ' + str(sigmoid(5)))
```

Output

```
Sigmoid of 5 is 0.9933071490757153
```

2.9 How to save and load NumPy Objects.

You can save numpy array to a file using numpy. save() and then later, load into an array using numpy. load().

Following is a quick code snippet where we use firstly use save() function to write array to file. Secondly, we use load() function to load the file to a numpy array.

```
# save array to file
numpy.save(file, array)
# load file to array
array = numpy.load(file)
```

1. Save numpy array to file

In the following example: we will initialize an array; create and open a file in write binary mode; and then write the array to the file using `numpy.save()` method.

```
import numpy as np

#initialize an array
arr = np.array([[[[11, 11, 9, 9],
                  [11, 0, 2, 0]
                 ],
                [[10, 14, 9, 14],
                 [0, 1, 11, 11]]])

# open a binary file in write mode
file = open("arr", "wb")
# save array to the file
np.save(file, arr)
# close the file
file.close
```

Output

```
<function BufferedWriter.close>
```

2. Load the saved numpy array from file

In this example, we will load the array from file. We will use the above example to save an array and continue that to read the array from file.

```

import numpy as np

#initialize an array
arr = np.array([[11, 11, 9, 9],
                [11, 0, 2, 0]
                ],
                [[10, 14, 9, 14],
                [0, 1, 11, 11]])

# open a binary file in write mode
file = open("arr", "wb")
# save array to the file
np.save(file, arr)
# close the file
file.close

# open the file in read binary mode
file = open("arr", "rb")
#read the file to numpy array
arr1 = np.load(file)
#close the file
print(arr1)

```

Output

```

[[[11 11  9  9]
  [11  0  2  0]]

```

UNIT-III

Pandas: User Guide: Object Creation, Viewing Data, Selection, Missing Data, Operations, Merge, Grouping, Reshaping, Time Series, Categorical, Getting Data In/ Out

Introduction to Data Structures: Series, Data Frame.

3.1 Introduction to Pandas

Python Pandas is defined as an open-source library that provides high-performance data manipulation in Python. It is used for data analysis in Python and developed by Wes McKinney in 2008.

Data analysis requires lots of processing, such as restructuring, cleaning or merging, etc. There are different tools are available for fast data processing, such as Numpy, Scipy, Cython, and Panda. But we prefer Pandas because working with Pandas is fast, simple and more expressive than other tools.

Pandas is built on top of the Numpy package, means Numpy is required for operating the Pandas.

Before Pandas, Python was capable for data preparation, but it only provided limited support for data analysis. So, Pandas came into the picture and enhanced the capabilities of data analysis. It can perform five significant steps required for processing and analysis of data irrespective of the origin of the data, i.e., load, manipulate, prepare, model, and analyze.

Benefits of Pandas

The benefits of pandas over using other language are as follows:

Data Representation: It represents the data in a form that is suited for data analysis through its DataFrame and Series.

Clear code: The clear API of the Pandas allows you to focus on the core part of the code. So, it provides clear and concise code for the user.

3.2 Python Pandas Data Structure

Pandas deals with the following three data structures –

- Series
- DataFrame
- Panel

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

Dimension & Description

The best way to think of these data structures is that the higher dimensional data structure is a container of its lower dimensional data structure. For example, DataFrame is a container of Series, Panel is a container of DataFrame.

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.

1) 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

Creating Series from Array:

Before creating a Series, Firstly, we have to import the numpy module and then use array() function in the program.

```
import pandas as pd
import numpy as np
info = np.array(['P', 'a', 'n', 'd', 'a', 's'])
a = pd.Series(info)
print(a)
```

Output

```
0    P
1    a
2    n
3    d
4    a
5    s
dtype: object
```

2) 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.

Data Type of Columns

The data types of the four columns are as follows –

Create a DataFrame using List:

Column	Type
Name	String

Age Integer

Gender String

Rating Float

Key Points

- Heterogeneous data
- Size Mutable
- Data Mutable

We can easily create a DataFrame in Pandas using list.

```
import pandas as pd
# a list of strings
x = ['Python', 'Pandas']

# Calling DataFrame constructor on list
df = pd.DataFrame(x)
print(df)
```

Output

```
      0
0  Python
1  Pandas
```

3) 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 DataFrame.

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

Pending

Key Points

- Heterogeneous data
- Size Mutable
- Data Mutable

a) Python Pandas Series

The Pandas Series can be defined as a one-dimensional array that is capable of storing various data types. We can easily convert the list, tuple, and dictionary into series using "series' method. The row labels of series are called the index. A Series cannot contain multiple columns. It has the following parameter:

data: It can be any list, dictionary, or scalar value.

index: The value of the index should be unique and hashable. It must be of the same length as data. If we do not pass any index, default `np.arange(n)` will be used.

dtype: It refers to the data type of series.

copy: It is used for copying the data.

Creating a Series:

We can create a Series in two ways:

- **Create an empty Series**
- **Create a Series using inputs.**

1. Create an Empty Series:

We can easily create an empty series in Pandas which means it will not have any value. The syntax that is used for creating an Empty Series:

```
<series object> = pandas.Series()
```

The below example creates an Empty Series type object that has no values and having default datatype, i.e., float64.

Example

```
import pandas as pd
x = pd.Series()
print (x)
```

Output

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

2. Creating a Series using inputs:

We can create Series by using various inputs:

- Array
- Dict
- Scalar value

Creating Series from Array:

Before creating a Series, firstly, we have to import the numpy module and then use array() function in the program. If the data is ndarray, then the passed index must be of the same length.

If we do not pass an index, then by default index of range(n) is being passed where n defines the length of an array, i.e., [0,1,2,...range(len(array))-1].

Example

```
import pandas as pd
import numpy as np
info = np.array(['P', 'a', 'n', 'd', 'a', 's'])
a = pd.Series(info)
print(a)
```

Output

```
0    P
1    a
2    n
3    d
4    a
5    s
dtype: object
```

Create a Series from dict

We can also create a Series from dict. If the dictionary object is being passed as an input and the index is not specified, then the dictionary keys are taken in a sorted order to construct the index.

If index is passed, then values correspond to a particular label in the index will be extracted from the dictionary.

```
#import the pandas library
import pandas as pd
import numpy as np
info = {'x' : 0., 'y' : 1., 'z' : 2.}
a = pd.Series(info)
print (a)
```

Output

```
x    0.0
y    1.0
z    2.0
dtype: float64
```

Create a Series using Scalar:

If we take the scalar values, then the index must be provided. The scalar value will be repeated for matching the length of the index.

```
#import pandas library
import pandas as pd
import numpy as np
x = pd.Series(4, index=[0, 1, 2, 3])
print (x)
```

Output

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


Accessing data from series with Position:

Once you create the Series type object, you can access its indexes, data, and even individual elements.

The data in the Series can be accessed similar to that in the ndarray.

```
import pandas as pd
x = pd.Series([1,2,3],index = ['a','b','c'])
#retrieve the first element
print (x[0])
```

Output

1

Series object attributes

The Series attribute is defined as any information related to the Series object such as size, datatype. etc. Below are some of the attributes that you can use to get the information about the Series object:

Attributes	Description
Series.index	Defines the index of the Series.
Series.shape	It returns a tuple of shape of the data.
Series.dtype	It returns the data type of the data.
Series.size	It returns the size of the data.
Series.empty	It returns True if Series object is empty, otherwise returns false.
Series.hasnans	It returns True if there are any NaN values, otherwise returns false.
Series.nbytes	It returns the number of bytes in the data.
Series.ndim	It returns the number of dimensions in the data.
Series.itemsize	It returns the size of the datatype of item.

Retrieving Index array and data array of a series object

We can retrieve the index array and data array of an existing Series object by using the attributes index and values.

```
import numpy as np
import pandas as pd
x=pd.Series(data=[2,4,6,8])
y=pd.Series(data=[11.2,18.6,22.5], index=['a', 'b', 'c'])
print(x.index)
print(x.values)
print(y.index)
print(y.values)
```

Output

```
RangeIndex(start=0, stop=4, step=1)
[2 4 6 8]
Index(['a', 'b', 'c'], dtype='object')
[11.2 18.6 22.5]
```

Retrieving Types (dtype) and Size of Type (itemsize)

You can use attribute dtype with Series object as <objectname> dtype for retrieving the data type of an individual element of a series object, you can use the itemsize attribute to show the number of bytes allocated to each data item.

```
import numpy as np
import pandas as pd
a=pd.Series(data=[1,2,3,4])
b=pd.Series(data=[4.9,8.2,5.6],
index=['x','y','z'])
print(a.dtype)
print(a.itemsize)
print(b.dtype)
print(b.itemsize)
```

Output

```
int64
8
float64
8
```

Series Functions

There are some functions used in Series which are as follows:

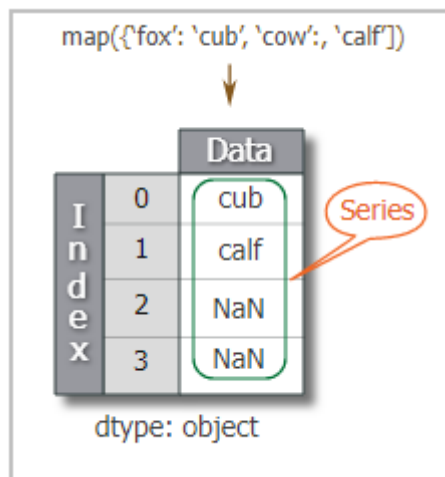
Functions	Description
Pandas Series.map()	Map the values from two series that have a common column.
Pandas Series.std()	Calculate the standard deviation of the given set of numbers, DataFrame, column, and rows.
Pandas Series.to_frame()	Convert the series object to the dataframe.
Pandas Series.value_counts()	Returns a Series that contain counts of unique values.

a) Pandas Series.map()

The map() function is used to map values of Series according to input correspondence.

Syntax:

```
Series.map(self, arg, na_action=None)
```



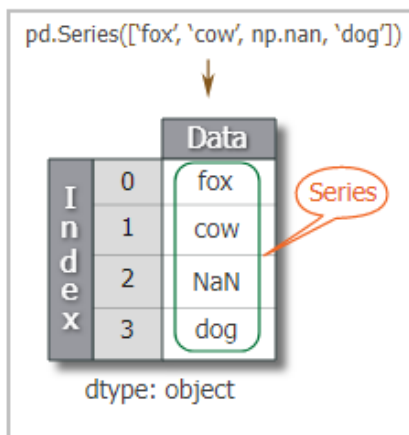
Parameters:

Name	Description	Type/Default Value	Required / Optional
arg	Mapping correspondence.	function, dict, or Series	Required
na_action	If 'ignore', propagate NaN values, without passing them to the mapping correspondence.	{None, 'ignore'} Default Value : None	Required

```
import numpy as np
import pandas as pd
s = pd.Series(['fox', 'cow', np.nan, 'dog'])
s
```

Output

```
0    fox
1    cow
2    NaN
3    dog
dtype: object
```



Example - map accepts a dict or a Series. Values that are not found in the dict are converted to NaN, unless the dict has a default value (e.g. defaultdict):

```
import numpy as np
import pandas as pd
s = pd.Series(['fox', 'cow', np.nan, 'dog'])
s.map({'fox': 'cub', 'cow': 'calf'})
```

Output

```
0    cub
1    calf
2    NaN
3    NaN
dtype: object
```

Example - It also accepts a function:

```
import numpy as np
import pandas as pd
s = pd.Series(['fox', 'cow', np.nan, 'dog'])
s.map('I am a {}'.format)
```

Output

```
0    I am a fox
1    I am a cow
2    I am a nan
3    I am a dog
dtype: object
```

b) Pandas Series.std()

Pandas Series.std() function return sample standard deviation over requested axis. The standard deviation is normalized by N-1 by default.

Example #1 : Use Series.std() function to find the standard deviation of the given Series object.

```
# importing pandas as pd
import pandas as pd

# Creating the Series
sr = pd.Series([100, 25, 32, 118, 24, 65])

# Print the series
print(sr)
sr.std()
```

Output

```
0    100
1     25
2     32
3    118
4     24
5     65
dtype: int64
40.72182052249956
```

C) Pandas Series.to_frame()

Pandas Series.to_frame() function is used to convert the given series object to a dataframe.

Example #1: Use Series.to_frame() function to convert the given series object to a dataframe.

```
# importing pandas as pd
import pandas as pd

# Creating the Series
sr = pd.Series([19.5, 16.8, 22.78, 20.124, 18.1002])

# Print the series
print(sr)
sr.to_frame()
```

Output

```
0    19.5000
1    16.8000
2    22.7800
3    20.1240
4    18.1002
dtype: float64
```

0



0 19.5000

1 16.8000

2 22.7800

3 20.1240

4 18.1002

D) Pandas Series.unique()

Pandas **Series.is_unique** attribute is used to check every element or value present in the pandas series object holds unique values or not. It returns True if the elements present in the given series object is unique, otherwise, it returns False.

Syntax

pandas.unique(values)

```
import pandas as pd
import numpy as np

# Create the Series
ser = pd.Series(['Spark', 'PySpark', 'Pandas', 'NumPy'])
print(ser)
# Usage Series.is_unique
ser2 = ser.is_unique
print(ser2)
```

Output

```
0    Spark
1  PySpark
2   Pandas
3   NumPy
dtype: object
True
```

```
import pandas as pd
ser=pd.Series(['Spark', 'Pyspark', 'Spark', 'Numpy'])
print(ser)
ser1=ser.is_unique
print(ser1)
```

Output

```
0      Spark
1    Pyspark
2      Spark
3      Numpy
dtype: object
False
```

E) Pandas Series.value_counts()

The value_counts() function returns a Series that contain counts of unique values. It returns an object that will be in descending order so that its first element will be the most frequently-occurred element. By default, it excludes NA values.

```
import pandas as pd
ser=pd.Series(['Spark', 'Pyspark', 'Spark', 'Numpy', 'Sci-Learn'])
print(ser)
ser.value_counts()
```

Output

```
0      Spark
1    Pyspark
2      Spark
3      Numpy
4    Sci-Learn
dtype: object
Spark      2
Pyspark    1
Numpy      1
Sci-Learn  1
dtype: int64
```

b) Python Pandas DataFrame

Pandas DataFrame is a widely used data structure which works with a two-dimensional array with labeled axes (rows and columns). DataFrame is defined as a standard way to store data that has two different indexes, i.e., row index and column index.

Create a DataFrame

We can create a DataFrame using following ways:

- dict
- Lists
- Numpy ndarrays
- Series

Create an empty DataFrame

The below code shows how to create an empty DataFrame in Pandas:

```
# importing the pandas library
import pandas as pd
df = pd.DataFrame()
print (df)
```

Output

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

Create a DataFrame using List:

We can easily create a DataFrame in Pandas using list.

```
# importing the pandas library
import pandas as pd
# a list of strings
x = ['Python', 'Pandas']

# Calling DataFrame constructor on list
df = pd.DataFrame(x)
print(df)
```

Output

```
      0
0  Python
1  Pandas
```

Create a DataFrame from Dict of ndarrays/ Lists

```
# importing the pandas library
import pandas as pd
info = {'ID' :[101, 102, 103], 'Department' :['B.Sc', 'B.Tech', 'M.Tech',]}
df = pd.DataFrame(info)
print (df)
```

Output

```
      ID Department
0  101      B.Sc
1  102      B.Tech
2  103      M.Tech
```

Create a DataFrame from Dict of Series:

```
# importing the pandas library
import pandas as pd

info = {'one' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f']),
        'two' : pd.Series([1, 2, 3, 4, 5, 6, 7, 8], index=['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])}

d1 = pd.DataFrame(info)
print (d1)
```

Output

	one	two
a	1.0	1
b	2.0	2
c	3.0	3
d	4.0	4
e	5.0	5
f	6.0	6
g	NaN	7
h	NaN	8

Column Selection

We can select any column from the DataFrame. Here is the code that demonstrates how to select a column from the DataFrame.

```
# importing the pandas library
import pandas as pd

info = {'one' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f']),
        'two' : pd.Series([1, 2, 3, 4, 5, 6, 7, 8], index=['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])}

d1 = pd.DataFrame(info)
print (d1 ['one'])
```

Output

```
a    1.0
b    2.0
c    3.0
d    4.0
e    5.0
f    6.0
g    NaN
h    NaN
Name: one, dtype: float64
```

Column Addition

```
# importing the pandas library
import pandas as pd

info = {'one' : pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e']),
        'two' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f'])}

df = pd.DataFrame(info)

# Add a new column to an existing DataFrame object

print ("Add new column by passing series")
df['three']=pd.Series([20,40,60],index=['a', 'b', 'c'])
print (df)

print ("Add new column using existing DataFrame columns")
df['four']=df['one']+df['three']

print (df)
```

Output

Add new column by passing series

	one	two	three
a	1.0	1	20.0
b	2.0	2	40.0
c	3.0	3	60.0
d	4.0	4	NaN
e	5.0	5	NaN
f	NaN	6	NaN

Add new column using existing DataFrame columns

	one	two	three	four
a	1.0	1	20.0	21.0
b	2.0	2	40.0	42.0
c	3.0	3	60.0	63.0
d	4.0	4	NaN	NaN
e	5.0	5	NaN	NaN
f	NaN	6	NaN	NaN

Column Deletion:

We can also delete any column from the existing DataFrame. This code helps to demonstrate how the column can be deleted from an existing DataFrame:

```
# Import pandas package
import pandas as pd

# create a dictionary with five fields each
data = {
    'A': ['A1', 'A2', 'A3', 'A4', 'A5'],
    'B': ['B1', 'B2', 'B3', 'B4', 'B5'],
    'C': ['C1', 'C2', 'C3', 'C4', 'C5'],
    'D': ['D1', 'D2', 'D3', 'D4', 'D5'],
    'E': ['E1', 'E2', 'E3', 'E4', 'E5']}

# Convert the dictionary into DataFrame
df = pd.DataFrame(data)

# Remove column name 'A'
df.drop(['A'], axis=1)
```

Output

	B	C	D	E
0	B1	C1	D1	E1
1	B2	C2	D2	E2
2	B3	C3	D3	E3
3	B4	C4	D4	E4
4	B5	C5	D5	E5

Example 2: Remove specific multiple columns.

```
# Import pandas package
import pandas as pd

# create a dictionary with five fields each
data = {
    'A': ['A1', 'A2', 'A3', 'A4', 'A5'],
    'B': ['B1', 'B2', 'B3', 'B4', 'B5'],
    'C': ['C1', 'C2', 'C3', 'C4', 'C5'],
    'D': ['D1', 'D2', 'D3', 'D4', 'D5'],
    'E': ['E1', 'E2', 'E3', 'E4', 'E5']}

# Convert the dictionary into DataFrame
df = pd.DataFrame(data)

# Remove two columns name is 'C' and 'D'
df.drop(['C', 'D'], axis=1)

# df.drop(columns=['C', 'D'])
```

Output

	A	B	E
0	A1	B1	E1
1	A2	B2	E2
2	A3	B3	E3
3	A4	B4	E4
4	A5	B5	E5

Row Selection, Addition, and Deletion

Row Selection:

We can easily select, add, or delete any row at anytime

a) Selection by Label:

We can select any row by passing the row label to a loc function.

```
# importing the pandas library
import pandas as pd

info = {'one' : pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e']),
        'two' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f'])}

df = pd.DataFrame(info)
print (df.loc['b'])
```

Output

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

b) Selection by integer location:

The rows can also be selected by passing the integer location to an iloc function.

```
# importing the pandas library
import pandas as pd
info = {'one' : pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e']),
        'two' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f'])}
df = pd.DataFrame(info)
print (df.iloc[3])
```

Output

```
one    4.0
two    4.0
Name: d, dtype: float64
```

c) Slice Rows

It is another method to select multiple rows using ':' operator.

```
# importing the pandas library
import pandas as pd
info = {'one' : pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e']),
       'two' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f'])}
df = pd.DataFrame(info)
print (df[2:5])
```

Output

```
      one  two
c  3.0    3
d  4.0    4
e  5.0    5
```

d) Addition of rows

We can easily add new rows to the DataFrame using append function. It adds the new rows at the end.

```
# importing the pandas library
import pandas as pd
d = pd.DataFrame([[7, 8], [9, 10]], columns = ['x','y'])
d2 = pd.DataFrame([[11, 12], [13, 14]], columns = ['x','y'])
d = d.append(d2)
print (d)
```

Output

	x	y
0	7	8
1	9	10
0	11	12
1	13	14

e) Deletion of rows

We can delete or drop any rows from a DataFrame using the index label. If in case, the label is duplicate then multiple rows will be deleted.

```
# importing the pandas library
import pandas as pd

a_info = pd.DataFrame([[4, 5], [6, 7]], columns = ['x','y'])
# Drop rows with label 0
a_info = a_info.drop(0)
a_info
```

Output

	x	y
1	6	7

f) DataFrame Functions

Functions	Description
Pandas DataFrame.append()	Add the rows of other dataframe to the end of the given dataframe.
Pandas DataFrame.apply()	Allows the user to pass a function and apply it to every single value of the Pandas series.
Pandas DataFrame.assign()	Add new column into a dataframe.
Pandas DataFrame.astype()	Cast the Pandas object to a specified dtype.astype() function.
Pandas DataFrame.concat()	Perform concatenation operation along an axis in the DataFrame.
Pandas DataFrame.count()	Count the number of non-NA cells for each column or row.
Pandas DataFrame.describe()	Calculate some statistical data like percentile, mean and std of the numerical values of the Series or DataFrame.
Pandas DataFrame.drop_duplicates()	Remove duplicate values from the DataFrame.
Pandas DataFrame.groupby()	Split the data into various groups.
Pandas DataFrame.head()	Returns the first n rows for the object based on position.
Pandas DataFrame.hist()	Divide the values within a numerical variable into "bins".
Pandas DataFrame.iterrows()	Iterate over the rows as (index, series) pairs.
Pandas DataFrame.mean()	Return the mean of the values for the requested axis.
Pandas DataFrame.melt()	Unpivots the DataFrame from a wide format to a long format.
Pandas DataFrame.merge()	Merge the two datasets together into one.
Pandas DataFrame.pivot_table()	Aggregate data with calculations such as Sum, Count, Average, Max, and Min.
Pandas DataFrame.query()	Filter the dataframe.
Pandas DataFrame.sample()	Select the rows and columns from the dataframe randomly.

Pandas DataFrame.shift()	Shift column or subtract the column value with the previous row value from the dataframe.
Pandas DataFrame.sort()	Sort the dataframe.
Pandas DataFrame.sum()	Return the sum of the values for the requested axis by the user.
Pandas DataFrame.to_excel()	Export the dataframe to the excel file.
Pandas DataFrame.transpose()	Transpose the index and columns of the dataframe.
Pandas DataFrame.where()	Check the dataframe for one or more conditions.

Pandas DataFrame.append()

Example #1: Create two data frames and append the second to the first one.

```
# Importing pandas as pd
import pandas as pd
# Creating the first Dataframe using dictionary
df1 = df = pd.DataFrame({"a": [1, 2, 3, 4],
                        "b": [5, 6, 7, 8]})

# Creating the Second Dataframe using dictionary
df2 = pd.DataFrame({"a": [1, 2, 3],
                    "b": [5, 6, 7]})

# Print df1
print(df1, "\n")

# Print df2
df2

# to append df2 at the end of df1 dataframe
df1.append(df2)
```

Output

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

Pandas DataFrame.apply()

Example

Return the sum of each row by applying a function:

```
import pandas as pd

def calc_sum(x):
    return x.sum()

data = {
    "x": [50, 40, 30],
    "y": [300, 1112, 42]
}

df = pd.DataFrame(data)

x = df.apply(calc_sum)

print(x)
```

Output

```
x      120
y     1454
dtype: int64
```

Pandas DataFrame.aggregate()

The main task of DataFrame.aggregate() function is to apply some aggregation to one or more column. Most frequently used aggregations are:

sum: It is used to return the sum of the values for the requested axis.

min: It is used to return the minimum of the values for the requested axis.

max: It is used to return the maximum values for the requested axis.

```
import pandas as pd
data = {
    "x": [50, 40, 30],
    "y": [300, 1112, 42]
}
df = pd.DataFrame(data)
x = df.aggregate(["sum"])
print(x)
```

Output

```
      x      y
sum  120  1454
```


Pandas DataFrame.assign()

The assign() method is also responsible for adding a new column into a DataFrame. If we re-assign an existing column, then its value will be overwritten.

```
import pandas as pd
data = {
    "age": [16, 14, 10],
    "qualified": [True, True, True]
}
df = pd.DataFrame(data)
newdf = df.assign(name = ["Emil", "Tobias", "Linus"])
newdf
```

Output

	age	qualified	name
0	16	True	Emil
1	14	True	Tobias
2	10	True	Linus

Pandas DataFrame.astype()

Example

Return a new DataFrame where the data type of all columns has been set to 'int64':

```

import pandas as pd

data = {
    "Duration": [50, 40, 45],
    "Pulse": [109, 117, 110],
    "Calories": [409.1, 479.5, 340.8]
}
df = pd.DataFrame(data)
newdf = df.astype('int64')
newdf

```

Output

	Duration	Pulse	Calories
0	50	109	409
1	40	117	479
2	45	110	340

Pandas DataFrame.count()

Example

Count the number of (not NULL) values in each row:

```

import pandas as pd
data = {
    "Duration": [50, 40, None, None, 90, 20],
    "Pulse": [109, 140, 110, 125, 138, 170]
}
df = pd.DataFrame(data)
print(df.count())

```

Output

```
Duration    4
Pulse       6
dtype: int64
```

Pandas DataFrame.describe()

Example

Multiply the values for each row with the values from the previous row:

```
import pandas as pd
data = [10, 18, 11]
df = pd.DataFrame(data)
print(df.describe())
```

Output

```
count    3.000000
mean     13.000000
std       4.358899
min      10.000000
25%      10.500000
50%      11.000000
75%      14.500000
max      18.000000
```

Pandas DataFrame.drop_duplicates()

Example

Remove duplicate rows from the DataFrame:

```
import pandas as pd

data = {
    "name": ["Sally", "Mary", "John", "Mary"],
    "age": [50, 40, 30, 40]
}
df = pd.DataFrame(data)
newdf = df.drop_duplicates()
newdf
```

Output

	name	age
0	Sally	50
1	Mary	40
2	John	30

Pandas DataFrame.groupby()

The groupby() method allows you to group your data and execute functions on these groups.

Example

Find the average co2 consumption for each car brand:

```
import pandas as pd

data = {
    'co2': [95, 90, 99, 104, 105, 94, 99, 104],
    'model': ['Citigo', 'Fabia', 'Fiesta', 'Rapid', 'Focus', 'Mondeo', 'Octavia', 'B-Max'],
    'car': ['Skoda', 'Skoda', 'Ford', 'Skoda', 'Ford', 'Ford', 'Skoda', 'Ford']
}
df = pd.DataFrame(data)
print(df.groupby(["car"]).mean())
```

Output

```
          co2
car
Ford    100.5
Skoda   97.0
```

Pandas DataFrame.head()

The head() method returns a specified number of rows, string from the top.

The head() method returns the first 5 rows if a number is not specified.

```
import pandas as pd
df = pd.read_csv('data.csv')
print(df.head())
```

Output

	Duration	Pulse	Maxpulse	Calories
0	60	110	130	409.1
1	60	117	145	479.0
2	60	103	135	340.0
3	45	109	175	282.4
4	45	117	148	406.0

Pandas DataFrame.hist()

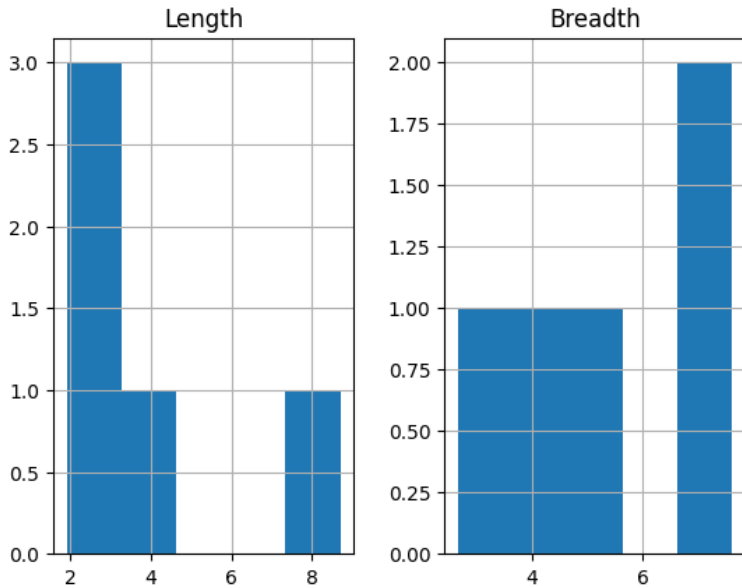
The hist() function is defined as a quick way to understand the distribution of certain numerical variables from the dataset. This function splits up the values into the numeric variables. Its main functionality is to make the Histogram of a given Data frame.

```
# Importing pandas library
import pandas as pd

# Creating a Data frame
values = pd.DataFrame({
    'Length': [2.7, 8.7, 3.4, 2.4, 1.9],
    'Breadth': [4.24, 2.67, 7.6, 7.1, 4.9]
})

# Creating Histograms of columns 'Length'
# and 'Breadth' using Dataframe.hist()
# function
hist = values.hist(bins=5)
```

Output



Pandas DataFrame.iterrows()

The `iterrows()` method generates an iterator object of the DataFrame, allowing us to iterate each row in the DataFrame.

Each iteration produces an index object and a row object

```
import pandas as pd

data = {
    "firstname": ["Sally", "Mary", "John"],
    "age": [50, 40, 30]
}

df = pd.DataFrame(data)

for index, row in df.iterrows():
    print(row["firstname"])
```

Output

```
Sally  
Mary  
John
```

Pandas DataFrame.join()

The join() method inserts column(s) from another DataFrame, or Series.

Example

Add the content of one DataFrame to another:

```
import pandas as pd  
  
data1 = {  
    "name": ["Sally", "Mary", "John"],  
    "age": [50, 40, 30]  
}  
  
data2 = {  
    "qualified": [True, False, False]  
}  
  
df1 = pd.DataFrame(data1)  
df2 = pd.DataFrame(data2)  
  
newdf = df1.join(df2)  
newdf
```


Output

	name	age	qualified
0	Sally	50	True
1	Mary	40	False
2	John	30	False

Pandas DataFrame.mean()

The mean() method returns a Series with the mean value of each column.

Mean, Median, and Mode:

Mean - The average value

Median - The mid point value

Mode - The most common value

Example

Return the average (mean) value for each column:

```
import pandas as pd
data = [[1, 1, 2], [6, 4, 2], [4, 2, 1], [4, 2, 3]]
df = pd.DataFrame(data)
print(df.mean())
```

Output

```
0    3.75
1    2.25
2    2.00
dtype: float64
```

The median() method returns a Series with the median value of each column.

```
import pandas as pd
data = [[1, 1, 2], [6, 4, 2], [4, 2, 1], [4, 2, 3]]
df = pd.DataFrame(data)
print(df.median())
```

Output

```
0    4.0
1    2.0
2    2.0
dtype: float64
```

The mode() method returns the mode value of each column.

```
import pandas as pd
data = [[1, 1, 2], [6, 4, 2], [4, 2, 1], [4, 2, 3]]
df = pd.DataFrame(data)
print(df.mode())
```

Output

```
0 1 2
0 4 2 2
```

3.3 Object Creation

Creating a Series by passing a list of values, letting pandas create a default integer index.

Complete list of Object creation with examples:

```
import pandas as pd
import numpy as np
info = np.array(['P', 'a', 'n', 'd', 'a', 's'])
a = pd.Series(info)
print(a)
```

Output

```
0    P
1    a
2    n
3    d
4    a
5    s
dtype: object
```

Create a Series from dict

```
#import the pandas library
import pandas as pd
import numpy as np
info = {'x' : 0., 'y' : 1., 'z' : 2.}
a = pd.Series(info)
print (a)
```

Output

```
x    0.0
y    1.0
z    2.0
dtype: float64
```

Create a DataFrame from Dict of ndarrays/ Lists

```
# importing the pandas library
import pandas as pd
info = {'ID' : [101, 102, 103], 'Department' : ['B.Sc', 'B.Tech', 'M.Tech',]}
df = pd.DataFrame(info)
print (df)
```

Output

	ID	Department
0	101	B.Sc
1	102	B.Tech
2	103	M.Tech

3.4 Viewing Data

The **head()** function is used for Viewing data in pandas from starting rows and **tail()** is used to view the data from the bottom, by default it views the 5 rows.

Using **describe()** we can get the description of the data, like mean, standard deviation, percentage of the same data, etc.

a) Pandas DataFrame.head()

The head() method returns a specified number of rows, string from the top.

The head() method returns the first 5 rows if a number is not specified.

```
import pandas as pd
df = pd.read_csv('data.csv')
print(df.head())
```

Output

	Duration	Pulse	Maxpulse	Calories
0	60	110	130	409.1
1	60	117	145	479.0
2	60	103	135	340.0
3	45	109	175	282.4
4	45	117	148	406.0

b) Pandas DataFrame.tail()

The tail() method returns a specified number of last rows. The tail() method returns the last 5 rows if a number is not specified.

```
import pandas as pd
df = pd.read_csv('data.csv')
print(df.tail())
```

Output

	Duration	Pulse	Maxpulse	Calories
164	60	105	140	290.8
165	60	110	145	300.4
166	60	115	145	310.2
167	75	120	150	320.4
168	75	125	150	330.4

C) Pandas DataFrame.describe()

Example

Multiply the values for each row with the values from the previous row:

```
import pandas as pd
data = [10, 18, 11]
df = pd.DataFrame(data)
print(df.describe())
```

Output

```
count    3.000000
mean     13.000000
std       4.358899
min      10.000000
25%      10.500000
50%      11.000000
75%      14.500000
max      18.000000
```

d) Pandas DataFrame sort_values()

The sort_values() method sorts the DataFrame by the specified label.

Example

Sort the DataFrame by age:

```
import pandas as pd
data = {
    "age": [50, 40, 30, 40, 20, 10, 30],
    "qualified": [True, False, False, False, False, True, True]
}
df = pd.DataFrame(data)
newdf = df.sort_values(by='age')
newdf
```

Output

	age	qualified
5	10	True
4	20	False
2	30	False
6	30	True
1	40	False
3	40	False
0	50	True

3.5 Selection

There are multiple ways to select rows and columns from Pandas DataFrames.

- Selecting a single column
- Selection by label
- Selection by position

a) Column Selection

We can select any column from the DataFrame. Here is the code that demonstrates how to select a column from the DataFrame.

```
# importing the pandas library
import pandas as pd

info = {'one' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f']),
        'two' : pd.Series([1, 2, 3, 4, 5, 6, 7, 8], index=['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])}

d1 = pd.DataFrame(info)
print (d1 ['one'])
```

Output

```
a    1.0
b    2.0
c    3.0
d    4.0
e    5.0
f    6.0
g    NaN
h    NaN
Name: one, dtype: float64
```

Row Selection:

b) Selection by Label:

We can select any row by passing the row label to a loc function.

```
# importing the pandas library
import pandas as pd

info = {'one' : pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e']),
        'two' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f'])}

df = pd.DataFrame(info)
print (df.loc['b'])
```

Output

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

c) Selection by integer location:

The rows can also be selected by passing the integer location to an iloc function.

```
# importing the pandas library
import pandas as pd
info = {'one' : pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e']),
        'two' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f'])}
df = pd.DataFrame(info)
print (df.iloc[3])
```

Output

```
one    4.0
two    4.0
Name: d, dtype: float64
```

d) Slice Rows

It is another method to select multiple rows using ':' operator.

```
# importing the pandas library
import pandas as pd
info = {'one' : pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e']),
        'two' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f'])}
df = pd.DataFrame(info)
print (df[2:5])
```

Output

	one	two
c	3.0	3
d	4.0	4
e	5.0	5

3.6 Missing Data

Missing Data can occur when no information is provided for one or more items or for a whole unit. Missing Data is a very big problem in a real-life scenarios. Missing Data can also refer to as NA(Not Available) values in pandas. In DataFrame sometimes many datasets simply arrive with missing data, either because it exists and was not collected or it never existed. For Example, Suppose different users being surveyed may choose not to share their income, some users may choose not to share the address in this way many datasets went missing.

In Pandas missing data is represented by two value:

None: None is a Python singleton object that is often used for missing data in Python code.

NaN : NaN (an acronym for Not a Number), is a special floating-point value recognized by all systems that use the standard IEEE floating-point representation

To facilitate this convention, there are several useful functions for detecting, removing, and replacing null values in Pandas DataFrame:

- **isnull()**
- **notnull()**
- **dropna()**
- **fillna()**
- **replace()**
- **interpolate()**

a) Pandas DataFrame isnull() Method

The isnull() method returns a DataFrame object where all the values are replaced with a Boolean value True for NULL values, and otherwise False.

Example

Replace all values in the DataFrame with True for NULL values, otherwise False: In this example we use a .csv file called data.csv

```

# importing pandas as pd
import pandas as pd
# Creating the Series
sr = pd.Series([10, 25, 3, 25, 24, 6])
# Create the Index
index_ = ['Coca Cola', 'Sprite', 'Coke', 'Fanta', 'Dew', 'ThumbsUp']
# set the index
sr.index = index_
# Print the series
print(sr)
# detect missing values
result = sr.isnull()

# Print the result
print(result)

```

Output

```

Coca Cola    10
Sprite       25
Coke         3
Fanta        25
Dew          24
ThumbsUp     6
dtype: int64
Coca Cola    False
Sprite       False
Coke         False
Fanta        False
Dew          False
ThumbsUp     False
dtype: bool

```

b) Pandas DataFrame notnull() Method

Pandas dataframe.notnull() function detects existing/ non-missing values in the dataframe. All of the non-missing values gets mapped to true and missing values get mapped to false.

```
# importing pandas as pd
import pandas as pd
# Creating the Series
sr = pd.Series([10, 25, 3, 25, 24, 6])
# Create the Index
index_ = ['Coca Cola', 'Sprite', 'Coke', 'Fanta', 'Dew', 'ThumbsUp']
# set the index
sr.index = index_
# Print the series
print(sr)
# detect missing values
result = sr.notnull()

# Print the result
print(result)
```

Output

```
Coca Cola    10
Sprite       25
Coke         3
Fanta       25
Dew         24
ThumbsUp     6
dtype: int64
Coca Cola    True
Sprite       True
Coke         True
Fanta       True
Dew         True
ThumbsUp    True
dtype: bool
```

c) Pandas DataFrame dropna() Method

The dropna() function is used to remove missing values.

```
# importing the pandas library
import pandas as pd

info = {'one' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f']),
        'two' : pd.Series([1, 2, 3, 4, 5, 6, 7, 8], index=['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])}

d1 = pd.DataFrame(info)
print (d1)
d1.dropna()
```

Output

	one	two		one	two
a	1.0	1	a	1.0	1
b	2.0	2	b	2.0	2
c	3.0	3	c	3.0	3
d	4.0	4	d	4.0	4
e	5.0	5	e	5.0	5
f	6.0	6			
g	NaN	7			
h	NaN	8	f	6.0	6

d) Pandas DataFrame fillna() Method

In order to fill null values in a datasets, we use fillna(), replace() and interpolate() function these function replace NaN values with some value of their own. All these function help in filling a null values in datasets of a DataFrame.

```

# importing the pandas library
import pandas as pd

info = {'one' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f']),
        'two' : pd.Series([1, 2, 3, 4, 5, 6, 7, 8], index=['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])}

d1 = pd.DataFrame(info)
print (d1)
d1.fillna(0)

```

Output

	one	two
a	1.0	1
b	2.0	2
c	3.0	3
d	4.0	4
e	5.0	5
f	6.0	6
g	NaN	7
h	NaN	8

	one	two
a	1.0	1
b	2.0	2
c	3.0	3
d	4.0	4
e	5.0	5
f	6.0	6
g	0.0	7
h	0.0	8

e) Pandas DataFrame interpolate () Method

Pandas dataframe.interpolate() function is basically used to fill NA values in the dataframe or series. But, this is a very powerful function to fill the missing values. It uses various interpolation

technique to fill the missing values rather than hard-coding the value.

Parameters :

method : {'linear', 'time', 'index', 'values', 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'barycentric', 'krogh', 'polynomial', 'spline', 'piecewise_polynomial', 'from_derivatives', 'pchip', 'akima'}

limit_direction : {'forward', 'backward', 'both'}, default 'forward'

```
import pandas as pd
import numpy as np
s = pd.Series([0, 1, np.nan, 3])
s
s.interpolate()
```

Output

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



```

# importing pandas as pd
import pandas as pd

# Creating the dataframe
df = pd.DataFrame({"A": [12, 4, 5, None, 1],
                  "B": [None, 2, 54, 3, None],
                  "C": [20, 16, None, 3, 8],
                  "D": [14, 3, None, None, 6]})

# Print the dataframe
df

# to interpolate the missing values
df.interpolate(method = 'linear', limit_direction = 'both')

```

Output

	A	B	C	D
0	12.0	2.0	20.0	14.0
1	4.0	2.0	16.0	3.0
2	5.0	54.0	9.5	4.0
3	3.0	3.0	3.0	5.0
4	1.0	3.0	8.0	6.0

f) Pandas DataFrame replace() Method

The replace() method replaces the specified value with another specified value. The replace() method searches the entire DataFrame and replaces every case of the specified value.

```

import pandas as pd
data = {
    "name": ["Bill", "Bob", "Betty"],
    "age": [50, 50, 30],
    "qualified": [True, False, False]
}
df = pd.DataFrame(data)

newdf = df.replace(50, 60)

print(newdf)

```

Output

	name	age	qualified
0	Bill	60	True
1	Bob	60	False
2	Betty	30	False

3.7 Operations

Pandas DataFrame.mean()

The mean() method returns a Series with the mean value of each column.

Mean, Median, and Mode:

Mean - The average value

Median - The mid point value

Mode - The most common value

Example

Return the average (mean) value for each column:

```
import pandas as pd
data = [[1, 1, 2], [6, 4, 2], [4, 2, 1], [4, 2, 3]]
df = pd.DataFrame(data)
print(df.mean())
```

Output

```
0    3.75
1    2.25
2    2.00
dtype: float64
```

Object Creation

Creating a Series by passing a list of values, letting pandas create a default integer index.

Complete list of Object creation with examples:

```
import pandas as pd
import numpy as np
info = np.array(['P', 'a', 'n', 'd', 'a', 's'])
a = pd.Series(info)
print(a)
```

Output

```
0    P
1    a
2    n
3    d
4    a
5    s
dtype: object
```

Create a Series from dict

```
#import the pandas library
import pandas as pd
import numpy as np
info = {'x' : 0., 'y' : 1., 'z' : 2.}
a = pd.Series(info)
print (a)
```

Output

```
x    0.0
y    1.0
z    2.0
dtype: float64
```

Create a DataFrame from Dict of ndarrays/ Lists

```
# importing the pandas library
import pandas as pd
info = {'ID' :[101, 102, 103], 'Department' :['B.Sc', 'B.Tech', 'M.Tech',]}
df = pd.DataFrame(info)
print (df)
```

Output

	ID	Department
0	101	B.Sc
1	102	B.Tech
2	103	M.Tech

3.8 Merge

The merge() method updates the content of two DataFrame by merging them together, using the specified method(s).

```

# Create Population DataFrame
df1 = pd.DataFrame({
    'Country': ['America', 'Indonesia', 'France'],
    'Location': ['New York', 'Jakarta', 'Paris'],
    'Population': [731800, 575030, 183305]
})

# Create Income DataFrame
df2 = pd.DataFrame({
    'Country': ['America', 'America', 'Indonesia', 'India', 'France', 'Greece'],
    'Location': ['New York', 'Chicago', 'Jakarta', 'Mumbai', 'Paris', 'Yunani'],
    'Income': [1000, 1500, 1400, 1100, 900, 1200]
})

# Merge Dataframe
merged_df = pd.merge(df1, df2, on='Country')
merged_df

```

Output

	Country	Location_x	Population	Location_y	Income
0	America	New York	731800	New York	1000
1	America	New York	731800	Chicago	1500
2	Indonesia	Jakarta	575030	Jakarta	1400
3	France	Paris	183305	Paris	900

```
# import pandas library
import pandas as pd
# create a series
a = pd.Series(["C++", "JAVA",
               "PYTHON", "DBMS",
               "C#"], name = "subjects")
# create a series
b = pd.Series(["30", "60",
               "90", "56",
               "50"], name = "marks")
# merge both series
df = pd.merge(a, b, right_index = True,
              left_index = True)
# show the dataframe
df
```

Output

	subjects	marks
0	C++	30
1	JAVA	60
2	PYTHON	90
3	DBMS	56
4	C#	50

3.9 Grouping

The `groupby()` method allows you to group your data and execute functions on these groups.

Example

Find the average `co2` consumption for each car brand:

```
import pandas as pd
data = {
    'co2': [95, 90, 99, 104, 105, 94, 99, 104],
    'model': ['Citigo', 'Fabia', 'Fiesta', 'Rapid', 'Focus', 'Mondeo', 'Octavia', 'B-Max'],
    'car': ['Skoda', 'Skoda', 'Ford', 'Skoda', 'Ford', 'Ford', 'Skoda', 'Ford']
}
df = pd.DataFrame(data)
print(df.groupby(["car"]).mean())
```

Output

	co2
car	
Ford	100.5
Skoda	97.0

3.10 Reshaping

Pandas has two methods that aid in reshaping the data into a desired format. Pandas has two methods namely, `melt()` and `pivot()`, to reshape the data.

```
# import pandas library
import pandas as pd
# make an array
array = [2, 4, 6, 8, 10, 12]
# create a series
series_obj = pd.Series(array)
# convert series object into array
arr = series_obj.values
# reshaping series
reshaped_arr = arr.reshape((3, 2))
# show
reshaped_arr
```

Output

```
array([[ 2,  4],
       [ 6,  8],
       [10, 12]])
```

The melt() method reshapes the DataFrame into a long table with one row for each column.

```
import pandas as pd
d1 = {"Name": ["Pankaj", "Lisa", "David"],
      "ID": [1, 2, 3],
      "Role": ["CEO", "Editor", "Author"]}
df = pd.DataFrame(d1)
print(df)
df_melted = pd.melt(df, id_vars=["ID"], value_vars=["Name", "Role"])
print(df_melted)
```


Output

```
      Name  ID  Role
0  Pankaj   1   CEO
1    Lisa   2  Editor
2   David   3  Author
      ID variable  value
0     1      Name  Pankaj
1     2      Name   Lisa
2     3      Name   David
3     1      Role    CEO
4     2      Role  Editor
5     3      Role  Author
```

`pivot()`

The `pivot()` function is used to reshaped a given DataFrame organized by given index / column values. This function does not support data aggregation, multiple values will result in a MultiIndex in the columns.

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

```
df = pd.DataFrame({'fff': ['one', 'one', 'one', 'two', 'two',
                           'two'],
                   'bbb': ['P', 'Q', 'R', 'P', 'Q', 'R'],
                   'baa': [2, 3, 4, 5, 6, 7],
                   'zzz': ['h', 'i', 'j', 'k', 'l', 'm']})
df
```

Output

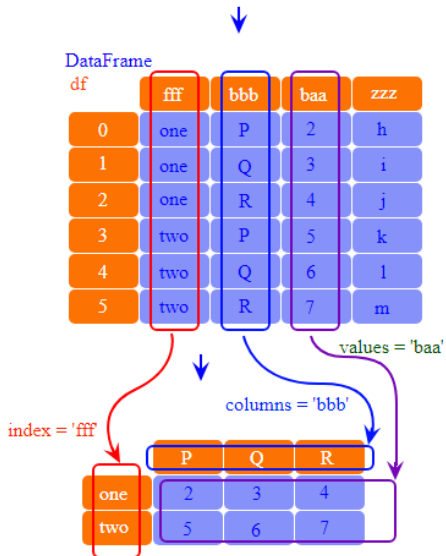
DataFrame

df	fff	bbb	baa	zzz
0	one	P	2	h
1	one	Q	3	i
2	one	R	4	j
3	two	P	5	k
4	two	Q	6	l
5	two	R	7	m

```
df.pivot(index='fff', columns='bbb', values='baa')
```

Output

```
df.pivot(index='fff', columns='bbb', values='baa')
```



3.10 Time Series

Series is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). The axis labels are collectively referred to as the index.

Time series data is a sequence of data points in chronological order that is used by businesses to analyze past data and make future predictions. These data points are a set of observations at specified times and equal intervals, typically with a datetime index and corresponding value. Common examples of time series data in our day-to-day lives include:

- Measuring weather temperatures
- Measuring the number of taxi rides per month
- Predicting a company's stock prices for the next day

Creating a time series DataFrame

To work with time series data in pandas, we use a DatetimeIndex as the index for our DataFrame (or Series). Let's see how to do this with our OPSD data set. First, we use the `read_csv()` function to read the data into a DataFrame, and then display its shape.

```
opsd_daily = pd.read_csv('opsd_germany_daily.csv')
opsd_daily.shape
```

Output

```
(4383, 5)
```

The DataFrame has 4383 rows, covering the period from January 1, 2006 through December 31, 2017. To see what the data looks like, let's use the head() and tail() methods to display the first three and last three rows.

```
opsd_daily.head(3)
```

Output

	Date	Consumption	Wind	Solar	Wind+Solar
0	2006-01-01	1069.184	NaN	NaN	NaN
1	2006-01-02	1380.521	NaN	NaN	NaN
2	2006-01-03	1442.533	NaN	NaN	NaN

```
opsd_daily.tail(3)
```

Output

	Date	Consumption	Wind	Solar	Wind+Solar
4380	2017-12-29	1295.08753	584.277	29.854	614.131
4381	2017-12-30	1215.44897	721.247	7.467	728.714
4382	2017-12-31	1107.11488	721.176	19.980	741.156

Next, let's check out the data types of each column.

```
opsd_daily.dtypes
```

Output

```
Date datetime64[ns]
Consumption float64
Wind float64
Solar float64
Wind+Solar float64
dtype: object
```

```
import pandas as pd
from datetime import datetime
import numpy as np

range_date = pd.date_range(start = '1/1/2019', end = '1/08/2019',
                           freq = 'Min')

print(range_date)
```

Output

```
DatetimeIndex(['2019-01-01 00:00:00', '2019-01-01 00:01:00',
               '2019-01-01 00:02:00', '2019-01-01 00:03:00',
               '2019-01-01 00:04:00', '2019-01-01 00:05:00',
               '2019-01-01 00:06:00', '2019-01-01 00:07:00',
               '2019-01-01 00:08:00', '2019-01-01 00:09:00',
               ...,
               '2019-01-07 23:51:00', '2019-01-07 23:52:00',
               '2019-01-07 23:53:00', '2019-01-07 23:54:00',
               '2019-01-07 23:55:00', '2019-01-07 23:56:00',
               '2019-01-07 23:57:00', '2019-01-07 23:58:00',
               '2019-01-07 23:59:00', '2019-01-08 00:00:00'],
              dtype='datetime64[ns]', length=10081, freq='T')
```

Here in this code, we have created the timestamp on the basis of minutes for date ranges from 1/1/2019 – 8/1/2019. We can vary

the frequency by hours to minutes or seconds. This function will help you to track the record of data stored per minute. As we can see in the output the length of the datetime stamp is 10081. Remember pandas use data type as datetime64[ns].

We can use the `to_datetime()` function to create Timestamps from strings in a wide variety of date/time formats. Let's import pandas and convert a few dates and times to Timestamps.

```
import pandas as pd
pd.to_datetime('2018-01-15 3:45pm')
pd.to_datetime('7/8/1952')
```

Output

```
Timestamp('1952-07-08 00:00:00')
```

3.11 Categorical

Object Creation

Categorical object can be created in multiple ways. The different ways have been described below –

category

By specifying the dtype as "category" in pandas object creation.

```
import pandas as pd

s = pd.Series(["a","b","c","a"], dtype="category")
print(s)
```

Output

```
0    a
1    b
2    c
3    a
dtype: category
Categories (3, object): ['a', 'b', 'c']
```

pd.Categorical

Using the standard pandas Categorical constructor, we can create a category object.

`pandas.Categorical(values, categories, ordered)`

```
import pandas as pd

cat = pd.Categorical(['a', 'b', 'c', 'a', 'b', 'c'])
print(cat)
```

Output

```
['a', 'b', 'c', 'a', 'b', 'c']
Categories (3, object): ['a', 'b', 'c']
```

Appending New Categories

Using the `Categorical.add.categories()` method, new categories can be appended.

```
import pandas as pd

s = pd.Series(["a","b","c","a"], dtype="category")
s = s.cat.add_categories([4])
print(s.cat.categories)
```

Output

```
Index(['a', 'b', 'c', 4], dtype='object')
```

Removing Categories

Using the `Categorical.remove_categories()` method, unwanted categories can be removed.

```
import pandas as pd

s = pd.Series(["a","b","c","a"], dtype="category")
print ("Original object:")
print (s)

print ("After removal:")
print(s.cat.remove_categories("a"))
```


Output

```
Original object:
0    a
1    b
2    c
3    a
dtype: category
Categories (3, object): ['a', 'b', 'c']
After removal:
0    NaN
1     b
2     c
3    NaN
dtype: category
Categories (2, object): ['b', 'c']
```

3.12 Getting Data In/ Out

a) Using the pandas `read_csv()` and `.to_csv()` Functions

A comma-separated values (CSV) file is a plaintext file with a `.csv` extension that holds tabular data. This is one of the most popular file formats for storing large amounts of data. Each row of the CSV file represents a single table row. The values in the same row are by default separated with commas, but you could change the separator to a semicolon, tab, space, or some other character.

Write a CSV File

You can save your pandas DataFrame as a CSV file with `.to_csv()`:

```
df.to_csv('data.csv')
```

That's it! You've created the file `data.csv` in your current working directory

This text file contains the data separated with commas. The first column contains the row labels. In some cases, you'll find them irrelevant. If you don't want to keep them, then you can pass the argument `index=False` to `.to_csv()`.

Read a CSV File

Once your data is saved in a CSV file, you'll likely want to load and use it from time to time. You can do that with the pandas `read_csv()` function:

```
df = pd.read_csv('data.csv', index_col=0)
```

df

b) Using pandas to Write and Read Excel Files

Microsoft Excel is probably the most widely-used spreadsheet software. While older versions used binary `.xls` files, Excel 2007 introduced the new XML-based `.xlsx` file. You can read and write Excel files in pandas, similar to CSV files. However, you'll need to install the following Python packages first:

- `xlwt` to write to `.xls` files
- `openpyxl` or `XlsxWriter` to write to `.xlsx` files
- `xlrd` to read Excel files

You can install them using `pip` with a single command:

```
pip install xlwt openpyxl xlsxwriter xlrd
```

You can also use Conda:

conda install xlwt openpyxl xlswriter xlrd

Write an Excel File

Once you have those packages installed, you can save your DataFrame in an Excel file with `.to_excel()`:

```
df.to_excel('data.xlsx')
```

The argument 'data.xlsx' represents the target file and, optionally, its path. The above statement should create the file data.xlsx in your current working directory. That file should look like this:

	A	B	C	D	E	F	G
		COUNTRY	POP	AREA	GDP	CONT	IND_DAY
2	CHN	China	1398.72	9596.96	12234.78	Asia	
3	IND	India	1351.16	3287.26	2575.67	Asia	1947-08-15
4	USA	US	329.74	9833.52	19485.39	N.America	1776-07-04
5	IDN	Indonesia	268.07	1910.93	1015.54	Asia	1945-08-17
6	BRA	Brazil	210.32	8515.77	2055.51	S.America	1822-09-07
7	PAK	Pakistan	205.71	881.91	302.14	Asia	1947-08-14
8	NGA	Nigeria	200.96	923.77	375.77	Africa	1960-10-01
9	BGD	Bangladesh	167.09	147.57	245.63	Asia	1971-03-26
10	RUS	Russia	146.79	17098.25	1530.75		1992-06-12
11	MEX	Mexico	126.58	1964.38	1158.23	N.America	1810-09-16
12	JPN	Japan	126.22	377.97	4872.42	Asia	
13	DEU	Germany	83.02	357.11	3693.2	Europe	
14	FRA	France	67.02	640.68	2582.49	Europe	1789-07-14
15	GBR	UK	66.44	242.5	2631.23	Europe	
16	ITA	Italy	60.36	301.34	1943.84	Europe	
17	ARG	Argentina	44.94	2780.4	637.49	S.America	1816-07-09
18	DZA	Algeria	43.38	2381.74	167.56	Africa	1962-07-05
19	CAN	Canada	37.59	9984.67	1647.12	N.America	1867-07-01
20	AUS	Australia	25.47	7692.02	1408.68	Oceania	
21	KAZ	Kazakhstan	18.53	2724.9	159.41	Asia	1991-12-16

The first column of the file contains the labels of the rows, while the other columns store data.

Read an Excel File

You can load data from Excel files with `read_excel()`:

```
df = pd.read_excel('data.xlsx', index_col=0)
```

```
df
```

UNIT-IV

Matplotlib: Features of Matplotlib, anatomy and customization of a Matplotlib Plot.

Plotting and Plot Customization: Creating a plot and figure, Axes, Subplots, Changing Figure sizes.

Customizing Plots: Plot Titles, Labels and Legends, Text, Ticks, Layouts. Changing Colour of Elements, Visualization Examples.

4.1 Introduction to Matplotlib

Matplotlib is the basic plotting library of the Python programming language. Among Python visualization packages, it is the most widely used.

Matplotlib is exceptionally fast at a variety of operations. In addition, it can export visualizations to all popular image formats, including PDF, SVG, JPG, PNG, BMP, and GIF.

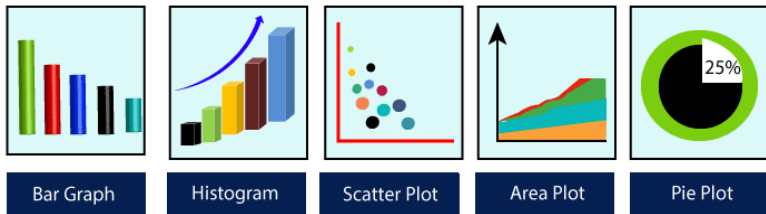
It can create line graphs, scatter plots, histograms, bar charts, error charts, pie charts, box plots, and many other visualization styles. Along with that, 3D charting is also possible with Matplotlib.

Matplotlib serves as the foundation for several Python libraries and was used to build Pandas and Seaborn, for example. They make it possible to access Matplotlib's methods with less code.

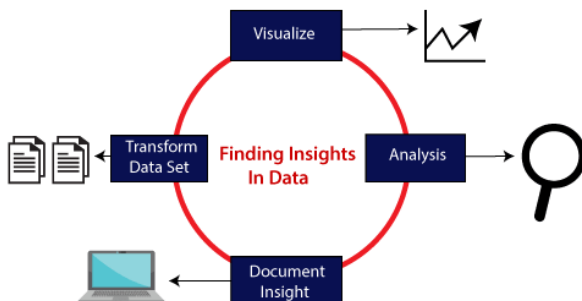
Matplotlib, an open-source plotting toolkit for the Python programming language, has become the most extensively used

plotting library. For example, it was used to visualize data during the 2008 landing of the Phoenix spacecraft.

There are five key plots that are used for data visualization.



There are five phases which are essential to make the decision for the organization:



Visualize: We analyze the raw data, which means it makes complex data more accessible, understandable, and more usable.

Analysis: Data analysis is defined as cleaning, inspecting, transforming, and modeling data to derive useful information.

Document Insight: Document insight is the process where the

useful data or information is organized in the document in the standard format.

Transform Data Set: Standard data is used to make the decision more effectively.

4.2 How to install matplotlib in Python?

Matplotlib is a Python library that helps to plot graphs. It is used in data visualization and graphical plotting. To use matplotlib, we need to install it.

Step 1 – Make sure Python and pip is preinstalled on your system
Type the following commands in the command prompt to check if python and pip is installed on your system. To check Python

```
python --version
```

If python is successfully installed, the version of python installed on your system will be displayed.

To check pip

```
pip -V
```

The version of pip will be displayed, if it is successfully installed on your system.

Step 2 – Install Matplotlib

Matplotlib can be installed using pip. The following command is run in the command prompt to install Matplotlib.

```
pip install matplotlib
```

This command will start downloading and installing packages related to the matplotlib library. Once done, the message of successful installation will be displayed.

Step 3 – Check if it is installed successfully

To verify that matplotlib is successfully installed on your system, execute the following command in the command prompt. If matplotlib is successfully installed, the version of matplotlib installed will be displayed.

```
import matplotlib  
matplotlib.__version__
```

4.3 Why is Matplotlib so Popular?

Matplotlib's popularity can be explained as follows:

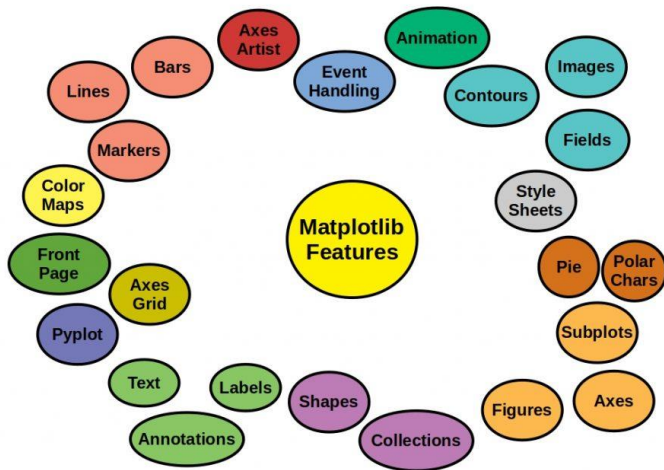
- For beginners, it is simple and straightforward.
- It is open-source and free.
- Matplotlib is a highly customized and robust library.
- Matplotlib is good at working with data frames and arrays. Figures and axes are treated as objects. It has

several stateful plotting APIs. As a result, methods like `plot()` can be used without any parameters.

- Those who have used MATLAB or other graph plotting tools will find Matplotlib easy to use.
- Matplotlib can be used in various contexts, such as Python scripts, the Python and iPython shells, and Jupyter Notebooks.
- Matplotlib is a 2-D plotting library. However, several extensions can produce complex visualizations such as 3-D graphs, etc.
- It offers high-quality photos and plots in various formats, including png, pdf, and pgf.
- Controls numerous aspects of a figure, including DPI, figure color, and figure size.

4.4 Features of Matplotlib

The library offers a wide range of visualization functions. Some of them are listed in the figure below.



Matplotlib is designed to effectively visualize the results of mathematical calculations. Visualization is an efficient and important data analysis tool.

The library is able to generate all the usual diagrams and figures by default. It is even possible to create animations that can be used to better understand the flow of certain algorithms.

4.5 Customization of a Matplotlib Plot

Let's create a simple plot. We will be plotting two lists containing the X, Y coordinates for the plot.

```
import matplotlib.pyplot as plt

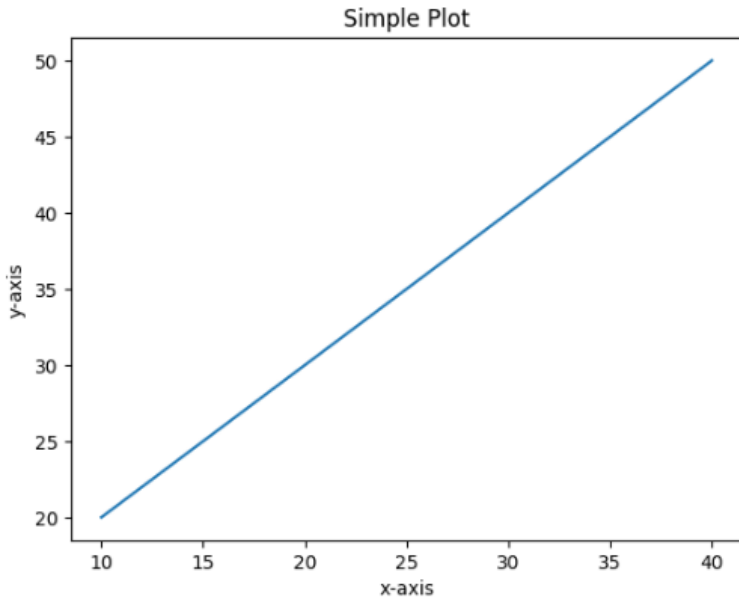
# initializing the data
x = [10, 20, 30, 40]
y = [20, 30, 40, 50]

# plotting the data
plt.plot(x, y)

# Adding the title
plt.title("Simple Plot")

# Adding the labels
plt.ylabel("y-axis")
plt.xlabel("x-axis")
plt.show()
```

Output



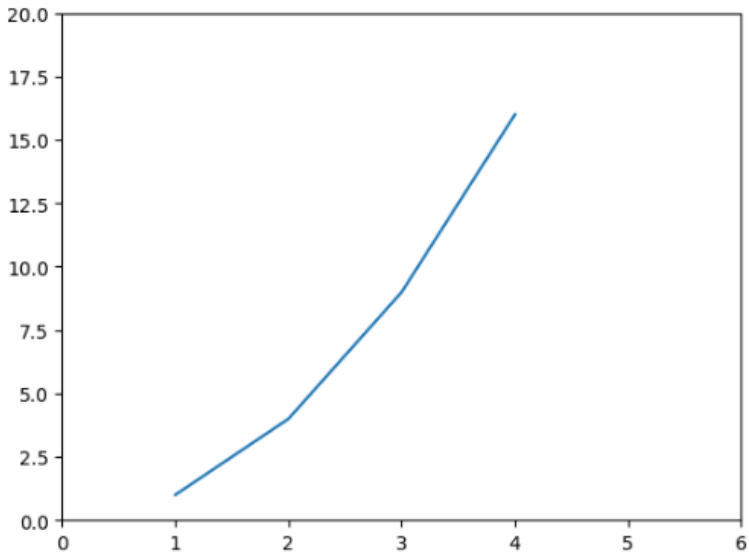
In the above example, the elements of X and Y provides the coordinates for the x axis and y axis and a straight line is plotted against those coordinates.

a) Pyplot

Pyplot is a Matplotlib module that provides a MATLAB-like interface. Pyplot provides functions that interact with the figure i.e. creates a figure, decorates the plot with labels, and creates a plotting area in a figure.

```
# Python program to show pyplot module
import matplotlib.pyplot as plt
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
plt.axis([0, 6, 0, 20])
plt.show()
```

Output



b) Figure class

Figure class is the top-level container that contains one or more axes. It is the overall window or page on which everything is drawn.

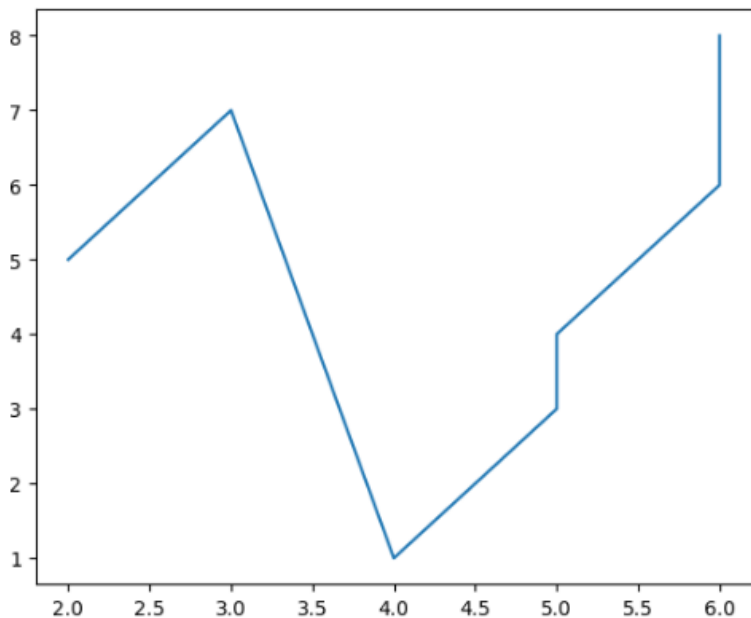
```
# Python program to show pyplot module
import matplotlib.pyplot as plt
from matplotlib.figure import Figure

# Creating a new figure with width = 5 inches
# and height = 4 inches
fig = plt.figure(figsize =(5, 4))

# Creating a new axes for the figure
ax = fig.add_axes([1, 1, 1, 1])

# Adding the data to be plotted
ax.plot([2, 3, 4, 5, 5, 6, 6],
        [5, 7, 1, 3, 4, 6 ,8])
plt.show()
```

Output

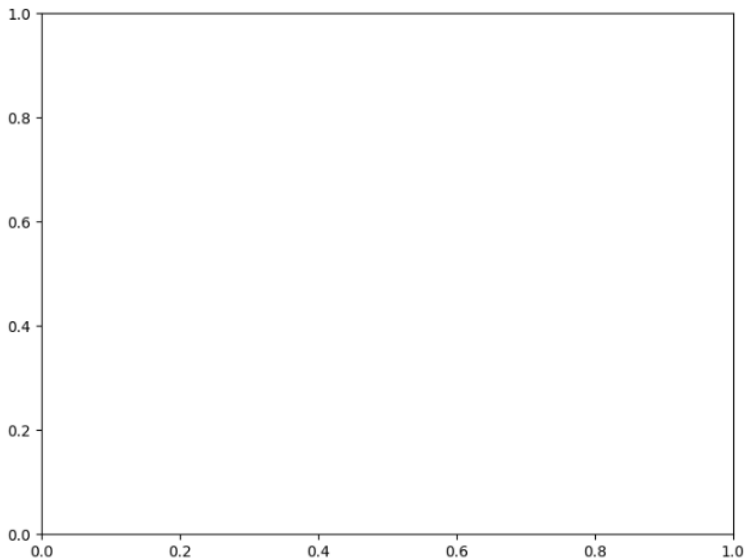


c) Axes Class

Axis class is the most basic and flexible unit for creating subplots. A given figure may contain many axes, but a given axis can only be present in one figure. The `axes()` function creates the axis object. Let's see the below example.

```
# Python program to show pyplot module
import matplotlib.pyplot as plt
from matplotlib.figure import Figure
# Creating the axis object with argument as
# [left, bottom, width, height]
ax = plt.axes([1, 1, 1, 1])
```

Output



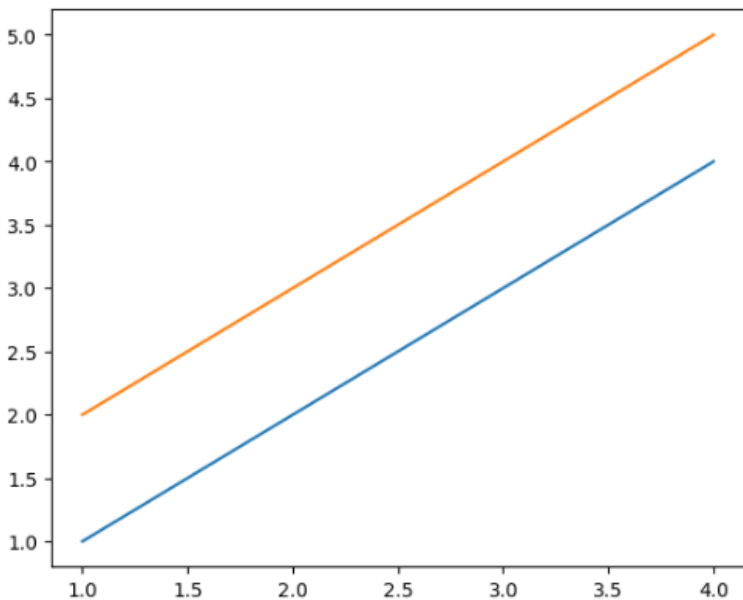
```
# Python program to show pyplot module
import matplotlib.pyplot as plt
from matplotlib.figure import Figure
fig = plt.figure(figsize = (5, 4))

# Adding the axes to the figure
ax = fig.add_axes([1, 1, 1, 1])

# plotting 1st dataset to the figure
ax1 = ax.plot([1, 2, 3, 4], [1, 2, 3, 4])

# plotting 2nd dataset to the figure
ax2 = ax.plot([1, 2, 3, 4], [2, 3, 4, 5])
plt.show()
```

Output



d) Subplot

Display Multiple Plots

With the subplot() function you can draw multiple plots in one figure:

```
import matplotlib.pyplot as plt
import numpy as np

#plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

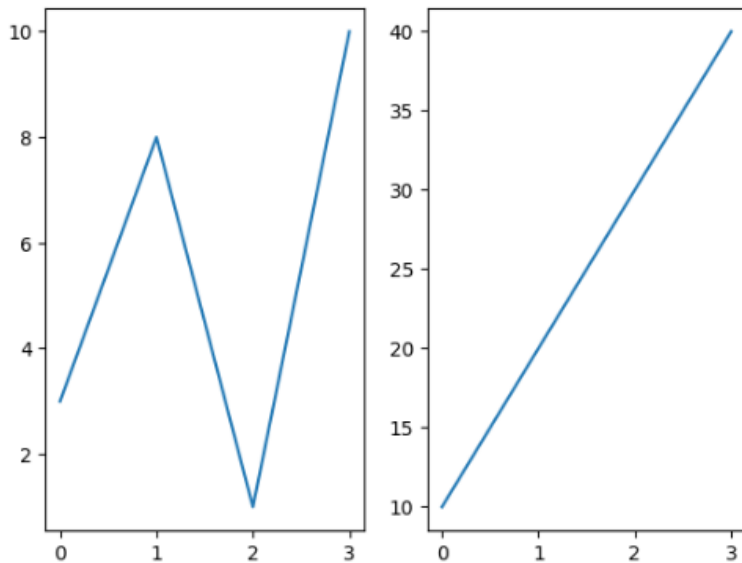
plt.subplot(1, 2, 1)
plt.plot(x,y)

#plot 2:
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(1, 2, 2)
plt.plot(x,y)

plt.show()
```

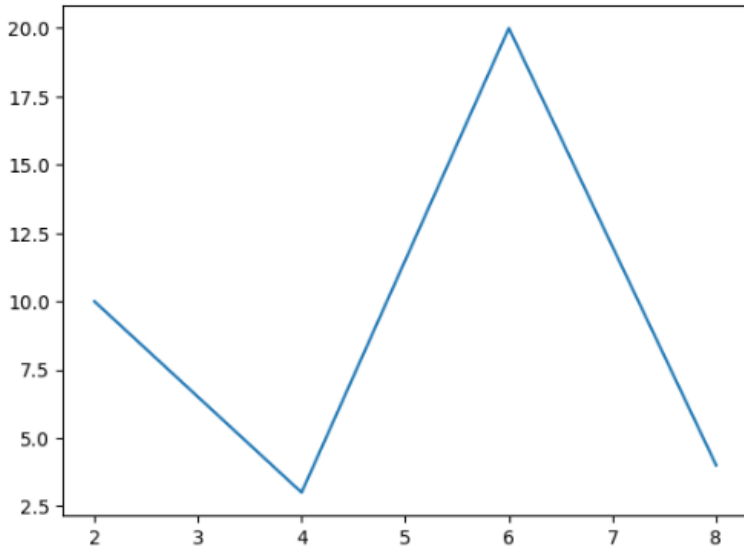
Output



e) How to Change Plot Size in Matplotlib with `plt.figure()`

```
import matplotlib.pyplot as plt  
x = [2,4,6,8]  
y = [10,3,20,4]  
plt.plot(x,y)  
plt.show()
```

Output



We can change the size of the plot above using the `figsize()` attribute of the `figure()` function.

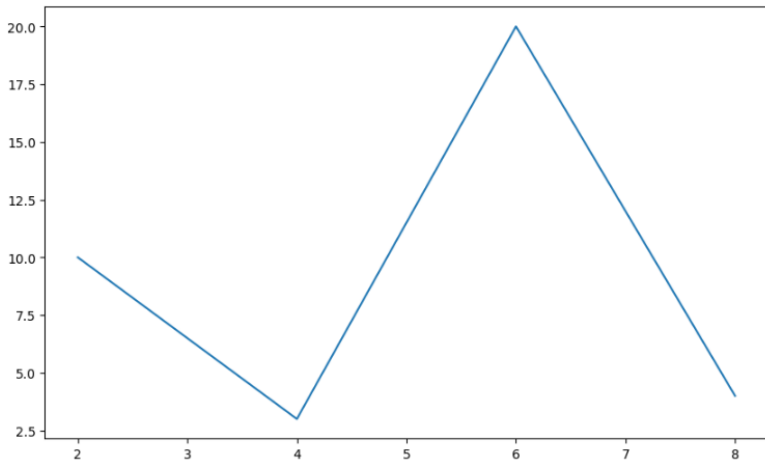
The `figsize()` attribute takes in two parameters — one for the width and the other for the height.

Here's what the syntax looks like:

`figure(figsize=(WIDTH_SIZE,HEIGHT_SIZE))`

```
import matplotlib.pyplot as plt
x = [2,4,6,8]
y = [10,3,20,4]
plt.figure(figsize=(10,6))
plt.plot(x,y)
plt.show()
```

Output

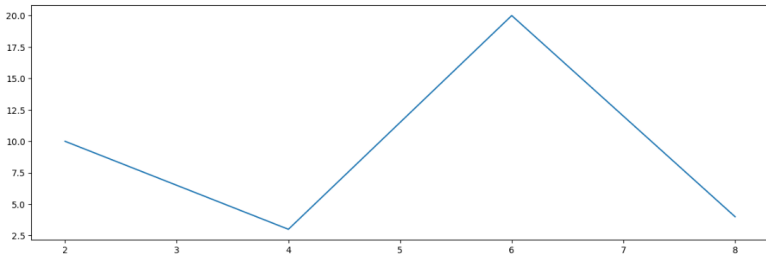


f) How to Change Plot Width in Matplotlib with `set_figwidth()`

You can use the `set_figwidth()` method to change the width of a plot.

```
import matplotlib.pyplot as plt
x = [2,4,6,8]
y = [10,3,20,4]
plt.figure().set_figwidth(15)
plt.plot(x,y)
plt.show()
```

Output

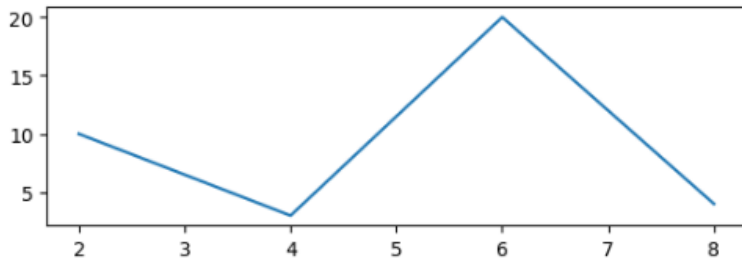


g) How to Change Plot Height in Matplotlib with `set_figheight()`

You can use the `set_figheight()` method to change the height of a plot.

```
import matplotlib.pyplot as plt
x = [2,4,6,8]
y = [10,3,20,4]
plt.figure().set_figheight(2)
plt.plot(x,y)
plt.show()
```

Output

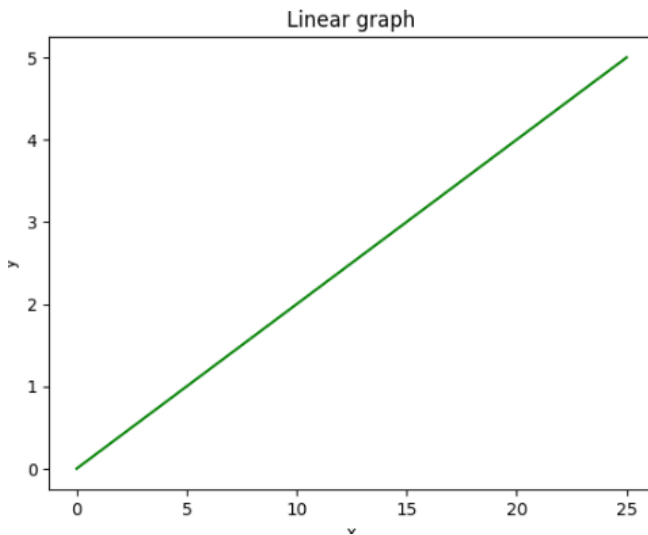


4.6 Matplotlib.pyplot.title()

The title() method in matplotlib module is used to specify title of the visualization depicted and displays the title using various attributes.

```
# importing module
import matplotlib.pyplot as plt
# assigning x and y coordinates
y = [0,1,2,3,4,5]
x= [0,5,10,15,20,25]
# depicting the visualization
plt.plot(x, y, color='green')
plt.xlabel('x')
plt.ylabel('y')
# displaying the title
plt.title("Linear graph")
plt.show()
```

Output



4.7 Matplotlib.pyplot.legend()

A legend is an area describing the elements of the graph. In the matplotlib library, there's a function called legend() which is used to Place a legend on the axes.

```
import numpy as np
import matplotlib.pyplot as plt

# X-axis values
x = [1, 2, 3, 4, 5]

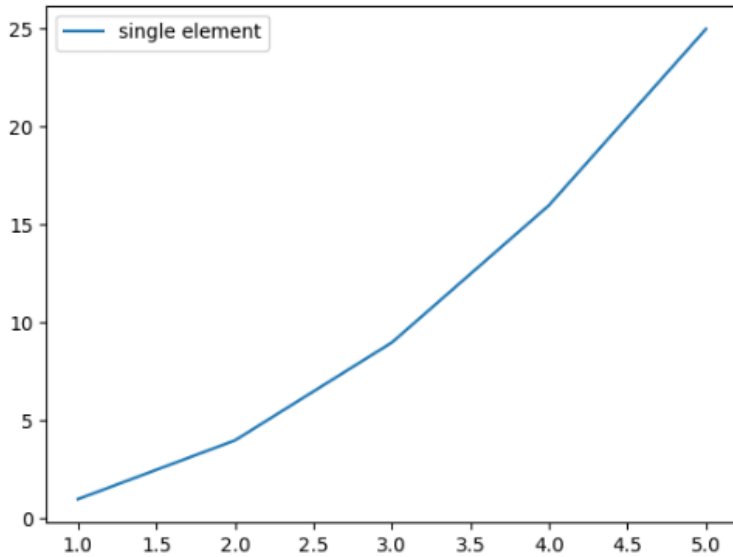
# Y-axis values
y = [1, 4, 9, 16, 25]

# Function to plot
plt.plot(x, y)

# Function add a legend
plt.legend(['single element'])

# function to show the plot
plt.show()
```

Output



4.8 Text

text() is the main function we can use to add text to a plot.

```
import numpy as np
import matplotlib.pyplot as plt

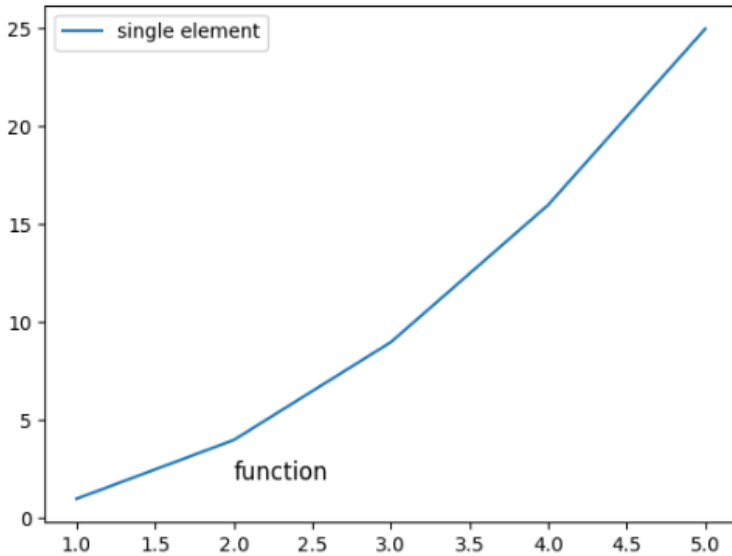
# X-axis values
x = [1, 2, 3, 4, 5]

# Y-axis values
y = [1, 4, 9, 16, 25]

# Function to plot
plt.plot(x, y)

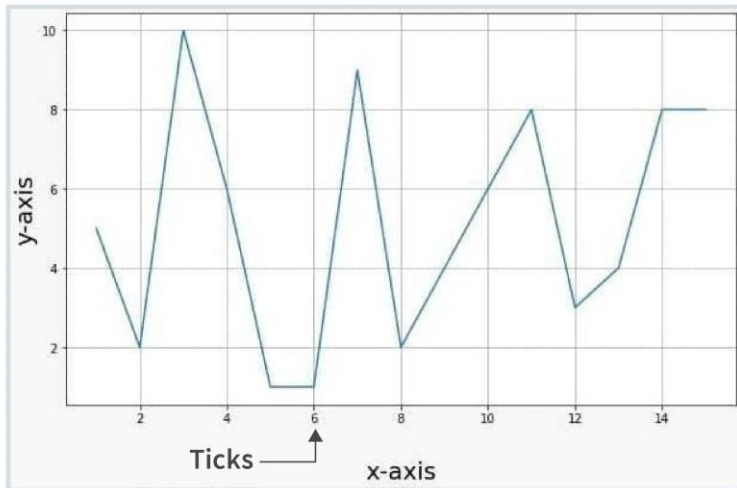
# Function add a legend
plt.legend(['single element'])
plt.text(2, 2, "function", fontsize=12)
# function to show the plot
plt.show()
```

Output



4.9 Ticks in Matplotlib

Ticks are the value on the axis to show the coordinates on the graph. It is the value on the axes by which we can visualize where will a specific coordinate lie on a graph. Whenever we plot a graph, ticks values are adjusted according to the data, which is sufficient in common situations, but it is not ideal whenever we plot data on a graph. So we will discuss the ticks in matplotlib and how we can customize it.



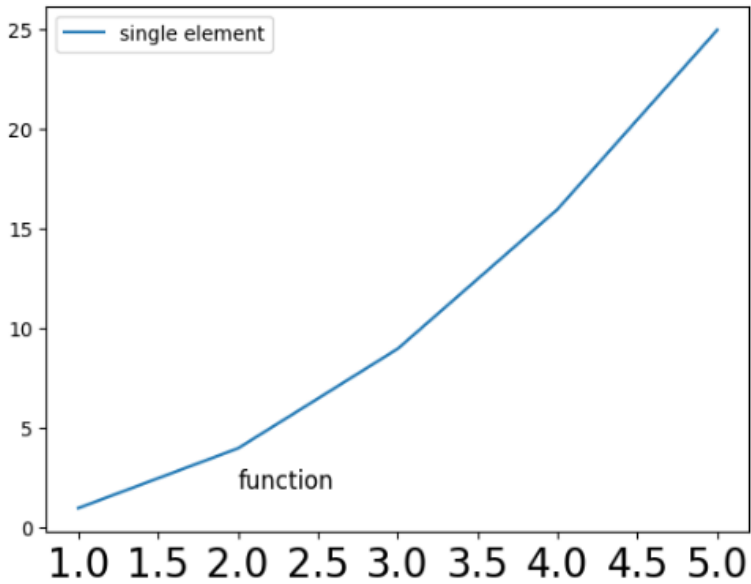
```
import numpy as np
import matplotlib.pyplot as plt

# X-axis values
x = [1, 2, 3, 4, 5]

# Y-axis values
y = [1, 4, 9, 16, 25]

# Function to plot
plt.plot(x, y)

# Function add a legend
plt.legend(['single element'])
plt.text(2, 2, "function", fontsize=12)
plt.xticks(fontsize=20)
# function to show the plot
plt.show()
```



4.10 Layouts

You can use the `tight_layout()` function in Matplotlib to automatically adjust the padding between and around subplots.

```
import matplotlib.pyplot as plt

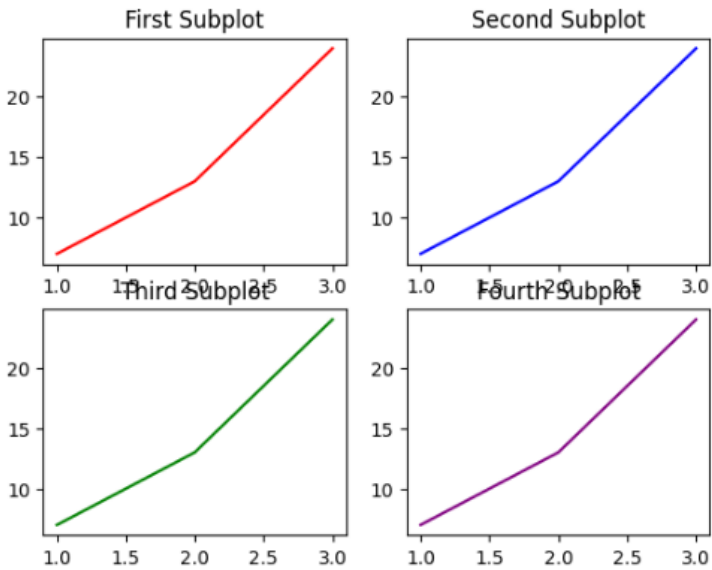
#define data
x = [1, 2, 3]
y = [7, 13, 24]

#define layout for subplots
fig, ax = plt.subplots(2, 2)

#define subplot titles
ax[0, 0].plot(x, y, color='red')
ax[0, 1].plot(x, y, color='blue')
ax[1, 0].plot(x, y, color='green')
ax[1, 1].plot(x, y, color='purple')

#add title to each subplot
ax[0, 0].set_title('First Subplot')
ax[0, 1].set_title('Second Subplot')
ax[1, 0].set_title('Third Subplot')
ax[1, 1].set_title('Fourth Subplot')
```

Output



Notice that there is minimal padding between the subplots, which causes the titles to overlap in some places. By specifying `fig.tight_layout()` we can automatically adjust the padding between and around subplots:

```
import matplotlib.pyplot as plt

#define data
x = [1, 2, 3]
y = [7, 13, 24]

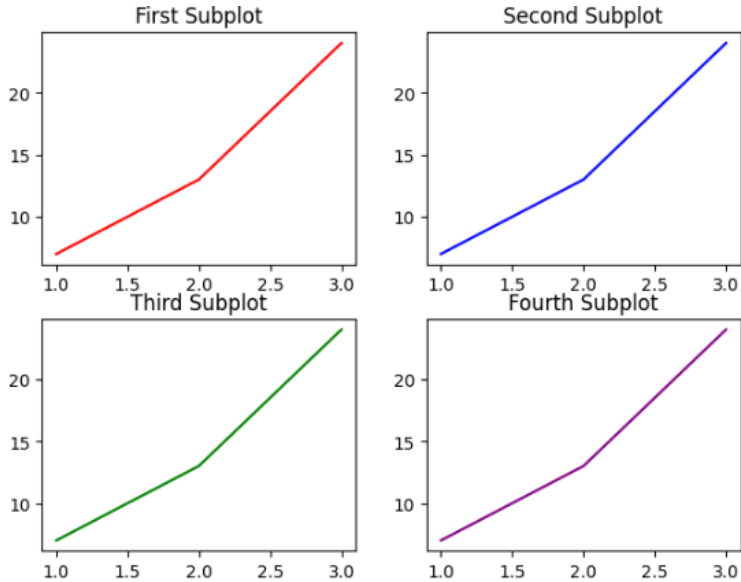
#define layout for subplots
fig, ax = plt.subplots(2, 2)

#specify a tight layout
fig.tight_layout()

#define subplot titles
ax[0, 0].plot(x, y, color='red')
ax[0, 1].plot(x, y, color='blue')
ax[1, 0].plot(x, y, color='green')
ax[1, 1].plot(x, y, color='purple')

#add title to each subplot
ax[0, 0].set_title('First Subplot')
ax[0, 1].set_title('Second Subplot')
ax[1, 0].set_title('Third Subplot')
ax[1, 1].set_title('Fourth Subplot')
```

Output



Notice that the padding between and around the subplots has been adjusted so that the plots no longer overlap in any area.

Note that the `tight_layout()` function takes a `pad` argument to specify the padding between the figure edge and the edges of subplots, as a fraction of the font size.

The default value for `pad` is 1.08. However, we can increase this value to increase the padding around the plots:

```
import matplotlib.pyplot as plt

#define data
x = [1, 2, 3]
y = [7, 13, 24]

#define layout for subplots
fig, ax = plt.subplots(2, 2)

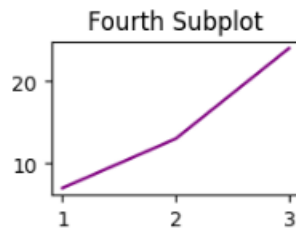
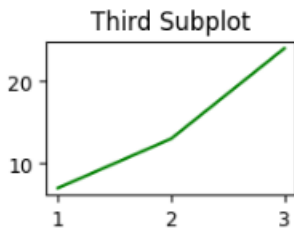
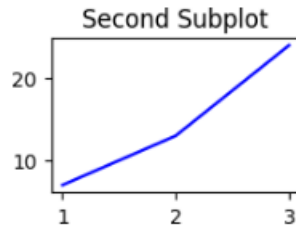
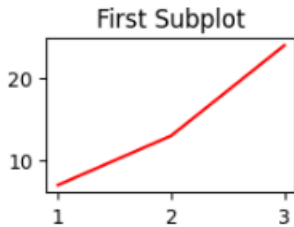
#specify a tight layout with increased padding
fig.tight_layout(pad=5)

#define subplot titles
ax[0, 0].plot(x, y, color='red')
ax[0, 1].plot(x, y, color='blue')
ax[1, 0].plot(x, y, color='green')
ax[1, 1].plot(x, y, color='purple')

#add title to each subplot
ax[0, 0].set_title('First Subplot')
ax[0, 1].set_title('Second Subplot')
ax[1, 0].set_title('Third Subplot')
ax[1, 1].set_title('Fourth Subplot')
```

Output

```
Text(0.5, 1.0, 'Fourth Subplot')
```



Notice that the padding around the plots has increased noticeably. Feel free to adjust the value for the `pad` argument to increase the padding around the plots as much as you'd like.

4.11 Changing Colour of Elements

Using `DataFrame.style` property

`df.style.set_properties`: By using this, we can use inbuilt functionality to manipulate data frame styling from font color to background color.


```
df = pd.DataFrame({'foo': ['one', 'one', 'one', 'two', 'two',  
                          'two'],  
                  'bar': ['A', 'B', 'C', 'A', 'B', 'C'],  
                  'baz': [1, 2, 3, 4, 5, 6],  
                  'zoo': ['x', 'y', 'z', 'q', 'w', 't']})  
  
df  
df.style.set_properties(**{'background-color': 'black',  
                           'color': 'green'})
```

Output

	foo	bar	baz	zoo
0	one	A	1	x
1	one	B	2	y
2	one	C	3	z
3	two	A	4	q
4	two	B	5	w
5	two	C	6	t

df.style.highlight_null : With the help of this, we can highlight the missing or null values inside the data frame.

```

# importing pandas as pd
import pandas as pd

# Creating the dataframe
df = pd.DataFrame({"A": [12, 4, 5, None, 1],
                  "B": [None, 2, 54, 3, None],
                  "C": [20, 16, None, 3, 8],
                  "D": [14, 3, None, None, 6]})

# Print the dataframe
df

# to interpolate the missing values
df.interpolate(method='linear', limit_direction='forward')
# Highlight the NaN values in DataFrame
print("\nModified Styling DataFrame:")
df.style.highlight_null(null_color='red')

```

Output

	A	B	C	D
0	12.000000	nan	20.000000	14.000000
1	4.000000	2.000000	16.000000	3.000000
2	5.000000	54.000000	nan	nan
3	nan	3.000000	3.000000	nan
4	1.000000	nan	8.000000	6.000000

df.style.highlight_min : For highlighting the minimum value in each column throughout the data frame.

```

# importing pandas as pd
import pandas as pd

# Creating the dataframe
df = pd.DataFrame({"A": [12, 4, 5, None, 1],
                  "B": [None, 2, 54, 3, None],
                  "C": [20, 16, None, 3, 8],
                  "D": [14, 3, None, None, 6]})

# Print the dataframe
df

# Highlight the Min values in each column
print("\nModified Styling DataFrame:")
df.style.highlight_min(axis=0)

```

Output

	A	B	C	D
0	12.000000	nan	20.000000	14.000000
1	4.000000	2.000000	16.000000	3.000000
2	5.000000	54.000000	nan	nan
3	nan	3.000000	3.000000	nan
4	1.000000	nan	8.000000	6.000000

df.style.highlight_max : For highlighting the maximum value in each column throughout the data frame.

```
# importing the pandas library
import pandas as pd
info = {'ID' : [101, 102, 103], 'Department' : ['B.Sc', 'B.Tech', 'M.Tech',]}
df = pd.DataFrame(info)
print (df)
# Highlight the Max values in each column
print("\nModified Stlying DataFrame:")
df.style.highlight_max(axis=0)
```

Output

	ID	Department
0	101	B.Sc
1	102	B.Tech
2	103	M.Tech

Modified Stlying DataFrame:

	ID	Department
0	101	B.Sc
1	102	B.Tech
2	103	M.Tech

4.12 Visualization Examples

Data Visualization is the process of presenting data in the form of graphs or charts. It helps to understand large and complex amounts of data very easily. It allows the decision-makers to make decisions very efficiently and also allows them in identifying new trends and patterns very easily. It is also used in high-level data analysis for Machine Learning and Exploratory Data Analysis (EDA). Data visualization can be done with various tools like Tableau, Power BI, Python.

Matplotlib is a low-level library of Python which is used for data visualization. It is easy to use and emulates MATLAB like graphs and visualization. This library is built on the top of NumPy arrays and consist of several plots like line chart, bar chart, histogram, etc.

Bar chart

A bar plot or bar chart is a graph that represents the category of data with rectangular bars with lengths and heights that is proportional to the values which they represent. The bar plots can be plotted horizontally or vertically. A bar chart describes the comparisons between the discrete categories. It can be created using the bar() method.

Syntax:

plt.bar(x, height, width, bottom, align)

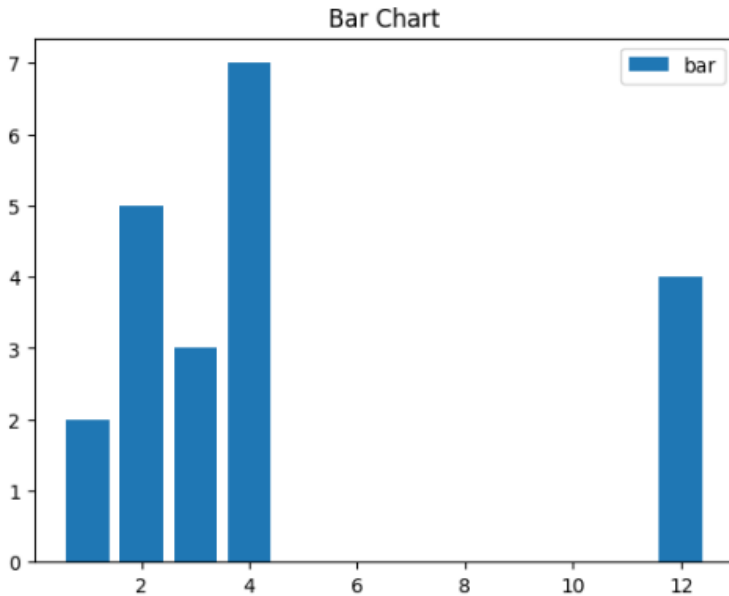
```
import matplotlib.pyplot as plt
# data to display on plots
x = [3, 1, 3, 12, 2, 4, 4]
y = [3, 2, 1, 4, 5, 6, 7]

# This will plot a simple bar chart
plt.bar(x, y)

# Title to the plot
plt.title("Bar Chart")

# Adding the legends
plt.legend(["bar"])
plt.show()
```

Output

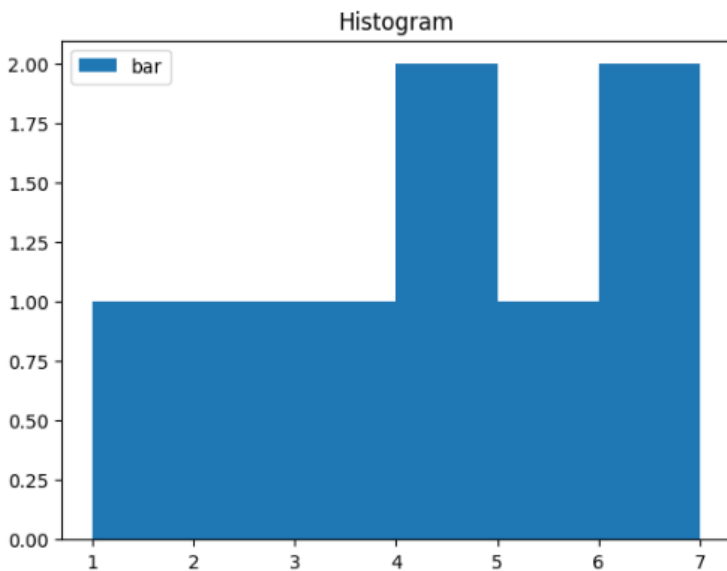


Histograms

A histogram is basically used to represent data in the form of some groups. It is a type of bar plot where the X-axis represents the bin ranges while the Y-axis gives information about frequency. To create a histogram the first step is to create a bin of the ranges, then distribute the whole range of the values into a series of intervals, and count the values which fall into each of the intervals. Bins are clearly identified as consecutive, non-overlapping intervals of variables. The `hist()` function is used to compute and create histogram of `x`.

```
import matplotlib.pyplot as plt
# data to display on plots
x = [1, 2, 3, 4, 5, 6, 7, 4]
# This will plot a simple histogram
plt.hist(x, bins = [1, 2, 3, 4, 5, 6, 7])
# Title to the plot
plt.title("Histogram")
# Adding the legends
plt.legend(["bar"])
plt.show()
```

Output



Scatter Plot

Scatter plots are used to observe the relationship between variables and use dots to represent the relationship between them. The scatter() method in the matplotlib library is used to draw a scatter plot.

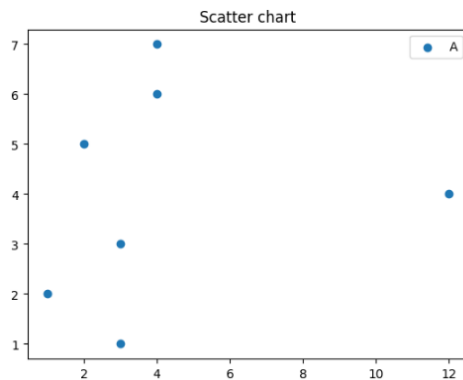
```
import matplotlib.pyplot as plt
# data to display on plots
x = [3, 1, 3, 12, 2, 4, 4]
y = [3, 2, 1, 4, 5, 6, 7]

# This will plot a simple scatter chart
plt.scatter(x, y)

# Adding legend to the plot
plt.legend("A")

# Title to the plot
plt.title("Scatter chart")
plt.show()
```

Output



Pie Chart

A Pie Chart is a circular statistical plot that can display only one series of data. The area of the chart is the total percentage of the given data. The area of slices of the pie represents the percentage of the parts of the data. The slices of pie are called wedges. The area of the wedge is determined by the length of the arc of the wedge. It can be created using the pie() method.

```
import matplotlib.pyplot as plt
# data to display on plots
x = [1, 2, 3, 4]

# this will explode the 1st wedge
# i.e. will separate the 1st wedge
# from the chart
e = (0.1, 0, 0, 0)

# This will plot a simple pie chart
plt.pie(x, explode = e)

# Title to the plot
plt.title("Pie chart")
plt.show()
```

Output

Pie chart



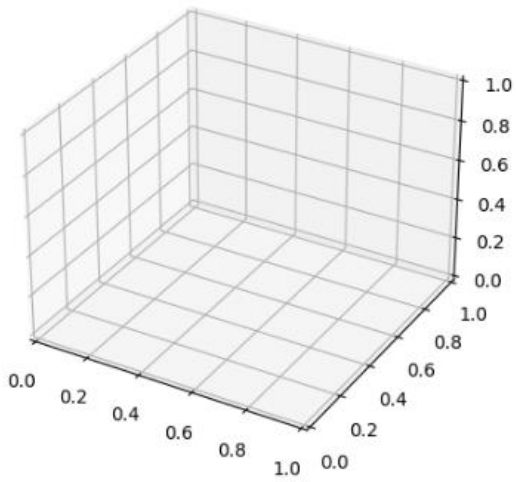
3D Plots

Matplotlib was introduced keeping in mind, only two-dimensional plotting. But at the time when the release of 1.0 occurred, the 3D utilities were developed upon the 2D and thus, we have a 3D implementation of data available today.

```
import matplotlib.pyplot as plt
# Creating the figure object
fig = plt.figure()

# keeping the projection = 3d
# creates the 3d plot
ax = plt.axes(projection = '3d')
```

Output



UNIT-V

Scikit-Learn: Introduction to Machine Learning with Scikit-Learn: Machine Learning: The Problem Setting, Loading an Example Datasets, Learning and Predicting, Model Persistence, Conventions.

A Tutorial on Statistical-Learning for Scientific Data Processing: Statistical Learning, Supervised Learning, Model Selection

<https://scikit-learn.org/stable/tutorial/basic/tutorial.html>

UNIT-VI

SciPy: Basic Functions, Special Functions, Compressed Sparse Graph Routines, Spatial Data Structures and Algorithms, Statistics, Building Specific Distributions.

6.1 SciPy Introduction

SciPy in Python is an open-source library used for solving mathematical, scientific, engineering, and technical problems. It allows users to manipulate the data and visualize the data using a wide range of high-level Python commands. SciPy is built on the Python NumPy extension. SciPy is also pronounced as “Sigh Pi.”

Sub-packages of SciPy:

- File input/output – `scipy.io`
- Special Function – `scipy.special`
- Linear Algebra Operation – `scipy.linalg`
- Interpolation – `scipy.interpolate`
- Optimization and fit – `scipy.optimize`
- Statistics and random numbers – `scipy.stats`
- Numerical Integration – `scipy.integrate`
- Fast Fourier transforms – `scipy.fftpack`
- Signal Processing – `scipy.signal`
- Image manipulation – `scipy.ndimage`

Why use SciPy

- SciPy contains varieties of sub packages which help to solve the most common issue related to Scientific Computation.
- SciPy package in Python is the most used Scientific library only second to GNU Scientific Library for C/C++ or Matlab's.
- Easy to use and understand as well as fast computational power.
- It can operate on an array of NumPy library.

Numpy VS SciPy

Numpy:

- Numpy is written in C and use for mathematical or numeric calculation.
- It is faster than other Python Libraries
- Numpy is the most useful library for Data Science to perform basic calculations.
- Numpy contains nothing but array data type which performs the most basic operation like sorting, shaping, indexing, etc.

SciPy:

- SciPy is built in top of the NumPy

- SciPy module in Python is a fully-featured version of Linear Algebra while Numpy contains only a few features.
- Most new Data Science features are available in Scipy rather than Numpy.

SciPy – Installation and Environment Setup

INSTALLING WITH PIP

You can install SciPy from PyPI with pip:

```
python -m pip install scipy
```

INSTALLING VIA CONDA

You can install SciPy from the defaults or conda-forge channels with conda:

```
conda install scipy
```

UBUNTU AND DEBIAN

Using apt-get:

```
sudo apt-get install python3-scipy
```

FEDORA

Using dnf:

```
sudo dnf install python3-scipy
```

MACOS

macOS doesn't have a preinstalled package manager, but you can install Homebrew and use it to install SciPy (and Python itself):

```
brew install scipy
```

6.2 Basic Functions

Interaction with NumPy:

SciPy builds on NumPy and therefore you can make use of NumPy functions itself to handle arrays. To know in-depth about these functions, you can simply make use of `help()`, `info()` or `source()` functions.

help():

To get information about any function, you can make use of the `help()` function. There are two ways in which this function can be used:

- without any parameters
- using parameters

Here is an example that shows both of the above methods:

```
from scipy import cluster
help(cluster)           #with parameter
help()                 #without parameter
```

When you execute the above code, the first `help()` returns the information about the `cluster` submodule. The second `help()` asks the user to enter the name of any module, keyword, etc for which the user desires to seek information. To stop the execution of this function, simply type 'quit' and hit enter.

info():

This function returns information about the desired functions, modules, etc.

`scipy.info(cluster)`

source():

The source code is returned only for objects written in Python. This function does not return useful information in case the methods or objects are written in any other language such as C. However in case you want to make use of this function, you can do it as follows:

`scipy.source(cluster)`

Common functions

We will explore the five most commonly used functions from the SciPy package:

- **cbirt():** The `cbirt()` function gets the cube root of a number. It accepts one parameter, i.e., the number whose cube root needs to be determined. The parameter could be a single real number or a list of real numbers. It returns the cube root of the number.
- **exp10():** The `exp10()` function calculates the value of the expressions 10^x , where x is the input parameter to the given function. The value of x could be any real number or a list containing the real numbers.
- **comb():** The `comb()` function calculates the value of the expression $x^C y$. This is the calculation of the number of ways we can select the objects with no effect on the order

of the objects. Mathematically, it is termed as combinations. This function accepts two parameters—the value of x and y — and returns the result of the combination.

- **perm():** The `perm()` function calculates xPy 's value. This is the calculation of the number of ways we can select y objects out of x . Here, the order of the objects matters. Mathematically, it is termed as permutations. This function accepts two parameters—the value of x and y — and returns the result of the permutation.
- **round():** The `round()` function returns the nearest possible integer from the given floating number. For example, if the input given is 1.9, the nearest integer would be 2. This function accepts a real number or a list containing the real numbers.

```
import numpy
from scipy import special

#Create a numpy array
x = numpy.array([27,45])

#Get cube root
print(special.cbrt(x))

#Exponential function
print(special.exp10(x))

#Permutation and combination
print(special.comb(3,2)) #Equivalent to 3C2
print(special.perm(3,2)) #Equivalent to 3P2

#round function
print(special.round([1.9, -2.4, 1.2]))
```

Output

```
[3.          3.5568933]
[1.e+27 1.e+45]
3.0
6.0
[ 2. -2.  1.]
```

6.3 Special Functions

Basic Functions:

a) Interaction with NumPy:

SciPy builds on NumPy and therefore you can make use of NumPy functions itself to handle arrays. To know in-depth about these functions, you can simply make use of `help()`, `info()` or `source()` functions.

help():

To get information about any function, you can make use of the `help()` function. There are two ways in which this function can be used:

- without any parameters
- using parameters

Here is an example that shows both of the above methods:

```
from scipy import cluster
help(cluster)           #with parameter
help()                  #without parameter
```

When you execute the above code, the first `help()` returns the information about the `cluster` submodule. The second `help()` asks the user to enter the name of any module, keyword, etc for which the user desires to seek information. To stop the execution of this function, simply type 'quit' and hit enter.

info():

This function returns information about the desired functions, modules, etc.

```
scipy.info(cluster)
```

source():

The source code is returned only for objects written in Python. This function does not return useful information in case the methods or objects are written in any other language such as C. However in case you want to make use of this function, you can do it as follows:

```
scipy.source(cluster)
```

b) Special Functions:

SciPy provides a number of special functions that are used in mathematical physics such as elliptic, convenience functions, gamma, beta, etc. To look for all the functions, you can make use of `help()` function as described earlier.

Exponential and Trigonometric Functions:

SciPy's Special Function package provides a number of functions through which you can find exponents and solve trigonometric problems.

Consider the following example:

```
from scipy import special
a = special.exp10(3)
print(a)

b = special.exp2(3)
print(b)

c = special.sindg(90)
print(c)

d = special.cosdg(45)
print(d)
```

Output

```
1000.0
8.0
1.0
0.7071067811865475
```

There are many other functions present in the special functions package of SciPy that you can try for yourself.

Integration Functions:

SciPy provides a number of functions to solve integrals. Ranging from ordinary differential integrator to using trapezoidal rules to compute integrals, SciPy is a storehouse of functions to solve all types of integrals problems.

General Integration:

SiPy provides a function named quad to calculate the integral of a function which has one variable. The limits can be $\pm\infty$ (\pm inf) to

indicate infinite limits. The syntax of the quad() function is as follows:

SYNTAX:

```
quad(func, a, b, args=(), full_output=0, epsabs=1.49e-08,
epsrel=1.49e-08, limit=50, points=None, weight=None,
wvar=None, wopts=None, maxp1=50, limlst=50)
```

Here, the function will be integrated between the limits a and b (can also be infinite).

```
from scipy import special
from scipy import integrate
a = lambda x: special.exp10(x)
b = integrate.quad(a, 0, 1)
print(b)
```

In the above example, the function ‘a’ is evaluated between the limits 0, 1. When this code is executed, you will see the following output.

Output

```
(3.9086503371292665, 4.3394735994897923e-14)
```

Double Integral Function:

SciPy provides dblquad that can be used to calculate double integrals. A double integral, as many of us know, consists of two real variables. The dblquad() function will take the function to be integrated as its parameter along with 4 other variables which define the limits and the functions dy and dx.

```

from scipy import integrate
a = lambda y, x: x*y**2
b = lambda x: 1
c = lambda x: -1
integrate.dblquad(a, 0, 2, b, c)

```

Output

```
(-1.3333333333333335, 1.4802973661668755e-14)
```

Fourier Transform Functions:

Fourier analysis is a method that deals with expressing a function as a sum of periodic components and recovering the signal from those components. The fft functions can be used to return the discrete Fourier transform of a real or complex sequence.

```

from scipy.fftpack import fft, ifft
import numpy as np
x = np.array([0,1,2,3])
y = fft(x)
print(y)

```

Output

```
[ 6.-0.j -2.+2.j -2.-0.j -2.-2.j]
```

Similarly, you can find the inverse of this by using the ifft function as follows:

```

from scipy.fftpack import fft, ifft
import numpy as np
x = np.array([0,1,2,3])
y = ifft(x)
print(y)

```


Output

```
[ 1.5-0.j -0.5-0.5j -0.5-0.j -0.5+0.5j]
```

Linear Algebra:

Linear algebra deals with linear equations and their representations using vector spaces and matrices. SciPy is built on ATLAS LAPACK and BLAS libraries and is extremely fast in solving problems related to linear algebra. In addition to all the functions from `numpy.linalg`, `scipy.linalg` also provides a number of other advanced functions. Also, if `numpy.linalg` is not used along with ATLAS LAPACK and BLAS support, `scipy.linalg` is faster than `numpy.linalg`.

Finding the Inverse of a Matrix:

Mathematically, the inverse of a matrix A is the matrix B such that $AB=I$ where I is the identity matrix consisting of ones down the main diagonal denoted as $B=A^{-1}$. In SciPy, this inverse can be obtained using the `linalg.inv` method.

```
import numpy as np
from scipy import linalg
A = np.array([[1,2], [4,3]])
B = linalg.inv(A)
print(B)
```

Output

```
[[ -0.6  0.4]
 [ 0.8 -0.2]]
```

Finding the Determinants:

The value derived arithmetically from the coefficients of the matrix is known as the determinant of a square matrix. In SciPy, this can be done using a function `det` which has the following syntax:

SYNTAX:

```
det(a, overwrite_a=False, check_finite=True)
```

where,

- `a : (M, M)` Is a square matrix
- `overwrite_a(bool, optional) :` Allow overwriting data in `a`
- `check_finite (bool, optional):` To check whether input matrix consist only of finite numbers

```
import numpy as np
from scipy import linalg
A = np.array([[1,2], [4,3]])
B = linalg.det(A)
print(B)
```

Output

-5.0

Sparse Eigenvalues:

Eigenvalues are a specific set of scalars linked with linear equations. The ARPACK provides that allow you to find eigenvalues (eigenvectors) quite fast. The complete functionality of ARPACK is packed within two high-level interfaces which are `scipy.sparse.linalg.eigs` and `scipy.sparse.linalg.eigsh`. `eigs`. The

eigs interface allows you to find the eigenvalues of real or complex nonsymmetric square matrices whereas the eigsh interface contains interfaces for real-symmetric or complex-hermitian matrices.

The eigh function solves a generalized eigenvalue problem for a complex Hermitian or real symmetric matrix.

```
from scipy.linalg import eigh
import numpy as np
A = np.array([[1, 2, 3, 4], [4, 3, 2, 1], [1, 4, 6, 3], [2, 3, 2, 5]])
a, b = eigh(A)
print("Selected eigenvalues :", a)
print("Complex ndarray :", b)
```

Output

```
Selected eigenvalues : [-2.53382695  1.66735639  3.69488657 12.17158399]
Complex ndarray : [[ 0.69205614  0.5829305  0.25682823 -0.33954321]
 [-0.68277875  0.46838936  0.03700454 -0.5595134 ]
 [ 0.23275694 -0.29164622 -0.72710245 -0.57627139]
 [ 0.02637572 -0.59644441  0.63560361 -0.48945525]]
```

6.4 Compressed Sparse Graph Routines

What is Sparse Data?

Sparse data is data that has mostly unused elements (elements that don't carry any information). It can be an array like this one:

```
[1, 0, 2, 0, 0, 3, 0, 0, 0, 0, 0]
```

Sparse Data: is a data set where most of the item values are zero.

Dense Array: is the opposite of a sparse array: most of the values are not zero.

How to Work With Sparse Data

SciPy has a module, `scipy.sparse` that provides functions to deal with sparse data.

There are primarily two types of sparse matrices that we use:

CSC - Compressed Sparse Column. For efficient arithmetic, fast column slicing.

CSR - Compressed Sparse Row. For fast row slicing, faster matrix vector products

CSR Matrix

We can create CSR matrix by passing an array into function `scipy.sparse.csr_matrix()`.

```
import numpy as np
from scipy.sparse import csr_matrix

arr = np.array([0, 0, 0, 0, 0, 1, 1, 0, 2])

print(csr_matrix(arr))
```

Output

```
(0, 5)      1
(0, 6)      1
(0, 8)      2
```

From the result we can see that there are 3 items with value.

The 1. item is in row 0 position 5 and has the value 1.

The 2. item is in row 0 position 6 and has the value 1.

The 3. item is in row 0 position 8 and has the value 2.

Sparse Matrix Methods

Viewing stored data (not the zero items) with the data property:

```
import numpy as np
from scipy.sparse import csr_matrix

arr = np.array([[0, 0, 0], [0, 0, 1], [1, 0, 2]])

print(csr_matrix(arr).data)
```

Output

```
[1 1 2]
```

Counting nonzeros with the count_nonzero() method:

```
import numpy as np
from scipy.sparse import csr_matrix

arr = np.array([[0, 0, 0], [0, 0, 1], [1, 0, 2]])

print(csr_matrix(arr).count_nonzero())
```

Output

```
3
```

Removing zero-entries from the matrix with the eliminate_zeros() method:

```
import numpy as np
from scipy.sparse import csr_matrix

arr = np.array([[0, 0, 0], [0, 0, 1], [1, 0, 2]])

mat = csr_matrix(arr)
mat.eliminate_zeros()

print(mat)
```

Output

```
(1, 2)      1
(2, 0)      1
(2, 2)      2
```

Eliminating duplicate entries with the `sum_duplicates()` method:

```
import numpy as np
from scipy.sparse import csr_matrix
arr = np.array([[0, 0, 0], [0, 0, 1], [1, 0, 2]])
mat = csr_matrix(arr)
mat.sum_duplicates()
print(mat)
```

Output

```
(1, 2)      1
(2, 0)      1
(2, 2)      2
```

Converting from csr to csc with the `tocsc()` method:

```
import numpy as np
from scipy.sparse import csr_matrix
arr = np.array([[0, 0, 0], [0, 0, 1], [1, 0, 2]])
newarr = csr_matrix(arr).tocsc()
print(newarr)
```

Output

```
(2, 0)      1
(1, 2)      1
(2, 2)      2
```

SciPy Graphs

Working with Graphs

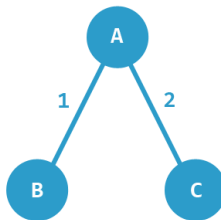
Graphs are an essential data structure.

SciPy provides us with the module `scipy.sparse.csgraph` for working with such data structures.

Adjacency Matrix

Adjacency matrix is a $n \times n$ matrix where n is the number of elements in a graph. And the values represents the connection between the elements.

Example:



For a graph like this, with elements A, B and C, the connections are:

A & B are connected with weight 1.

A & C are connected with weight 2.

C & B is not connected.

The Adjacency Matrix would look like this:

```
A B C
A: [0 1 2]
B: [1 0 0]
C: [2 0 0]
```

Below follows some of the most used methods for working with adjacency matrices.

Connected Components

Find all of the connected components with the `connected_components()` method.

```
import numpy as np
from scipy.sparse.csgraph import connected_components
from scipy.sparse import csr_matrix
arr = np.array([
    [0, 1, 2],
    [1, 0, 0],
    [2, 0, 0]
])

newarr = csr_matrix(arr)

print(connected_components(newarr))
```

Output

```
(1, array([0, 0, 0], dtype=int32))
```

Dijkstra

Use the `dijkstra` method to find the shortest path in a graph from one element to another.

It takes following arguments:

return_predecessors: boolean (True to return whole path of traversal otherwise False).

indices: index of the element to return all paths from that element only.

limit: max weight of path.

Example

Find the shortest path from element 1 to 2:

```
import numpy as np
from scipy.sparse.csgraph import dijkstra
from scipy.sparse import csr_matrix

arr = np.array([
    [0, 1, 2],
    [1, 0, 0],
    [2, 0, 0]
])

newarr = csr_matrix(arr)

print(dijkstra(newarr, return_predecessors=True, indices=0))
```

Output

```
(array([0., 1., 2.]), array([-9999,    0,    0], dtype=int32))
```

Floyd Warshall

Use the `floyd_warshall()` method to find shortest path between all pairs of elements.

Example

Find the shortest path between all pairs of elements:

```

import numpy as np
from scipy.sparse.csgraph import floyd_warshall
from scipy.sparse import csr_matrix

arr = np.array([
    [0, 1, 2],
    [1, 0, 0],
    [2, 0, 0]
])

newarr = csr_matrix(arr)

print(floyd_warshall(newarr, return_predecessors=True))

```

Output

```

(array([[0., 1., 2.],
        [1., 0., 3.],
        [2., 3., 0.])), array([[ -9999,    0,    0],
        [    1, -9999,    0],
        [    2,    0, -9999]], dtype=int32))

```

Bellman Ford

The `bellman_ford()` method can also find the shortest path between all pairs of elements, but this method can handle negative weights as well.

Example

Find shortest path from element 1 to 2 with given graph with a negative weight:

```
import numpy as np
from scipy.sparse.csgraph import bellman_ford
from scipy.sparse import csr_matrix

arr = np.array([
    [0, -1, 2],
    [1, 0, 0],
    [2, 0, 0]
])

newarr = csr_matrix(arr)

print(bellman_ford(newarr, return_predecessors=True, indices=0))
```

Output

```
(array([ 0., -1.,  2.]), array([-9999,    0,    0], dtype=int32))
```

Depth First Order

The `depth_first_order()` method returns a depth first traversal from a node.

This function takes following arguments:

- the graph.
- the starting element to traverse graph from.

Example

Traverse the graph depth first for given adjacency matrix:

```

import numpy as np
from scipy.sparse.csgraph import depth_first_order
from scipy.sparse import csr_matrix

arr = np.array([
    [0, 1, 0, 1],
    [1, 1, 1, 1],
    [2, 1, 1, 0],
    [0, 1, 0, 1]
])

newarr = csr_matrix(arr)

print(depth_first_order(newarr, 1))

```

Output

```
(array([1, 0, 3, 2], dtype=int32), array([ 1, -9999, 1, 0], dtype=int32))
```

Breadth First Order

The `breadth_first_order()` method returns a breadth first traversal from a node.

This function takes following arguments:

- the graph.
- the starting element to traverse graph from.

Example

Traverse the graph breadth first for given adjacency matrix:

```

import numpy as np
from scipy.sparse.csgraph import breadth_first_order
from scipy.sparse import csr_matrix

arr = np.array([
    [0, 1, 0, 1],
    [1, 1, 1, 1],
    [2, 1, 1, 0],
    [0, 1, 0, 1]
])

newarr = csr_matrix(arr)

print(breadth_first_order(newarr, 1))

```

Output

```
(array([1, 0, 2, 3], dtype=int32), array([ 1, -9999, 1, 1], dtype=int32))
```

6.5 Spatial Data Structures and Algorithms

Working with Spatial Data

Spatial data refers to data that is represented in a geometric space.

E.g. points on a coordinate system.

We deal with spatial data problems on many tasks.

E.g. finding if a point is inside a boundary or not.

SciPy provides us with the module `scipy.spatial`, which has functions for working with spatial data.

Triangulation

A Triangulation of a polygon is to divide the polygon into multiple triangles with which we can compute an area of the polygon.

A Triangulation with points means creating surface composed triangles in which all of the given points are on at least one vertex of any triangle in the surface.

One method to generate these triangulations through points is the Delaunay() Triangulation.

Example

Create a triangulation from following points:

```
import numpy as np
from scipy.spatial import Delaunay
import matplotlib.pyplot as plt

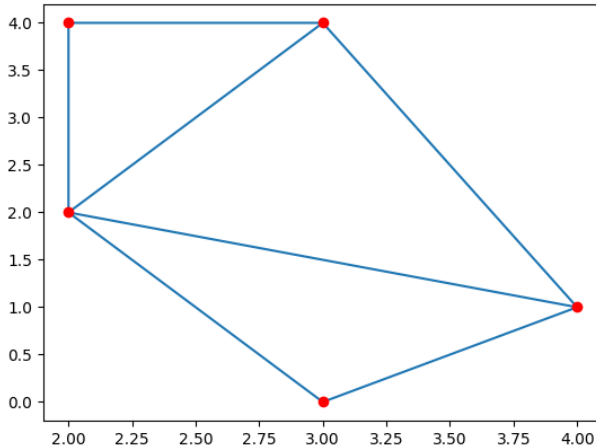
points = np.array([
    [2, 4],
    [3, 4],
    [3, 0],
    [2, 2],
    [4, 1]
])

simplices = Delaunay(points).simplices

plt.triplot(points[:, 0], points[:, 1], simplices)
plt.scatter(points[:, 0], points[:, 1], color='r')

plt.show()
```

Output



Distance Matrix

There are many Distance Metrics used to find various types of distances between two points in data science, Euclidean distance, cosine distance etc.

The distance between two vectors may not only be the length of straight line between them, it can also be the angle between them from origin, or number of unit steps required etc.

Many of the Machine Learning algorithm's performance depends greatly on distance metrics. E.g. "K Nearest Neighbors", or "K Means" etc.

Let us look at some of the Distance Metrics:

Euclidean Distance

Find the euclidean distance between given points.

Example

```
from scipy.spatial.distance import euclidean
p1 = (1, 0)
p2 = (10, 2)
res = euclidean(p1, p2)
print(res)
```

Output

9.219544457292887

Cityblock Distance (Manhattan Distance)

Is the distance computed using 4 degrees of movement.

E.g. we can only move: up, down, right, or left, not diagonally.

Example

Find the cityblock distance between given points:

```
from scipy.spatial.distance import cityblock
p1 = (1, 0)
p2 = (10, 2)
res = cityblock(p1, p2)
print(res)
```

Output

11

Cosine Distance

Is the value of cosine angle between the two points A and B.

Example

Find the cosine distance between given points:


```
from scipy.spatial.distance import cosine
p1 = (1, 0)
p2 = (10, 2)
res = cosine(p1, p2)
print(res)
```

Output

0.019419324309079777

Hamming Distance

Is the proportion of bits where two bits are different.

It's a way to measure distance for binary sequences.

Example

Find the hamming distance between given points:

```
from scipy.spatial.distance import hamming
p1 = (True, False, True)
p2 = (False, True, True)
res = hamming(p1, p2)
print(res)
```

Output

0.6666666666666666

6.6 Statistics

Statistics, in general, is the method of collection of data, tabulation, and interpretation of numerical data. It is an area of applied mathematics concerned with data collection analysis,

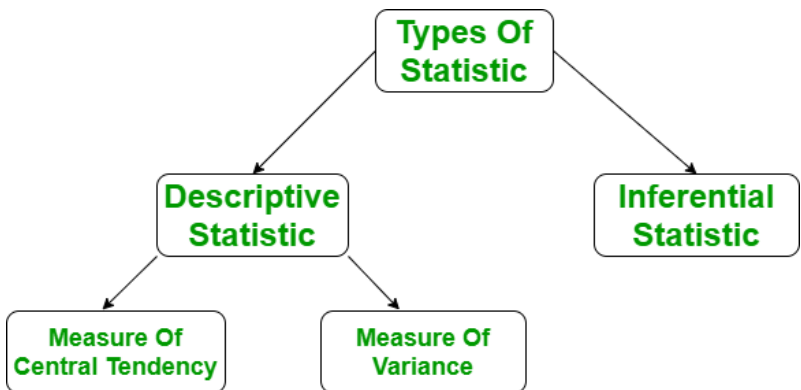
interpretation, and presentation. With statistics, we can see how data can be used to solve complex problems.

Understanding the Descriptive Statistics

In layman's terms, descriptive statistics generally means describing the data with the help of some representative methods like charts, tables, Excel files, etc. The data is described in such a way that it can express some meaningful information that can also be used to find some future trends. Describing and summarizing a single variable is called univariate analysis. Describing a statistical relationship between two variables is called bivariate analysis. Describing the statistical relationship between multiple variables is called multivariate analysis.

There are two types of Descriptive Statistics:

- The measure of central tendency
- Measure of variability



Measure of Central Tendency

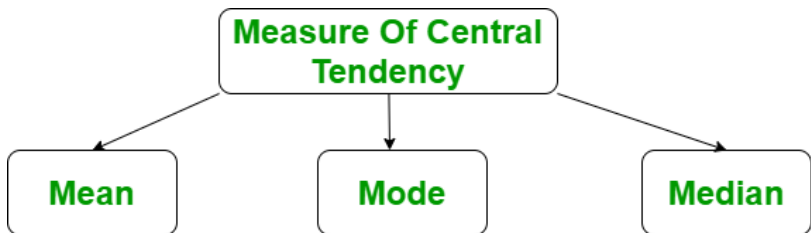
The measure of central tendency is a single value that attempts to describe the whole set of data. There are three main features of central tendency:

Mean

Median

- Median Low
- Median High

Mode



Mean

It is the sum of observations divided by the total number of observations. It is also defined as average which is the sum divided by count.

$$Mean(\bar{x}) = \frac{\sum x}{n}$$

The mean() function returns the mean or average of the data passed in its arguments. If the passed argument is empty, StatisticsError is raised.

Example: Python code to calculate mean

```
import statistics
# initializing list
li = [1, 2, 3, 3, 2, 2, 1]
# using mean() to calculate average of list
# elements
print ("The average of list values is : ",end="")
print (statistics.mean(li))
```

Output

```
The average of list values is : 2
```

Median

It is the middle value of the data set. It splits the data into two halves. If the number of elements in the data set is odd then the center element is the median and if it is even then the median would be the average of two central elements. It first sorts the data and then performs the median operation

For Odd Numbers:

$$\frac{n+1}{2}$$

For Even Numbers:

$$\frac{\frac{n}{2} + (\frac{n}{2} + 1)}{2}$$

The median() function is used to calculate the median, i.e. middle element of data. If the passed argument is empty, StatisticsError is raised.

Example: Python code to calculate Median

Median Low

The `median_low()` function returns the median of data in case of odd number of elements, but in case of even number of elements, returns the lower of two middle elements. If the passed argument is empty, `StatisticsError` is raised

Example: Python code to calculate Median Low

Median High

The `median_high()` function returns the median of data in case of odd number of elements, but in case of even number of elements, returns the higher of two middle elements. If passed argument is empty, `StatisticsError` is raised.

Example: Python code to calculate Median High

Mode

It is the value that has the highest frequency in the given data set. The data set may have no mode if the frequency of all data points is the same. Also, we can have more than one mode if we encounter two or more data points having the same frequency.

The mode() function returns the number with the maximum number of occurrences. If the passed argument is empty, StatisticsError is raised.

Example: Python code to calculate Mode

<https://www.geeksforgeeks.org/statistics-with-python/>

6.7 Building Specific Distributions

Reference

- https://numpy.org/doc/stable/user/absolute_beginners.html
- <https://www.geeksforgeeks.org/why-numpy-is-faster-in-python/>
- <https://www.javatpoint.com/numpy-tutorial>
- <https://www.geeksforgeeks.org/how-to-install-numpy-on-linux/>
- <https://www.programsbuzz.com/article/difference-between-numpy-array-and-list>
- <https://www.datacamp.com/tutorial/python-arrays>

- <https://aiplanet.com/learn/introduction-to-numpy/attributes-of-an-array-and-creating-basic-array/89/shape-size-and-data-type-of-an-array>
- <https://www.educba.com/matrix-in-numpy/#:~:text=We%20can%20create%20a%20matrix,structuring%20as%20an%20input%20type.>
- <https://www.educba.com/matrix-in-numpy/>
- <https://www.datacamp.com/tutorial/numpy-random>
- <https://sparkbyexamples.com/numpy/numpy-transpose-matrix/>
- <https://www.geeksforgeeks.org/python-numpy-matrix-transpose/>
- https://www.w3schools.com/python/numpy/numpy_array_reshape.asp
- <https://www.stechies.com/python-docstrings/>
- https://www.tutorialspoint.com/python_pandas/python_pandas_panel.htm
- <https://www.w3resource.com/pandas/object-creation.php>
- <https://www.javatpoint.com/python-pandas-series>
- <https://www.w3resource.com/pandas/series/series-map.php>
- <https://www.dataquest.io/blog/tutorial-time-series-analysis-with-pandas/>
- <https://www.edureka.co/blog/scipy-tutorial/#special>

•