Basic
Numpy

LEARN PYTHON FOR DATA SCIENCE

NUMPY

Learn Numpy and How to use it for Data Science.

CONCEPTS

Learn Different methods in Numpy and concepts to easily work with Data.

EXAMPLES

Learn Numpy with Examples.

@ds.learn

YOUR FREE GUIDE FOR PYTHON

TABLE OF CONTENT

ABOUT THE E-BOOK

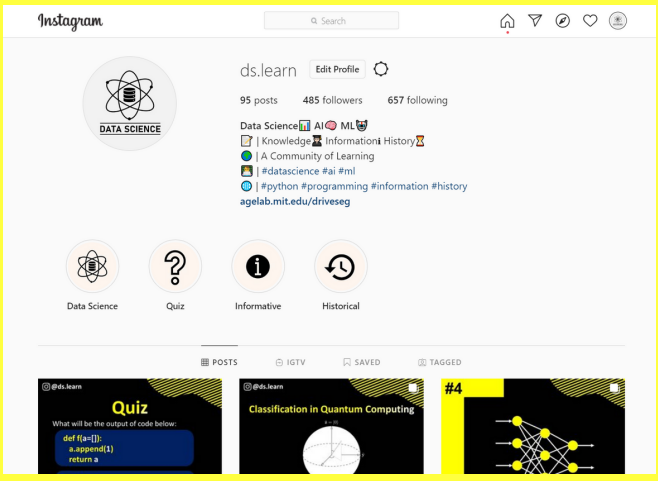
E-BOOK

This entire E-Book is created to help beginners in learning about NumPy Library's basic concepts and functions for Python. This will help in getting better in your journey of becoming a good at Data Science.

This E-Book is made with the purpose that one can have a basic guide of NumPy handy or can revise the topic quickly.

ABOUT BOOK CREATOR

Hey, If this book has help you in learning then check out our **Instagram account "@ds.learn"**. And make sure to support it, it would be helpful.



HISTORY OF NUMPY

NumPy

ORIGINAL AUTHOR: TRAVIS OLIPHANT

Python Language was not initially created for Numerical Computing but later it attracted attention of Scientific and Engineering Communities.

Initial version known as "Numeric" was created which was working Faster for small Arrays as compare to another package "Numarray". But, "Numarray" was more flexible option than "Numeric" because it was faster on Large Array. So, because of these for some time these two were parallelly used for different use. until Travis Oliphant combined Numarray's features to Numeric.

In 2005, Travis Oliphant created NumPy by incorporating features of the competing Numarray into Numeric, with extensive modifications. NumPy is open-source software and has many contributors.

INTRODUCTION TO NUMPY

INTRODUCTION TO NUMPY

Numpy is a open source and fundamental package for array computing with the Python programming language. Numpy is very useful in working with large, multi-dimensional arrays and matrices along with large collection of high-level mathematical functions to operate on these arrays.

WHAT NUMPY PROVIDES?

- Sophisticated Functions
- Capability of working with linear algebra, Fourier transform, and random number.
- N-dimensional powerful array objects.
- Seamless Integration with wide variety of Databases, et al.

COMMAND TO INSTALL NUMPY

$ pip install numpy
Or
$ pip3 install numpy

BASICS OF NUMPY

TOPICS

THE BASICS

- Array Creation
- Array Operations
- Array Functions
- Indexing & Slicing

SHAPE MANIPULATION

- Shape Manipulation
- Stacking Arrays

COPIES & VIEWS

THE BASICS

THE BASICS

np.arange(a,b,c)

Here, **"a"** is for **starting point**.
"b" is for **ending point**.
"c" is for **step**.

np.arange(b)

Here, for only one element it takes from **0 to (b-1)**.

ndarray.reshape((a,b,c))

Here, using **a,b,c** we can change the dimension into different shape.

numpy.reshape(a, nshape)

This is another way of reshaping the array. Where first we specify **name of array** then **new shape**.

```
import numpy as np
a = np.arange(20)
print(a)
#Reshaping
a = a.reshape(5,4)
print(a)
#Another Way
a = a.reshape(2,10)
print(a)
#Another Way (3-Dimensional)
a = a.reshape(2,5,2)
print(a)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]]
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]]
[[[ 0  1]
  [ 2  3]
  [ 4  5]
  [ 6  7]
  [ 8  9]]

 [[10 11]
  [12 13]
  [14 15]
  [16 17]
  [18 19]]]
```

THE BASICS

THE BASICS

ndarray.ndim

Here, "**.ndim**" is used to get the no. of axes of an array.

.

ndarray.shape

Here, "**.shape**" is used to get the **shape (a,b)** of an array. Where **"a"** is **no. of rows** and **"b"** is **no. of columns**.

ndarray.size

Here, "**.size**" gives total **no.of elements** of an array.

ndarray.dtype

Here, "**.dtype**" is used to get the **data-type of elements** of array.

ndarray.itemsize

Here, "**.itemsize**" is used to get the **size of each element** of an array in bytes. Here. output is **4** because defined **dtype='int32'**. (32/8) = 4

```
import numpy as np
a = np.arange(20,dtype='int32').reshape(5,4)
print(a)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]]
```

```
print("No.of axes: ",a.ndim)
print("Shape: ",a.shape)
print("Size: ",a.size)
print("Type of elements: ",a.dtype)
print("Item size: ",a.itemsize)
```

```
No.of axes:  2
Shape:  (5, 4)
Size:  20
Type of elements:  int32
Item size:  4
```

THE BASICS

ARRAY CREATION

np.array(obj.)

To create array.

.

np.zeros(shape)

Here, **".zeros"** is used to create **array full of zeros** with **specified shape**.

np.zeros_like(array_name)

Here, **".zeros_like"** gives array with **same shape and type** filled with zeros.

np.ones(shape)

Here, ".ones" is used to create array **full of ones** with **specified shape**.

np.ones_like(array_name)

Here, **".ones_like"** gives array with **same shape** and **type** filled with ones.

```python
import numpy as np
b = np.array([[1,2,3,4],[5,6,7,8]])
print("Creating array from lists:")
print(b)
#Creating array full of zeros
c = np.zeros((3,4))
print("Creating array full of zeros:")
print(c)
#Making an existing array (b) elements into zero
d = np.zeros_like(b)
print("Making an existing array (b) elements into zero")
print(d)
#Creating array full of ones
e = np.ones((3,4))
print("Creating array full of ones:")
print(e)
#Making an existing array (b) elements into ones
f = np.ones_like(b)
print("Making an existing array (b) elements into ones:")
print(f)
```

```
Creating array from lists:
[[1 2 3 4]
 [5 6 7 8]]
Creating array full of zeros:
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
Making an existing array (b) elements into zero
[[0 0 0 0]
 [0 0 0 0]]
Creating array full of ones:
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
Making an existing array (b) elements into ones:
[[1 1 1 1]
 [1 1 1 1]]
```

THE BASICS

ARRAY CREATION

np.linspace(start,stop,n)

Return **evenly spaced numbers** over a specified interval.

np.random.rand(a,b)

Random values in a given shape.

np.random.randn(a,b)

Return a **sample** from the **“standard normal” distribution**.

```
import numpy as np
#Generating specified no.of Elements with same spacing
a = np.linspace(1,5,num=5,dtype='int16')
print("Generating specified no.of Elements with same spacing")
print(a)
#Generating Randomly
b = np.random.rand(3,5)
print("Generating Elements Randomly:")
print(b)
#Generating normal distributed elements
c = np.random.rand(2,3)
print("Generating normal distributed elements:")
print(c)
```

```
Generating specified no.of Elements with same spacing
[1 2 3 4 5]
Generating Elements Randomly:
[[0.6437415  0.5673874  0.11289488 0.00294212 0.54080059]
 [0.92523186 0.96763268 0.17796876 0.93620187 0.27166045]
 [0.94284786 0.23079898 0.4257908  0.04270658 0.17463475]]
Generating normal distributed elements:
[[0.22224691 0.46003247 0.70472392]
 [0.8533196  0.98959457 0.62901579]]
```

THE BASICS

ARRAY OPERATIONS

ARITHMETIC OPERATION

Condition: Arrays must be either of the same shape or should conform to array broadcasting rules.

np.add(a,b) ----------> To Add Two Arrays

np.subtract(a,b)----------> To Subtract Two Arrays

np.multiply(a,b) ----------> To Multiply Two Arrays

np.divide(a,b) ----------> To Divide Two Arrays

MATRIX PRODUCT

A * B ----------> Elementwise Product

A @ B ----------> Matrix Product

A.dot(B) ----------> Another Matrix Product

THE BASICS

ARRAY FUNCTIONS

np.all(a) ----------> Checks array elements along a given axis evaluate to True (Except 0).

np.any(a)----------> Checks any array element along a given axis evaluates to True (Except 0).

np.max(a) ----------> Returns Mamimum Element of array.

np.min(a) ----------> Returns Minimum Element of Array.

np.argmax(a) -------> Returns index of Maximum Element.

np.argmin(a) --------> Returns index of Minimum Element.

np.ptp(a) ----------> Returns difference between max and min value.

np.sum(a) ----------> Sum of all Elements.

np.var(a) ----------> Returns Variance.

np.std(a) ----------> Returns Standard Deviation.

np.mean(a) ----------> Returns Mean of an array.

np.median(a) ---------> Returns Median of an Array.

THE BASICS

ARRAY FUNCTIONS

np.cumsum(a) ----> Return the Cumulative Sum of the elements along a given axis. (Axis=1 --> Row-wise & Axis=0 --> Col wise)

ndarray.clip(a,b) ---> Clip (limit) the values in an array.

np.round(a) ---------> Return number rounded to ndigitd precision.

np.sort(a) ------------> To Sort the Array.

np.transpose(a) ----> To Transpose the Array.

```
import numpy as np
a = np.array([[50,60,70,80,1.1,2.2,3.3,4.4]])
print("Main Array:\n",a)
print("All elements are not 0:",np.all(a))
print("Any element is not 0:",np.any(a))
print("Maximum Element:",np.max(a))
print("Minimum Element:",np.min(a))
print("Index of maximum element:",np.argmax(a))
print("Index of minimum element:",np.argmin(a))
print("Difference Between max and min value:",np.ptp(a))
print("Sum:",np.sum(a))
print("Variance:",np.var(a))
print("Standard Deviation:",np.std(a))
print("Mean:",np.mean(a))
print("Median:",np.median(a))
print("Cumulative Sum: \n", np.cumsum(a)) #Axis=1/0 "Row-wise/col-wise"
print("Clipping the array with given limit:\n ",a.clip(3,5))
print("Rounding Array: \n",np.round(a))
print("Sorted Array:\n",np.sort(a))
print("Transposed Array:\n",np.transpose(a))
```

THE BASICS

ARRAY FUNCTIONS

OUTPUT

```
Main Array:
 [[50.  60.  70.  80.   1.1  2.2  3.3  4.4]]
All elements are not 0: True
Any element is not 0: True
Maximum Element: 80.0
Minimum Element: 1.1
Index of maximum element: 3
Index of minimum element: 4
Difference Between max and min value: 78.9
Sum: 271.0
Variance: 1032.021875
Standard Deviation: 32.12509727611731
Mean: 33.875
Median: 27.2
Cumulative Sum:
 [ 50.  110.  180.  260.  261.1 263.3 266.6 271. ]
Clipping the array with given limit:
  [[5.  5.  5.  5.  3.  3.  3.3 4.4]]
Rounding Array:
 [[50. 60. 70. 80.  1.  2.  3.  4.]]
Sorted Array:
 [[ 1.1  2.2  3.3  4.4 50.  60.  70.  80. ]]
Transposed Array:
 [[50. ]
 [60. ]
 [70. ]
 [80. ]
 [ 1.1]
 [ 2.2]
 [ 3.3]
 [ 4.4]]
```

THE BASICS

INDEXING & SLICING

ONE DIMENSIONAL

a[Starting_index, Ending_Index] ----> Accessing Specific Element.

a[Start_Ind, End_Ind, Step] -----> Accessing Range of elements with steps.

TWO DIMENSIONAL

a[row/s, col/s] ----> Accessing Specific Element in Multi-D Array.

```
import numpy as np
#ONE DIMENSIONAL
a = np.arange(20)
print("One Dimensional Array:",a)
print("Accessing Specific Element:",a[3])
print("Accessing Elements in Range:",a[2:16])
print("Accessing Elements in Range with Steps:", a[2:16:2])

#TWO-DIMENSIONAL
b = np.arange(20,dtype='int16').reshape(5,4)
print("\n Multi-Dimensional Array:\n",b)
print("Accessing Specific Element of Multi-Dim. Array:",b[3,3])
print("Accessing Multiple Elements of Multi-dim Array:\n",b[0:2,1:3])
print("Accessing Specific Element of diff. rows and cols:",b[[0,1,4],[0,2,3]])
```

```
One Dimensional Array: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
Accessing Specific Element: 3
Accessing Elements in Range: [ 2  3  4  5  6  7  8  9 10 11 12 13 14 15]
Accessing Elements in Range with Steps: [ 2  4  6  8 10 12 14]

 Multi-Dimensional Array:
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]]
Accessing Specific Element of Multi-Dim. Array: 15
Accessing Multiple Elements of Multi-dim Array:
 [[1 2]
 [5 6]]
Accessing Specific Element of diff. rows and cols: [ 0  6 19]
```

SHAPE MANIPULATION

STACKING ARRAYS

We have already discussed about **ndarray.reshape()** and **ndarray.shape**

STACKING ARRAYS

np.hstack((a,b)) ------> Stacking Arrays Horizontally.

np.vstack((a,b)) ------> Stacking Arrays Vertically.

```
import numpy as np
a = np.arange(8).reshape(2,4)
print("Array a:\n",a)
b = np.arange(10,18).reshape(2,4)
print("Array b:\n",b)
print("Stacking arrays Horizontally (Col-wise):\n",np.hstack((a,b)))
print("Stacking Arrays Vertically (Row-wise):\n",np.vstack((a,b)))
```

```
Array a:
 [[0 1 2 3]
 [4 5 6 7]]
Array b:
 [[10 11 12 13]
 [14 15 16 17]]
Stacking arrays Horizontally (Col-wise):
 [[ 0  1  2  3 10 11 12 13]
 [ 4  5  6  7 14 15 16 17]]
Stacking Arrays Vertically (Row-wise):
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [10 11 12 13]
 [14 15 16 17]]
```

COPIES AND VIEWS

COPIES & VIEWS

NO COPY:

(b = a) generates no copy. Basically, it points to the same object, no new object is generated. Hence, changes happen in both.

VIEW:

The **view()** method creates a new array object that looks at the same data. Here, changes in shape does not apply to parent object but changes in data does.

```
import numpy as np
a = np.arange(8)
#No Copy ----Both are same.
b = a
b.shape = 2,4
print("For Assigning:",b is a)
print("Shape of b:",b.shape)
print("Shape of a:",a.shape)
#View
c = np.arange(8)
d = c.view()
d.shape = 2,4
d[1,2] = 12
print("\nFor View:",d is c)
print("Shape of d:",d.shape)
print("Shape of c:",c.shape)
print("Array d:\n",d)
print("Array c:\n",c)
```

```
For Assigning: True
Shape of b: (2, 4)
Shape of a: (2, 4)

For View: False
Shape of d: (2, 4)
Shape of c: (8,)
Array d:
 [[ 0  1  2  3]
 [ 4  5 12  7]]
Array c:
 [ 0  1  2  3  4  5 12  7]
```

COPIES AND VIEWS

COPIES & VIEWS

DEEP COPY:

The **copy()** method makes a complete copy of the array and its data. That is why both remain **uninfluenced** by each other.

```
import numpy as np
#DEEP COPY
c = np.arange(8)
d = c.copy()
d.shape = 2,4
d[1,2] = 12
print("\nFor Deep Copy:",d is c)
print("Shape of d:",d.shape)
print("Shape of c:",c.shape)
print("Array d:\n",d)
print("Array c:\n",c)
```

```
For Deep Copy: False
Shape of d: (2, 4)
Shape of c: (8,)
Array d:
 [[ 0  1  2  3]
 [ 4  5 12  7]]
Array c:
 [0 1 2 3 4 5 6 7]
```

THE END

THE END

Hope you guys liked the E-Book, and it would have contributed in your learning.