NumPy (Numerical Python)

- Efficient storage and manipulation of numerical arrays is absolutely fundamental to the process of doing data science.
- Numy is a specialized tool for handling such numerical arrays.
- NumPy arrays are like Python's built-in list type, but NumPy arrays provide much more efficient storage and data operations as the arrays grow larger in size.
- NumPy arrays form the core of nearly the entire ecosystem of data science tools in Python.
- Thus, learning NumPy to use it effectively will be valuable no matter what aspect of data science interests you.

Import Numpy

In [1]:
```
import numpy
```

But, by convention, most of the people in Python data science world will import numpy as `np` as an alias.

In [2]:
```
import numpy as np
```

Numpy Version

In [3]:
```
np.__version__
```

Out[3]:
```
'1.20.3'
```

Creating Arrays from Python Lists

In [4]:
```
# Integer array:
np.array([2, 4, 6, 8, 10])
```

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

Remember that, NumPy arrays contain the same type of data. If does not match, it will upcast if possible.

In [5]:
```
# Upcast to floating point array:
np.array([2.5, 4, 6.7, 8, 10])
```

Out[5]:
```
array([ 2.5,  4. ,  6.7,  8. , 10. ])
```

In [6]:
```
# Explicitly set the datatype using `dtype` keyword.
np.array([2, 4, 6, 8, 10], dtype = 'float32')
```

Out[6]:
```
array([ 2.,  4.,  6.,  8., 10.], dtype=float32)
```

Creating Arrays from Scratch

Especially for larger arrays, it is more efficient to create arrays from scratch using *routines* built into NumPy.

In [7]:
```
# Create a length-10 integer array filled with zeros
np.zeros(10, dtype=np.int16)
```

Out[7]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=int16)
```

In [8]:
```
# Create a length-10 floating point array filled with zeros
np.zeros(10, dtype=np.float64) # float64 is default size for float of this
```

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

In [9]:
```
# Create a 3x4 floating-point array filled with ones
np.ones((3, 4), dtype=np.int64) # int64 is default size for int of this sy
```

Out[9]:
```
array([[1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 1, 1, 1]])
```

In [10]:
```
# Create a 3x4 array filled with 2.1416
np.full((3, 4), 2.1416)
```

Out[10]:
```
array([[2.1416, 2.1416, 2.1416, 2.1416],
       [2.1416, 2.1416, 2.1416, 2.1416],
       [2.1416, 2.1416, 2.1416, 2.1416]])
```

In [11]:
```
# Create an array filled with a linear sequence
# using np.arange(start, end, step)
# (this is similar to the built-in range() function)
np.arange(0, 20, 2)
```

Out[11]:
```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

In [12]:
```
# Create an array of five values evenly spaced between 0 and 1
np.linspace(0, 1, 5)
```

Out[12]:
```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

In [13]:
```
# Create a 3x3 array of uniformly distributed
# random values between 0 and 1
np.random.random((3, 3))
```

Out[13]:
```
array([[0.69332175, 0.78430349, 0.5463991 ],
       [0.42231673, 0.27326955, 0.9838995 ],
       [0.34921557, 0.01309714, 0.50318551]])
```

In [14]:
```
# Create a 3x3 array of normally distributed random values
# with mean 0 and standard deviation 1
np.random.normal(0, 1, (3, 3))
```

Out[14]:
```
array([[-0.27156065,  0.17642552,  0.49530464],
       [ 0.61002561,  0.34116023, -0.48144036],
       [ 1.06474987, -2.11048442,  0.75108846]])
```

In [15]:
```
# Create a 3x3 array of random integers in the interval [0, 10)
np.random.randint(0, 10, (3, 3))
```

Out[15]:
```
array([[8, 6, 6],
       [5, 8, 1],
       [9, 5, 0]])
```

In [16]:
```
# Create a 5x5 identity matrix
np.eye(5)
```

Out[16]:
```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

In [17]:
```
# Create an uninitialized array of five integers
# The values will be whatever happens to already exist at that memory loca
np.empty(5)
```

Out[17]:
```
array([1., 1., 1., 1., 1.])
```

NumPy Array Attributes

In [18]:
```
# seed for reproducibility
# Ensure that the same random arrays are generated each time this code is 
np.random.seed(0)

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
```

In [19]:
```
x1 # One-dimensional array
```

Out[19]:
```
array([5, 0, 3, 3, 7, 9])
```

In [20]:
```
x2 # Two-dimensional array
```

Out[20]:
```
array([[3, 5, 2, 4],
       [7, 6, 8, 8],
       [1, 6, 7, 7]])
```

In [21]:
```
x3 # Three-dimensional array
```

Out[21]:
```
array([[[8, 1, 5, 9, 8],
        [9, 4, 3, 0, 3],
        [5, 0, 2, 3, 8],
        [1, 3, 3, 3, 7]],

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

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

In [22]:
```
# Example of NumPy array attributes
print("x3 ndim: ", x3.ndim) # ndim (the number of dimensions)
print("x3 shape:", x3.shape) # shape (the size of each dimension)
print("x3 size: ", x3.size) # size (the total size of the array)
print("x3 dtype:", x3.dtype) # data type of the array
print("x3 itemsize:", x3.itemsize, "bytes") # size (in bytes) of each arra
print("x3 nbytes:", x3.nbytes, "bytes") # total size (in bytes) of the arr
```

```
x3 ndim:  3
x3 shape: (3, 4, 5)
x3 size:  60
x3 dtype: int64
x3 itemsize: 8 bytes
x3 nbytes: 480 bytes
```

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

Array Indexing

In a one-dimensional array, the *ith* value (counting from zero) can be accessed by specifying the desired index in square brackets, just as with Python lists.

In [23]:
```
x1
```

Out[23]:
```
array([5, 0, 3, 3, 7, 9])
```

In [24]:
```
x1[0] # first element
```

Out[24]:
```
5
```

In [25]:
```
x1[-1] # Negative indxing, last element
```

Out[25]:
```
9
```

In a multi-dimensional array, items can be accessed using a comma-separated tuple of indices.

In [26]:
```
x2
```

Out[26]:
```
array([[3, 5, 2, 4],
       [7, 6, 8, 8],
       [1, 6, 7, 7]])
```

In [27]:
```
x2[0,0] # first row, first column
```

Out[27]:
```
3
```

In [28]:
```
x2[2, -1] # 3rd row, last index
```

Out[28]:
```
7
```

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

In [29]:
```
x2[0, 0] = 9
x2
```

Out[29]:
```
array([[9, 5, 2, 4],
       [7, 6, 8, 8],
       [1, 6, 7, 7]])
```

But remember that, unlike Python lists, NumPy arrays have a fixed type. This means, for example, that if you attempt to insert a floating-point value to an integer array, the value will be silently truncated. *Don't be caught unaware by this behavior!*

In [30]:
```
x1[0] = 2.1416  # this will be truncated!
x1
```

Out[30]:
```
array([2, 0, 3, 3, 7, 9])
```

Array Slicing

The NumPy slicing syntax follows that of the standard Python list.

`array_name[start:stop:step]`

If any of these are unspecified, they default to the values `start=0`, `stop=size of dimension`, `step=1`.

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

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

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

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

In [33]:
```
x[5:]  # elements from index 5
```

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

In [34]:
```
x[3:7]  # sub-array from index 3 to 7
```

Out[34]:
```
array([3, 4, 5, 6])
```

In [35]:
```
x[::2]  # starting from 0 and step is 2
```

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

In [36]:
```
x[1::2]  # starting from 1 and step is 2
```

Out[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.

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

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

In [38]:
```
x
```

Out[38]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [39]:
```
x[5::-2]  # reversed every other from index 5
```

Out[39]:
```
array([5, 3, 1])
```

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

In [40]:
```
x2
```

Out[40]:
```
array([[9, 5, 2, 4],
       [7, 6, 8, 8],
       [1, 6, 7, 7]])
```

In [41]:
```
x2[:2, :3]  # two rows, three columns
```

Out[41]:
```
array([[9, 5, 2],
       [7, 6, 8]])
```

In [42]:
```
x2[:3, ::2]  # all rows, every other column
```

Out[42]:
```
array([[9, 2],
       [7, 8],
       [1, 7]])
```

In [43]:
```
x2[::-1, ::-1] # reversed
```

Out[43]:
```
array([[7, 7, 6, 1],
       [8, 8, 6, 7],
       [4, 2, 5, 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 (:).

In [44]:
```
x2
```

Out[44]:
```
array([[9, 5, 2, 4],
       [7, 6, 8, 8],
       [1, 6, 7, 7]])
```

In [45]:
```
print(x2[:, 0])  # first column of x2
```

```
[9 7 1]
```

In [46]:
```
print(x2[0, :])  # first row of x2
```

```
[9 5 2 4]
```

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

In [47]:
```
print(x2[0])  # equivalent to x2[0, :]
```

```
[9 5 2 4]
```

Subarrays as no-copy views

NumPy array slicing differs from Python list slicing:

- in lists, slices will be copies, and
- in NumPy array slices return views.

In [48]:
```
print(x2)
```

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

In [49]:
```
# extract a 2×2 subarray from x2 array
x2_sub = x2[:2, :2]
print(x2_sub)
```

```
[[9 5]
 [7 6]]
```

In [50]:
```
# modify x2_sub subarray
x2_sub[0, 0] = 88
print(x2_sub)
```

```
[[88  5]
 [ 7  6]]
```

In [51]:
```
# original array x2 is also changed
print(x2)
```

```
[[88  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

This default behavior is useful when we work with large datasets, we can access and process pieces of these datasets without the need to copy the underlying data buffer.

Creating copies of arrays

- `copy()` method can be used to copy an array or subarray.
- In this case if we now modify this new array or subarray, the original array is not touched/changd.

In [52]:
```
x2 # original array
```

Out[52]:
```
array([[88,  5,  2,  4],
       [ 7,  6,  8,  8],
       [ 1,  6,  7,  7]])
```

In [53]:
```
# copy a subarray
x2_sub_copy = x2[:2, :2].copy()
print(x2_sub_copy)
```

```
[[88  5]
 [ 7  6]]
```

In [54]:
```
# now, modify the subarray
x2_sub_copy[0, 0] = 44
print(x2_sub_copy)
```

```
[[44  5]
 [ 7  6]]
```

In [55]:
```
# original array x2 is not touched
print(x2)
```

```
[[88  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

Reshaping of Arrays

- Using reshape() method
- Size of the initial array must match the size of the reshaped array

In [56]:
```
# 3×3
grid = np.arange(1, 10).reshape((3,3))
print(grid)
```

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

In [57]:
```
x = np.array([1, 2, 3])
x
```

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

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

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

In [59]:
```
# column vector via reshape
x.reshape((3, 1))
```

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

Array Concatenation

Concatenation, or joining of two arrays in NumPy, is primarily accomplished using `np.concatenate`, `np.vstack`, and `np.hstack`.

In [60]:
```
# Using concatenate() method
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
np.concatenate([x, y])
```

Out[60]:
```
array([1, 2, 3, 3, 2, 1])
```

In [61]:
```
# More than two arrays at once
z = [99, 99, 99]
np.concatenate([x, y, z])
```

Out[61]:
```
array([ 1,  2,  3,  3,  2,  1, 99, 99, 99])
```

In [62]:
```
# concatenate two dimentional array
grid = np.array([[1, 2, 3],
                 [4, 5, 6]])
# concatenate along the first axis
np.concatenate([grid, grid])
```

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

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

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

For working with arrays of mixed dimensions, it can be clearer to use the `np.vstack` (vertical stack) and `np.hstack` (horizontal stack) functions.

In [64]:
```
x = np.array([1, 2, 3])
grid = np.array([[9, 8, 7],
                 [6, 5, 4]])

# vertically stack the arrays
np.vstack([x, grid])
```

Out[64]:
```
array([[1, 2, 3],
       [9, 8, 7],
       [6, 5, 4]])
```

In [65]:
```
# horizontally stack the arrays
y = np.array([[99],
              [99]])
np.hstack([grid, y])
```

Out[65]:
```
array([[ 9,  8,  7, 99],
       [ 6,  5,  4, 99]])
```

Array Splitting

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

In [66]:
```
x = [1, 2, 3, 99, 99, 3, 2, 1]
x1, x2, x3 = np.split(x, [3, 5])
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.

In [67]:
```
grid = np.arange(16).reshape((4, 4))
grid
```

Out[67]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

In [68]:
```
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]]
```

In [69]:
```
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]]
```

Iterating Over NumPy Array

NumPy contains an iterator object `numpy.nditer`, which is an efficient multidimensional iterator object to iterate over an array.

In [78]:
```
arr = np.arange(1, 10).reshape((3,3))
arr
```

Out[78]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [81]:
```
# iterating the array, arr, using numpy.nditer()
for i in np.nditer(arr):
    print(i)
```

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

In [ ]: