02022021 - Numpy

February 27, 2021

1 Limitations of Lists

You have already seen what lists are and what you can do with them. To quickly recap, a list is a basic Python data structure which can hold multiple values of multiple data types such as integers, floats, strings, and so on. It also allows you to add, update, or delete individual values in it.

In this example, we have two lists "distance" and "speed." Each list holds four different readings. These readings correspond to each other. For instance, the first element in the 'distance' list is 100 miles. This value corresponds to the first element of the "speed" list which is 20 hours.

[1]:
```
distance = [100, 15, 900, 60]
speed = [20, 0.75, 90, 5]
```

[2]:
```
time = distance/speed
```

```
       ␣
↪---------------------------------------------------------------------------

     TypeError                                 Traceback (most recent call␣
↪last)

     <ipython-input-2-4e2958124023> in <module>
 ----> 1 time = distance/speed


     TypeError: unsupported operand type(s) for /: 'list' and 'list'
```

Using these lists, if you try to calculate the time, which is equal to distance over time, it will give you an error. That's because mathematical functions can't be applied over an entire list. Now, let's see how NumPy solves this problem.

2 Numpy

NumPy is a Python library that supports a data container called arrays. **Arrays are like lists, but they can do something that lists can't.** They easily allow you to **apply mathematical operations over the entire dataset**. Let's see how this works.

First, you need to import the NumPy library. You then need to convert the 'distance' and 'time' lists into NumPy arrays. Apply the formula for time using these arrays. You can see that NumPy easily generates an output for each of the four readings.

This property of NumPy makes a Data Scientist's job much easier, because it helps them to easily manipulate data by applying mathematical functions over a given dataset.

2.1 Installation Instructions

It is highly recommended you install Python using the Anaconda distribution to make sure all underlying dependencies (such as Linear Algebra libraries) all sync up with the use of a conda install. If you have Anaconda, install NumPy by going to your terminal or command prompt and typing:

```
conda install numpy
pip install numpy
```

[1]:
```python
import numpy as np # calling the library with importand giving it an alias
# sqrt = np.sqrt
```

[4]:
```python
np_distance = np.array(distance)
np_speed = np.array(speed)
```

[6]:
```python
time = np_distance/np_speed
print (time)
print (type(time))
```

```
[ 5. 20. 10. 12.]
<class 'numpy.ndarray'>
```

2.1.1 Arrays

- Arrays can be one-dimensional, two-dimensional, three-dimensional, or multi-dimensional.
- The best way to visualize an array is as rows and columns. You can also look at it by its dimensional axes or rank.
- You can think of a one-dimensional array as a single row of values. So, a one-dimensional array has one axis or rank 1.
- Two-dimensional arrays can be visualized with rows and columns. This means that it has two axes or rank 2.
- A three-dimensional array has three axes and can be visualized as a cube which has height, width, and length.
- Similarly, multi-dimensional arrays have multiple axes.

3 Creating and Printing an array

[7]:
```python
# A 1-dimensional array == Row

x = np.array([1,2,3])
print (x)
print ()
print (type(x))

# numpy.ndarray == numpy object and n- dimensional array
```

```
[1 2 3]

<class 'numpy.ndarray'>
```

[8]:
```
# A 2-dimensional array == Matrix

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

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

<class 'numpy.ndarray'>
```

[9]:
```
# A 2-dimensional array == Matrix

x = np.array([[1,2,3],[4,5,6],[7,8,9]])
print (x)
print ()
print (type(x))
```

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

<class 'numpy.ndarray'>
```

[10]:
```
# A 3-dimensional array == Matrices, Rows and Columns

x = np.array([[[1,2,3],
              [4,5,6],
              [7,8,9]],

              [[11,12,13],
              [14,15,16],
              [17,18,19]],

              [[21,22,23],
              [24,25,26],
              [27,28,29]]])
```

```
print (x)
```

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

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

 [[21 22 23]
  [24 25 26]
  [27 28 29]]]
```

4 Class and Attributes of ndarray

Numpy's array class is "ndarray," also referred to as "numpy.ndarray." The attributes of ndarray are:

4.1 dim

ndarray.ndim – This refers to the number of axes or the dimensions of the array, which means that it will indicate whether it is a one-dimensional, two-dimensional, or three-dimensional array. It is also called the rank of the array.

```
[11]: np_city = np.array(['NYC', 'LA', 'Miami', 'Houston'])

print (np_city)
print ()
print (np_city.ndim)
```

```
['NYC' 'LA' 'Miami' 'Houston']

1
```

```
[12]: np_city_with_state = np.array([['NYC', 'LA', 'Miami', 'Houston'], ["NY", "CA",␣
 ↪"Fl", "TX"]])

print (np_city_with_state)
print ()
print (np_city_with_state.ndim)
```

```
[['NYC' 'LA' 'Miami' 'Houston']
 ['NY' 'CA' 'Fl' 'TX']]

2
```

```
[13]: x = np.array([[[1,2,3],
                     [4,5,6],
                     [7,8,9]],

                     [[11,12,13],
                     [14,15,16],
                     [17,18,19]],

                     [[21,22,23],
                     [24,25,26],
                     [27,28,29]]])
      print (x.ndim)
```

```
3
```

4.2 shape

This consists of a tuple of integers showing the size of the array in each dimension. The length of the "shape tuple" is the rank or ndim.

ndarray.shape describes the structure of an array. It consists of a tuple of integers indicating the size of the array in each dimension. For a matrix with n rows and m columns, the shape will be (n, m). So, the length of the shape tuple is the rank or ndim.

The shape tuple of both the arrays indicate their size along each dimension.

```
[17]: np_city = np.array(['NYC', 'LA', 'Miami', 'Houston'])

      print(np_city)
      print ()
      print (np_city.shape)
```

```
['NYC' 'LA' 'Miami' 'Houston']

(4,)
```

```
[18]: np_city_with_state = np.array([['NYC',"LA","Miami", "Houston"], ["NY", "CA",␣
       ↪"Fl", "TX"]])

      print (np_city_with_state)
      print ()
      print (np_city_with_state.shape)
```

```
[['NYC' 'LA' 'Miami' 'Houston']
 ['NY' 'CA' 'Fl' 'TX']]

(2, 4)
```

```
[19]: x = np.array([[[1,2,3],
                    [4,5,6],
                    [7,8,9]],

                    [[11,12,13],
                    [14,15,16],
                    [17,18,19]],

                    [[21,22,23],
                    [24,25,26],
                    [27,28,29]]])
      print (x.shape)

      # (3, 3, 3) == (no. of matrices, no. of rows, no. of cols)
```

```
(3, 3, 3)
```

4.3 size

It gives the total number of elements in the array. It is equal to the product of the elements of the shape tuple. ndarray.size indicates the number of elements present in an array. It is equal to the product of the elements of its shape tuple.

```
[20]: np_city = np.array(['NYC', 'LA', 'Miami', 'Houston'])

      print (np_city.size)
```

```
4
```

```
[21]: np_city_with_state = np.array([['NYC',"LA","Miami", "Houston"], ["NY", "CA",␣
       ↪"Fl", "TX"]])

      print (np_city_with_state.size)
```

```
8
```

```
[28]: np_city_with_state = np.array([['NYC',"LA","Miami", "Houston"], ["NY", "CA",␣
       ↪"Fl"]])

      print (np_city_with_state)
      print (np_city_with_state.size)
      print (np_city_with_state.ndim)
      print (np_city_with_state.shape)
```

```
[list(['NYC', 'LA', 'Miami', 'Houston']) list(['NY', 'CA', 'Fl'])]
2
1
(2,)
```

[22]:
```
x = np.array([[[1,2,3],
              [4,5,6],
              [7,8,9]],

              [[11,12,13],
              [14,15,16],
              [17,18,19]],

              [[21,22,23],
              [24,25,26],
              [27,28,29]]])
print (x.size)
```

```
27
```

5 Basic Operations

[29]:
```
first_trial_cyclist = [10,15,17,26]
second_trial_cyclist = [12,11,21,24]
```

[30]:
```
np_first_trial_cyclist = np.array(first_trial_cyclist)
np_second_trial_cyclist = np.array(second_trial_cyclist)
```

[31]:
```
print (np_first_trial_cyclist)
print (np_second_trial_cyclist)
```

```
[10 15 17 26]
[12 11 21 24]
```

[32]:
```
np_first_trial_cyclist + np_second_trial_cyclist
```

[32]:
```
array([22, 26, 38, 50])
```

[33]:
```
np_first_trial_cyclist * np_second_trial_cyclist
```

[33]:
```
array([120, 165, 357, 624])
```

[34]:
```
np_first_trial_cyclist - np_second_trial_cyclist
```

[34]:
```
array([-2,  4, -4,  2])
```

[35]:
```
np_first_trial_cyclist ** 2
```

[35]:
```
array([100, 225, 289, 676])
```

6 Accessing Array Elements: Indexing

```
[36]: cyclist_trials = np.array([[10,15,17,26],[12,11,21,24]]) # Create 2D array␣
       ↪using cyclist trial data shown earlier
      print (cyclist_trials)
```

```
[[10 15 17 26]
 [12 11 21 24]]
```

```
[37]: first_trial = cyclist_trials[0] # first row
      print (first_trial)
```

```
[10 15 17 26]
```

```
[38]: second_trial = cyclist_trials[1] # second row
      print (second_trial)
```

```
[12 11 21 24]
```

How to access [10,12]?

```
[39]: first_cyclist_firstTrial = cyclist_trials[0][0]
      first_cyclist_firstTrial
```

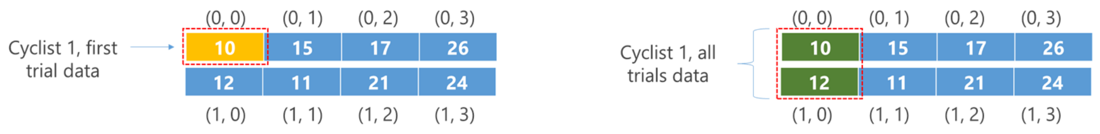
```
[39]: 10
```

```
[40]: cyclist_trials[[0][0],[1][0]]
```

```
[40]: 15
```

```
[41]: from IPython.display import Image
      Image(filename='Indexing.png')
```

[41]:



```
[42]: first_cyclist_firstTrial = cyclist_trials[:,0]
      first_cyclist_firstTrial

      # values before the ',' stand for rows and values after the ',' stand for␣
       ↪columns
      # ":" before the comma is to select all the rows for the said col
```

```
[42]: array([10, 12])
```

[43]:
```
cyclist_trials[(0,1),0]
```

[43]:
```
array([10, 12])
```

[44]:
```
cyclist_trials[0,:]
```

[44]:
```
array([10, 15, 17, 26])
```

[45]:
```
# 4th col
cyclist_trials[:,-1]
```

[45]:
```
array([26, 24])
```

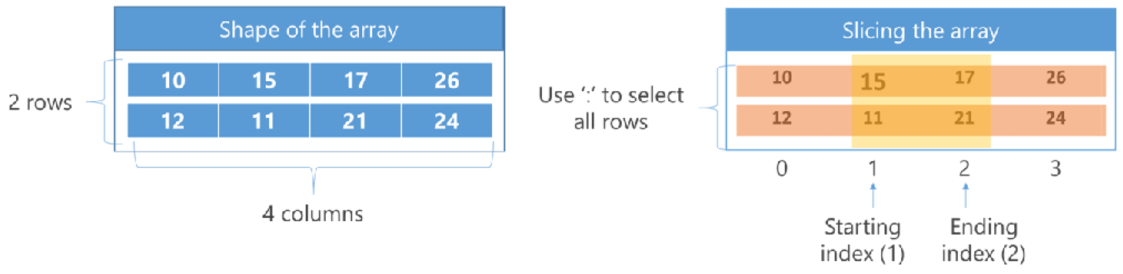
7 Slicing

[46]:
```
print (cyclist_trials.shape) # Shape of the array
print (cyclist_trials)
```

```
(2, 4)
[[10 15 17 26]
 [12 11 21 24]]
```

[47]:
```
from IPython.display import Image
Image(filename='slicing.png')
```

[47]:



[49]:
```
# cyclist_trials[:,(1,2)]
cyclist_trials[:,1:3]

# Slicing the array data [ : , 1 : 3] where 1 is inclusive but 3 is not.␣
↪elements from 1 to (3-1) index
```

[49]:
```
array([[15, 17],
       [11, 21]])
```

[50]:
```
cyclist_trials[:,2:]
```

```
[50]: array([[17, 26],
             [21, 24]])
```

```
[56]: cyclist_trials[:,(0,-1)]
      # cyclist_trials[-[:,1:2]] wrong
```

```
[56]: array([[10, 26],
             [12, 24]])
```

8 Shape Manipulation

The shape of a basic array can be changed according to the requirement using NumPy library functions. These array shape manipulation methods really come in handy during the data wrangling phase and is used extensively by Data Scientists. Following are some common methods for manipulating shapes: - Ravel/Flatten - Reshape - Resize

Let's look at each in the following:

```
[3]: new_cyclist_trials = np.array([[10,15,17,26,13,19],[12,11,21,24,14,23]])
     print (new_cyclist_trials)
     print ()
     print (new_cyclist_trials.ndim)
     print ()
     print (new_cyclist_trials.shape)
     print ()
     print (new_cyclist_trials.size)
```

```
[[10 15 17 26 13 19]
 [12 11 21 24 14 23]]

2

(2, 6)

12
```

Ravel This function returns a flattened one-dimensional array. The returned array will have the same type as that of the input array. The function takes one parameter.

```
[6]: print (new_cyclist_trials.ravel())
     print ()
     print (new_cyclist_trials.ravel().shape)
     print (new_cyclist_trials.ravel().ndim)
     print ()
     print (new_cyclist_trials)
     print ()
     print (new_cyclist_trials.shape)
     print (new_cyclist_trials.ndim)
```

```
[10 15 17 26 13 19 12 11 21 24 14 23]

(12,)
1

[[10 15 17 26 13 19]
 [12 11 21 24 14 23]]

(2, 6)
2
```

- **The ravel function flattens the dataset into a single row.**
- **It will not modify the original array**

Reshape This function gives a new shape to an array without changing the data.

[10]:
```
new_cyclist_trials = np.array([[10,15,17,26,13,19],[12,11,21,24,14,23]])
print (new_cyclist_trials)
print ()
print (new_cyclist_trials.shape)
print ()
print (new_cyclist_trials.reshape(3,4)) # reshaped array
print (new_cyclist_trials.reshape(3,4).shape) # reshaped array
print ()
print (new_cyclist_trials.shape)
```

```
[[10 15 17 26 13 19]
 [12 11 21 24 14 23]]

(2, 6)

[[10 15 17 26]
 [13 19 12 11]
 [21 24 14 23]]
(3, 4)

(2, 6)
```

[12]:
```
new_cyclist_trials = new_cyclist_trials.reshape(3,4)
print (new_cyclist_trials.shape)
print (new_cyclist_trials)
```

```
(3, 4)
[[10 15 17 26]
```

```
 [13 19 12 11]
 [21 24 14 23]]
```

- `Reshape is a function which reshapes the dataset`
- `Here, the dataset is reshaped from 2 rows and 6 columns to 3 rows and 4 columns`
- `It will not modify the original array`

9 Work with same new__cyclist__trials and reshape it to (1,12)

[16]:
```
new_cyclist_trials = new_cyclist_trials.reshape(1,12)
print (new_cyclist_trials.shape)
print (new_cyclist_trials.ndim)
print (new_cyclist_trials)
```

```
(1, 12)
2
[[10 15 17 26 13 19 12 11 21 24 14 23]]
```

10 Work with same new__cyclist__trials and reshape it to (3,5)

[14]:
```
new_cyclist_trials = np.array([[10,15,17,26,13,19],[12,11,21,24,14,23]])
print (new_cyclist_trials)
print ()
print (new_cyclist_trials.shape)
print ()
print (new_cyclist_trials.reshape(3,5))
print (new_cyclist_trials.reshape(3,5).shape)
print ()
print (new_cyclist_trials.shape)

# ValueError: cannot reshape array of size 12 into shape (3,5)
```

```
[[10 15 17 26 13 19]
 [12 11 21 24 14 23]]

(2, 6)
```

```
       ␣
 ↪---------------------------------------------------------------------------

       ValueError                                Traceback (most recent call␣
 ↪last)

       <ipython-input-14-e7cf28b3b8f6> in <module>
```

```
          4 print (new_cyclist_trials.shape)
          5 print ()
----> 6 print (new_cyclist_trials.reshape(3,5))
          7 print (new_cyclist_trials.reshape(3,5).shape)
          8 print ()


     ValueError: cannot reshape array of size 12 into shape (3,5)
```

Resize This function returns a new array with the specified size.

[18]:
```
new_cyclist_trials = np.array([[10,15,17,26,13,19],[12,11,21,24,14,23]])
print (new_cyclist_trials)
print ()
print (new_cyclist_trials.shape)
print ()
print (new_cyclist_trials.resize(3,4)) # no output - updates the original array
print ()
print (new_cyclist_trials.shape)
print ()
print (new_cyclist_trials)
```

```
[[10 15 17 26 13 19]
 [12 11 21 24 14 23]]

(2, 6)

None

(3, 4)

[[10 15 17 26]
 [13 19 12 11]
 [21 24 14 23]]
```

- Ravel - Does not update the original array
- Reshape - Does not update the original array
- Resize - **Updates** the original array

11 Work with same new__cyclist__trials and resize it to (3,5)

[19]:
```
new_cyclist_trials = np.array([[10,15,17,26,13,19],[12,11,21,24,14,23]])
print (new_cyclist_trials)
print ()
print (new_cyclist_trials.shape)
print ()
print (new_cyclist_trials.resize(3,5)) # no output - updates the original array
print ()
print (new_cyclist_trials.shape)
print ()
print (new_cyclist_trials)
```

```
[[10 15 17 26 13 19]
 [12 11 21 24 14 23]]

(2, 6)

None

(3, 5)

[[10 15 17 26 13]
 [19 12 11 21 24]
 [14 23  0  0  0]]
```

12 Work with same new_cyclist_trials and resize it to (2,5)

[20]:
```
new_cyclist_trials = np.array([[10,15,17,26,13,19],[12,11,21,24,14,23]])
print (new_cyclist_trials)
print ()
print (new_cyclist_trials.shape)
print ()
print (new_cyclist_trials.resize(2,5)) # no output - updates the original array
print ()
print (new_cyclist_trials.shape)
print ()
print (new_cyclist_trials)
```

```
[[10 15 17 26 13 19]
 [12 11 21 24 14 23]]

(2, 6)

None

(2, 5)

[[10 15 17 26 13]
 [19 12 11 21 24]]
```

13 Broadcasting

NumPy uses broadcasting to carry out arithmetic operations between arrays of different shapes.

To understand how this works, let's look at the examples shown here. Array_a and array_b have the same shape, which is one row and four columns. So, to calculate the product of the two arrays, NumPy conducts an element-wise multiplication.

However, c is a single scalar value. Its shape doesn't match with array_a. In this case, NumPy doesn't have to create copies of the scalar value to multiply it element-wise with the array elements. Instead, it broadcasts the scalar value over the entire array to find the product. This saves memory space as an array takes a lot more memory than a scalar.

Though broadcasting can help carry out mathematical operations between different-shaped arrays, they are subject to certain constraints as listed below:

When NumPy operates on two arrays, it compares their shapes element-wise. It finds these shapes compatible only if: Their dimensions are the same or one of them has a dimension of size 1. If these conditions are not met, a ``ValueError'' is thrown, indicating that the arrays have incompatible shapes.

[24]:
```
# **element-wise arithmetic operation is performed**

array_a = np.array([2,  3,   5,  8])
array_b = np.array([0.3,0.3,0.3,0.3])

print (array_a * array_b)
```

```
[0.6 0.9 1.5 2.4]
```

[22]:
```
array_d = np.array([4,3])
print (array_a * array_d)
```

```
      ␣
↪---------------------------------------------------------------------------

      ValueError                                Traceback (most recent call␣
↪last)

      <ipython-input-22-36c96c0fa9cf> in <module>
        1 array_d = np.array([4,3])
  ----> 2 print (array_a * array_d)


      ValueError: operands could not be broadcast together with shapes (4,)␣
↪(2,)
```

```
[26]: # **broadcasting**
      c = 0.3
      print (array_a * c)
```

```
[0.6 0.9 1.5 2.4]
```

14 Creating some generic arrays

np.ones

```
[27]: x = np.ones(5) # default dtype is float
      print (x)
      print ()
      print (x.shape)
      print ()
      print (type(x))
```

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

(5,)

<class 'numpy.ndarray'>
```

```
[28]: x = np.ones(5, dtype = int)
      print (x)
      print ()
      print (x.shape)
      print ()
      print (type(x))
```

```
[1 1 1 1 1]

(5,)

<class 'numpy.ndarray'>
```

```
[29]: x = np.ones(5, dtype = str)
      print (x)
      print ()
      print (x.shape)
      print ()
      print (type(x))
```

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

(5,)

<class 'numpy.ndarray'>
```

```
[30]: y = np.ones((2,2))
      print (y)
```

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

np.zeros

```
[32]: y = np.zeros((5,), dtype=int)
      print (y)
```

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

```
[33]: y = np.zeros((5,3))
      print (y)
```

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

Application to **initialize the weights during the first iteration in TensorFlow** Deep Learning

np.arange

```
[2]: # Return evenly spaced values within a given interval
     np.arange(3,9)
```

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

```
[3]: np.arange(3,9.0)
```

```
[3]: array([3., 4., 5., 6., 7., 8.])
```

```
[4]: print (np.arange(3,9,2)) # start, stop and step == start from 3 and give every␣
      ↪second value
```

```
[3 5 7]
```

Linspace Return evenly spaced numbers over a specified interval.

Returns num evenly spaced samples, calculated over the interval [start, stop].

The endpoint of the interval can optionally be excluded.

```
[37]: np.linspace(2.0, 3.0, num = 5) # (3-2)/(5-1)
```

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

```
[38]: np.linspace(2.0, 3.0, num = 5, endpoint = False) # (3-2)/5
```

```
[38]: array([2. , 2.2, 2.4, 2.6, 2.8])
```

```
[39]: np.linspace(2.0, 3.0, num = 5, retstep=True)
```

```
[39]: (array([2.  , 2.25, 2.5 , 2.75, 3.  ]), 0.25)
```

```
[40]: np.linspace(2.0, 3.0, num = 5, endpoint = False, retstep=True)
```

```
[40]: (array([2. , 2.2, 2.4, 2.6, 2.8]), 0.2)
```

```
[42]: np.linspace(2.0, 3.0, num = 10, endpoint = False, retstep=True)
```

```
[42]: (array([2. , 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9]), 0.1)
```

What is the application of arange and linspace?

We use arange and linspace while creating x-axis of graphs for visualization.

15 Universal Functions

```
[5]: np_sqrt = np.sqrt([2, 4, 9, 16])
     print (np_sqrt)
```

```
[1.41421356 2.         3.         4.        ]
```

```
[6]: np_log = np.log([2, 4, 9, 16]) # natural log - log to the base e
     print (np_log)
```

```
[0.69314718 1.38629436 2.19722458 2.77258872]
```

```
[7]: np_log10 = np.log10([2, 4, 9, 16]) # base 10
     print (np_log10)
```

```
[0.30103    0.60205999 0.95424251 1.20411998]
```

```
[8]: print (np.exp(np_log))
```

```
[ 2.  4.  9. 16.]
```

```
[9]: print (np.floor([1.2,1.3,4.9,2.6,3.3,5.6])) # nearest lowest integer
```

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

```
[10]: print (np.round([1.2,1.3,4.9,2.6,3.3,5.6]))
```

```
[1. 1. 5. 3. 3. 6.]
```

```
[11]: print (np.ceil([1.2,1.3,4.9,2.6,3.3,5.6])) # nearest highest integer
```

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

```
[12]: np_cbrt = np.cbrt([2, 4, 8, 16])
      print (np_cbrt)
```

```
[1.25992105 1.58740105 2.         2.5198421 ]
```

16 Comparision Operations

16.0.1 numpy.greater

Return the truth value of (x1 > x2) element-wise

```
[13]: print(np.greater([4,2],[2,2]))
```

```
[ True False]
```

16.0.2 numpy.greater_equal

Return the truth value of (x1 >= x2) element-wise

```
[14]: print(np.greater_equal([4,2,1],[2,2,2]))
```

```
[ True  True False]
```

16.0.3 numpy.less

Return the truth value of (x1 < x2) element-wise

```
[15]: print(np.less([1, 2], [2, 2]))
```

```
[ True False]
```

16.0.4 numpy.less_equal

Return the truth value of (x1 <= x2) element-wise

```
[17]: print(np.less_equal([4, 2, 1], [2, 2, 2]))
```

```
[False  True  True]
```

16.0.5 numpy.equal

Return the truth value of (x1 == x2) element-wise

```
[18]: print([0, 1, 3])
      print ()
```

```
print(np.arange(3))
print ()
print(np.equal([0, 1, 3], np.arange(3)))
```

```
[0, 1, 3]

[0 1 2]

[ True  True False]
```

16.0.6 numpy.not_equal

Return the truth value of (x1 != x2) element-wise

[19]:
```
print(np.not_equal([1.,2.], [1., 3.]))
```

```
[False  True]
```

17 Statistical Functions

[20]:
```
y = np.linspace(1, 12, 12)
print (y)
print ()
print (y.shape)
print ()
y = y.reshape(3,4)
print (y)
print ()
print (y.shape)
```

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

(12,)

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

(3, 4)
```

[21]:
```
print (y.min())
```

```
1.0
```

[22]:
```
print (y.max())
```

```
12.0
```

```
[23]: print (y.min(axis = 0)) # col
```

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

```
[24]: print (y.min(axis = 1)) # row
```

```
[1. 5. 9.]
```

```
[25]: print (y.mean())
```

```
6.5
```

When just working with arrays --- for an operation on col use axis = 0 and for row vice-versa

```
[26]: print (y)
      print (y.min())
      print (y.argmin()) # index of min value
```

```
[[ 1.  2.  3.  4.]
 [ 5.  6.  7.  8.]
 [ 9. 10. 11. 12.]]
1.0
0
```

```
[27]: print (y)
      print (y.max())
      print (y.argmax()) # index of max value
```

```
[[ 1.  2.  3.  4.]
 [ 5.  6.  7.  8.]
 [ 9. 10. 11. 12.]]
12.0
11
```

18 Find Min Value of the second row and the max value of 3rd row

```
[32]: print (y[1,:].min()) # min value of second row
      print (y[-1,:].min()) # min value of third row

      print (y[:,1].max()) # max value of second col
      print (y[:,2].max()) # max value of third col
```

```
5.0
9.0
10.0
11.0
```

18.0.1 Matrix Multiplication

[33]:
```
y = np.linspace(1, 9, 9)
print (y)
print ()
print (y.shape)
print ()
y = y.reshape(3,3)
print (y)
print ()
print (y.shape)
```

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

(9,)

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

(3, 3)
```

[34]:
```
print (y * y)
```

```
[[ 1.  4.  9.]
 [16. 25. 36.]
 [49. 64. 81.]]
```

[35]:
```
print (np.matmul(y, y))
```

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

19 Copy and Views - Simple Assignment

[36]:
```
NYC_Borough = np.array(['Manhattan','Bronx','Brooklyn','Staten␣
 ↪Island','Queens'])
print (NYC_Borough)
```

```
['Manhattan' 'Bronx' 'Brooklyn' 'Staten Island' 'Queens']
```

[37]:
```
Borough_in_NYC = NYC_Borough
print (Borough_in_NYC)
```

```
['Manhattan' 'Bronx' 'Brooklyn' 'Staten Island' 'Queens']
```

[38]:
```python
Borough_in_NYC is NYC_Borough # Shows both objects are the same
```

[38]:
```
True
```

[40]:
```python
NYC_Borough[4] = 10000 # updating the original array
print (NYC_Borough)
```

```
['Manhattan' 'Bronx' 'Brooklyn' 'Staten Island' '10000']
```

[41]:
```python
print (Borough_in_NYC)
```

```
['Manhattan' 'Bronx' 'Brooklyn' 'Staten Island' '10000']
```

20 Copy and Views - View

[42]:
```python
NYC_Borough = np.array(['Manhattan','Bronx','Brooklyn','Staten␣
 ↪Island','Queens'])
print (NYC_Borough)
```

```
['Manhattan' 'Bronx' 'Brooklyn' 'Staten Island' 'Queens']
```

[43]:
```python
View_of_Borough_in_NYC = NYC_Borough.view() # creating view of array
print (View_of_Borough_in_NYC)
```

```
['Manhattan' 'Bronx' 'Brooklyn' 'Staten Island' 'Queens']
```

[44]:
```python
NYC_Borough is View_of_Borough_in_NYC # Views are different objects
```

[44]:
```
False
```

[45]:
```python
NYC_Borough[4] = 10000 # updating the original array
print (NYC_Borough)
```

```
['Manhattan' 'Bronx' 'Brooklyn' 'Staten Island' '10000']
```

[46]:
```python
print (View_of_Borough_in_NYC) # the view also gets update dwhen the orginal␣
 ↪array is updated.
```

```
['Manhattan' 'Bronx' 'Brooklyn' 'Staten Island' '10000']
```

21 Copy and Views - Deep Copy

[47]:
```python
NYC_Borough = np.array(['Manhattan','Bronx','Brooklyn','Staten␣
 ↪Island','Queens'])
print (NYC_Borough)
```

```
['Manhattan' 'Bronx' 'Brooklyn' 'Staten Island' 'Queens']
```

[48]:
```
Copy_of_Borough_in_NYC = NYC_Borough.copy() # creating view of array
print (Copy_of_Borough_in_NYC)
```

```
['Manhattan' 'Bronx' 'Brooklyn' 'Staten Island' 'Queens']
```

[49]:
```
NYC_Borough is Copy_of_Borough_in_NYC
```

[49]:
```
False
```

[50]:
```
NYC_Borough[4] = 10000 # updating the original array
print (NYC_Borough)
```

```
['Manhattan' 'Bronx' 'Brooklyn' 'Staten Island' '10000']
```

[ ]:
```
print (Copy_of_Borough_in_NYC)
```

[ ]:
```

```