NumPy Fundamentals

What is NumPy?

- NumPy stands for Numerical Python.
- NumPy is a Python library used for working with arrays.
- It also has functions for working in domain of linear algebra, fourier transform, and matrices.
- NumPy was created in 2005 by Travis Oliphant. It is an open source project and you can use it freely.

Why Use NumPy?

- Arrays are very frequently used in data science, where speed and resources are very important.
- In Python we have lists that serve the purpose of arrays, but they are slow to process.
- NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.
- Its primary goal is to facilitate complex mathematical and scientific operations by introducing array-oriented computing capabilities. NumPy's design allows for seamless integration with other scientific libraries, enabling faster execution of numerical tasks.
- The array object in NumPy is called `ndarray`. A lot of supporting functions have been made available that make working with `ndarray` very easy.

Installation of NumPy

- If you have Python and PIP already installed on a system, then installation of NumPy is very easy.
- Install it using this command:

```
C:\Users\Your Name>pip install numpy
```

- If this command fails, then use a python distribution that already has NumPy installed like, Anaconda, Spyder etc.

NumPy Basics

- Once NumPy is installed, import it in your applications by adding the import keyword:

```
import numpy
```

- Now NumPy is imported and ready to use.

Example

```python
import numpy

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

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

- NumPy is usually imported under the np alias.

Example

```python
import numpy as np

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

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

- The array function copies its argument's dimensions. Let's create an array from a two-row-by-three-column list.

```python
import numpy as np

arr2D = np.array([[1, 2, 3], [4, 5, 6]])
print(arr2D)
```

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

- We can create a one-dimensional array from a list comprehension also.

```python
import numpy as np

arr = np.array([x for x in range(2, 21, 2)])
print(arr)
```

```
[ 2  4  6  8 10 12 14 16 18 20]
```

- We can create a two-dimensional array from a list comprehension also.

```python
import numpy as np

arr2D = np.array([[3*x+2*y for y in range(5)] for x in range(10)])
print(arr2D)
```

```
[[ 0  2  4  6  8]
 [ 3  5  7  9 11]
 [ 6  8 10 12 14]
 [ 9 11 13 15 17]
 [12 14 16 18 20]
 [15 17 19 21 23]
 [18 20 22 24 26]
 [21 23 25 27 29]
 [24 26 28 30 32]
 [27 29 31 33 35]]
```

Accessing the array Index

- In a NumPy array, indexing or accessing the array index can be done in multiple ways.
- To print a range of an array, slicing is done. Slicing of an array is defining a range in a new array which is used to print a range of elements from the original array.
- Since, sliced array holds a range of elements of the original array, modifying content with the help of sliced array modifies the original array content.

Example

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

# Initial Array
arr = np.array([[-1, 2, 0, 4],
                [4, -0.5, 6, 0],
                [2.6, 0, 7, 8],
                [3, -7, 4, 2.0]])
print("Initial Array: ")
print(arr)

# Printing a range of Array with the use of slicing method
sliced_arr = arr[:2, ::2]
print ("\nArray with first 2 rows and"
    " alternate columns(0 and 2):\n", sliced_arr)

```

```
# Printing elements at a specific Index
print('\nElement at index (1,3):', arr[1][3])
# Printing elements at multiple Indices
Index_arr = arr[[1, 1, 0, 3],
                [3, 2, 1, 0]]
print ("\nElements at indices (1, 3), "
    "(1, 2), (0, 1), (3, 0):\n", Index_arr)
```

```
Initial Array:
[[-1.   2.   0.   4. ]
 [ 4.  -0.5  6.   0. ]
 [ 2.6  0.   7.   8. ]
 [ 3.  -7.   4.   2. ]]

Array with first 2 rows and alternate columns(0 and 2):
 [[-1.  0.]
 [ 4.  6.]]

Element at index (1,3): 0.0

Elements at indices (1, 3), (1, 2), (0, 1), (3, 0):
 [0. 6. 2. 3.]
```

Basic Array Operations

- In NumPy, arrays allow a wide range of operations which can be performed on a particular array or a combination of Arrays.
- These operations include some basic Mathematical operation as well as Unary and Binary operations.

Example

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

# Defining Array 1
a = np.array([[1, 2],
              [3, 4]])

# Defining Array 2
b = np.array([[4, 3],
              [2, 1]])
print('a=\n', a, '\nb=\n', b)

# Adding 1 to every element
print ("Adding 1 to every element of a:\n", a + 1)

# Subtracting 2 from each element
print ("\nSubtracting 2 from each element of b:\n", b - 2)

# sum of array elements Performing Unary operations
print ("\nSum of all array elements of a: ", a.sum())

# Adding two arrays Performing Binary operations
print ("\nArray sum (a+b):\n", a + b)
```

```
a=
 [[1 2]
 [3 4]]
b=
 [[4 3]
 [2 1]]
Adding 1 to every element of a:
 [[2 3]
 [4 5]]

Subtracting 2 from each element of b:
 [[ 2  1]
 [ 0 -1]]

Sum of all array elements of a:  10

Array sum (a+b):
 [[5 5]
 [5 5]]
```

Creating arrays

- Creating Integer arrays with `arange`:

```
import numpy as np
print(np.arange(5))
print(np.arange(5, 10))
print(np.arange(1, 10, 3))
print(np.arange(10, 1, -2))
```

```
[0 1 2 3 4]
[5 6 7 8 9]
[1 4 7]
[10  8  6  4  2]
```

- Creating Floating-Point arrays with `linspace`:

```
import numpy as np
print(np.linspace(0.0, 1.0, 21))
```

```
[0.   0.05 0.1  0.15 0.2  0.25 0.3  0.35 0.4  0.45 0.5  0.55 0.6  0.65
 0.7  0.75 0.8  0.85 0.9  0.95 1.  ]
```

- Filling arrays with Specific Values using `zeros`, `ones` and `full`:
    - NumPy provides functions `zeros`, `ones` and `full` for creating **arrays** containing 0s, 1s or a specified value, respectively.
    - By default, `zeros` and `ones` create arrays containing `float64` values.
    - The array returned by `full` contains elements with the second argument's value and type.

```
import numpy as np
print(np.zeros(5))
print(np.ones(7))
print(np.full((3, 5), 13))
```

```
[0. 0. 0. 0. 0.]
[1. 1. 1. 1. 1. 1. 1.]
[[13 13 13 13 13]
 [13 13 13 13 13]
 [13 13 13 13 13]]
```

Creating arrays of random numbers

- Random number does NOT mean a different number every time. Random means something that cannot be predicted logically.
- 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.
- Random numbers generated through a generation algorithm are called **pseudo random**.
- NumPy offers the `random` module to work with random numbers.
- Syntax

  `numpy.random.rand(d0, d1, …, dn)`
    - **Parameters:**

      d0, d1, ..., dn : [int, optional]Dimension of the returned array we require, If no argument is given a single Python float is returned.

      Create an array of the given shape and populate it with random samples from a **uniform distribution** over [0, 1).
    - **Return:**

      Array of defined shape, filled with random values.

`numpy.random.randint(low, high, size, dtype)`

- Return random integers from low (inclusive) to high (exclusive).
- Return random integers from the "discrete uniform" distribution of the specified dtype in the "half-open" interval [low, high). If high is None (the default), then results are from [0, low).

`random.randn(d0, d1, ..., dn)`

- Return a sample (or samples) from the "**standard normal**" distribution (zero-mean, unit variance).
- **Parameters:**
  `d0, d1, …, dnint, optional`
  The dimensions of the returned array, must be non-negative. If no argument is given a single Python float is returned.
- **Returns:**
  `ndarray` or `float`
- A (d0, d1, ..., dn)-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

```python
# Python Program illustrating numpy.random.rand() method
import numpy as np

# 1D Floating Point Array
array1 = np.random.rand(5)
print("1D Array :\n", array1);
array2 = np.random.rand(5,3)
print("2D Array :\n", array2);
print()
# 1D Integer Array
array3 = np.random.randint(10)
print("Scalar :\n", array3);
array4 = np.random.randint(10, size=5)
print("1D Array :\n", array4);
array5 = np.random.randint(10, size=(3,5))
print("2D Array :\n", array5);
```

```
1D Array :
 [0.5077935  0.99150585 0.05332898 0.963807   0.34448639]
2D Array :
 [[0.70514576 0.20538881 0.84848959]
 [0.3158575  0.18197601 0.04306961]
 [0.23590684 0.91817376 0.80755967]
 [0.64218403 0.46194465 0.99963507]
 [0.52500989 0.95899822 0.06057017]]

Scalar :
 4
1D Array :
 [0 5 3 0 8]
2D Array :
 [[5 7 0 5 7]
 [4 8 1 7 6]
 [7 9 8 8 5]]
```

Reshaping an array

- You also can create an array from a range of elements, then use array method `reshape` to transform the one-dimensional array into a multidimensional array.
- The number of elements in both the arrays must be same otherwise `ValueError` is generated.

```python
import numpy as np

print(np.arange(1, 16).reshape(3, 5), '\n')
print(np.zeros(15).reshape(3, 5), '\n')
print(np.ones(15).reshape(3, 5), '\n')

```

```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]]

[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]

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

Array Operators

- Element-wise operations

```python
import numpy as np

numbers = np.arange(1, 6)
print(numbers, '\n')
print(numbers*2, '\n')
print(numbers*6.5, '\n')
print(numbers**2, '\n')
```

```
[1 2 3 4 5]

[ 2  4  6  8 10]

[ 6.5 13.  19.5 26.  32.5]

[ 1  4  9 16 25]
```

- **Arithmetic Operations Between arrays**
  - You may perform elementwise arithmetic operations and augmented assignments between arrays of the same shape.

```python
import numpy as np

numbers2 = np.arange(1, 5)
numbers1 = np.linspace(11, 50, 4)

print(numbers1,)
print(numbers2, '\n')

print(numbers1+numbers2)
print(numbers1-numbers2)
print(numbers1*numbers2)
print(numbers1/numbers2)
print(numbers1**numbers2)
```

```
[11. 24. 37. 50.]
[1 2 3 4]

[12. 26. 40. 54.]
[10. 22. 34. 46.]
[ 11.  48. 111. 200.]
[11.         12.         12.33333333 12.5       ]
[1.1000e+01 5.7600e+02 5.0653e+04 6.2500e+06]
```

- **Comparison Operations Between arrays**

- Comparisons are performed elementwise. Such comparisons produce arrays of Boolean values in which each element's `True` or `False` value indicates the comparison result.

```
import numpy as np

arr1 = np.array([10, 14, 13, 16, 15])
arr2 = np.array([11, 16, 14, 12, 17])

print(arr1,)
print(arr2, '\n')

print(arr1>=15)
print(arr2==12)
print(arr1<arr2)
```

```
[10 14 13 16 15]
[11 16 14 12 17]

[False False False  True  True]
[False False False  True False]
[ True  True  True False  True]
```

NumPy Calculation Methods

- We can use methods to calculate sum, minimum, maximum, mean, standard deviation and variance of the arrays in NumPy.

```
import numpy as np
grades = np.array([[87,  96, 70],
                   [100, 87, 90],
                   [94,  77, 90],
                   [100, 81, 82]])
print('     Sum = ', grades.sum())
print('     Max = ', grades.max())
print('     Min = ', grades.min())
print(' Average = ', grades.mean())
print(' Std Dev = ', grades.std())
print('Variance = ', grades.var())
```

```
     Sum =  1054
     Max =  100
     Min =  70
 Average =  87.83333333333333
 Std Dev =  8.792357792739987
Variance =  77.30555555555556
```

Calculations by Row or Column

- Specifying `axis=0` performs the calculation on all the row values within each column.
- Specifying `axis=1` performs the calculation on all the column values within each column.

```
import numpy as np
grades = np.array([[87,  96, 70],
                   [100, 87, 90],
                   [94,  77, 90],
                   [100, 81, 82]])
print('Max of Cols = ', grades.max(axis = 0))
print('Max of Rows = ', grades.max(axis = 1))

```

```
Max of Cols =  [100  96  90]
Max of Rows =  [ 96 100  94 100]
```

NumPy Universal functions

- NumPy Universal functions (`ufuncs` in short) are simple mathematical functions that operate on `ndarray` (N-dimensional array) in an element-wise fashion.
- It supports array broadcasting, type casting, and several other standard features.
- NumPy provides various universal functions like standard trigonometric functions, functions for arithmetic operations, handling complex numbers, statistical functions, etc.

- **Basic Universal Functions in NumPy**
  - All of the arithmetic operations are simply convenient wrappers around specific `ufuncs` built into NumPy.

Operator	Equivalent ufunc	Description
`+`	`np.add`	Addition (e.g., 1 + 1 = 2)
`-`	`np.subtract`	Subtraction (e.g., 3 - 2 = 1)
`-`	`np.negative`	Unary negation (e.g., -2)
`*`	`np.multiply`	Multiplication (e.g., 2 * 3 = 6)
`/`	`np.divide`	Division (e.g., 3 / 2 = 1.5)
`//`	`np.floor_divide`	Floor division (e.g., 3 // 2 = 1)
`**`	`np.power`	Exponentiation (e.g., 2 ** 3 = 8)
`%`	`np.mod`	Modulus/remainder (e.g., 9 % 4 = 1)
`abs`	`np.abs`	Absolute value or Magnitude (e.g., abs(-9)=9)

- **Trigonometric functions**

Function	Description
`sin, cos, tan`	compute the sine, cosine, and tangent of angles
`arcsin, arccos, arctan`	calculate inverse sine, cosine, and tangent
`hypot`	calculate the hypotenuse of the given right triangle
`sinh, cosh, tanh`	compute hyperbolic sine, cosine, and tangent
`arcsinh, arccosh, arctanh`	compute inverse hyperbolic sine, cosine, and tangent
`deg2rad`	convert degree into radians
`rad2deg`	convert radians into degree

```python
# Python code to demonstrate trigonometric function
import numpy as np

# create an array of angles
angles = np.array([0, 30, 45, 60, 90, 180])

# conversion of degree into radians using deg2rad function
radians = np.deg2rad(angles)

# sine of angles
print('Sine of angles in the array:')
sine_value = np.sin(radians)
print(np.sin(radians))

# inverse sine of sine values
print('Inverse Sine of sine values:')
print(np.rad2deg(np.arcsin(sine_value)))

# hyperbolic sine of angles
print('Sine hyperbolic of angles in the array:')
sineh_value = np.sinh(radians)
print(np.sinh(radians))
```

```python
# inverse sine hyperbolic
print('Inverse Sine hyperbolic:')
print(np.sin(sineh_value))

# hypot function demonstration
base, height = 3, 4
print('hypotenuse of right triangle is:')
print(np.hypot(base, height))
```

```
Sine of angles in the array:
[0.00000000e+00 5.00000000e-01 7.07106781e-01 8.66025404e-01
 1.00000000e+00 1.22464680e-16]
Inverse Sine of sine values:
[0.0000000e+00 3.0000000e+01 4.5000000e+01 6.0000000e+01 9.0000000e+01
 7.0167093e-15]
Sine hyperbolic of angles in the array:
[ 0.          0.54785347  0.86867096  1.24936705  2.3012989  11.54873936]
Inverse Sine hyperbolic:
[ 0.          0.52085606  0.76347126  0.94878485  0.74483916 -0.85086591]
hypotenuse of right triangle is:
5.0
```

- **Statistical functions**
  - These functions calculate the **mean**, **median**, **variance**, **minimum**, **etc.** of array elements.
  - They are used to perform statistical analysis of array elements.

Function	Description
`amin, amax`	returns minimum or maximum of an array or along an axis
`ptp`	returns range of values (maximum-minimum) of an array or along an axis
`percentile(a, p, axis)`	calculate the p^{th} percentile of the array or along a specified axis
`median`	compute the median of data along a specified axis
`mean`	compute the mean of data along a specified axis
`std`	compute the standard deviation of data along a specified axis
`var`	compute the variance of data along a specified axis
`average`	compute the average of data along a specified axis

```python
# Python code demonstrate statistical function
import numpy as np

# construct a weight array
weight = np.array([50.7, 52.5, 50, 58, 55.63, 73.25, 49.5, 45])
# minimum and maximum
print('Minimum and maximum weight of the students: ', np.amin(weight), np.amax(weight))
# range of weight i.e. max weight-min weight
print('Range of the weight of the students: ', np.ptp(weight))
# percentile
print('Weight below which 70 % student fall: ', np.percentile(weight, 70))
# mean
print('Mean weight of the students: ', np.mean(weight))
# median
print('Median weight of the students: ', np.median(weight))
# standard deviation
print('Standard deviation of weight of the students: ', np.std(weight))
# variance
print('Variance of weight of the students: ', np.var(weight))
# average
print('Average weight of the students: ', np.average(weight))
```

```
Minimum and maximum weight of the students:  45.0 73.25
Range of the weight of the students:  28.25
Weight below which 70 % student fall:  55.317
Mean weight of the students:  54.3225
Median weight of the students:  51.6
Standard deviation of weight of the students:  8.052773978574091
Variance of weight of the students:  64.84716875
Average weight of the students:  54.3225
```

- **Bit-twiddling functions**
  - These functions accept integer values as input arguments and perform **bitwise operations** on binary representations of those integers.

Function	Description
`bitwise_and`	performs bitwise and operation on two array elements
`bitwies_or`	performs bitwise or operation on two array elements
`bitwise_xor`	performs bitwise xor operation on two array elements
`invert`	performs bitwise inversion of an array of elements
`left_shift`	shift the bits of elements to the left
`right_shift`	shift the bits of elements to the left

```python
# Python code to demonstrate bitwise-function
import numpy as np

# construct an array of even and odd numbers
even = np.array([0, 2, 4, 6, 8, 16, 32])
odd = np.array([1, 3, 5, 7, 9, 17, 33])

# bitwise_and
print('bitwise_and of two arrays: ', np.bitwise_and(even, odd))
# bitwise_or
print('bitwise_or of two arrays: ', np.bitwise_or(even, odd))
# bitwise_xor
print('bitwise_xor of two arrays: ', np.bitwise_xor(even, odd))
# invert or not
print('inversion of even no. array: ', np.invert(even))
# left_shift
print('left_shift of even no. array: ', np.left_shift(even, 1))
# right_shift
print('right_shift of even no. array: ', np.right_shift(even, 1))
```

```
bitwise_and of two arrays:  [ 0  2  4  6  8 16 32]
bitwise_or of two arrays:  [ 1  3  5  7  9 17 33]
bitwise_xor of two arrays:  [1 1 1 1 1 1 1]
inversion of even no. array:  [ -1  -3  -5  -7  -9 -17 -33]
left_shift of even no. array:  [ 0  4  8 12 16 32 64]
right_shift of even no. array:  [ 0  1  2  3  4  8 16]
```

- **Exponents and Logarithms**
  - Other common operations available in NumPy `ufuncs` are the exponentials and logarithms.

```python
# Python code to demonstrate exponetials and logarithms

import numpy as np
x = np.array([1, 2, 3])
print("x   =", x)
print("e^x =", np.exp(x))
print("2^x =", np.exp2(x))
print("3^x =", np.power(3., x))
print()
print("ln(x)    =", np.log(x))
print("log2(x)  =", np.log2(x))
print("log10(x) =", np.log10(x))
```

```
x   = [1 2 3]
e^x = [ 2.71828183  7.3890561  20.08553692]
2^x = [2. 4. 8.]
3^x = [ 3.  9. 27.]

ln(x)    = [0.         0.69314718 1.09861229]
log2(x)  = [0.        1.         1.5849625]
log10(x) = [0.         0.30103    0.47712125]
```

Computational time using NumPy

- Most array operations execute significantly faster than corresponding list operations.
- To demonstrate the same the `%timeit` command can be used.
- By default, `%timeit` executes a statement in a loop, and it runs the loop seven times.
- If you do not indicate the number of loops, `%timeit` chooses an appropriate value.

```python
# Computational time for lists and NumPy arrays

import numpy as np
np.random.seed(0)

def compute_reciprocals(values):
    output = np.empty(len(values))
    for i in range(len(values)):
        output[i] = 1.0 / values[i]
    return output

# Using traditional method
values = np.random.randint(1, 10, size=5)
print(compute_reciprocals(values))

# using NumPy ufuncs
print(1.0 / values)

big_array = np.random.randint(1, 100, size=1000000)
%timeit compute_reciprocals(big_array)
%timeit (1.0 / big_array)
```

```
[0.16666667 1.         0.25       0.25       0.125      ]
[0.16666667 1.         0.25       0.25       0.125      ]
2.48 s ± 82.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
4.56 ms ± 343 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```