tutorial011-Numpy-Introduction

October 13, 2022

1 `NumPy`: Numerical operations in Python

This tutorial uses material also found in the SciPy 1.0 Nature Methods and NumPy Array IEEE article and at `https://numpy.org/`.

1.0.1 Learning outcomes:

- Understand Python Library `NumPy`
- Use `numpy arrays`
- Apply loops and logical operators on `numpy arrays`

Data scientists spend a lot of time – wait for it – working with ***data***! To work with **data** it is critical to organize the data in a way that facilitate the work on the potential analyses we might need to do. So organizing data means guessing what type of work we will want to do with the dataset. And, odd is it may seem, good guessing requires some practice. The data organization process will require:

- store the data a clear and systematic way
- provide methods to access the data that are simple and straightforward
- be flexible enough so to and allow to modify the format of the data for various needs

NumPy is the fundamental `library` for mathematical operations and computations in Python.

The NumPy array is a multidimensional array object. A variety of fast operations on arrays are provided by NumPy. These include operations that are mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

NumPy is the base of scientific computing and data science libraries such as Pandas scipy.org, and scikit-learn.org among many others.

In other (simpler?) words, Numpy arrays are grids or tables for holding, accessing, and manipulating data. They are created and accessed in ways that are very similar to the ways Python `lists` can be accessed.

So what we are going to do is to recall how `lists` work (lists are handy!), then we will graduate to `numpy arrays` and see what they can do.

1.0.2 Python lists

We have covered Python `lists` (and other datatypes) in previous tutorials. Python `lists` (a list of things) is build by collecting, ahem, a list of things using `[square brackets]`.

For example:

[17]:
```
mylist = ['this', 3, 'list', 4+2j, 6.66]
```

We can address elements in a list by using indices and the `:` (colon) operator.

[18]:
```
mylist[0:3]
```

[18]:
```
['this', 3, 'list']
```

We can read this as "Give me all the elements in the interval between 0 **inclusive** to 3 **exclusive**."

I know this is weird. But at least for any two indexes `a` and `b`, the number of elements you get back from `mylist[a,b]` is always equal to `b` minus `a`, so I guess that's good!

We can get any consecutive hunk of elements using `:`.

[19]:
```
mylist[2:5]
```

[19]:
```
['list', (4+2j), 6.66]
```

If you omit the indexes, Python will assume you want everything.

[20]:
```
mylist[:]
```

[20]:
```
['this', 3, 'list', (4+2j), 6.66]
```

List can obviously host also homogeneous types of data, such as `int` or `float`:

[21]:
```
mylistHomogeneous = [2, 3.14, 10.5, 11.13, 12.7, 4.31]
```

1.0.3 Numpy Arrays

Numpy arrays were designed to be lists with superpowers, so everything we learned about `lists` will apply to `numpy arrays` as well!

A NumPy array is a multidimensional, uniform collection of elements (that is, all elements occupy the same number of bytes in memory). An array is characterized by - the type of elements it contains and - its shape.

For example, a matrix might be represented as an array of shape M×N that contains numbers, such as floating-point or complex numbers. Unlike matrices, NumPy arrays can have up to 32 dimensions; they might also contain other kinds of elements (or even combinations of elements), such as Booleans or dates. Ref. Van Der Walt et al. IEEE

Not to state the obvious, but to use `numpy arrays`, we'll need to `import` the library `numpy`. The standard is to import `numpy` as `np`:

[22]:
```
import numpy as np
```

`arrays` can be made by simply asking for one and filling it out with values, in much the same way we make a `list`.

[23]:
```
myarr = np.array([2, 4, 6, 8, 9, 10])
```

The command `print` can be used also on `NumPy arrays` and it returns the content of the array:

[24]:
```
print(myarr)
```

```
[ 2  4  6  8  9 10]
```

By simply returning the array object name (`myarr`) the output is a bit more informative and it returns the type (`array`):

[25]:
```
myarr
```

[25]:
```
array([ 2,  4,  6,  8,  9, 10])
```

From then on, all the indexing we've learned so far applies directly! Square brackets are used for indexing and the same type of addressing can be used as we have learned for `lists`:

[26]:
```
myarr[4]
```

[26]:
```
9
```

[27]:
```
myarr[-3:]
```

[27]:
```
array([ 8,  9, 10])
```

Complete the following exercise.

- Create a `NumPy array` containing both `int`, `float` and `complex` datatypes.

[Use the cell below to show your code]

[28]:
```
numpyarr = np.array([8, 8.8, 8 + 8j])
```

- Create a `NumPy array` containing both `str` as datatypes.

[Use the cell below to show your code]

[29]:
```
numpyarr2 = np.array(['i','do','not','know'])
```

1.0.4 Operations on `numpy arrays`: the difference between `lists` and `arrays`

Indeed, we can make an array directly out of a list.

[30]:
```
myArrFromList = np.array(mylistHomogeneous)
```

[31]:
```
myArrFromList
```

[31]:
```
array([ 2.  ,  3.14, 10.5 , 11.13, 12.7 ,  4.31])
```

And then of course we can index it exactly the same way, so… *Wait, why are we making arrays now? What's the difference?*

One **huge** difference is that if we wanted to do some math with basic Python lists, the fact that they can hold multiple types of data elements does not assure that the mathematical operations will perform.

[32]:
```
mylistHomogeneous + 5
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [32], line 1
----> 1 mylistHomogeneous + 5

TypeError: can only concatenate list (not "int") to list
```

`numpy` arrays instead contain numerical elements by definition. This definition assures the ability to perform math ith the arrays. So, whereas the addition above did not work when using the `list`, it does work when using the `numpy` array, even though both `list` and `array` contain the same elements!

[34]:
```
myArrFromList + 5
```

[34]:
```
array([ 7.  ,  8.14, 15.5 , 16.13, 17.7 ,  9.31])
```

Now **that** seems like it might be useful!

Complete the following exercise.

- What happens when you add a number to a `NumPy array`? How do the content of the array change?

The math operation is performed on each object in the array. In this case the operation adds 5 to each element.

- Create a new `array` and multiply the array by a complex number:

[Use the cell below to show your code]

[35]:
```
newarr = np.array([8, 8.8, 6, 6.6, 8.6, 6.8])
```

[37]:
```
newarr * (6 + 6j)
```

[37]:
```
array([48. +48.j , 52.8+52.8j, 36. +36.j , 39.6+39.6j, 51.6+51.6j,
       40.8+40.8j])
```

`[Use the cell below to describe in your own words what happnes to the elements of the array af`

Each of the elements is multipled by the complex number.

1.0.5 More operations on arrays

Two arrays can be added, or subtracted, or multiplied or whatever!

[38]:
```
myarr + myArrFromList
```

[38]:
```
array([ 4.  ,  7.14, 16.5 , 19.13, 21.7 , 14.31])
```

[39]:
```
myarr * myArrFromList
```

[39]:
```
array([  4.  ,  12.56,  63.  ,  89.04, 114.3 ,  43.1 ])
```

[40]:
```
myarr / myArrFromList
```

[40]:
```
array([1.        , 1.27388535, 0.57142857, 0.71877808, 0.70866142,
       2.32018561])
```

We can also **combine** our 2 arrays into a single ***two dimensional (2D) array***.

[41]:
```
twoDarr = np.array([myarr, myArrFromList])
```

[42]:
```
twoDarr
```

[42]:
```
array([[ 2.  ,  4.  ,  6.  ,  8.  ,  9.  , 10.  ],
       [ 2.  ,  3.14, 10.5 , 11.13, 12.7 ,  4.31]])
```

Simple though this may seem, *2D arrays just like this are the bedrock of data analysis!* Arrays of real data are usually larger – sometimes much much larger! – but all the principles are the same and all you as a Data Scientists need to remember is the dimensionality of the data arrays. Python will then compute what you ask for.

But, hold on one second. Remembering the dimensionality of the array is ** *very* ** imoortant. Indeed Python can perform some operations if two arrays do *not* have the same dimensions, but other operations are likely to fail.

For example, imagine two `arrays` with different dimensions:

[43]:
```
smallArray = [2, 3, 4]
largeArray = [2, 3, 4, 5, 6, 7]
```

The two arrays can be added together by using the symbol `+`:

[44]:
```
smallArray + largeArray
```

[44]:
```
[2, 3, 4, 2, 3, 4, 5, 6, 7]
```

Yet, the same two arrays cannot be multiplied:

[45]:
```
smallArray * largeArray
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [45], line 1
----> 1 smallArray * largeArray

TypeError: can't multiply sequence by non-int of type 'list'
```

This is because Python cannot identify elements to match during the multiplication.

Complete the following exercise.

- What happens when you add two arrays of different dimensions? Say one array with 6 complex numers and one with 4 `float` numbers?

  [Use the cell below to show how to create and add the two arrays]

[56]:
```
onea = np.array([6 + 6j, 6 - 6j, 1 + 2j, 1 - 2j, 8 + 8j, 8 - 8j])
twoa = np.array([6.6, 8.8, 8.6, 6.8])

onea + twoa
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In [56], line 4
      1 onea = np.array([6 + 6j, 6 - 6j, 1 + 2j, 1 - 2j, 8 + 8j, 8 - 8j])
      2 twoa = np.array([6.6, 8.8, 8.6, 6.8])
----> 4 onea + twoa

ValueError: operands could not be broadcast together with shapes (6,) (4,)
```

[Use the cell below to describe in your own words what happens to the arrays as result of the a

The arrays could not be added due to being different shapes. The matrices do not line up.

1.0.6 Methods of `numpy arrays`

So the shape of the array (the dimensionality) is key, especially if we plan on doing math with the arrays, whihc is the primary goal of the arrays!

`numpy arrays` are Python objects and as such they have `methods`. A variety of methods exist for the array and `shape` is the one that allow us to retrieve the dimensionality of an array.

[57]:
```
twoDarr.shape
```

[57]: (2, 6)

Unlike lists, which are always just lists, arrays can come in any shape. So it's *really* convenient that they can tell us what shape they are straight away.

Indexing into 2D arrays is a straightforward extension of indexing into 1D arrays or lists. We just provide a second index after a , (comma). Like this.

[58]:
```
twoDarr[1,3]
```

[58]:
```
11.13
```

The first index refers to the **row index**, and the second to the **column index**. In this case, we're asking for the value in the second row and the fourth column, which is indeed 7 (remember *the first row and column are index=0!*).

We can play all the same games indexing with 2D arrays as we can with 1D arrays, we just have to remember that everything before the comma , refers to the *rows* in that it specifies locations along the *vertical dimension*, and everything after the comma , refers to the *columns* in that it specifies locations along the *horizontal dimension*.

So this:

[59]:
```
twoDarr[:,0:3]
```

[59]:
```
array([[ 2.  ,  4.  ,  6.  ],
       [ 2.  ,  3.14, 10.5 ]])
```

means "Give me all the rows (the colon :) in the first 3 columns (the"`0:3`)."

I told you that the colon all alone by itself would end up being useful!!! In this case for example, by using the : you do not need to type many indices (one per row) and you even do not need to remmeber how many rows there are, just use : and Python will return all the elements.

A few more examples:

[61]:
```
# the last row (regardless of the number of rows,
# again you do not need to knowhow many rows exist)
twoDarr[-1,:]
```

[61]:
```
array([ 2.  ,  3.14, 10.5 , 11.13, 12.7 ,  4.31])
```

[62]:
```
twoDarr[:,-2:] # last two columns
```

[62]:
```
array([[ 9.  , 10.  ],
       [12.7 ,  4.31]])
```

[63]:
```
twoDarr[0,::2] # first row, every other column
```

[63]:
```
array([2., 6., 9.])
```

To get good at this, you don't need natural born talent or anything like that. Like so much in life, the key is *practice, practice, practice*!!! So play around! You can't break your computer or anything!

Another neat trick that arrays can do is *transpose* themselves, flipping the rows for columns.

(Hold your right hand in front of your face so that you're looking at your palm with your fingers pointing towards the left. Now flip your hand so that you're looking at the back of your hand with your fingers pointing up. You just *transposed* your hand such that the first row (your pointer finger) became the first column!)

[64]:
```
colarr = twoDarr.T
```

[65]:
```
colarr
```

[65]:
```
array([[ 2.  ,  2.  ],
       [ 4.  ,  3.14],
       [ 6.  , 10.5 ],
       [ 8.  , 11.13],
       [ 9.  , 12.7 ],
       [10.  ,  4.31]])
```

Why would we want to do that? By convention, *variables* in datasets should correspond to the columns, and *observations* should correspond to the rows. So we have taken data in which this was not so and turned it into an array in which the columns are the first few non-prime numbers and the prime numbers, respectively, and the rows correspond to the instances in order (1st , 2nd, 3rd, ….).

We have just done a little of what is known as **data wrangling**. While not as fun as data visualization, data wrangling is often a big part of any analysis project!

Now that we have the data into shape, we can unleash all the powers of numpy arrays, powers which pandas DataFrames will inherit and build upon!

For example, who's bigger overall, the primes or the non-primes?

[66]:
```
colarr.sum(0)
```

[66]:
```
array([39.  , 43.78])
```

The primes win! In `colarr.sum(0)`, the 0 means "the first (vertical) dimension", i.e., sum the values *across the rows* within each column. To sum along the second dimension, we do:

[67]:
```
colarr.sum(1)
```

[67]:
```
array([ 4.  ,  7.14, 16.5 , 19.13, 21.7 , 14.31])
```

So any numpy array knows how to add up the numbers in it by row or by column (see what happens if you leave off the dimension, like this `colarr.sum()`. The list of things that numpy arrays can do themselves is pretty impressive.

Check it out here.

(or paste this into your browser: https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html)

Complete the following exercise.

- How many methods does a numpy array have? [I think 71]

- Create a new 2-dimensional array, and show the use of two methods not used above (`prod` and `round` could be two simple ones, but no pressure):

[Use the cell below to show how to create and add the two arrays]

Hint: The symbol `?` can be used at the end of a method and that can help understand how to use the method, for example, `myarray.shape?`

[68]:
```
dir(twoa);
twodarr = np.array([myarr, newarr])
twodarr.max(axis = None)
```

[68]:
```
10.0
```

[69]:
```
twodarr.round(decimals = 0)
```

[69]:
```
array([[ 2.,  4.,  6.,  8.,  9., 10.],
       [ 8.,  9.,  6.,  7.,  9.,  7.]])
```

1.0.7 NumPy methods to create arrays

Often, we want to create a array that we know we're going to put values in later. For example, we might be planning on doing a computation that will result in 3 sets of 7 values, and we want be able to store them directly into an array. We can pre-make an array filled with zeros with `np.zeros(r, c)`.

[70]:
```
myzeros = np.zeros((7, 3))
```

[71]:
```
myzeros
```

[71]:
```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

Another handy method to create arrays is `ones`

[72]:
```
myones = np.ones((3,4))
```

[73]:
```
myones
```

[73]:
```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

We will encounter other `NumPy` methods in later tutorials. For the time being one last method that will turn out very handy when modelling data:

[74]:
```
myRandomNumArray = np.random.randn(10,1)
print(myRandomNumArray)
```

```
[[-0.31872981]
 [ 1.09571481]
 [ 0.3745555 ]
 [ 0.20080004]
 [ 0.79226134]
 [ 0.44206305]
 [ 0.45347135]
 [ 1.559471  ]
 [ 1.01436733]
 [ 0.2230868 ]]
```

The **numpy** submodule **random** contains a variety of methods to create arrays containing random numbers. Generating random numbers is helpful in many applications, for example, they can be used to create normally distributed noise, or data with normally distributed noise, etc.

Complete the following exercise.

- Create a new 1-dimensional array of uniformly-distributed random number:

[Use the cell below to show your code]

[75]:
```
qone = np.random.randn(1,6)
print(qone)
```

```
[[0.77329995 0.26438115 1.17279823 0.12795692 0.42116764 0.33433447]]
```

- Create a new 2-dimensional array of normally-distributed random number:

[Use the cell below to show your code]

[76]:
```
qtwo = np.random.randn(2,3)
print(qtwo)
```

```
[[-1.8019753  -0.24326913 -2.8728046 ]
 [-0.56343371  1.21549927  0.42334764]]
```

Let's now create a simple 1-D array and explore what happens when we add a number to the values and what happens when we multiply the numbers in the array:

[77]:
```
size  = 20
origArray = np.random.randn(size,1)
```

Let's look at the values in the array.

[78]:
```
print(origArray)
```

```
[[ 0.08286634]
 [-0.87741913]
 [ 0.12070416]
```

```
 [ 1.58799583]
 [-0.53860435]
 [ 0.42527486]
 [-0.26790419]
 [-0.59768107]
 [ 0.57101397]
 [ 0.54133089]
 [ 1.20672778]
 [-0.22848642]
 [-1.39402897]
 [ 0.1112637 ]
 [ 1.47326446]
 [-0.48154559]
 [-0.73290416]
 [-0.96962954]
 [ 0.16522422]
 [ 0.61820278]]
```

There seem to be a variety of numbers, some positive, some negative, as expected because **randn** is supposed to generate numbers centered at 0 (i.e., with mean 0) and standard deviation of 1.

Let's compute the standard deviation and mean of these numbers. Numpy provides to handy methods:

[79]:
```
mean = np.mean(origArray)
sd = np.std(origArray)
print(['The mean is:', mean])
print(['the STD is:', sd])
```

```
['The mean is:', 0.04078327871324032]
['the STD is:', 0.7922153439813938]
```

Well, okay, the mean is not quite close to 0, but perhaps close enough? The standard deviation seems pretty close to the expected value of 1.

Complete the following exercise.

- Create a new 1-dimensional array of 100 normally-distributed random numbers:

[Use the cell below to show your code]

[80]:
```
hundred = np.random.randn(100,1)
```

- What happens to the mean and standard deviation after increasing the size of the array? Are they closer of further from the expected values? Why?

[Use the cell below to show your code]

[81]:
```
mean = np.mean(hundred)
sd = np.std(hundred)
print(['The mean is: ', mean])
print(['The standard dev. is: ', sd])
```

```
['The mean is: ', -0.045359436221237025]
['The standard dev. is: ', 0.9734025338621172]
```

Yes, the mean and standard deviation are closer to the expected values. The more random numbers generated in the array the closer the values get to a normal distribution.

What happens if we add 5 to the array?

[82]:
```
x  = 5 + origArray
```

[83]:
```
print(x)
```

```
[[5.08286634]
 [4.12258087]
 [5.12070416]
 [6.58799583]
 [4.46139565]
 [5.42527486]
 [4.73209581]
 [4.40231893]
 [5.57101397]
 [5.54133089]
 [6.20672778]
 [4.77151358]
 [3.60597103]
 [5.1112637 ]
 [6.47326446]
 [4.51845441]
 [4.26709584]
 [4.03037046]
 [5.16522422]
 [5.61820278]]
```

It looks like the values shifted. But how much? It looks like they recentered at 5, the value we added. So we can perhaps assume that now the distribution of numbers is normally distributed but with a mean of 5. The standard deviation has not bee changed. It is still at 1, trust me for the moment and let try multiplying the numbers.

[84]:
```
x  = 2 * origArray
print(x)
```

```
[[ 0.16573269]
 [-1.75483826]
 [ 0.24140832]
 [ 3.17599167]
 [-1.0772087 ]
 [ 0.85054972]
 [-0.53580837]
 [-1.19536213]
 [ 1.14202793]
```

```
[ 1.08266179]
[ 2.41345555]
[-0.45697284]
[-2.78805795]
[ 0.2225274 ]
[ 2.94652892]
[-0.96309119]
[-1.46580832]
[-1.93925908]
[ 0.33044844]
[ 1.23640557]]
```

It looks like the values increased. There seem to be a larger variality, more bigger numbers, both negative and positive. So perhaps the STD is not at 1 anymore. Could it be at 2?

Complete the following exercise.

- Compute the mean, std and median of `x`:

[Use the cell below to show your code]

[85]:
```
x_mean = np.mean(x)
x_sd = np.std(x)
x_median = np.median(x)
print(x_mean, x_sd, x_median)
```

```
0.08156655742648064 1.5844306879627876 0.19413004285683166
```

- What are the mean, std and median of `x`? Why, what is going on here?

[Use the cell below to show answer in your own words]

The mean is 0.379, the standard deviation is 2.18, and the median is 0.299. The mean is more than the median meaning that the data is skewed to the right.

1.0.8 Summary

So in this tutorial we have shown how to organize and manipulate data using Python `numpy arrays`.

So those are the basics of numpy arrays. They:

- store values in rows and columns
- each dimension starts at index zero (like lists)
- can be accessed using
  - square brackets `[]` with row and column indexes separated by a comma
  - integer indexes (including negative "start from the end" indexes)
  - a colon `:` (or two if you want a step value other than 1)
- can have maths done to every element in one go
- can be added, subtracted, etc. from one another
- have superpowers! they can compute stuff along their rows and columns!

The operations that are available for these two data types will be the base for many things that you might need to do as a Data Scientist.

`Numpy arrays` Can host a variety of data types. Although this might be too much for now, below a table of all the data types an `array` can support:

Type	Description
bool	Boolean (True or False) stored as a bit (0, 1)
inti	Platform integer (normally either int32 or int64)
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2 ** 31 to 2 ** 31 -1)
int64	Integer (-2 ** 63 to 2 ** 63 -1)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 2 ** 32 – 1)
uint64	Unsigned integer (0 to 2 ** 64 – 1)
float16	Half precision float: sign bit, 5 bits exponent, and 10 bits mantissa
float32	Single precision float: sign bit, 8 bits exponent, and 23 bits mantissa
float64 or float	Double precision float: sign bit, 11 bits exponent, and 52 bits mantissa
complex64	Complex number, represented by two 32-bit floats (real and imaginary components)
complex128 or complex	Complex number, represented by two 64-bit floats (real and imaginary components)

Complete the following exercise.

- Generate an 1-D array of mean = 10 and std = 1.5:

[Use the cell below to show your code]

[86]:
```
value = np.random.normal(loc = 10, scale = 1.5, size = 10000)
value_mean = np.mean(value)
value_sd = np.std(value)
print('mean = ', value_mean)
print('sd = ', value_sd)
```

```
mean =  10.017866455884677
sd =  1.5029862490529473
```