03_Numpy_Notebook

February 29, 2020

Introduction to numpy:

Package for scientific computing with Python

Numerical Python, or "Numpy" for short, is a foundational package on which many of the most common data science packages are built. Numpy provides us with high performance multi-dimensional arrays which we can use as vectors or matrices.

The key features of numpy are:

- ndarrays: n-dimensional arrays of the same data type which are fast and space-efficient. There are a number of built-in methods for ndarrays which allow for rapid processing of data without using loops (e.g., compute the mean).
- Broadcasting: a useful tool which defines implicit behavior between multi-dimensional arrays of different sizes.
- Vectorization: enables numeric operations on ndarrays.
- Input/Output: simplifies reading and writing of data from/to file.

Additional Recommended Resources: Numpy Documentation Python for Data Analysis by Wes McKinney Python Data science Handbook by Jake VanderPlas

Getting started with ndarray

ndarrays are time and space-efficient multidimensional arrays at the core of numpy. Like the data structures in Week 2, let's get started by creating ndarrays using the numpy package.

How to create Rank 1 numpy arrays:

[1]:
```
import numpy as np

an_array = np.array([3, 33, 333])  # Create a rank 1 array

print(type(an_array))              # The type of an ndarray is: "<class 'numpy.
 ↪ndarray'>"
```

```
<class 'numpy.ndarray'>
```

[2]:
```
# test the shape of the array we just created, it should have just one dimension␣
 ↪(Rank 1)
print(an_array.shape)
```

```
(3,)
```

```
[3]: # because this is a 1-rank array, we need only one index to accesss each element
print(an_array[0], an_array[1], an_array[2])
```

```
3 33 333
```

```
[4]: an_array[0] =888            # ndarrays are mutable, here we change an element of␣
 ↪the array

print(an_array)
```

```
[888  33 333]
```

How to create a Rank 2 numpy array:

A rank 2 **ndarray** is one with two dimensions. Notice the format below of [[row] , [row]]. 2 dimensional arrays are great for representing matrices which are often useful in data science.

```
[5]: another = np.array([[11,12,13],[21,22,23]])   # Create a rank 2 array

print(another)  # print the array

print("The shape is 2 rows, 3 columns: ", another.shape)  # rows x columns      ␣
 ↪

print("Accessing elements [0,0], [0,1], and [1,0] of the ndarray: ", another[0,␣
 ↪0], ", ",another[0, 1],", ", another[1, 0])
```

```
[[11 12 13]
 [21 22 23]]
The shape is 2 rows, 3 columns:  (2, 3)
Accessing elements [0,0], [0,1], and [1,0] of the ndarray:  11 ,  12 ,  21
```

There are many way to create numpy arrays:

Here we create a number of different size arrays with different shapes and different pre-filled values. numpy has a number of built in methods which help us quickly and easily create multidimensional arrays.

```
[6]: import numpy as np

# create a 2x2 array of zeros
ex1 = np.zeros((2,2))
print(ex1)
```

```
[[0. 0.]
 [0. 0.]]
```

```
[7]: # create a 2x2 array filled with 9.0
ex2 = np.full((2,2), 9.0)
print(ex2)
```

```
[[9. 9.]
 [9. 9.]]
```

[8]:
```
# create a 2x2 matrix with the diagonal 1s and the others 0
ex3 = np.eye(2,2)
print(ex3)
```

```
[[1. 0.]
 [0. 1.]]
```

[9]:
```
# create an array of ones
ex4 = np.ones((1,2))
print(ex4)
```

```
[[1. 1.]]
```

[10]:
```
# notice that the above ndarray (ex4) is actually rank 2, it is a 2x1 array
print(ex4.shape)

# which means we need to use two indexes to access an element
print()
print(ex4[0,1])
```

```
(1, 2)

1.0
```

[11]:
```
# create an array of random floats between 0 and 1
ex5 = np.random.random((2,2))
print(ex5)
```

```
[[0.50609804 0.39284153]
 [0.9443509  0.78096497]]
```

Array Indexing

Slice indexing:

Similar to the use of slice indexing with lists and strings, we can use slice indexing to pull out sub-regions of ndarrays.

[12]:
```
import numpy as np

# Rank 2 array of shape (3, 4)
an_array = np.array([[11,12,13,14], [21,22,23,24], [31,32,33,34]])
print(an_array)
```

```
[[11 12 13 14]
 [21 22 23 24]
 [31 32 33 34]]
```

Use array slicing to get a subarray consisting of the first 2 rows x 2 columns.

[13]:
```
a_slice = an_array[:2, 1:3]
print(a_slice)
```

```
[[12 13]
 [22 23]]
```

When you modify a slice, you actually modify the underlying array.

[14]:
```
print("Before:", an_array[0, 1])   #inspect the element at 0, 1
a_slice[0, 0] = 1000    # a_slice[0, 0] is the same piece of data as an_array[0,␣
↪1]
print("After:", an_array[0, 1])
```

```
Before: 12
After: 1000
```

Use both integer indexing & slice indexing

We can use combinations of integer indexing and slice indexing to create different shaped matrices.

[15]:
```
# Create a Rank 2 array of shape (3, 4)
an_array = np.array([[11,12,13,14], [21,22,23,24], [31,32,33,34]])
print(an_array)
```

```
[[11 12 13 14]
 [21 22 23 24]
 [31 32 33 34]]
```

[16]:
```
# Using both integer indexing & slicing generates an array of lower rank
row_rank1 = an_array[1, :]     # Rank 1 view

print(row_rank1, row_rank1.shape)  # notice only a single []
```

```
[21 22 23 24] (4,)
```

[17]:
```
# Slicing alone: generates an array of the same rank as the an_array
row_rank2 = an_array[1:2, :]  # Rank 2 view

print(row_rank2, row_rank2.shape)   # Notice the [[ ]]
```

```
[[21 22 23 24]] (1, 4)
```

[18]:
```
#We can do the same thing for columns of an array:

print()
col_rank1 = an_array[:, 1]
col_rank2 = an_array[:, 1:2]
```

```
print(col_rank1, col_rank1.shape)  # Rank 1
print()
print(col_rank2, col_rank2.shape)  # Rank 2
```

```
[12 22 32] (3,)

[[12]
 [22]
 [32]] (3, 1)
```

Array Indexing for changing elements:

Sometimes it's useful to use an array of indexes to access or change elements.

[19]:
```
# Create a new array
an_array = np.array([[11,12,13], [21,22,23], [31,32,33], [41,42,43]])

print('Original Array:')
print(an_array)
```

```
Original Array:
[[11 12 13]
 [21 22 23]
 [31 32 33]
 [41 42 43]]
```

[20]:
```
# Create an array of indices
col_indices = np.array([0, 1, 2, 0])
print('\nCol indices picked : ', col_indices)

row_indices = np.arange(4)
print('\nRows indices picked : ', row_indices)
```

```

Col indices picked :  [0 1 2 0]

Rows indices picked :  [0 1 2 3]
```

[21]:
```
# Examine the pairings of row_indices and col_indices.  These are the elements␣
↪we'll change next.
for row,col in zip(row_indices,col_indices):
    print(row, ", ",col)
```

```
0 ,  0
1 ,  1
2 ,  2
3 ,  0
```

[22]:
```
# Select one element from each row
print('Values in the array at those indices: ',an_array[row_indices,␣
↪col_indices])
```

```
Values in the array at those indices:  [11 22 33 41]
```

[23]:
```
# Change one element from each row using the indices selected
an_array[row_indices, col_indices] += 100000

print('\nChanged Array:')
print(an_array)
```

```

Changed Array:
[[100011     12     13]
 [    21 100022     23]
 [    31     32 100033]
 [100041     42     43]]
```

Boolean Indexing

Array Indexing for changing elements:

[24]:
```
# create a 3x2 array
an_array = np.array([[11,12], [21, 22], [31, 32]])
print(an_array)
```

```
[[11 12]
 [21 22]
 [31 32]]
```

[25]:
```
# create a filter which will be boolean values for whether each element meets␣
↪this condition
filter = (an_array > 15)
filter
```

[25]:
```
array([[False, False],
       [ True,  True],
       [ True,  True]])
```

Notice that the filter is a same size ndarray as an_array which is filled with True for each element whose corresponding element in an_array which is greater than 15 and False for those elements whose value is less than 15.

[26]:
```
# we can now select just those elements which meet that criteria
print(an_array[filter])
```

```
[21 22 31 32]
```

[27]:
```
# For short, we could have just used the approach below without the need for the
↪separate filter array.

an_array[an_array > 15]
```

[27]:
```
array([21, 22, 31, 32])
```

What is particularly useful is that we can actually change elements in the array applying a similar logical filter. Let's add 100 to all the even values.

[28]:
```
an_array[an_array % 2 == 0] +=100
print(an_array)
```

```
[[ 11 112]
 [ 21 122]
 [ 31 132]]
```

Datatypes and Array Operations

Datatypes:

[29]:
```
ex1 = np.array([11, 12]) # Python assigns the  data type
print(ex1.dtype)
```

```
int32
```

[30]:
```
ex2 = np.array([11.0, 12.0]) # Python assigns the  data type
print(ex2.dtype)
```

```
float64
```

[31]:
```
ex3 = np.array([11, 21], dtype=np.int64) #You can also tell Python the  data type
print(ex3.dtype)
```

```
int64
```

[32]:
```
# you can use this to force floats into integers (using floor function)
ex4 = np.array([11.1,12.7], dtype=np.int64)
print(ex4.dtype)
print()
print(ex4)
```

```
int64

[11 12]
```

[33]:
```
# you can use this to force integers into floats if you anticipate
# the values may change to floats later
ex5 = np.array([11, 21], dtype=np.float64)
```

```
print(ex5.dtype)
print()
print(ex5)
```

```
float64

[11. 21.]
```

Arithmetic Array Operations:

[34]:
```
x = np.array([[111,112],[121,122]], dtype=np.int)
y = np.array([[211.1,212.1],[221.1,222.1]], dtype=np.float64)

print(x)
print()
print(y)
```

```
[[111 112]
 [121 122]]

[[211.1 212.1]
 [221.1 222.1]]
```

[35]:
```
# add
print(x + y)          # The plus sign works
print()
print(np.add(x, y))  # so does the numpy function "add"
```

```
[[322.1 324.1]
 [342.1 344.1]]

[[322.1 324.1]
 [342.1 344.1]]
```

[36]:
```
# subtract
print(x - y)
print()
print(np.subtract(x, y))
```

```
[[-100.1 -100.1]
 [-100.1 -100.1]]

[[-100.1 -100.1]
 [-100.1 -100.1]]
```

[37]:
```
# multiply
print(x * y)
print()
```

```
print(np.multiply(x, y))
```

```
[[23432.1 23755.2]
 [26753.1 27096.2]]

[[23432.1 23755.2]
 [26753.1 27096.2]]
```

[38]:
```
# divide
print(x / y)
print()
print(np.divide(x, y))
```

```
[[0.52581715 0.52805281]
 [0.54726368 0.54930212]]

[[0.52581715 0.52805281]
 [0.54726368 0.54930212]]
```

[39]:
```
# square root
print(np.sqrt(x))
```

```
[[10.53565375 10.58300524]
 [11.         11.04536102]]
```

[40]:
```
# exponent (e ** x)
print(np.exp(x))
```

```
[[1.60948707e+48 4.37503945e+48]
 [3.54513118e+52 9.63666567e+52]]
```

Statistical Methods, Sorting, and Set Operations:

Basic Statistical Operations:

[41]:
```
# setup a random 2 x 4 matrix
arr = 10 * np.random.randn(2,5)
print(arr)
```

```
[[ -7.49447723  -8.04204722 -14.39816862  10.30620247  -0.91667579]
 [  3.93942862 -10.35762464  20.01316616  -3.35620275  -0.13719788]]
```

[42]:
```
# compute the mean for all elements
print(arr.mean())
```

```
-1.044359687791173
```

[43]:
```
# compute the means by row
print(arr.mean(axis = 1))
```

```
[-4.10903328  2.0203139 ]
```

[44]:
```python
# compute the means by column
print(arr.mean(axis = 0))
```

```
[-1.77752431 -9.19983593  2.80749877  3.47499986 -0.52693683]
```

[45]:
```python
# sum all the elements
print(arr.sum())
```

```
-10.44359687791173
```

[46]:
```python
# compute the medians
print(np.median(arr, axis = 1))
```

```
[-7.49447723 -0.13719788]
```

Sorting:

[47]:
```python
# create a 10 element array of randoms
unsorted = np.random.randn(10)

print(unsorted)
```

```
[-0.76495927  0.69111965  1.05181618  1.99343156 -0.99541698 -0.22206029
  0.51861726  1.67135647 -1.43911626 -0.74007184]
```

[48]:
```python
# create copy and sort
sorted = np.array(unsorted)
sorted.sort()

print(sorted)
print()
print(unsorted)
```

```
[-1.43911626 -0.99541698 -0.76495927 -0.74007184 -0.22206029  0.51861726
  0.69111965  1.05181618  1.67135647  1.99343156]

[-0.76495927  0.69111965  1.05181618  1.99343156 -0.99541698 -0.22206029
  0.51861726  1.67135647 -1.43911626 -0.74007184]
```

[49]:
```python
# inplace sorting
unsorted.sort()

print(unsorted)
```

```
[-1.43911626 -0.99541698 -0.76495927 -0.74007184 -0.22206029  0.51861726
  0.69111965  1.05181618  1.67135647  1.99343156]
```

Finding Unique elements:

[50]:
```
array = np.array([1,2,1,4,2,1,4,2])

print(np.unique(array))
```

```
[1 2 4]
```

Set Operations with np.array data type:

[51]:
```
s1 = np.array(['desk','chair','bulb'])
s2 = np.array(['lamp','bulb','chair'])
print(s1, s2)
```

```
['desk' 'chair' 'bulb'] ['lamp' 'bulb' 'chair']
```

[52]:
```
print( np.intersect1d(s1, s2) )
```

```
['bulb' 'chair']
```

[53]:
```
print( np.union1d(s1, s2) )
```

```
['bulb' 'chair' 'desk' 'lamp']
```

[54]:
```
print( np.setdiff1d(s1, s2) )# elements in s1 that are not in s2
```

```
['desk']
```

[55]:
```
print( np.in1d(s1, s2) )#which element of s1 is also in s2
```

```
[False  True  True]
```

Broadcasting:

Introduction to broadcasting. For more details, please see: https://docs.scipy.org/doc/numpy-1.10.1/user/basics.broadcasting.html

[56]:
```
import numpy as np

start = np.zeros((4,3))
print(start)
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

[57]:
```
# create a rank 1 ndarray with 3 values
add_rows = np.array([1, 0, 2])
print(add_rows)
```

```
[1 0 2]
```

[58]:
```
y = start + add_rows  # add to each row of 'start' using broadcasting
print(y)
```

```
[[1. 0. 2.]
 [1. 0. 2.]
 [1. 0. 2.]
 [1. 0. 2.]]
```

[59]:
```
# create an ndarray which is 4 x 1 to broadcast across columns
add_cols = np.array([[0,1,2,3]])
add_cols = add_cols.T

print(add_cols)
```

```
[[0]
 [1]
 [2]
 [3]]
```

[60]:
```
# add to each column of 'start' using broadcasting
y = start + add_cols
print(y)
```

```
[[0. 0. 0.]
 [1. 1. 1.]
 [2. 2. 2.]
 [3. 3. 3.]]
```

[61]:
```
# this will just broadcast in both dimensions
add_scalar = np.array([1])
print(start+add_scalar)
```

```
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

Example from the slides:

[62]:
```
# create our 3x4 matrix
arrA = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
print(arrA)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

[63]:
```
# create our 4x1 array
arrB = [0,1,0,2]
print(arrB)
```

```
[0, 1, 0, 2]
```

[64]:
```
# add the two together using broadcasting
print(arrA + arrB)
```

```
[[ 1  3  3  6]
 [ 5  7  7 10]
 [ 9 11 11 14]]
```

Speedtest: ndarrays vs lists

First setup paramaters for the speed test. We'll be testing time to sum elements in an ndarray versus a list.

[65]:
```
from numpy import arange
from timeit import Timer

size    = 1000000
timeits = 1000
```

[66]:
```
# create the ndarray with values 0,1,2...,size-1
nd_array = arange(size)
print( type(nd_array) )
```

```
<class 'numpy.ndarray'>
```

[67]:
```
# timer expects the operation as a parameter,
# here we pass nd_array.sum()
timer_numpy = Timer("nd_array.sum()", "from __main__ import nd_array")

print("Time taken by numpy ndarray: %f seconds" %
      (timer_numpy.timeit(timeits)/timeits))
```

```
Time taken by numpy ndarray: 0.000869 seconds
```

[68]:
```
# create the list with values 0,1,2...,size-1
a_list = list(range(size))
print (type(a_list) )
```

```
<class 'list'>
```

[ ]:
```
# timer expects the operation as a parameter, here we pass sum(a_list)
timer_list = Timer("sum(a_list)", "from __main__ import a_list")

print("Time taken by list:  %f seconds" %
```

```
    (timer_list.timeit(timeits)/timeits))
```

Read or Write to Disk:

Binary Format:

```
[ ]: x = np.array([ 23.23, 24.24] )
```

```
[ ]: np.save('an_array', x)
```

```
[ ]: np.load('an_array.npy')
```

Text Format:

```
[ ]: np.savetxt('array.txt', X=x, delimiter=',')
```

```
[ ]: !cat array.txt
```

```
[ ]: np.loadtxt('array.txt', delimiter=',')
```

Additional Common ndarray Operations

Dot Product on Matrices and Inner Product on Vectors:

```
[ ]: # determine the dot product of two matrices
     x2d = np.array([[1,1],[1,1]])
     y2d = np.array([[2,2],[2,2]])

     print(x2d.dot(y2d))
     print()
     print(np.dot(x2d, y2d))
```

```
[ ]: # determine the inner product of two vectors
     a1d = np.array([9 , 9 ])
     b1d = np.array([10, 10])

     print(a1d.dot(b1d))
     print()
     print(np.dot(a1d, b1d))
```

```
[ ]: # dot produce on an array and vector
     print(x2d.dot(a1d))
     print()
     print(np.dot(x2d, a1d))
```

Sum:

```
[ ]: # sum elements in the array
     ex1 = np.array([[11,12],[21,22]])
```

```
print(np.sum(ex1))           # add all members
```

[ ]:
```
print(np.sum(ex1, axis=0))   # columnwise sum
```

[ ]:
```
print(np.sum(ex1, axis=1))   # rowwise sum
```

Element-wise Functions:

For example, let's compare two arrays values to get the maximum of each.

[ ]:
```
# random array
x = np.random.randn(8)
x
```

[ ]:
```
# another random array
y = np.random.randn(8)
y
```

[ ]:
```
# returns element wise maximum between two arrays

np.maximum(x, y)
```

Reshaping array:

[ ]:
```
# grab values from 0 through 19 in an array
arr = np.arange(20)
print(arr)
```

[ ]:
```
# reshape to be a 4 x 5 matrix
arr.reshape(4,5)
```

Transpose:

[ ]:
```
# transpose
ex1 = np.array([[11,12],[21,22]])

ex1.T
```

Indexing using where():

[ ]:
```
x_1 = np.array([1,2,3,4,5])

y_1 = np.array([11,22,33,44,55])

filter = np.array([True, False, True, False, True])
```

[ ]:
```
out = np.where(filter, x_1, y_1)
print(out)
```

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

[ ]:
```
np.where( mat > 0.5, 1000, -1)
```

"any" or "all" conditionals:

[ ]:
```
arr_bools = np.array([ True, False, True, True, False ])
```

[ ]:
```
arr_bools.any()
```

[ ]:
```
arr_bools.all()
```

Random Number Generation:

[ ]:
```
Y = np.random.normal(size = (1,5))[0]
print(Y)
```

[ ]:
```
Z = np.random.randint(low=2,high=50,size=4)
print(Z)
```

[ ]:
```
np.random.permutation(Z) #return a new ordering of elements in Z
```

[ ]:
```
np.random.uniform(size=4) #uniform distribution
```

[ ]:
```
np.random.normal(size=4) #normal distribution
```

Merging data sets:

[ ]:
```
K = np.random.randint(low=2,high=50,size=(2,2))
print(K)

print()
M = np.random.randint(low=2,high=50,size=(2,2))
print(M)
```

[ ]:
```
np.vstack((K,M))
```

[ ]:
```
np.hstack((K,M))
```

[ ]:
```
np.concatenate([K, M], axis = 0)
```

[ ]:
```
np.concatenate([K, M.T], axis = 1)
```