What are Arrays?

**What is NumPy?**

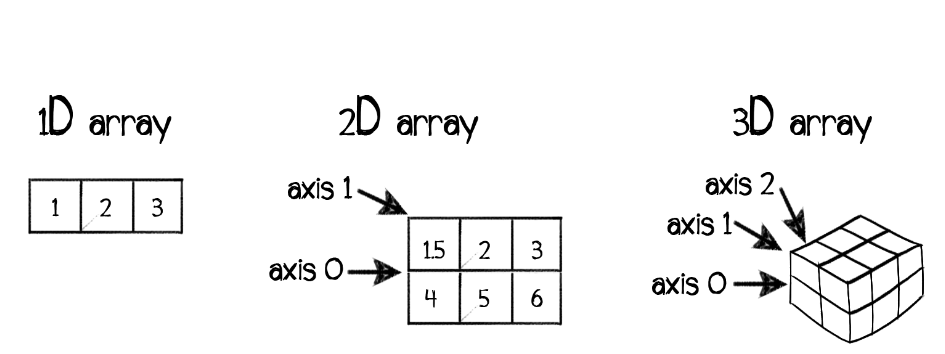
It stands for "Numerical Python". NumPy is a Python module that provides fast and efficient array operations of **homogeneous data**. It is the core library for scientific computing in Python providing a high-performance multidimensional array object, and tools for working with arrays.

NumPy is one of the many packages that are extremely essential in your data science journey because this library equips you with an array data structure that offers some benefits over the traditional data structures of Python like lists.

**NumPy Arrays**

The central feature of NumPy is the array object class, also called the **ndarray**. Arrays are very similar to lists in Python, **except that every element of an array must be of the same type** (in lists you can hold data which have different types), typically a numeric type like float or int. It is very much similar to an n-dimensional matrix which looks like:

Arrays make operations with large amounts of numeric data very fast and are generally much more efficient than lists. You can chose to create arrays of *n* dimensions (Python list is an array of pointers to Python objects, at least 4 bytes per pointer plus 16 bytes for even the smallest Python object; 4 for type pointer, 4 for reference count, 4 for value and the memory allocators rounds up to 16. A NumPy array is an array of uniform values -- single-precision numbers takes 4 bytes each, double-precision ones, 8 bytes).



**Creating NumPy arrays**

The syntax of creating a NumPy array is:

numpy.array(object, dtype = None, copy = True, order = None, subok = False, ndmin = 0)

Here, the arguments

* object: Any object exposing the array interface
* dtype: Desired data type of array, optional
* copy: Optional. By default (true), the object is copied
* order: C (row-major) or F (column-major) or A (any) (default)
* subok: By default, returned array forced to be a base class array. If true, sub-classes passed through
* ndim: Specifies minimum dimensions of the resultant array

Let's see how you can create a simple array using NumPy by first importing the package numpy as np

**import** numpy **as** np

a = np.array([1,2,3,4]) *# creates a 1-dimensional array*

b = np.array([[1,2,3,4], [5,6,7,8]]) *# creates a 2-dimensional array*

print(a)

print('----')

print(b)

Its output will be

[1 2 3 4]

----

[[1 2 3 4]

[5 6 7 8]]

**Advantages of using NumPy**

* Absolutely free since open-sourced
* Faster access in reading and writing items
* Time and space complexity of tasks is much lower when compared with traditional data structures
* Has a lot of built-in functions for linear algebra

# Attributes of NumPy arrays

Now that you know how to create a NumPy array, let us look at the most essential features of one and discuss them in details. We will be taking two arrays to illustrate the features

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

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

The attributes of both the arrays a and b are discussed below:

## Shape

It returns a tuple consisting of array dimensions i.e. tells us how many items are present in each dimension and can be found using the .shape the attribute of the ndarray object.

print('The shape of the array a is ', a.shape)

print('The shape of the array b is ', b.shape)

**Output**

'The shape of the array a is (4,)'

'The shape of the array b is (2,4)'

Note that the 1-D array *a* has a shape of (4, ) and not (4, 1)

## ****Dimensions****

It gives the number of dimensions and can be found using the .ndim the attribute of ndarray object.

print('The dimensions of array a is ', a.ndim)

print('The dimensions of array b is ', b.ndim)

**Output**

'The dimensions of array a is 1'

'The dimensions of array b is 2'

## Size

It tells the total number of items in the array as a whole. More precisely it is the product of the elements of the .shape the attribute of the array.

print('The size of the array a is ', a.size)

print('The size of the array b is ', b.size)

**Output**

'The shape of the array a is 4'

'The shape of the array b is 8'

## Datatype

As the name suggests, it informs about the type of data in the array. Since a NumPy array consists of homogeneous data only, you will get only a single dtype.

print('The datatype of the array a is ', a.dtype)

print('The datatype of the array b is ', b.dtype)

**Output**

'The datatype of the array a is int64'

'The datatype of the array b is int64'

NumPy offers support to a much greater variety of numerical types than base Python does like int8, int16, float32, float16, bool\_, complex\_ etc.

## Itemsize

It represents the number of bytes in each element of the array.

print('The number of bytes in each element of the array a is ', a.itemsize)

print('The number of bytes in each element of the array b is ', b.itemsize)

**Output**

'The number of bytes in each element of the array a is 8'

'The number of bytes in each element of the array b is 8'

With the help of the following attributes, you can get the necessary information of the NumPy array as a whole along with its elements.

Creating with Low-level ndarray Constructor(1/4)

50

LESSON

# Creating New NumPy Arrays

You already know to create a NumPy array using the numpy.array() command. Now, let's look at some of the other ways to make NumPy arrays taking the help of the low-level ndarray constructor (we will be using np as an alias for numpy).

## np.empty()

Creates an uninitialized (arbitrary) array of specified shape and dtype

np.empty((3,4),dtype='int8')

**Output**

array([[ 0, 0, 0, 0],

[ 0, 0, 0, 96],

[124, 0, -65, 21]], dtype=int8)

## np.zeros()

Creates a new array of specified size, filled with zeros

np.zeros((3,4),dtype='int8')

**Output**

array([[0, 0, 0, 0],

[0, 0, 0, 0],

[0, 0, 0, 0]], dtype=int8)

## np.ones()

Creates a new array of specified size and type, filled with ones

np.ones((3,4),dtype='int8')

**Output**

array([[1, 1, 1, 1],

[1, 1, 1, 1],

[1, 1, 1, 1]], dtype=int8)

## np.full()

Creates a new array of given shape and type, filled with a constant value

np.full((2,2),7)

**Output**

array([[7, 7],

[7, 7]])

## np.eye()

Creates a 2-D array with ones on the diagonal and zeros elsewhere

np.eye(3)

**Output**

array([[1., 0., 0.],

[0., 1., 0.],

[0., 0., 1.]])

PR

Creating with Existing Data(2/4)

0

# Arrays with Existing Data

New NumPy arrays can also be created from already existing ones. Let us look at some ways of doing so.

## np.asarray( )

This command is very similar to the **np.array()** command. Some examples are:

* Converting a list to array

*#python list*

a=[1,2,3]

*#convert to NumPy array*

b=np.asarray(a)

print(b)

**Output**

[1 2 3]

* Converting tuples into array

*#python tuples*

a=((1,2),(3,4))

*#convery to NumPy array*

b=np.asarray(a)

print(b)

**Output**

[[1 2]

[3 4]]

## np.fromiter( )

This function creates a NumPy ndarray from an iterable. For example:

a=np.fromiter([1,2,3,4],dtype='int8')

b=np.fromiter((1,2,3,4),dtype='int8')

c=np.fromiter(range(1,5),dtype='int8')

d=np.fromiter('string',dtype='S50')

print("Array a is ",a)

print("Array b is ",b)

print("Array c is ",c)

print("Array d is ",d)

**Output**

Array a **is** [1 2 3 4]

Array b **is** [1 2 3 4]

Array c **is** [1 2 3 4]

Array d **is** [b's' b't' b'r' b'i' b'n' b'g']

Here we are creating arrays from list a, tuple b, range( ) and finally a string d.

# Creating new arrays

In vector mathematics, it is necessary to generate a set of numbers within some predefined range. You can create them easily with the help of some NumPy functions. Let's look at some of them.

## np.arange( )

It returns an array containing evenly spaced values within a given range.

**Syntax** :numpy.arange(start, stop, step, dtype).

Here, numbers are created in the range of [start, stop-1]. We can also specify steps and data type using the step and dtype hyperparameters.

*# NumPy array from 1 to 19*

print(np.arange(1,20,dtype='int32'))

*# NumPy array from 1 to 19 with step size 2*

print(np.arange(1,20,2,dtype='int8'))

**Output**

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

[ 1 3 5 7 9 11 13 15 17 19]

In the above example, we first created an array from 1 to 19 with step size 1 and dtype int32 and in the second one we created one from 1 to 19 with step size 2 and dtype int8.

## np.linspace( )

It also returns an array within a range but not according to the step size as in the case of .arange() but according to the number of values we want within that range.

**Syntax**: numpy.linspace(start, stop, num, endpoint, retstep, dtype)

Here, start and stop means the same as in .arange( ) but the difference lies in num, which gives us the number of equally spaced numbers you want to insert within the range [start, stop-1]. The endpoint argument generally is by default set to the stop value (try changing it for some interesting results)

*# NumPy array from 1 to 20 with 100 numbers in between*

print(np.linspace(1,20,100))

**Output**

[ 1. 1.19191919 1.38383838 1.57575758 1.76767677 1.95959596

2.15151515 2.34343434 2.53535354 2.72727273 2.91919192 3.11111111

3.3030303 3.49494949 3.68686869 3.87878788 4.07070707 4.26262626

4.45454545 4.64646465 4.83838384 5.03030303 5.22222222 5.41414141

5.60606061 5.7979798 5.98989899 6.18181818 6.37373737 6.56565657

6.75757576 6.94949495 7.14141414 7.33333333 7.52525253 7.71717172

7.90909091 8.1010101 8.29292929 8.48484848 8.67676768 8.86868687

9.06060606 9.25252525 9.44444444 9.63636364 9.82828283 10.02020202

10.21212121 10.4040404 10.5959596 10.78787879 10.97979798 11.17171717

11.36363636 11.55555556 11.74747475 11.93939394 12.13131313 12.32323232

12.51515152 12.70707071 12.8989899 13.09090909 13.28282828 13.47474747

13.66666667 13.85858586 14.05050505 14.24242424 14.43434343 14.62626263

14.81818182 15.01010101 15.2020202 15.39393939 15.58585859 15.77777778

15.96969697 16.16161616 16.35353535 16.54545455 16.73737374 16.92929293

17.12121212 17.31313131 17.50505051 17.6969697 17.88888889 18.08080808

18.27272727 18.46464646 18.65656566 18.84848485 19.04040404 19.23232323

19.42424242 19.61616162 19.80808081 20. ]

In the above example, we have created 100 evenly spaced numbers in the range [1, 20]

## np.logspace( )

This function returns an array containing numbers that are evenly spaced on a log scale. Start and stop endpoints of the scale are indices of the base, usually 10.

**Syntax**: numpy.logspace(start, stop, num, endpoint, base, dtype)

Here, the range of values is [{base}^{start}, {base}^{stop}][*basestart*,*basestop*] with num being the number of equally spaced values on **log** scale within the range.

*# NumPy array from 10^0 to 10^2 with 100 numbers in log scale*

print(np.logspace(0,2,100))

**Output**

[ 1. 1.04761575 1.09749877 1.149757 1.20450354

1.26185688 1.32194115 1.38488637 1.45082878 1.51991108

1.59228279 1.66810054 1.7475284 1.83073828 1.91791026

2.009233 2.10490414 2.20513074 2.3101297 2.42012826

2.53536449 2.65608778 2.7825594 2.91505306 3.05385551

3.19926714 3.35160265 3.51119173 3.67837977 3.85352859

4.03701726 4.22924287 4.43062146 4.64158883 4.86260158

5.09413801 5.33669923 5.59081018 5.85702082 6.13590727

6.42807312 6.73415066 7.05480231 7.39072203 7.74263683

8.11130831 8.49753436 8.90215085 9.32603347 9.77009957

10.23531022 10.72267222 11.23324033 11.76811952 12.32846739

12.91549665 13.53047775 14.17474163 14.84968262 15.55676144

16.29750835 17.07352647 17.88649529 18.73817423 19.6304065

20.56512308 21.5443469 22.5701972 23.64489413 24.77076356

25.95024211 27.18588243 28.48035868 29.8364724 31.2571585

32.74549163 34.30469286 35.93813664 37.64935807 39.44206059

41.320124 43.28761281 45.34878508 47.50810162 49.77023564

52.14008288 54.62277218 57.22367659 59.94842503 62.80291442

65.79332247 68.92612104 72.20809018 75.64633276 79.24828984

83.02175681 86.97490026 91.11627561 95.45484567 100. ]

In the example above, there are 100 values in the range of [10^0, 10^2].

Indexing and Slicing(1/5)

50

LESSON

# Indexing

You now know how to create different types of NumPy arrays and check their features. But how about accessing a particular value or taking a chunk of values from the array itself? In this topic, we are going to discuss exactly that. **Like Python lists, the index starts at 0 for arrays as well**.

Array indexing and slicing are exactly similar like Python indexing and slicing. It follows the same pattern of array[start:stop: step]. Let us look at an example to observe this behaviour.

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

*# Pull out second element of third row*

print(a[2][1])

print('==========')

*# Pull out first two rows and columns*

print(a[:2,:2])

print('==========')

*# Pull all elements of the third row*

print(a[2,:])

**Output**

8

==========

[[1 2]

[4 5]]

==========

[7 8 9]

## Integer array indexing

Integer array indexing allows you to construct arbitrary arrays using the data from another array. Let us understand from the example

*# An example of integer array indexing*

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

print(a[[0,1,2],[0,1,0]])

print('==========')

print(np.array([a[0,0],a[1,1],a[2,0]]))

print('==========')

print(a[[0,0],[1,1]])

print('==========')

print(np.array([a[0,1],a[0,1]]))

**Output**

[1 4 5]

==========

[1 4 5]

==========

[2 2]

==========

[2 2]

Explanation: The print statements in line numbers 4 and 7 yields the same result, likewise in lines 10 and 13. In the first case, a[[0, 1, 2], [0, 1, 0]] essentially means we are indexing the value in first row-first column, second row-second column and third row-first column, which is the same as a[0, 0], a[1, 1], a[2, 0]]. Similarly, you should be able to deduce the logic behind the second case.

## Boolean indexing

This type is generally used for comparison purposes. For ex: How about checking if how many numbers in the array are greater than 50(say)? It can be performed using a simple comparison operator (>=, >, ==, <, <=)

A boolean index array is of the same shape as the array-to-be-filtered and it contains only True and False values. You can filter those you want using the concept of **masking**. For ex: If for some array a and boolean condition condition = a > 2, a[condition] will result in an array that contains only the numbers in array a that is greater than 2.

Let us look at an example below:

a = np.array([[4,7,1],[2,5,7],[7,1,1]])

*# Boolean condition for values greater than 3*

mask = a > 3

print(mask)

*# Masking for the above boolean condition in the array*

print(a[mask])

**Output**

[[ **True** **True** **False**]

[**False** **True** **True**]

[ **True** **False** **False**]]

[4 7 5 7 7]

PRACTICE

Vectorization(2/5)

50

LESSON

# What is vectorization?

Vectorization is the ability of NumPy by which we can perform operations on entire arrays rather than on a single element. When looping over an array or any data structure in Python, there’s a lot of overhead involved. Vectorized operations in NumPy delegate the looping internally to highly optimized C and Fortran functions, making for a cleaner and faster Python code.

## Examples

You have already come across such an example in Boolean indexing.

**import** numpy **as** np

a = np.array([1,2,3,4,5,6,7])

print(a[a > 2])

The above codeblock will output the array [3, 4, 5, 6, 7] as it compares each element being greater than or less than **2**.

# Vectorized operations

No, let us look at how you can do some elementary vectorized operations like addition, subtraction, multiplication etc. Images below depict the type of operations and their corresponding output.

## Addition

Two ways to go about it, using either + or np.add()

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

b=np.array([[10,11,12],[13,14,15],[16,17,18]])

*# element wise addition*

print(a+b)

print('==========')

print(np.add(a,b))

**Output**

[[11 13 15]

[17 19 21]

[23 25 27]]

==========

[[11 13 15]

[17 19 21]

[23 25 27]]

## Subtraction

Two ways, - or np.subtract()

*# element wise subtractions*

print(a-b)

print('==========')

print(np.subtract(a,b))

**Output**

[[-9 -9 -9]

[-9 -9 -9]

[-9 -9 -9]]

==========

[[-9 -9 -9]

[-9 -9 -9]

[-9 -9 -9]]

## Multiplication

Two ways, \* or np.multiply()

*# element wise multiplication*

print(a\*b)

print('==========')

print(np.multiply(a,b))

**Output**

[[ 10 22 36]

[ 52 70 90]

[112 136 162]]

==========

[[ 10 22 36]

[ 52 70 90]

[112 136 162]]

## Division

Two ways, / or np.divide()

*# element wise division*

print(a/b)

print('==========')

print(np.divide(a,b))

**Output**

[[0.1 0.18181818 0.25 ]

[0.30769231 0.35714286 0.4 ]

[0.4375 0.47058824 0.5 ]]

==========

[[0.1 0.18181818 0.25 ]

[0.30769231 0.35714286 0.4 ]

[0.4375 0.47058824 0.5 ]]

## Square root transform

Use np.sqrt()

*# element wise square root transform*

a = np.array([[1,4,9],[16,25,36]])

print(np.sqrt(a))

**Output**

[[1. 2. 3.]

[4. 5. 6.]]

## Log transform

Use np.log()

*# element wise square root transform*

a = np.array([[1,4,9],[16,25,36]])

print(np.log(a))

**Output**

[[0. 1.38629436 2.19722458]

[2.77258872 3.21887582 3.58351894]]

# Aggregrate operations

Aggregration operations are those where we perform some operation on the entire array. Some commonly used aggregrate operations are listed below:

| **Command** | **Description** |
| --- | --- |
| a.sum() | Array-wise sum |
| a.min() | Array-wise minimum value |
| a.max(axis=0) | Maximum value of an array row |
| a.cumsum(axis=1) | Cumulative sum of the elements |
| a.mean() | Mean |
| np.median(a) | Median |
| np.corrcoef(a) | Correlation coefficient |
| np.std(a) | Standard deviation |

# Array comparison

You already saw how you can perform element-wise comparison of array elements. With NumPy you also perform entire array comparisons. Use the command np.array\_equal() for array comparison. It is illustrated with examples below:

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

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

print(np.array\_equal(a,b))

**Output**

**True**

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

b = np.array([5,6,7,8])

print(np.array\_equal(a,b))

**Output**

**False**

# Understanding Axes notation

In NumPy, an axis refers to a single dimension of a multidimensional array. By changing axis you can compute across dimensions, whereas not specifying axis will result in computation over the entire array.

a = np.array([[1,4,9],[16,25,36]])

*# computes sum over columns*

print(a.sum(Axis=0))

print('==========')

*# computes sum over rows*

print(a.sum(axis=1))

print('==========')

*# computes total sum*

print(a.sum())

**Output**

[17 29 45]

==========

[14 77]

==========

91

In the image above you can calculate the sum over the rows, columns and the entire array just by playing around the parameter axis. Try it more on arrays with 3 or more dimensions!

PRACTICE

**What is broadcasting?**

Have you wondered how this operation np.array([1, 2, 3]) + 4 was successfully carried out? It was all due to the broadcasting power of NumPy arrays. Let's discuss this property in details.

In NumPy you do not need arrays to be of the same shape while performing operations among them until these conditions are satisfied:

* If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.
* The two arrays are said to be compatible in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
* The arrays can be broadcast together if they are compatible in all dimensions.
* After broadcasting, each array behaves as if it had shape equal to the element-wise maximum of shapes of the two input arrays.
* In any dimension where one array had size 1 and the other array had a size greater than 1, the first array behaves as if it were copied along that dimension.

**Visual intuition of broadcasting**

