10_NumPy

April 17, 2020

[1]:
```
height = [1.73, 1.68, 1.71, 1.89, 1.79]
```

[2]:
```
height
```

[2]:
```
[1.73, 1.68, 1.71, 1.89, 1.79]
```

[3]:
```
weight = [65.4, 59.2, 63.6, 88.4, 68.7]
```

[4]:
```
weight
```

[4]:
```
[65.4, 59.2, 63.6, 88.4, 68.7]
```

[5]:
```
weight / height ** 2
```

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

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-5-6a4c0c70e3b9> in <module>
----> 1 weight / height ** 2


TypeError: unsupported operand type(s) for ** or pow(): 'list' and 'int'
```

Solution: NumPy (short for Numerical Python): Provide much more efficient storage and data operations as the size grows. - Alternative to Python List: NumPy Array, - Calculations over entire arrays - Easy and Fast

[6]:
```
import numpy as np
```

[7]:
```
np.__version__
```

[7]:
```
'1.18.1'
```

[8]:
```
np_height = np.array(height)
```

[9]:
```
np_height
```

[9]:
```
array([1.73, 1.68, 1.71, 1.89, 1.79])
```

[10]:
```
type(np_height)
```

[10]:
```
numpy.ndarray
```

[11]:
```
np_weight = np.array(weight)
```

[12]:
```
np_weight
```

[12]:
```
array([65.4, 59.2, 63.6, 88.4, 68.7])
```

[13]:
```
bmi = np_weight / np_height ** 2
```

[14]:
```
bmi
```

[14]:
```
array([21.85171573, 20.97505669, 21.75028214, 24.7473475 , 21.44127836])
```

We'll cover a few categories of basic array manipulations here:

- *Attributes of arrays*: Determining the size, shape, memory consumption, and data types of arrays
- *Indexing of arrays*: Getting and setting the value of individual array elements
- *Slicing of arrays*: Getting and setting smaller subarrays within a larger array
- *Reshaping of arrays*: Changing the shape of a given array
- *Joining and splitting of arrays*: Combining multiple arrays into one, and splitting one array into many

0.1 NumPy Array Attributes

First let's discuss some useful array attributes. We'll start by defining three random arrays, a one-dimensional, two-dimensional, and three-dimensional array. We'll use NumPy's random number generator, which we will *seed* with a set value in order to ensure that the same random arrays are generated each time this code is run:

[15]:
```
np.random.seed(1)  # seed for reproducibility

x1 = np.random.randint(10, size=6)  # One-dimensional array
x2 = np.random.randint(10, size=(3, 4))  # Two-dimensional array
x3 = np.random.randint(10, size=(3, 4, 5))  # Three-dimensional array
```

[16]:
```
x1
```

[16]:
```
array([5, 8, 9, 5, 0, 0])
```

[17]:
```
x2 #3 rows, 4 columns
```

[17]:
```
array([[1, 7, 6, 9],
       [2, 4, 5, 2],
       [4, 2, 4, 7]])
```

[18]:
```
x3 #4 rows, 5 columns and 3 items
```

[18]:
```
array([[[7, 9, 1, 7, 0],
        [6, 9, 9, 7, 6],
        [9, 1, 0, 1, 8],
        [8, 3, 9, 8, 7]],

       [[3, 6, 5, 1, 9],
        [3, 4, 8, 1, 4],
        [0, 3, 9, 2, 0],
        [4, 9, 2, 7, 7]],

       [[9, 8, 6, 9, 3],
        [7, 7, 4, 5, 9],
        [3, 6, 8, 0, 2],
        [7, 7, 9, 7, 3]]])
```

Each array has attributes `ndim` (the number of dimensions), `shape` (the size of each dimension), and `size` (the total size of the array):

[19]:
```
print("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
```

```
x3 ndim:  3
x3 shape: (3, 4, 5)
x3 size:  60
```

Another useful attribute is the `dtype`, the data type of the array.

[20]:
```
print("dtype:", x3.dtype)
```

```
dtype: int32
```

Other attributes include `itemsize`, which lists the size (in bytes) of each array element, and `nbytes`, which lists the total size (in bytes) of the array:

[21]:
```
print("itemsize:", x3.itemsize, "bytes")
print("nbytes:", x3.nbytes, "bytes") # nbytes = itemsize * size
```

```
itemsize: 4 bytes
nbytes: 240 bytes
```

In general, we expect that `nbytes` is equal to `itemsize` times `size`.

0.2 Array Indexing: Accessing Single Elements

If you are familiar with Python's standard list indexing, indexing in NumPy will feel quite familiar. In a one-dimensional array, the i^{th} value (counting from zero) can be accessed by specifying the desired index in square brackets, just as with Python lists:

[22]:
```
x1
```

[22]:
```
array([5, 8, 9, 5, 0, 0])
```

[23]:
```
x1[0]
```

[23]:
```
5
```

[24]:
```
x1[4]
```

[24]:
```
0
```

To index from the end of the array, you can use negative indices:

[25]:
```
x1[-1] # refers the last element
```

[25]:
```
0
```

[26]:
```
x1[-2]
```

[26]:
```
0
```

Values can also be modified using any of the above index notation:

[27]:
```
x2
```

[27]:
```
array([[1, 7, 6, 9],
       [2, 4, 5, 2],
       [4, 2, 4, 7]])
```

[28]:
```
x2[1,2]
```

[28]:
```
5
```

[29]:
```
x2[0, 0] = 15
x2
```

[29]:
```
array([[15,  7,  6,  9],
       [ 2,  4,  5,  2],
       [ 4,  2,  4,  7]])
```

[30]:
```
x2[1,1] = 12
x2
```

[30]:
```
array([[15,  7,  6,  9],
       [ 2, 12,  5,  2],
       [ 4,  2,  4,  7]])
```

0.3 Array Slicing: Accessing Subarrays

Just as we can use square brackets to access individual array elements, we can also use them to access subarrays with the *slice* notation, marked by the colon (`:`) character. The NumPy slicing syntax follows that of the standard Python list; to access a slice of an array `x`, use this:

`x[start:stop:step]`

If any of these are unspecified, they default to the values `start=0`, `stop=`*`size of dimension`*, `step=1`. We'll take a look at accessing sub-arrays in one dimension and in multiple dimensions.

0.3.1 One-dimensional subarrays

[31]:
```
x = np.arange(10)
x
```

[31]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[32]:
```
x[:5]  # first five elements
```

[32]:
```
array([0, 1, 2, 3, 4])
```

[33]:
```
x[5:]  # elements after index 5
```

[33]:
```
array([5, 6, 7, 8, 9])
```

[34]:
```
x[4:7]  # middle sub-array
```

[34]:
```
array([4, 5, 6])
```

[35]:
```
x[::2]  # every other element
```

[35]:
```
array([0, 2, 4, 6, 8])
```

[36]:
```
x[1::2]  # every other element, starting at index 1
```

[36]:
```
array([1, 3, 5, 7, 9])
```

A potentially confusing case is when the `step` value is negative. In this case, the defaults for `start` and `stop` are swapped. This becomes a convenient way to reverse an array:

[37]:
```
x[::-1]  # all elements, reversed
```

[37]:
```
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

[38]:
```
x[5::-1]  # reversed every other from index 5
```

[38]:
```
array([5, 4, 3, 2, 1, 0])
```

0.3.2 Multi-dimensional subarrays

Multi-dimensional slices work in the same way, with multiple slices separated by commas. For example:

[39]:
```
x2
```

[39]:
```
array([[15,  7,  6,  9],
       [ 2, 12,  5,  2],
       [ 4,  2,  4,  7]])
```

[40]:
```
x2[1,:] # second row
```

[40]:
```
array([ 2, 12,  5,  2])
```

[41]:
```
x2[1:,1:3]
```

[41]:
```
array([[12,  5],
       [ 2,  4]])
```

[42]:
```
x2[:,1:3]
```

[42]:
```
array([[ 7,  6],
       [12,  5],
       [ 2,  4]])
```

[43]:
```
x2[0:2, :2]  # two rows, two columns
```

[43]:
```
array([[15,  7],
       [ 2, 12]])
```

[44]:
```
x2[:3, :]  # all rows, every other column
```

[44]:
```
array([[15,  7,  6,  9],
       [ 2, 12,  5,  2],
       [ 4,  2,  4,  7]])
```

[45]:
```
x2[:, 1::2]
```

[45]:
```
array([[ 7,  9],
       [12,  2],
       [ 2,  7]])
```

Finally, subarray dimensions can even be reversed together:

[46]:
```
x2
```

[46]:
```
array([[15,  7,  6,  9],
       [ 2, 12,  5,  2],
       [ 4,  2,  4,  7]])
```

[47]:
```
x2[::-1, ::-1] # all rows and columns reversed
```

[47]:
```
array([[ 7,  4,  2,  4],
       [ 2,  5, 12,  2],
       [ 9,  6,  7, 15]])
```

[48]:
```
x3
```

[48]:
```
array([[[7, 9, 1, 7, 0],
        [6, 9, 9, 7, 6],
        [9, 1, 0, 1, 8],
        [8, 3, 9, 8, 7]],

       [[3, 6, 5, 1, 9],
        [3, 4, 8, 1, 4],
        [0, 3, 9, 2, 0],
        [4, 9, 2, 7, 7]],

       [[9, 8, 6, 9, 3],
        [7, 7, 4, 5, 9],
        [3, 6, 8, 0, 2],
        [7, 7, 9, 7, 3]]])
```

[49]:
```
x3[0,0:2,:] # element no, row, column
```

[49]:
```
array([[7, 9, 1, 7, 0],
       [6, 9, 9, 7, 6]])
```

[50]:
```
x3[2,0:2,0::2]
```

[50]:
```
array([[9, 6, 3],
       [7, 4, 9]])
```

Accessing array rows and columns One commonly needed routine is accessing of single rows or columns of an array. This can be done by combining indexing and slicing, using an empty slice marked by a single colon (:):

[51]:
```
print(x2[:, 0])  # first column of x2
```

```
[15  2  4]
```

[52]:
```
print(x2[0, :])  # first row of x2
```

```
[15  7  6  9]
```

In the case of row access, the empty slice can be omitted for a more compact syntax:

[53]:
```
print(x2[0])  # equivalent to x2[0, :]
```

```
[15  7  6  9]
```

0.4 Reshaping of Arrays

Another useful type of operation is reshaping of arrays. The most flexible way of doing this is with the `reshape` method. For example, if you want to put the numbers 1 through 9 in a 3×3 grid, you can do the following:

[54]:
```
grid = np.arange(1, 10).reshape((3, 3)) #converting 1D array to 2D
print(grid)
```

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

Note that for this to work, the size of the initial array must match the size of the reshaped array.

Another common reshaping pattern is the conversion of a one-dimensional array into a two-dimensional row or column matrix. This can be done with the `reshape` method, or more easily done by making use of the `newaxis` keyword within a slice operation:

[55]:
```
x = np.array([1, 2, 3])
# row vector via reshape
x.reshape((1, 3))
```

[55]:
```
array([[1, 2, 3]])
```

[56]:
```
# row vector via newaxis
x[np.newaxis, :]
```

[56]:
```
array([[1, 2, 3]])
```

[57]:
```
x[:, np.newaxis] # reshaping on column
```

[57]:
```
array([[1],
       [2],
       [3]])
```

[58]:
```
# column vector via reshape
x.reshape((1, 3))
```

[58]:
```
array([[1, 2, 3]])
```

0.5 Array Concatenation and Splitting

All of the preceding routines worked on single arrays. It's also possible to combine multiple arrays into one, and to conversely split a single array into multiple arrays. We'll take a look at those operations here.

0.5.1 Concatenation of arrays

Concatenation, or joining of two arrays in NumPy, is primarily accomplished using the routines `np.concatenate`, `np.vstack`, and `np.hstack`. `np.concatenate` takes a tuple or list of arrays as its first argument, as we can see here:

[59]:
```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
np.concatenate([x, y])
```

[59]:
```
array([1, 2, 3, 3, 2, 1])
```

You can also concatenate more than two arrays at once:

[60]:
```
z = np.array([99, 99, 99])
print(np.concatenate([x, y, z]))
```

```
[ 1  2  3  3  2  1 99 99 99]
```

It can also be used for two-dimensional arrays:

[61]:
```
grid = np.array([[1, 2, 3],[4, 5, 6]])
grid
```

[61]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

[62]:
```
# concatenate along the first axis
np.concatenate([grid, grid])
```

[62]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [1, 2, 3],
       [4, 5, 6]])
```

[63]:
```
# concatenate along the second axis (zero-indexed)
np.concatenate([grid, grid], axis=1)
```

[63]:
```
array([[1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6]])
```

0.5.2 Splitting of arrays

The opposite of concatenation is splitting, which is implemented by the functions `np.split`, `np.hsplit`, and `np.vsplit`. For each of these, we can pass a list of indices giving the split points:

[64]:
```
x = np.array([1, 2, 3, 99, 99, 3, 2, 1])
x1, x2= np.split(x, [2])
print(x1, x2)
```

```
[1 2] [ 3 99 99  3  2  1]
```

[65]:
```
x = np.array([1, 2, 3, 99, 99, 3, 2, 1])
x1,x2, x3 = np.split(x, [2, 4])
print(x1, x2, x3)
```

```
[1 2] [ 3 99] [99  3  2  1]
```

Notice that N split-points, leads to $N + 1$ subarrays. The related functions `np.hsplit` and `np.vsplit` are similar: - Split an array into multiple sub-arrays vertically (row-wise). - Split an array into multiple sub-arrays horizontally (column-wise)

[66]:
```
grid = np.arange(16).reshape((4, 4))
grid
```

[66]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

[67]:
```
upper, lower = np.vsplit(grid, [2])
print(upper)
print(lower)
```

```
[[0 1 2 3]
 [4 5 6 7]]
[[ 8  9 10 11]
 [12 13 14 15]]
```

[68]:
```
left, right = np.hsplit(grid, [2])
print(left)
print(right)
```

```
[[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]
[[ 2  3]
 [ 6  7]
```

```
 [10 11]
 [14 15]]
```

[69]:
```
np.hsplit?
```