2 - NumPy

October 14, 2021

Table of Contents

1 Overview

The following information is from the NumPy documentation.

https://numpy.org/doc/stable/user/absolute_beginners.html#

The sequence of information presented in this notebook follows the documentation sequence.

Welcome to NumPy!

NumPy (**Numerical Python**) is an open source Python library that's used in almost every field of science and engineering. It's the universal standard for working with numerical data in Python, and it's at the core of the scientific Python and PyData ecosystems. NumPy users include everyone from beginning coders to experienced researchers doing state-of-the-art scientific and industrial research and development. The NumPy API is used extensively in Pandas, SciPy, Matplotlib, scikit-learn, scikit-image and most other data science and scientific Python packages.

The NumPy library contains multidimensional array and matrix data structures (you'll find more information about this in later sections). It provides **ndarray**, a homogeneous n-dimensional array object, with methods to efficiently operate on it. NumPy can be used to perform a wide variety of mathematical operations on arrays. It adds powerful data structures to Python that guarantee efficient calculations with arrays and matrices and it supplies an enormous library of high-level mathematical functions that operate on these arrays and matrices.

What's the difference between a Python list and a NumPy array?

NumPy gives you an enormous range of fast and efficient ways of creating arrays and manipulating numerical data inside them. While a Python list can contain different data types within a single list, all of the elements in a NumPy array should be homogeneous. The mathematical operations that are meant to be performed on arrays would be extremely inefficient if the arrays weren't homogeneous.

Why use NumPy?

NumPy arrays are faster and more compact than Python lists. An array consumes less memory and is convenient to use. NumPy uses much less memory to store data and it provides a mechanism of specifying the data types. This allows the code to be optimized even further.

What is an array?

An array is a central data structure of the NumPy library. An array is a grid of values and it contains information about the raw data, how to locate an element, and how to interpret an element. It has a grid of elements that can be indexed in various ways. The elements are all of the same type, referred to as the array dtype.

An array can be indexed by a tuple of nonnegative integers, by booleans, by another array, or by integers. The rank of the array is the number of dimensions. The shape of the array is a tuple of integers giving the size of the array along each dimension.

One way we can initialize NumPy arrays is from Python lists, using nested lists for two- or higher-dimensional data.

For example:

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

or:

```
>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

We can access the elements in the array using square brackets. When you're accessing elements, remember that indexing in NumPy starts at 0. That means that if you want to access the first element in your array, you'll be accessing element "0".

```
>>> print(a[0])
[1 2 3 4]
```

2 Import NumPy

```python
[13]: import numpy as np

# This code is specific to use in kaggle.

#import os
#for dirname, _, filenames in os.walk('/kaggle/input'):
#    for filename in filenames:
#        print(os.path.join(dirname, filename))
```

3 Arrays

3.1 Creating a NumPy array:

- np.array(), np.zeros(), np.ones(), np.empty(), np.arange(), np.linspace(), dtype

```python
[14]: # Create an array from a list

      nums = [2,4,5,3,9]
      np.array(nums)
```

```
[14]: array([2, 4, 5, 3, 9])
```

```python
[15]: # Create a matrix from a list of lists

      nums_l = ([2,4,5,3,9],[4,7,12,34,76],[99,56,89,23,54])
      np.array(nums_l)
```

```
[15]: array([[ 2,  4,  5,  3,  9],
             [ 4,  7, 12, 34, 76],
             [99, 56, 89, 23, 54]])
```

```python
[16]: #

      np.array([[1,2,3], [3,2,1],[9,9,9]])   # Notice the extra square brackets!
```

```
[16]: array([[1, 2, 3],
             [3, 2, 1],
             [9, 9, 9]])
```

```python
[17]: # An array of zeros

      np.zeros(3)
```

```
[17]: array([0., 0., 0.])
```

```python
[18]: np.zeros((5,5))
```

```
[18]: array([[0., 0., 0., 0., 0.],
             [0., 0., 0., 0., 0.],
             [0., 0., 0., 0., 0.],
             [0., 0., 0., 0., 0.],
             [0., 0., 0., 0., 0.]])
```

```python
[19]: # An array of ones

      np.ones(3)
```

```
[19]: array([1., 1., 1.])
```

```python
[20]: np.ones((3,3))
```

```
[20]: array([[1., 1., 1.],
             [1., 1., 1.],
```

```
           [1., 1., 1.]])
```

[21]:
```
# An empty array.  Initial content is random numbers.

np.empty(2)
```

[21]:
```
array([ 1., 50.])
```

[22]:
```
np.empty((2,2))
```

[22]:
```
array([[-2.31584178e+077, -1.49458095e-154],
       [ 3.11109944e+231,  2.78134232e-309]])
```

3.1.1 arange()

[23]:
```
# An array with a range of elements

y=np.arange(0,10)    # start at zero and go for 10 positions
```

[24]:
```
type(y)
```

[24]:
```
numpy.ndarray
```

[25]:
```
np.arange(0,11,2)    # start at zero, end at eleven and increment by two
```

[25]:
```
array([ 0,  2,  4,  6,  8, 10])
```

3.1.2 np.linspace()

Return evenly spaced numbers over a specified interval.

[26]:
```
np.linspace(0,10,3)
```

[26]:
```
array([ 0.,  5., 10.])
```

[27]:
```
np.linspace(0,10,50)
```

[27]:
```
array([ 0.        , 0.20408163, 0.40816327, 0.6122449 , 0.81632653,
        1.02040816, 1.2244898 , 1.42857143, 1.63265306, 1.83673469,
        2.04081633, 2.24489796, 2.44897959, 2.65306122, 2.85714286,
        3.06122449, 3.26530612, 3.46938776, 3.67346939, 3.87755102,
        4.08163265, 4.28571429, 4.48979592, 4.69387755, 4.89795918,
        5.10204082, 5.30612245, 5.51020408, 5.71428571, 5.91836735,
        6.12244898, 6.32653061, 6.53061224, 6.73469388, 6.93877551,
        7.14285714, 7.34693878, 7.55102041, 7.75510204, 7.95918367,
        8.16326531, 8.36734694, 8.57142857, 8.7755102 , 8.97959184,
        9.18367347, 9.3877551 , 9.59183673, 9.79591837, 10.        ])
```

```
[28]: np.ones(2, dtype=np.int64)
```

```
[28]: array([1, 1])
```

```
[29]: np.ones(2, dtype=np.float64)
```

```
[29]: array([1., 1.])
```

3.2   Arrays with Random Numbers

- np.random, np.randon.rand, np.randon.randn, np.random.randint

```
[30]: np.random                          # random is a module within numpy
```

```
[30]: <module 'numpy.random' from
      '/opt/anaconda3/envs/cdc_training/lib/python3.8/site-
      packages/numpy/random/__init__.py'>
```

```
[31]: np.random.rand(2)               # Creates sample with a uniform distribution
```

```
[31]: array([0.64901909, 0.63752075])
```

```
[32]: np.random.seed(1234)            # Seed ensures we get the same random numbers
      np.random.rand(2)
```

```
[32]: array([0.19151945, 0.62210877])
```

```
[33]: np.random.randn(2)              # Creates a sample using the standard normal␣
       ↪distribution
```

```
[33]: array([ 1.43270697, -0.3126519 ])
```

```
[34]: np.random.randint(1,100)        # Return random integers from low (inclusive) to␣
       ↪high (exclusive)
```

```
[34]: 16
```

```
[35]: np.random.randint(1,100,10)
```

```
[35]: array([50, 24, 27, 31, 44, 31, 27, 59, 93, 70])
```

```
[36]: # You can explicity import random and then np. can be left off.

      from numpy import random
```

```
[37]: random.seed(1234)
      random.rand(2)
```

```
[37]: array([0.19151945, 0.62210877])
```

3.2.1 Sort

```
[38]: my_arr = np.array([2, 1, 5, 3, 7, 4, 6, 8])
      np.sort(my_arr)
```

```
[38]: array([1, 2, 3, 4, 5, 6, 7, 8])
```

3.2.2 Concatenate

```
[39]: a = np.array([1, 2, 3, 4])
      b = np.array([5, 6, 7, 8])
      c = np.concatenate((a, b))
      c
```

```
[39]: array([1, 2, 3, 4, 5, 6, 7, 8])
```

```
[40]: a = np.array([[1, 2, 3, 4]])   # Notice those extra square brackets again!
      b = np.array([[5, 6, 7, 8]])
      d = np.concatenate((a, b))
      d
```

```
[40]: array([[1, 2, 3, 4],
             [5, 6, 7, 8]])
```

```
[41]: # From the documentation

      q = [[1, 2], [3, 4]]

      a = np.array([[1, 2], [3, 4]])   # This is a 2x2 matrix
      b = np.array([[5, 6]])           # This is a 1-d array

      print(q, type(q))
      print(a, type(a))
      print(b, type(b))
```

```
      [[1, 2], [3, 4]] <class 'list'>
      [[1 2]
       [3 4]] <class 'numpy.ndarray'>
      [[5 6]] <class 'numpy.ndarray'>
```

```
[42]: np.concatenate((a, b), axis=0)
```

```
[42]: array([[1, 2],
             [3, 4],
             [5, 6]])
```

```
[43]: np.concatenate((a, b.T), axis=1)
```

```
[43]: array([[1, 2, 5],
              [3, 4, 6]])
```

```
[44]: np.concatenate((a, b), axis=None)
```

```
[44]: array([1, 2, 3, 4, 5, 6])
```

3.3   Shape and Reshape

ndim, size & shape are attributes of the array (not methods). Notice that () are not used.

```
[45]: a = np.array([[1, 2], [3, 4]])
      a.ndim
```

```
[45]: 2
```

```
[46]: a.size
```

```
[46]: 4
```

```
[47]: a.shape
```

```
[47]: (2, 2)
```

```
[48]: # Reshape an array

      m = np.arange(50)
      m
```

```
[48]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
             17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
             34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49])
```

```
[49]: m.reshape(5,10)
```

```
[49]: array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
             [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
             [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
             [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
             [40, 41, 42, 43, 44, 45, 46, 47, 48, 49]])
```

```
[50]: m.reshape(2,25)
```

```
[50]: array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15,
             16, 17, 18, 19, 20, 21, 22, 23, 24],
             [25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
             41, 42, 43, 44, 45, 46, 47, 48, 49]])
```

3.4 Transpose (swap axes) a matrix

```
[51]: nums_l = ([2,4,5,3,9],[4,7,12,34,76],[99,56,89,23,54])
      x = np.array(nums_l)
      x
```

```
[51]: array([[ 2,  4,  5,  3,  9],
             [ 4,  7, 12, 34, 76],
             [99, 56, 89, 23, 54]])
```

```
[52]: x.transpose()
```

```
[52]: array([[ 2,  4, 99],
             [ 4,  7, 56],
             [ 5, 12, 89],
             [ 3, 34, 23],
             [ 9, 76, 54]])
```

```
[53]: y = np.flip(x)
      y
```

```
[53]: array([[54, 23, 89, 56, 99],
             [76, 34, 12,  7,  4],
             [ 9,  3,  5,  4,  2]])
```

4 Indexing, Slicing & Selecting

```
[54]: m = np.arange(12)
      m
```

```
[54]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

```
[55]: m[6]
```

```
[55]: 6
```

```
[56]: m[2:5]            # Remember, 5 is not included
```

```
[56]: array([2, 3, 4])
```

4.1 Indexing a 2d array (matrix)

```
[57]: m2 = np.array(([2,4,5,3,9],[4,7,12,34,76],[99,56,89,23,54]))
      m2
```

```
[57]: array([[ 2,  4,  5,  3,  9],
             [ 4,  7, 12, 34, 76],
```

```
         [99, 56, 89, 23, 54]])
```

Format is **array_name[row][col]** or **array_name[row,col]**. Comma notation is more common.

```
[58]: #Indexing row
      m2[1]
```

```
[58]: array([ 4,  7, 12, 34, 76])
```

```
[59]: # Getting individual element value
      m2[1][0]
```

```
[59]: 4
```

```
[60]: # Getting individual element value
      m2[1,0]
```

```
[60]: 4
```

```
[61]: # 2D array slicing

      m2[:2,1:]   # Get all values in rows 0 & 1, get all values in those rows after␣
       ↪the first column.
```

```
[61]: array([[ 4,  5,  3,  9],
             [ 7, 12, 34, 76]])
```

```
[62]: m2[2]
```

```
[62]: array([99, 56, 89, 23, 54])
```

```
[63]: m2[2,:]
```

```
[63]: array([99, 56, 89, 23, 54])
```

4.2 Selecting (if time permits)

```
[64]: m2 = np.array(([2,4,5,3,9],[4,7,12,34,76],[99,56,89,23,54]))
      m2
```

```
[64]: array([[ 2,  4,  5,  3,  9],
             [ 4,  7, 12, 34, 76],
             [99, 56, 89, 23, 54]])
```

```
[65]: m2 > 4
```

```
[65]: array([[False, False,  True, False,  True],
             [False,  True,  True,  True,  True],
             [ True,  True,  True,  True,  True]])
```

```
[66]: bool_m2 = m2 > 4
      bool_m2
```

```
[66]: array([[False, False,  True, False,  True],
             [False,  True,  True,  True,  True],
             [ True,  True,  True,  True,  True]])
```

```
[67]: m2[bool_m2]
```

```
[67]: array([ 5,  9,  7, 12, 34, 76, 99, 56, 89, 23, 54])
```

5 Operations

```
[68]: # Add arrays together

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

```
[68]: array([[ 6,  8, 10, 12]])
```

```
[69]: # Perform more math on arrays
      c = a*b
      d = a/b
      e = a-b
      print(c)
      print(d)
      print(e)
```

```
[[ 5 12 21 32]]
[[0.2        0.33333333 0.42857143 0.5       ]]
[[-4 -4 -4 -4]]
```

```
[70]: # Aggregate functions - Sum

      a.sum()
```

```
[70]: 10
```

```
[71]: # Aggregate functions - min, max, mean

      f = a.min()
      g = a.max()
      h = a.mean()
      i = a.std()

      print(f)
```

```
print(g)
print(h)
print(i)
```

```
1
4
2.5
1.118033988749895
```

5.1 Numpy Array Aggregate Functions

Functions	Description
np.mean()	Compute the arithmetic mean along the specified axis.
np.std()	Compute the standard deviation along the specified axis.
np.var()	Compute the variance along the specified axis.
np.sum()	Sum of array elements over a given axis.
np.prod()	Return the product of array elements over a given axis.
np.cumsum()	Return the cumulative sum of the elements along a given axis.
np.cumprod()	Return the cumulative product of elements along a given axis.
np.min(), np.max()	Return the minimum / maximum of an array or minimum along an axis.
np.argmin(), np.argmax()	Returns the indices of the minimum / maximum values along an axis
np.all()	Test whether all array elements along a given axis evaluate to True.
np.any()	Test whether any array element along a given axis evaluates to True.

https://www.pythonprogramming.in/numpy-aggregate-and-statistical-functions.html[image7_aggregate.png]

5.2 Broadcasting

```
[72]: a
```

```
[72]: array([[1, 2, 3, 4]])
```

```
[73]: a*2
```

```
[73]: array([[2, 4, 6, 8]])
```

6 Exercises

```
[74]: # Create an array with 20 zeros
      np.zeros((20))
```

```
[74]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
             0., 0., 0.])
```

```
[75]: # Create a 2d array with 20 zeros
      np.zeros((4,5))
```

```
[75]: array([[0., 0., 0., 0., 0.],
             [0., 0., 0., 0., 0.],
             [0., 0., 0., 0., 0.],
             [0., 0., 0., 0., 0.]])
```

```
[76]: # Create an array of 20 ones
      np.ones((20))
```

```
[76]: array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
             1., 1., 1.])
```

```
[77]: # Create an array of 20 fives (requires 2 steps)
      x = np.ones((4,5))
      x*5
```

```
[77]: array([[5., 5., 5., 5., 5.],
             [5., 5., 5., 5., 5.],
             [5., 5., 5., 5., 5.],
             [5., 5., 5., 5., 5.]])
```

```
[78]: # Create an array with all the odd numbers between 0 and 50
      np.arange(1,50,2)
```

```
[78]: array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33,
             35, 37, 39, 41, 43, 45, 47, 49])
```

```
[79]: # Create an array with all the numbers between 15 and 50 incremented by 5
      np.arange(15,50,5)
```

```
[79]: array([15, 20, 25, 30, 35, 40, 45])
```

```
[80]: # Create a 5 x 5 matrix with numbers ranging from 5 to 25
      t = np.arange(5,30).reshape(5,5)
      t
```

```
[80]: array([[ 5,  6,  7,  8,  9],
             [10, 11, 12, 13, 14],
             [15, 16, 17, 18, 19],
             [20, 21, 22, 23, 24],
             [25, 26, 27, 28, 29]])
```

```
[81]:  # Using indexing from t, pick out the value 22

       t[3,2]
```

[81]: 22

```
[82]:  # Using t, create the following output

       # array([16, 17, 18])

       t[2,1:4]
```

[82]: array([16, 17, 18])

```
[83]:  # Using t, create the following output

       # array([[15, 16, 17],
       #        [20, 21, 22]])

       t[2:5,0:3]
```

[83]: array([[15, 16, 17],
              [20, 21, 22],
              [25, 26, 27]])

```
[84]:  # Using t, create the following output

       # array([[ 6],
       #        [11],
       #        [16],
       #        [21]])

       t[:4,1:2]
```

[84]: array([[6],
              [11],
              [16],
              [21]])
```