

The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. Uses include data cleaning and transformation, numerical simulation, statistical modelling, data visualization, machine learning, and much more.

### **Notebook:**

A notebook integrates code and its output into a single document that combines visualizations, narrative text, mathematical equations, and other rich media. In other words: it's a single document where you can run code, display the output, and also add explanations, formulas, charts, and make your work more transparent, understandable, repeatable, and shareable. A notebook file name has extension .ipynb.

### **Installing Jupyter Notebook using pip**

When you install Python directly from its [official website](#), it does not include Jupyter Notebook in its standard library. In this case, you need to install Jupyter Notebook using the pip. The process is as follows:

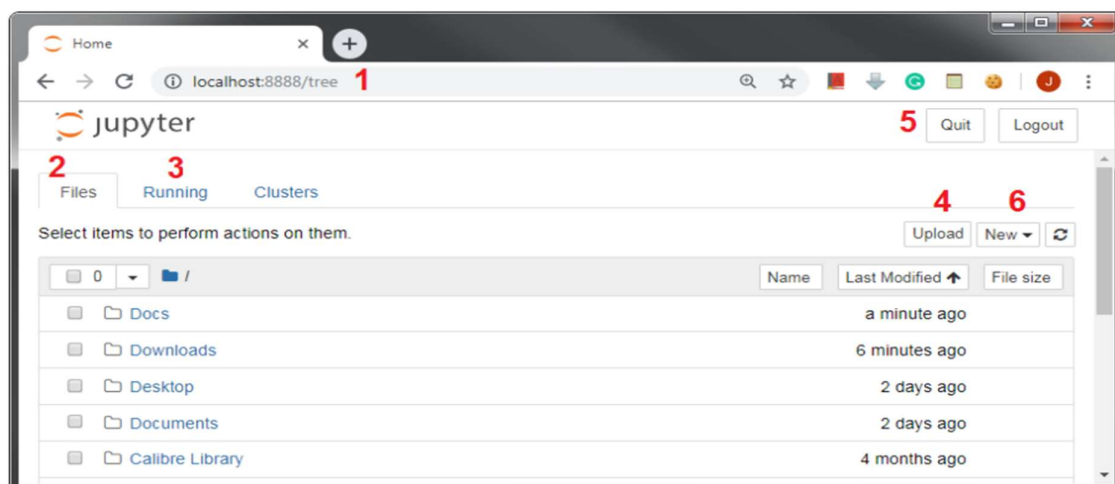
1. Open a new command prompt (Windows) or terminal (Mac/Linux)
2. Execute the following command to install Jupyter Notebook

```
python -m pip install jupyter  
or if you are using Python 3  
python3 -m pip install jupyter
```

or simply

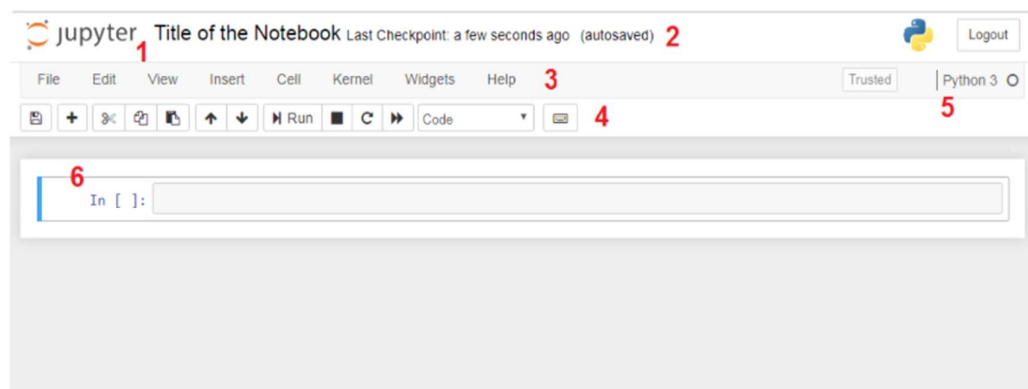
```
pip install Jupyter
```

### **Jupyter Dashboard:**



1. It is the URL on which Jupyter server is running. If you are running the Jupyter on localhost, this would be the same URL shown in the console when you started Jupyter software.
2. The Files tab lists the directories and files in the home folder, which usually is the home directory of the user logged in to the computer.
3. The Running tab shows you a list of all open notebooks. When you start a new notebook or open an existing notebook, a kernel will get attached to it. All such running kernels will be listed under this tab.
4. If you want to open an existing Jupyter notebook(that ends with .ipynb extension), it needs to be listed in the Files tab. If it is not listed, you need to upload it using the Upload button which will open a file browser for you to load the file.
5. Quit and Logout buttons allow you to logout and shutdown the server. When you quit, all the opened notebooks will be closed, and the server will get shutdown.
6. The New button allows you to create a new notebook, text file, folder, or terminal.

You can create a new notebook by clicking on the respective language name. Regardless of what language you choose, the new notebook that you create will have the same appearance. The difference would be in terms of the kernel attached to it. If I click on Python 3 on the dropdown opened, a new notebook with Python kernel attached to it will be created.



1. The title is the name of the notebook. The title you set becomes the file name for the notebook, and it will have the extension as .ipynb which stands for IPython NoteBook.
2. The checkpoint shows you the time when your notebook was saved last.
3. The menu bar lists various menus that allow you to download the notebook(in multiple formats), open a new notebook, edit the notebook, customise the headers, manipulate cells, nudge the kernel, access help and so on.

4. The shortcut bar lists commonly used shortcuts such as save to save the notebook, add a cell, cut, copy to manipulate cells, up and down to navigate between cells, run to execute the cell, and so on. Any extension that you add to Jupyter will have its shortcut on this bar. We will learn what extensions are in the latter part of this article.
5. The **Kernel** shows you the current kernel associated with the notebook. kernel is a “computational engine” that executes the code contained in a notebook document. In other words A kernel is a program that interprets and executes the user’s code. The circle beside the kernel shows the status of the kernel. The hollow circle represents that it is ready to take input and run a cell. When a kernel is executing code or processing anything, it changes to solid. The Jupyter Notebook App has an inbuilt kernel for Python code, but there are also kernels available for other programming languages. In our case, the kernel is Python 3.
6. A cell is a container for text to be displayed in the notebook or code to be executed by the notebook’s kernel.


Cell: A cell is a container for text to be displayed in the notebook or code to be executed by the notebook’s kernel. Main types of cells are as follows:

Code — This is where you type your code and when executed the kernel will display its output below the cell.

Markdown — This is where you type your text formatted using Markdown and the output is displayed in place when it is run.

Raw NBConvert — It’s a command line tool to convert your notebook into another format (like HTML, PDF, etc.)

## NumPy (Numerical Python)



**SYLLABUS:** NumPy Basics: creating ndarray, datatypes for ndarray, Arithmetic with NumPy Arrays, Basic Indexing and slicing, Transposing Arrays and Swapping Axes, Universal function, Mathematical and statistical function, Linear Algebra

### **Introduction**

1. NumPy stands for 'Numerical Python'. It is a package for data analysis and scientific computing with Python.
2. The NumPy package provides basic routines for manipulating large arrays and matrices of numeric data.
3. NumPy also uses a multidimensional array object, and has functions and tools for working with these arrays. The powerful n-dimensional array in NumPy speeds-up data processing.
4. NumPy can be easily interfaced with other Python packages and provides tools for integrating with other programming languages like C, C++ etc.

### **Arrays:**

An array is a data type used to store multiple values using a single variable name. An array contains an ordered collection of data elements where each element is of the same type and can be referenced by its index (position).

The important characteristics of an array are:

1. Each element of the array is of same data type, though the values stored in them may be different.
2. The entire array is stored contiguously in memory. This makes operations on array fast.
3. Each element of the array is identified or referred using the name of the Array along with the index of that element, which is start from zero.

### **NumPy Array**

The most important object defined in NumPy is an N-dimensional array type called *ndarray*. It describes the collection of items of the same type. Items in the collection can be accessed using a zero-based index. The *ndarray* are used to store lists of numerical data, vectors and matrices.

The NumPy library has a large set of routines (built-in functions) for creating, manipulating, and transforming *ndarray*.

Following are few differences between list and Array.

List	ndarray
List can have elements of different data types for example, [1,3.4, 'hello', 'a@']	All elements of an array are of same data type for example, an array of floats may be: [1.2, 5.4, 2.7]
Elements of a list are not stored contiguously in memory.	Array elements are stored in contiguous memory locations. This makes operations on arrays faster than lists.
Lists do not support element wise operations, for example, addition, multiplication, etc. because elements may not be of same type.	Arrays support element wise operations. For example, if A1 is an array, it is possible to say A1/3 to divide each element of the array by 3.
Lists can contain objects of different datatype that Python must store the type information for every element along with its element value. Thus, lists take more space in memory and are less efficient.	NumPy array takes up less space in memory as compared to a list because arrays do not require to store datatype of each element separately.

### Creation of NumPy Arrays from List

There are several ways to create arrays. To create an array and to use its methods, first we need to import the NumPy library.

```
#NumPy is loaded as np (we can assign any name), numpy must be
written in lowercase
>>> import numpy as np
```

The NumPy's array() function converts a given list into an array. For example,

```
#Create an array called array1 from the given list.
>>> array1 = np.array([10,20,30])
#Display the contents of the array
>>> array1
array([10, 20, 30])
```

- **Creating a 1-D Array**

An array with only single row of elements is called 1-D array.

```
>>> array2 = np.array([5,-7.4,'a',7.2])
>>> array2
array(['5', '-7.4', 'a', '7.2'],
```

Observe that since there is a string value in the list, all integer and float values have been promoted to string, while converting the list to array.

- **Creating a 2-D Array**

We can create a two dimensional (2-D) arrays by passing nested lists to the array() function.

```
>>> array3 = np.array([[2.4,3],[4.91,7],[0,-1]])
>>> array3
array([[ 2.4 ,  3.  ],
       [ 4.91,  7.  ],
       [ 0.  , -1.  ]])
```

Observe that the integers 3, 7, 0 and -1 have been promoted to floats.

### **Attributes of NumPy Array**

Some important attributes of a NumPy ndarray object are:

- `ndarray.ndim`: gives the number of dimensions of the array as an integer value. Arrays can be 1-D, 2-D or n-D. NumPy calls the dimensions as axes (plural of axis). Thus, a 2-D array has two axes. The row-axis is called axis-0 and the column-axis is called axis-1. The number of axes is also called the array's rank.

For example:

```
In [7]: array3.ndim
Out[7]: 2
```

- `ndarray.shape`: It gives the sequence of integers indicating the size of the array for each dimension.

For example

```
In [8]: array1.shape
Out[8]: (3,)
In [9]: array3.shape
Out[9]: (3,2)
```

- `ndarray.size`: It gives the total number of elements of the array. This is equal to the product of the elements of shape.

For example:

```
>>> array1.size
3
>>> array3.size
6
```

- `ndarray.dtype`: is the data type of the elements of the array. All the elements of an array are of same data type. Common data types are int32, int64, float32, float64, U32, etc.

For example:

```
>>> array1.dtype
```

```
dtype('int32')
>>> array2.dtype
dtype('<U32>')
>>> array3.dtype
type('float64')
```

- v. `ndarray.itemsize`: It specifies the size in bytes of each element of the array. Data type `int32` and `float32` means each element of the array occupies 32 bits in memory. 8 bits form a byte. Thus, an array of elements of type `int32` has `itemsize 32/8=4` bytes. Likewise, `int64/float64` means each item has `itemsize 64/8=8` bytes.

```
>>> array1.itemsize
4 # memory allocated to integer
>>> array2.itemsize
128 # memory allocated to string
>>> array3.itemsize
8 #memory allocated to float type
```

### Other Ways of Creating NumPy Arrays

1. We can specify data type (integer, float, etc.) while creating array using `dtype` as an argument to `array()`. This will convert the data automatically to the mentioned type. For example:

```
>>> array4 = np.array( [ [1,2], [3,4] ],
dtype=float)
>>> array4
array([[1., 2.],
       [3., 4.]])
```

2. We can create an array with all elements initialized to 0 using the function `zeros()`. By default, the data type of the array created by `zeros()` is float. For example

```
>>> array5 = np.zeros((3,4))

>>> array5
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

3. We can create an array with all elements initialized to 1 using the function `ones()`. By default, the data type of the array created by `ones()` is float.

```
>>> array6 = np.ones((3,2))
>>> array6
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
```

4. We can create an array with all elements filled with a fixed number using `full()` function. For example:

```
>>> array7 = np.full((3,2),3.14)
>>> array7

array([[3.14, 3.14],
       [3.14, 3.14],
       [3.14, 3.14]])
```

5. We can create an array with numbers in a given range and sequence using the `arange()` function. This function is analogous to the `range()` function of Python.

```
>>> array7 = np.arange(6)
# an array of 6 elements is created with start value 5 and step
size 1
>>> array7
array([0, 1, 2, 3, 4, 5])
# Creating an array with start value -2, end # value 24 and
step size 4
>>> array8 = np.arange( -2, 24, 4 )
>>> array8
array([-2, 2, 6, 10, 14, 18, 22])
```

## NumPy Standard Data Types:

Data type	Description
<code>bool_</code>	Boolean (True or False) stored as a byte
<code>int_</code>	Default integer type (same as C long; normally either <code>int64</code> or <code>int32</code> )
<code>intc</code>	Identical to C <code>int</code> (normally <code>int32</code> or <code>int64</code> )
<code>intp</code>	Integer used for indexing (same as C <code>ssize_t</code> ; normally either <code>int32</code> or <code>int64</code> )
<code>int8</code>	Byte (–128 to 127)
<code>int16</code>	Integer (–32768 to 32767)
<code>int32</code>	Integer (–2147483648 to 2147483647)
<code>int64</code>	Integer (–9223372036854775808 to 9223372036854775807)
<code>uint8</code>	Unsigned integer (0 to 255)
<code>uint16</code>	Unsigned integer (0 to 65535)
<code>uint32</code>	Unsigned integer (0 to 4294967295)
<code>uint64</code>	Unsigned integer (0 to 18446744073709551615)
<code>float_</code>	Shorthand for <code>float64</code>
<code>float16</code>	Half-precision float: sign bit, 5 bits exponent, 10 bits mantissa
<code>float32</code>	Single-precision float: sign bit, 8 bits exponent, 23 bits mantissa
<code>float64</code>	Double-precision float: sign bit, 11 bits exponent, 52 bits mantissa
<code>complex_</code>	Shorthand for <code>complex128</code>
<code>complex64</code>	Complex number, represented by two 32-bit floats
<code>complex128</code>	Complex number, represented by two 64-bit floats

## Array Indexing: Accessing Single Elements:

Accessing single elements in array is similar to that of lists.

1. In a one-dimensional array, the *i*th value (counting from zero) can be access by specifying the desired index in square brackets, just as with Python lists:

```
In[5]: x1=np.array([5,0,3,3,7,9])
Out[5]: array([5, 0, 3, 3, 7, 9])
```



```
In[6]: x1[0]
Out[6]: 5
In[7]: x1[4]
Out[7]: 7
```

2. To index from the end of the array, you can use negative indices:

```
In[8]: x1[-1]
Out[8]: 9
In[9]: x1[-2]
Out[9]: 7
```

3. In a multidimensional array, you access items using a comma-separated tuple of indices:

```
In[10]: x2
Out[10]: array([[3, 5, 2, 4],
               [7, 6, 8, 8],
               [1, 6, 7, 7]])

In[11]: x2[0, 0]
Out[11]: 3
In[12]: x2[2, 0]
Out[12]: 1
In[13]: x2[2, -1]
Out[13]: 7
```

4. You can also modify values using any of the above index notation:

```
In[14]: x2[0, 0] = 12
x2
Out[14]: array([[12, 5, 2, 4],
               [ 7, 6, 8, 8],
               [ 1, 6, 7, 7]])
```

### **Array Slicing: Accessing Subarrays:**

A part of array (subarray) can be access using slice (:) operator.

Basic syntax for slicing array x is:

`x[start : stop, size]`

If any of these are unspecified, they default to the values start=0, stop=size of dimension, step=1.

```
In[16]: x = np.arange(10)
x
Out[16]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
In[17]: x[:5] # first five elements
Out[17]: array([0, 1, 2, 3, 4])
In[18]: x[5:] # elements after index 5
Out[18]: array([5, 6, 7, 8, 9])
In[19]: x[4:7] # middle subarray
```

```

Out[19]: array([4, 5, 6])
In[20]: x[::2] # every other element
Out[20]: array([0, 2, 4, 6, 8])
In[21]: x[1::2] # every other element, starting at index 1
Out[21]: array([1, 3, 5, 7, 9])

```

When the step value is negative, then default values of start and stop get interchanged. This become a convenient way to reverse an array:

```

In[22]: x[::-1] # all elements, reversed
Out[22]: array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
In[23]: x[5::-2] # reversed every other from index 5
Out[23]: array([5, 3, 1])

```

### **Multidimensional subarrays:**

Multidimensional slices work in the same way, with multiple slices separated by commas.

For example:

```

In[24]: x2
Out[24]: array([[12, 5, 2, 4],
               [ 7, 6, 8, 8],
               [ 1, 6, 7, 7]])
In[25]: x2[:2, :3] # two rows, three columns
Out[25]: array([[12, 5, 2],
               [ 7, 6, 8]])
In[26]: x2[:3, ::2] # all rows, every other column
Out[26]: array([[12, 2],
               [ 7, 8],
               [ 1, 7]])

```

Finally, subarray dimensions can even be reversed together:

```

In[27]: x2[::-1, ::-1]
Out[27]: array([[ 7, 7, 6, 1],
               [ 8, 8, 6, 7],
               [ 4, 2, 5, 12]])

```

```
In [30]: a = np.array([[ -7, 0, 10, 20],  
[ -5, 1, 40, 200],  
[ -1, 1, 4, 30]])  
print(a)
```

```
[[ -7   0  10  20]  
 [ -5   1  40 200]  
 [ -1   1   4  30]]
```

```
In [31]: # access all the elements in the 3rd column  
a[0:3,2]
```

```
Out[31]: array([10, 40,  4])
```

```
In [33]: # access all the elements in the first row and 3rd column  
a[0:1,2]
```

```
Out[33]: array([10])
```

```
In [35]: # access elements from first, second row and 3rd column  
print(a[0:2,2])
```

```
[10 40]
```

```
In [38]: # access all the elements in the second row till last row and 3rd column till last column  
print(a[1:,2:])
```

```
[[ 40 200]  
 [  4  30]]
```

```
In [36]: # access elements from first two rows and first two columns  
print(a[0:2,0:2])
```

```
[[ -7   0]  
 [ -5   1]]
```

```
In [40]: # access elements from first till second last rows and first to second last columns  
print(a[:-1,:-1])
```

```
[[ -7   0  10]  
 [ -5   1  40]]
```

```
In [43]: # reverse the List  
print(a[::-1,:-1])
```

```
[[ 30   4   1  -1]  
 [200  40   1  -5]  
 [ 20  10   0  -7]]
```

## Subarrays as no-copy views

One important—and extremely useful—thing to know about array slices is that they return views rather than copies of the array data. This is one area in which NumPy array slicing differs from Python list slicing: in lists, slices will be copies. Consider our two-dimensional array from before:

```
In[31]: print(x2)
[[12 5 2 4]
 [ 7 6 8 8]
 [ 1 6 7 7]]
```

Let's extract a 2×2 subarray from this:

```
In[32]: x2_sub = x2[:2, :2]
print(x2_sub)
[[12 5]
 [ 7 6]]
```

Now if we modify this subarray, we'll see that the original array is changed! Observe:

```
In[33]: x2_sub[0, 0] = 99
print(x2_sub)
[[99 5]
 [ 7 6]]
In[34]: print(x2)
[[99 5 2 4]
 [ 7 6 8 8]
 [ 1 6 7 7]]
```

This default behavior is actually quite useful: it means that when we work with large datasets, we can access and process pieces of these datasets without the need to copy the underlying data buffer.

## Copy() function:

The `copy()` function is used to create a new array from an array. The newly created array is independent of the array from which it is created. For example

```
In[35]: x2_sub_copy = x2[:2, :2].copy()
print(x2_sub_copy)
[[99 5]
 [ 7 6]]
```

If we now modify this subarray, the original array is not touched:

```
In[36]: x2_sub_copy[0, 0] = 42
print(x2_sub_copy)
[[42 5]
 [ 7 6]]
In[37]: print(x2)
[[99 5 2 4]
 [ 7 6 8 8]
 [ 1 6 7 7]]
```

# Operation in Array

Once the arrays are created we can do many operations on that.

## Arithmetic Operatios

Arithmetic operations on NumPy arrays are fast and simple. When we perform a basic arithmetic operation like addition, subtraction, multiplication, division etc. on two arrays, these operations are done componentwise. It is important to note that for componentwise operations the size of both the arrays must be same.

In [3]: `import numpy as np`

In [14]: `a1=np.array([12,3])  
a2=np.array([-1,3])  
a3=a1+a2  
a4=a1-a2  
a5=a1*a2  
a6=a1/a2  
a7=a1**3  
a8=a1%a2  
print('a1 = ',a1)  
print('a2 = ',a2)  
print('a3 = ',a3)  
print('a4 = ',a4)  
print('a5 = ',a5)  
print('a6 = ',a6)  
print('a7 = ',a7)  
print('b8 = ',a8)`

a1 = [12 3]  
a2 = [-1 3]  
a3 = [11 6]  
a4 = [13 0]  
a5 = [-12 9]  
a6 = [-12. 1.]  
a7 = [1728 27]  
b8 = [0 0]

In [16]: `b1 = np.array([[3,6],[4,2]])  
b2 = np.array([[10,20],[15,12]])  
print('b3 = ',b1+b2)  
print('b4 = ',b1-b2)  
print('b5 = ',b1*b2)  
print('b6 = ',b1+b2)  
print('b7 = ',b1**3)  
print('b8 = ',b1%b2)`

b3 = [[13 26]  
[19 14]]  
b4 = [[ -7 -14]  
[-11 -10]]  
b5 = [[ 30 120]  
[ 60 24]]  
b6 = [[13 26]  
[19 14]]  
b7 = [[ 27 216]  
[ 64 8]]  
b8 = [[3 6]  
[4 2]]

## randn() Function

The `numpy.random.randn()` function creates an array of specified shape and fills it with random values as per standard normal distribution. For example

In [21]: `data = np.random.randn(7, 4)  
print(data)`

[[-1.77351625 -1.10390642 0.30808929 -1.1366876 ]  
[-1.58186217 0.98418295 -0.04660054 -0.2191429 ]  
[ 0.18969758 0.45400306 0.12902892 1.47863921]  
[-0.80875858 -0.35306668 -0.52944731 -0.85369725]  
[-1.09200602 0.18737241 1.6906511 0.60859627]  
[ 0.15779789 -1.26639515 -1.03407842 0.22595627]  
[ 0.04007042 0.43850161 -1.05063144 0.39280082]]

In [23]: `data1 = np.random.randn(4)  
print(data1)`

[-2.33515463 1.40166829 -0.38801288 0.18185373]

## Universal Functions

A universal function, or ufunc, is a function that performs element-wise operations on data in ndarrays. They are of two types: unary ufuncs and binary ufuncs. The unary ufuncs take only one array to be operate. Some of the unary ufuncs are as follows:

- 1. **sqrt:** Gives the square root of each element of the array.

In [26]: `u1=np.arange(10)  
print('The array U1 is')  
print(u1)  
print('Square root of each element of array u1 '  
print(np.sqrt(u1))`

The array U1 is  
[0 1 2 3 4 5 6 7 8 9]  
Square root of each element of array u1  
[0. 1. 1.41421356 1.73205081 2. 2.23606798  
2.44948974 2.64575131 2.82842712 3. ]

- 1. **exp:** This function gives the e to the power of each element of the array

In [27]: `u1=np.arange(10)  
print('The array U1 is')  
print(u1)  
print('Exponential each element of array u1 '  
print(np.exp(u1))`

The array U1 is  
[0 1 2 3 4 5 6 7 8 9]  
Exponential each element of array u1  
[1.00000000e+00 2.71828183e+00 7.38905610e+00 2.00855369e+01  
5.45981500e+01 1.48413159e+02 4.03428793e+02 1.09663316e+03  
2.98095799e+03 8.10308393e+03]

Some of the unary function is as follows

Function	Description
<b>abs, fabs</b>	Compute the absolute value element-wise for integer, floating-point, or complex values
<b>sqrt</b>	Compute the square root of each element (equivalent to <code>arr ** 0.5</code> )
<b>square</b>	Compute the square of each element (equivalent to <code>arr ** 2</code> )
<b>exp</b>	Compute the exponent $e^x$ of each element
<b>log, log10, log2, log1p</b>	Natural logarithm (base e), log base 10, log base 2, and $\log(1 + x)$ , respectively
<b>sign</b>	Compute the sign of each element: 1 (positive), 0 (zero), or $-1$ (negative)
<b>ceil</b>	Compute the ceiling of each element (i.e., the smallest integer greater than or equal to that number)
<b>floor</b>	Compute the floor of each element (i.e., the largest integer less than or equal to each element)
<b>rint</b>	Round elements to the nearest integer, preserving the dtype
<b>modf</b>	Return fractional and integral parts of array as a separate array
<b>isnan</b>	Return boolean array indicating whether each value is NaN (Not a Number)
<b>isfinite, isinf</b>	Return boolean array indicating whether each element is finite (non- <code>inf</code> , non-NaN) or infinite, respectively
<b>cos, cosh, sin, sinh, tan, tanh</b>	Regular and hyperbolic trigonometric functions
<b>arccos, arccosh, arcsin, arcsinh, arctan, arctanh</b>	Inverse trigonometric functions
<b>logical_not</b>	Compute truth value of not $x$ element-wise (equivalent to <code>~arr</code> ).

In [ ]:



There some function which change the structure of the array.

- 1. Transpose:** Transposing an array turns its rows into columns and columns into rows just like matrices in mathematics. The function transpose is used for that

In [1]: `import numpy as np`

In [10]: `al=np.random.randn(2,5)
print('The array A1 is')
print('')
print(al)
print('')
print('The transpose of the array A1 is')
print('')
print(np.transpose(al))`

The array A1 is

```
[[ 0.41038006  1.03842333 -0.98126279 -0.54972313  0.14663553]
 [ 0.89326192  1.46211564  0.94852804  1.05110377  0.78072158]]
```

The transpose of the array A1 is

```
[[ 0.41038006  0.89326192]
 [ 1.03842333  1.46211564]
 [-0.98126279  0.94852804]
 [-0.54972313  1.05110377]
 [ 0.14663553  0.78072158]]
```

- 1. Sorting:** Sorting is to arrange the elements of an array in ascending order.

In [9]: `s1=np.array([2,1,12,45,6,23,78])
print('The original array is')
print('')
print(s1)
print('')
print('The sorted array is')
print('')
print(np.sort(s1))`

The original array is

```
[ 2  1 12 45  6 23 78]
```

The sorted array is

```
[ 1  2  6 12 23 45 78]
```

In 2-D array, sorting can be done along either of the axes i.e., row-wise or column-wise. By default, sorting is done row-wise (i.e., on axis = 1). It means to arrange elements in each row in ascending order. When axis=0, sorting is done column-wise, which means each column is sorted in ascending order.

In [14]: `s2=np.random.randn(2,5)
print('The oringinal array')
print('')
print(s2)
print('')
print('The sorted array along the first axis (i.e. along row)')
print('')
print(np.sort(s2, axis=1))
print('')
print('The sorted array along the second axis (i.e. along column)')
print('')
print(np.sort(s2, axis=0))
print('')
print('The sorted array along the no axis')
print('')
print(np.sort(s2, axis=None))`

The oringinal array

```
[[ 2.17576624 -1.52018752  1.1285782   0.28196439 -2.17054025]
 [ 0.23584126  0.88219983  0.73243289 -1.74814777 -0.35624128]]
```

The sorted array along the first axis (i.e. along row)

```
[[-2.17054025 -1.52018752  0.28196439  1.1285782   2.17576624]
 [-1.74814777 -0.35624128  0.23584126  0.73243289  0.88219983]]
```

The sorted array along the second axis (i.e. along column)

```
[[ 0.23584126 -1.52018752  0.73243289 -1.74814777 -2.17054025]
 [ 2.17576624  0.88219983  1.1285782   0.28196439 -0.35624128]]
```

The sorted array along the no axis

```
[-2.17054025 -1.74814777 -1.52018752 -0.35624128  0.23584126  0.28196439
 0.73243289  0.88219983  1.1285782   2.17576624]
```

- 3. Concatenating Arrays:** Concatenation means joining two or more arrays. Concatenating 1-D arrays means appending the sequences one after another. This can be done by NumPy.concatenate() function.

In [26]: `c1=np.arange(2,15,3)
c2=np.arange(4,16,5)
c3=np.arange(3,12,5)
print(c1,c2,sep=',')
print('')
print('Array after concatenating')
print('')
print(np.concatenate([c1,c2,c3]))`

```
[ 2  5  8 11 14],[ 2  5  8 11 14],[ 4  9 14]
```

Array after concatenating

```
[ 2  5  8 11 14  4  9 14  3  8]
```

In 2-D or more concatenation can be done either by row-wise or column wise. For this all the dimensions of the arrays to be concatenated must match exactly except for the dimension or axis along which they need to be joined. Any mismatch in the dimensions results in an error. By default, the concatenation of the arrays happens along axis=0.

In [50]: `c4=np.random.randn(2,2)
c5=np.random.randn(2,2)
c6=np.random.randn(2,3)
print(c4,c5,c6 ,sep=',')
print('')
print('Concatenating along the x axis')
print('')
print(np.concatenate([c4,c5]))
print('')
print('Concatenating along the x axis')
print('')
print(np.concatenate([c5,c6],axis=1))
print('')
print('Concatenating along the y axis')
print('')
print(np.concatenate([c5,c6],axis=0))`

```
[[ 0.09500508 -1.27580884]
 [-0.33916872 -1.56900857]],[[ 0.15441892 -0.86156922]
 [ 1.78138543 -0.10746438]],[[ 0.73486473  1.65666169 -0.64059904]
 [-0.87095297  0.70251646  0.3031635 ]]
```

Concatenating along the x axis

```
[[ 0.09500508 -1.27580884]
 [-0.33916872 -1.56900857]
 [ 0.15441892 -0.86156922]
 [ 1.78138543 -0.10746438]]
```

Concatenating along the x axis

```
[[ 0.15441892 -0.86156922  0.73486473  1.65666169 -0.64059904]
 [ 1.78138543 -0.10746438 -0.87095297  0.70251646  0.3031635 ]]
```

Concatenating along the y axis

```
-----
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_16648\192846319.py in <module>
     14 print('Concatenating along the y axis')
     15 print('')
--> 16 print(np.concatenate([c5,c6],axis=0))

<_array_function__ internals> in concatenate(*args, **kwargs)

ValueError: all the input array dimensions for the concatenation axis must match exactly, but along dimension 1, the array at index 0 has size 2 and the array at index 1 has size 3
```

- 4. Reshaping Arrays:**

We can modify the shape of an array using the reshape() function. Reshaping an array cannot be used to change the total number of elements in the array. Attempting to change the number of elements in the array using reshape() results in an error.

In [37]: `r1=np.arange(12,24)
print(r1)
print('')
print('Making an array of size (3,4)')
print('')
print(r1.reshape(3,4))
print('')
print('Making an array of size (4,3)')
print('')
print(r1.reshape(4,3))
print('')
print('Making an array of size (2,6)')
print('')
print(r1.reshape(2,6))`

```
[12 13 14 15 16 17 18 19 20 21 22 23]
```

Making an array of size (3,4)

```
[[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
```

Making an array of size (4,3)

```
[[12 13 14]
 [15 16 17]
 [18 19 20]
 [21 22 23]]
```

Making an array of size (2,6)

```
[[12 13 14 15 16 17]
 [18 19 20 21 22 23]]
```

## Statistical Operations on Arrays

NumPy provides functions to perform al most all statistical operations on arrays. Some of which is as follows:

- 1. max() and min() function:** These function gives the maximum and minimum of the array elements.

- 2. sum() function:** Find the sum of all elements.

- 3. mean() function:** Find the mean of the elements of the array.

- 4. std() Function:** Find the standard deviation of elements of array.

In [49]: `st1=np.random.randn(1,5)
st2=np.random.randn(2,3)
print('st1= ',st1)
print('')
print('st2= ',st2)
print('')
print('The maximum in st1', np.max(st1))
print('')
print('The row wise maximum in st2',np.max(st2,axis=1))
print('')
print('The column wise maximum in st2',np.max(st2,axis=0))
print('')
print('The minimum in st1', np.min(st1))
print('')
print('The row wise minimum in st2',np.min(st2,axis=1))
print('')
print('The column wise minimum in st2',np.min(st2,axis=0))
print('')
print('The sum of all elements of st1', np.sum(st1))
print('')
print('The row wise sum of all elements of st2',np.sum(st2,axis=1))
print('')
print('The column wise sum of all elements of st2',np.sum(st2,axis=0))
print('')
print('The mean of elements of st1', np.mean(st1))
print('')
print('The row wise mean of all elements of st2',np.mean(st2,axis=1))
print('')
print('The column wise mean of all elements of st2',np.mean(st2,axis=0))
print('')
print('The standard deviation of elements of st1', np.std(st1))
print('')
print('The row wise standard deviation of all elements of st2',np.std(st2,axis=1))
print('')
print('The column wise standard deviation of all elements of st2',np.std(st2,axis=0))`

```
st1= [[-0.25227306  1.31967614  0.64491644  0.15350853  1.41716489]]
```

```
st2= [[ 0.15455361  1.2828916  0.148810935]
 [-1.38551377 -0.15488507  0.17706999]]
```

The maximum in st1 1.4171648914698607

The row wise maximum in st2 [2.00810935 0.17706999]

The column wise maximum in st2 [0.15455361 1.2828916 2.00810935]

The minimum in st1 -0.2522730614189392

The row wise minimum in st2 [ 0.15455361 -1.38551377]

The column wise minimum in st2 [-1.38551377 -0.15488507 0.17706999]

The sum of all elements of st1 3.2829929465085828

The row wise sum of all elements of st2 [ 3.44555456 -1.36332885]

The column wise sum of all elements of st2 [-1.23096015 1.12800653 2.18517934]

The mean of elements of st1 0.6565985893017166

The row wise mean of all elements of st2 [ 1.14851819 -0.45444295]

The column wise mean of all elements of st2 [-0.61548008 0.56400326 1.09258967]

The standard deviation of elements of st1 0.6476753348188751

The row wise standard deviation of all elements of st2 [0.76265299 0.67216971]

The column wise standard deviation of all elements of st2 [0.77003369 0.71888834 0.91551968]

Some more statistical function is given in following table:

Function Name	NaN-safe Version	Description
<code>np.sum</code>	<code>np.nansum</code>	Compute sum of elements
<code>np.prod</code>	<code>np.nanprod</code>	Compute product of elements
<code>np.mean</code>	<code>np.nanmean</code>	Compute median of elements
<code>np.std</code>	<code>np.nanstd</code>	Compute standard deviation
<code>np.var</code>	<code>np.nanvar</code>	Compute variance
<code>np.min</code>	<code>np.nanmin</code>	Find minimum value
<code>np.max</code>	<code>np.nanmax</code>	Find maximum value
<code>np.argmin</code>	<code>np.nanargmin</code>	Find index of minimum value
<code>np.argmax</code>	<code>np.nanargmax</code>	Find index of maximum value
<code>np.median</code>	<code>np.nanmedian</code>	Compute median of elements
<code>np.percentile</code>	<code>np.nanpercentile</code>	Compute rank-based statistics of elements
<code>np.any</code>	N/A	Evaluate whether any elements are true
<code>np.all</code>	N/A	Evaluate whether all elements are true

**Guru Nanak Khalsa College- (Autonomous)**  
**Department of Mathematics & Statistics**

**SEM VI Python Programming**  
**Chapter 5 Linear Algebra with NumPy**

Almost all concept of linear algebra can be done with the help of NumPy. Some examples of the same is as given below.

A linear algebra matrix is nothing but the array created in NumPy. We have following operation which multiply two arrays differently.

1. `array1*array2`: The `*` operator will work only when both the array are of same size. This will return a new array of size same as of `array1`, in which each element of new array is multiplication of corresponding elements.
2. `np.dot(array1, array2)`: This is simply usual matrix multiplication of two array provided they are compatible for matrix multiplication. One more advantage with `dot()` is that it multiply a scalar with each element of the matrix.
3. `np.vdot()`: This function is same as the `dot()` function with only difference that both the arrays in `vdot()` must be of same sizes otherwise it display the error.
4. `Np.matmul()`: This function will calculate the matrix multiplication. One of the difference between `dot()` and `matmul()` is that scalar multiplication is not allowed in `matmul()`.

```
In [18]: a1=np.array([[1,2,3],[4,5,6],[7,8,9]])  
         a2=np.array([[7,8,9],[1,2,3],[4,5,6]])
```

```
In [19]: print(a1*a2)  
         print('')  
         print(np.dot(a1,a2))  
         print('')  
         print(a1.dot(a2)) # other way to use dot()  
         print('')  
         print(np.matmul(a1,a2))  
         print(np.vdot(a1,a2))
```

```
In [14]: a3=a1[0:,0:2]
print(a3)
print('')
print(np.matmul(a1,a3))
print('')
print(np.dot(a1,a3))
print('')
print(np.vdot(a1,a3)) # This is displaying error.
```

```
In [15]: print(np.dot(5,a1))
```

```
[[ 5 10 15]
 [20 25 30]
 [35 40 45]]
```

```
In [16]: print(np.matmul(5,a1)) # Error
```

Linalg: linalg is a package in NumPy consisting of various linear algebraically operators. These operators are used with linalg as prefix.

1. det(): np.linalg.det() is used to find the determinant of the given matrix.

```
In [27]: print(np.linalg.det(np.array([[1,2],[2,1]])))

-2.9999999999999996
```

2. trace(): np.trace() is used to find the trace of the square matrix.

```
In [29]: print(np.trace(a1))

15
```

3. transpose(): np.transpose(), returns the transpose of the given matrix. The same effect can be obtained using .T operator.

```
In [31]: print(np.transpose(a1))
print(a1.T)
```

4. matrix\_rank(): The statement np.linalg.matrix\_rank() returns the rank of the given matrix.

```
In [34]: print(np.linalg.matrix_rank(a1))

2
```



5. `identity()`. This function generate the identity matrix of given size and datatype.

```
In [44]: print(np.identity(3,int))
```

```
[[1 0 0]
 [0 1 0]
 [0 0 1]]
```

6. `inv()`: The statement `np.linalg.inv()` retruns the inverse of the matrix if it exists.

```
In [35]: print(np.linalg.inv(np.array([[1,2],[2,1]])))
```

```
[[ -0.33333333  0.66666667]
 [ 0.66666667 -0.33333333]]
```

7. `matrix_power()`: This function takes two argument, one is the matrix and second is an integer. This function return the matrix raised to the given power.

```
In [36]: print(np.linalg.matrix_power(a1,2))
```

```
[[ 30  36  42]
 [ 66  81  96]
 [102 126 150]]
```

8. `eig()`: The function `np.linalg.eig()` will compute the eigen values and eigen vectors of a square matrix.

```
In [40]: c,d=np.linalg.eig(a1)
print('The eigen values are ',c)
print('The eigen vectors are ')
print('')
print(d)
```

The eigen values are [ 1.61168440e+01 -1.11684397e+00 -3.38433605e-16]

The eigen vectors are

```
[[ -0.23197069 -0.78583024  0.40824829]
 [ -0.52532209 -0.08675134 -0.81649658]
 [ -0.8186735   0.61232756  0.40824829]]
```

9. solve(): The function `np.linalg.solve()` take two arguments, the first one is the coefficient matrix of system of linear equations whereas the other arguments is the column matrix which is the right hand side of equations.

```
In [43]: a4=a1[0:3,0:1]
          print(a4)
          print(np.linalg.solve(a1,a4))
```

```
[[1]
 [4]
 [7]]
[[-0.225]
 [ 2.45 ]
 [-1.225]]
```

**Guru Nanak Khalsa College- (Autonomous)**  
**Department of Mathematics & Statistics**

**SEM VI Python Programming**  
**Chapter 6 File Handling with NumPy**

Sometimes, we may have data in files and we may need to load that data in an array for processing. `numpy.loadtxt()` and `numpy.genfromtxt()` are the two functions that can be used to load data from text files. The most commonly used file type to handle large amount of data is called CSV (Comma Separated Values). Each row in the text file must have the same number of values in order to load data from a text file into a numpy array. Both `loadtxt()` and `genfromtxt()` load txt file.

`loadtxt()`: This function create numpy array using text file. Following is the syntax of `loadtxt()`.

`numpy.loadtxt(fname, dtype=<class 'float'>, delimiter=None, skiprows=0)`

`fname` – File, filename, path of txt file to be imported.

`dtype` – Data-type of the resulting array.

`delimiter` – The string is used to separate values. By default, this is any whitespace. The delimiter to use for parsing the content of txt a file.

`skiprows` – Skip the first skip rows lines, including comments; default: 0.

```
In [4]: smark=np.loadtxt('E:/Class 22 - 23/filenumpy.txt',skiprows=1, delimiter=',',dtype=int)
smark
Out[4]: array([[110, 23, 34, 45],
               [111, 34, 43, 12]])
```

We can load each row or column of the data file into different numpy arrays using the `unpack` parameter. By default, `unpack=False` means we can extract each row of data as separate arrays. When `unpack=True`, the returned array is transposed means we can extract the columns as separate arrays.

```
In [8]: stud1, stud2=np.loadtxt('E:/Class 22 - 23/filenumpy.txt',skiprows=1,delimiter=',',dtype=int)
print(stud1, stud2)

[110 23 34 45] [111 34 43 12]

In [10]: r,maths,phy,chl=np.loadtxt('E:/Class 22 - 23/filenumpy.txt',skiprows=1, delimiter=',',dtype=int,unpack=True)
print(r,maths,phy,chl)

[110 111] [23 34] [34 43] [45 12]
```

`genfromtxt()`: NumPy this function can read a text file in which some data is missing. The `genfromtxt()` function converts missing values and character strings in numeric columns to `nan`. But if we specify `dtype` as `int`, it converts the

missing or other non-numeric values to -1. We can also convert these missing values and character strings in the data files to some specific value using the parameter filling values.

```
In [13]: smark1=np.genfromtxt('E:/Class 22 - 23/file2.txt',skip_header=1, delimiter=',')
smark1
```

```
Out[13]: array([[101.,  34.,  45.,  65.],
               [102.,  34.,  56.,  nan],
               [103.,  45.,  nan,  56.],
               [104.,  nan,  34.,  56.]])
```

```
In [14]: smark2=np.genfromtxt('E:/Class 22 - 23/file2.txt',skip_header=1, delimiter=',',dtype=int)
smark2
```

```
Out[14]: array([[101,  34,  45,  65],
               [102,  34,  56,  -1],
               [103,  45,  -1,  56],
               [104,  -1,  34,  56]])
```

```
In [15]: smark3=np.genfromtxt('E:/Class 22 - 23/file2.txt',skip_header=1, delimiter=',',dtype=int, filling_values=-999)
smark3
```

```
Out[15]: array([[ 101,  34,  45,  65],
               [ 102,  34,  56, -999],
               [ 103,  45, -999,  56],
               [ 104, -999,  34,  56]])
```

## Saving NumPy Arrays in Files on Disk

The savetxt() function is used to save a NumPy array to a text file.

```
In [20]: np.savetxt('E:/Class 22 - 23/file3.txt',smark3, delimiter=',',fmt='%i')
```

We have used parameter fmt to specify the format in which data are to be saved. The default is float.