00-NumPy-Arrays

October 19, 2022

For more information, visit us at www.pieriandata.com

1 NumPy

NumPy is a powerful linear algebra library for Python. What makes it so important is that almost all of the libraries in the PyData ecosystem (pandas, scipy, scikit-learn, etc.) rely on NumPy as one of their main building blocks. Plus we will use it to generate data for our analysis examples later on!

NumPy is also incredibly fast, as it has bindings to C libraries. For more info on why you would want to use arrays instead of lists, check out this great StackOverflow post.

We will only learn the basics of NumPy. To get started we need to install it!

1.1 Installation Instructions

1.1.1 NumPy is already included in your environment! You are good to go if you are using the tsa_course env!

For those not using the provided environment: 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
```

If you do not have Anaconda and can not install it, please refer to Numpy's official documentation on various installation instructions.

1.2 Using NumPy

Once you've installed NumPy you can import it as a library:

[1]:
```
import numpy as np
```

NumPy has many built-in functions and capabilities. We won't cover them all but instead we will focus on some of the most important aspects of NumPy: vectors, arrays, matrices and number generation. Let's start by discussing arrays.

2 NumPy Arrays

NumPy arrays are the main way we will use NumPy throughout the course. NumPy arrays essentially come in two flavors: vectors and matrices. Vectors are strictly 1-dimensional (1D) arrays and matrices are 2D (but you should note a matrix can still have only one row or one column).

Let's begin our introduction by exploring how to create NumPy arrays.

2.1 Creating NumPy Arrays

2.1.1 From a Python List

We can create an array by directly converting a list or list of lists:

[2]:
```
my_list = [1,2,3]
my_list
```

[2]:
```
[1, 2, 3]
```

[3]:
```
np.array(my_list)
```

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

[4]:
```
my_matrix = [[1,2,3],[4,5,6],[7,8,9]]
my_matrix
```

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

[5]:
```
np.array(my_matrix)
```

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

2.2 Built-in Methods

There are lots of built-in ways to generate arrays.

2.2.1 arange

Return evenly spaced values within a given interval. [reference]

[6]:
```
np.arange(0,10)
```

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

[7]:
```
np.arange(0,11,2)
```

[7]:
```
array([ 0,  2,  4,  6,  8, 10])
```

2.2.2 zeros and ones

Generate arrays of zeros or ones. [reference]

[8]:
```
np.zeros(3)
```

[8]:
```
array([0., 0., 0.])
```

[9]:
```
np.zeros((5,5))
```

[9]:
```
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.]])
```

[10]:
```
np.ones(3)
```

[10]:
```
array([1., 1., 1.])
```

[11]:
```
np.ones((3,3))
```

[11]:
```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

2.2.3 linspace

Return evenly spaced numbers over a specified interval. [reference]

[12]:
```
np.linspace(0,10,3)
```

[12]:
```
array([ 0.,  5., 10.])
```

[13]:
```
np.linspace(0,5,20)
```

[13]:
```
array([0.        , 0.26315789, 0.52631579, 0.78947368, 1.05263158,
       1.31578947, 1.57894737, 1.84210526, 2.10526316, 2.36842105,
       2.63157895, 2.89473684, 3.15789474, 3.42105263, 3.68421053,
       3.94736842, 4.21052632, 4.47368421, 4.73684211, 5.        ])
```

Note that `.linspace()` *includes* the stop value. To obtain an array of common fractions, increase the number of items:

[14]:
```
np.linspace(0,5,21)
```

[14]:
```
array([0.  , 0.25, 0.5 , 0.75, 1.  , 1.25, 1.5 , 1.75, 2.  , 2.25, 2.5 ,
       2.75, 3.  , 3.25, 3.5 , 3.75, 4.  , 4.25, 4.5 , 4.75, 5.  ])
```

2.2.4 eye

Creates an identity matrix [reference]

[15]:
```
np.eye(4)
```

[15]:
```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

2.3 Random

Numpy also has lots of ways to create random number arrays:

2.3.1 rand

Creates an array of the given shape and populates it with random samples from a uniform distribution over `[0, 1)`. [reference]

[16]:
```
np.random.rand(2)
```

[16]:
```
array([0.37065108, 0.89813878])
```

[17]:
```
np.random.rand(5,5)
```

[17]:
```
array([[0.03932992, 0.80719137, 0.50145497, 0.68816102, 0.1216304 ],
       [0.44966851, 0.92572848, 0.70802042, 0.10461719, 0.53768331],
       [0.12201904, 0.5940684 , 0.89979774, 0.3424078 , 0.77421593],
       [0.53191409, 0.0112285 , 0.3989947 , 0.8946967 , 0.2497392 ],
       [0.5814085 , 0.37563686, 0.15266028, 0.42948309, 0.26434141]])
```

2.3.2 randn

Returns a sample (or samples) from the "standard normal" distribution [= 1]. Unlike **rand** which is uniform, values closer to zero are more likely to appear. [reference]

[18]:
```
np.random.randn(2)
```

[18]:
```
array([-0.36633217, -1.40298731])
```

[19]:
```
np.random.randn(5,5)
```

[19]:
```
array([[-0.45241033,  1.07491082,  1.95698188,  0.40660223, -1.50445807],
       [ 0.31434506, -2.16912609, -0.51237235,  0.78663583, -0.61824678],
       [-0.17569928, -2.39139828,  0.30905559,  0.1616695 ,  0.33783857],
       [-0.2206597 , -0.05768918,  0.74882883, -1.01241629, -1.81729966],
       [-0.74891671,  0.88934796,  1.32275912, -0.71605188,  0.0450718 ]])
```

2.3.3 randint

Returns random integers from `low` (inclusive) to `high` (exclusive). [reference]

[20]:
```
np.random.randint(1,100)
```

[20]:
```
61
```

[21]:
```
np.random.randint(1,100,10)
```

[21]:
```
array([39, 50, 72, 18, 27, 59, 15, 97, 11, 14])
```

2.3.4 seed

Can be used to set the random state, so that the same "random" results can be reproduced. [reference]

[22]:
```
np.random.seed(42)
np.random.rand(4)
```

[22]:
```
array([0.37454012, 0.95071431, 0.73199394, 0.59865848])
```

[23]:
```
np.random.seed(42)
np.random.rand(4)
```

[23]:
```
array([0.37454012, 0.95071431, 0.73199394, 0.59865848])
```

2.4 Array Attributes and Methods

Let's discuss some useful attributes and methods for an array:

[24]:
```
arr = np.arange(25)
ranarr = np.random.randint(0,50,10)
```

[25]:
```
arr
```

[25]:
```
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])
```

[26]:
```
ranarr
```

[26]:
```
array([38, 18, 22, 10, 10, 23, 35, 39, 23,  2])
```

2.5 Reshape

Returns an array containing the same data with a new shape. [reference]

[27]:
```
arr.reshape(5,5)
```

[27]:
```
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]])
```

2.5.1 max, min, argmax, argmin

These are useful methods for finding max or min values. Or to find their index locations using argmin or argmax

[28]:
```
ranarr
```

[28]:
```
array([38, 18, 22, 10, 10, 23, 35, 39, 23,  2])
```

[29]:
```
ranarr.max()
```

[29]:
```
39
```

[30]:
```
ranarr.argmax()
```

[30]:
```
7
```

[31]:
```
ranarr.min()
```

[31]:
```
2
```

[32]:
```
ranarr.argmin()
```

[32]:
```
9
```

2.6 Shape

Shape is an attribute that arrays have (not a method): [reference]

[33]:
```
# Vector
arr.shape
```

[33]:
```
(25,)
```

[34]:
```
# Notice the two sets of brackets
arr.reshape(1,25)
```

[34]:
```
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]])
```

[35]:
```
arr.reshape(1,25).shape
```

[35]:
```
(1, 25)
```

[36]:
```
arr.reshape(25,1)
```

[36]:
```
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]])
```

[37]:
```
arr.reshape(25,1).shape
```

[37]:
```
(25, 1)
```

2.6.1 dtype

You can also grab the data type of the object in the array: [reference]

[38]:
```
arr.dtype
```

[38]:
```
dtype('int32')
```

[39]:
```
arr2 = np.array([1.2, 3.4, 5.6])
arr2.dtype
```

[39]:
```
dtype('float64')
```

3 Great Job!