CS 2302 Data Structures

Arrays in Python

Author: Olac Fuentes

Last modified: 12/21/2022

Arrays

Arrays are not part of the language. We will use them through the numpy library. Arrays are similar to lists, but all elements must be of the same type.

The numpy library provides implementations of many useful operations on arrays of any dimensionality.

The following statement imports the library and declares np as short for numpy. We can access functions and modules in the numpy library by using the dot notation.

```
import numpy as np
```

Arrays can be created in several ways.

We can convert a list to a 1-D array.

```
a = np.array([1,2,3,4])
print(a)
```

```
[1 2 3 4]
```

We can convert a list of lists to a 2-D array (or a list of lists of lists to a 3D array, and so on).

```
a = np.array([[1,2,3,4],[5,6,7,8]])
print(a)
```

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

Unlike Java, numpy does not allow jagged arrays.

```
a = np.array([[1,2,3,4],[5,6,7]])
print(a)
```

```
[list([1, 2, 3, 4]) list([5, 6, 7])]
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: VisibleDeprecationWarning: Creating an ndarray from ragged ne
  """Entry point for launching an IPython kernel.
```

The array type is inferred from the data provided.

```
a = np.array([5,6,7,8])
print(a)
print(type(a))
print(type(a[0]))
```

```
[5 6 7 8]
<class 'numpy.ndarray'>
<class 'numpy.int64'>
```

```
a = np.array([5,6,7.,8])
print(a)
print(type(a))
```

```
print(type(a[0]))
```

```
[5. 6. 7. 8.]
<class 'numpy.ndarray'>
<class 'numpy.float64'>
```

The *tuple* (an immutable list) **shape** contains the size of the array along each of its dimensions.

len(a.shape) contains the number of dimension in array a.

For a 2D array, shape[0] is the number of rows, shape[1] is the number of columns.

len(a) is the same as a.shape[0]

```
a = np.array([5,6,7.,8])
print(a)
print(a.shape)
print(len(a.shape))
```

```
[5. 6. 7. 8.]
(4,)
1
```

```
a = np.array([[1,2,3,4],[5,6,7,8]])
print(a)
print(len(a.shape))
print(a.shape)
print(len(a))
```

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

```
a = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
print(a)
print(len(a.shape))
print(a.shape)
print(len(a))
```

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

```
a = np.array([])
print(a)
print(len(a.shape))
print(a.shape)
print(len(a))
```

```
[]
1
(0,)
0
```

We can have a 2D array with a single row (or column).

```
a = np.array([[1,2,3,4]])
print(a)
print(len(a.shape))
print(a.shape)
print(len(a))
```

```
[[1 2 3 4]]
2
(1, 4)
1
```

```
a = np.array([[1],[2],[3],[4]])
print(a)
print(len(a.shape))
print(a.shape)
print(len(a))
```

```
[[1]
 [2]
 [3]
 [4]]
2
(4, 1)
4
```

```
print(a)
print(a.T)
a = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
print(a)
print(a.T)
```

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

The function zeros(s) creates an array of shape s where all elements are zero.

Similarly, ones(s) creates an array of shape s where all elements are one.

By default, the dtype of the array is float64.

```
a = np.zeros(5) # Create 1-D array
print(a)
print(len(a.shape))
print(a.shape)
print(len(a))
print(type(a[0]))
```

```
[0. 0. 0. 0. 0.]
1
(5,)
5
<class 'numpy.float64'>
```

```
a = np.ones((5,3)) # Create 2-D array with 5 rows and 3 columnns
print(a)
print(len(a.shape))
print(a.shape)
print(len(a))
print(type(a[0,0]))
```

```
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
2
(5, 3)
5
<class 'numpy.float64'>
```

```
a = np.zeros((5,3,2)) # Create 3-D array
print(a)
print(len(a.shape))
print(a.shape)
print(len(a))
print(type(a[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.]
  [0. 0.]]

 [[0. 0.]
  [0. 0.]
  [0. 0.]]]
3
(5, 3, 2)
5
<class 'numpy.float64'>
```

You can also specify the type of the elements in the array.

```
a = np.ones((2,3),dtype=np.int16) # Create 2-D array of integers with 2 rows and 3 columnns
print(a)
print(len(a.shape))
print(a.shape)
print(type(a[0,0]))
```

```
[[1 1 1]
 [1 1 1]]
2
(2, 3)
<class 'numpy.int16'>
```

We can also create arrays of integers using the numpy **arange** function, which is analogous to the Python built-in range.

```
a= np.arange(16)
print(a)
```

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

```
a= np.arange(20,100,10)
print(a)
```

```
[20 30 40 50 60 70 80 90]
```

The reshape operation can be used to change the dimensionality of an array (but the number of elements cannot change).

```
a= np.arange(20)
print(a)
b = a.reshape(4,5)
print(b)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]]
```

```
a= np.arange(20)
print(a)
a = a.reshape(4,5)
print(a)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]]
```

```
a= np.arange(16)
b = a.reshape(4,5)
print(b)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-21-c4bf174c9b86> in <module>
      1 a= np.arange(16)
----> 2 b = a.reshape(4,5)
      3 print(b)

ValueError: cannot reshape array of size 16 into shape (4,5)
```

SEARCH STACK OVERFLOW

```
a= np.arange(24)
b = a.reshape(2,4,3)
print(b)
print(a)
```

```
[[[ 0  1  2]
  [ 3  4  5]
  [ 6  7  8]
  [ 9 10 11]]

 [[12 13 14]
  [15 16 17]
  [18 19 20]
  [21 22 23]]]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
```

```
a= np.arange(24)
a.reshape(6,4)
print(a)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
```

If we want to reshape an array to n dimensions, we only need to specify n-1 of the dimensions, since the other can be inferrered from the size of the original array (the product of the elements in shape in the original and reshaped arrays must be the same). We pass a value of -1 corresponding to the unknown dimension.

```
a= np.arange(24)
print(a.reshape(-1,6))
```

```
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]]
```

```
print(a.reshape(5,-1))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-25-1e73aa8568ab> in <module>
----> 1 print(a.reshape(5,-1))

ValueError: cannot reshape array of size 24 into shape (5,newaxis)
```

SEARCH STACK OVERFLOW

```
a= np.arange(24)
print(a)
a = a.reshape(6,4)
print(a)
a = a.reshape(-1)
print(a)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
```

```
print(a.reshape(-1,12))
```

```
[[ 0  1  2  3  4  5  6  7  8  9 10 11]
 [12 13 14 15 16 17 18 19 20 21 22 23]]
```

```
print(a.reshape(4,-1,3))
```

```
[[[ 0  1  2]
  [ 3  4  5]]

 [[ 6  7  8]
  [ 9 10 11]]

 [[12 13 14]
  [15 16 17]]

 [[18 19 20]
  [21 22 23]]]
```

```
print(a.reshape(-1,4))
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
```

```
print(a.reshape(-1,6))
```

```
print(a.reshape(-1,8))
```

```
print(a.reshape(-1,24))
```

```
print(a.reshape(-1,5))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-33-a3842817d650> in <module>
----> 1 print(a.reshape(-1,5))

ValueError: cannot reshape array of size 24 into shape (5)
```

SEARCH STACK OVERFLOW

Important: reshape returns a **shallow** copy of the array, without modifying the original array.

```
a= np.arange(16)
a.reshape(4,4)
print(a)
```

```
a = np.arange(16)
b = a.reshape(4,4)
print(a)
print(b)
print(a.shape)
print(b.shape)
a[0] = 9
print(a)
print(b)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
(16,)
(4, 4)
[ 9  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
[[ 9  1  2  3]
 [ 4  5  6  7]
```

```
       [ 8  9 10 11]
       [12 13 14 15]]
```

```
a= np.arange(16)
b = np.copy(a.reshape(4,4))
print(a)
print(b)
a[0] = 9
print(a)
print(b)
```

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

Arithmetic operators on arrays apply elementwise. Usually, there's no need to write *for* loops to perform array operations!!!

```
a= np.arange(16).reshape(4,4)
print(a)
b = a + 5
print(b)
print(a)
```

```
     [[ 0  1  2  3]
      [ 4  5  6  7]
      [ 8  9 10 11]
      [12 13 14 15]]
     [[ 5  6  7  8]
      [ 9 10 11 12]
      [13 14 15 16]
      [17 18 19 20]]
     [[ 0  1  2  3]
      [ 4  5  6  7]
      [ 8  9 10 11]
      [12 13 14 15]]
```

Similarly:

```
c = a*3
print(c)
```

```
     [[ 0  3  6  9]
      [12 15 18 21]
      [24 27 30 33]
      [36 39 42 45]]
```

```
d = a**2
print(d)
```

```
     [[  0   1   4   9]
      [ 16  25  36  49]
      [ 64  81 100 121]
      [144 169 196 225]]
```

Functions are applied elementwise to whole arrays

```
d = np.sqrt(a)
print(d)
```

```
     [[0.         1.         1.41421356 1.73205081]
      [2.         2.23606798 2.44948974 2.64575131]
      [2.82842712 3.         3.16227766 3.31662479]
      [3.46410162 3.60555128 3.74165739 3.87298335]]
```

```
d = np.sin(a)
print(d)
```

```
[[ 0.          0.84147098  0.90929743  0.14112001]
 [-0.7568025  -0.95892427 -0.2794155   0.6569866 ]
 [ 0.98935825  0.41211849 -0.54402111 -0.99999021]
 [-0.53657292  0.42016704  0.99060736  0.65028784]]
```

Indexing and slicing

Integer indexing works the same way as in Java

```
a= np.arange(16).reshape(4,4)
print(a)
print(a[2,3])
print(a[0,2])
```

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

As with lists, negative indices (counting from the end) are allowed

```
print(a[0,-1])
print(a[-1,0])
```

```
3
12
```

Slicing works the same way as in lists, with one slice per dimension.

```
a = np.arange(10)*10
print(a)
print(a[:5])
print(a[2:5])
print(a[4:])
print(a[3:8:2])
print(a[3::2])
```

```
[ 0 10 20 30 40 50 60 70 80 90]
[ 0 10 20 30 40]
[20 30 40]
[40 50 60 70 80 90]
[30 50 70]
[30 50 70 90]
```

For slicing in two dimensions, the results is the **intersection** of the two slices provided.

```
a= np.arange(16).reshape(4,4)
print(a)
print(a[:,:])
print(a[:,2])
print(a[:3,2:])
print(a[1:-1,1:-1])
```

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

```
print(a[:2,1:])  # Select rows 0 and 1 and columns 1,2, and 3
```

```
[[1 2 3]
 [5 6 7]]
```

```
print(a[::2,1::2])  # Select rows 0 and 2 and columns 1 and 3
```

```
[[ 1  3]
 [ 9 11]]
```

```
print(a[:,::-1])  # Reverse the order of columns
```

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

```
print(a[::-1,:])  # Reverse the order of rows
```

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

```
print(a[::-1])  # Reverse the order of rows, trailing ':' may be ommited
```

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

Lists as indices: We can use lists of the same length as indices. This resturns a 1D array with the same length as the index list, independently of the size of the original array.

```
a= np.arange(16).reshape(4,4)
print(a)
ind1 = [0,1,2]
ind2 = [2,1,0]

print(a[ind1,ind2])
print(a[ind2,ind1])
print(a[ind1,ind1])
print(a[ind2,ind2])
```

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

```
a= np.arange(24).reshape(4,6)
print(a)
ind1 = [0,1,2]
ind2 = [2,1,0]

print(a[ind1,ind2])
print(a[ind2,ind1])
print(a[ind1,ind1])
print(a[ind2,ind2])
```

```
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]]
[ 2  7 12]
[12  7  2]
[ 0  7 14]
[14  7  0]
```

We can assign values to array elements and slices

```
a= np.arange(16).reshape(4,4)
print(a)
a[2,3] = -100
print(a)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
[[   0    1    2    3]
 [   4    5    6    7]
 [   8    9   10 -100]
 [  12   13   14   15]]
```

```
a= np.arange(16).reshape(4,4)
print(a)
a[:2,1:] = -100
print(a)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
[[   0 -100 -100 -100]
 [   4 -100 -100 -100]
 [   8    9   10   11]
 [  12   13   14   15]]
```

```
a= np.arange(24).reshape(4,6)
print(a)
ind1 = [0,1,2,0,2,3]
ind2 = [2,1,5,0,2,5]

a[ind1,ind2] = -1

print(a)
```

```
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]]
[[-1  1 -1  3  4  5]
 [ 6 -1  8  9 10 11]
 [12 13 -1 15 16 -1]
 [18 19 20 21 22 -1]]
```

If we omit a trailing dimension, all elements in that dimension are included in the slice.

```
a= np.arange(24).reshape(4,6)
print(a)
a[1:3]= 0 # This is the same as a[1:3,:] = 0
print(a)
```

```
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]]
[[ 0  1  2  3  4  5]
 [ 0  0  0  0  0  0]
 [ 0  0  0  0  0  0]
 [18 19 20 21 22 23]]
```

You can perform elementwise operations on array slices if they are the same size

```
a= np.arange(16).reshape(4,4)
print(a)
b = np.arange(6).reshape(2,3)
print(b)
a[:2,1:] = a[:2,1:] - b
print(a)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
```

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

Warning- arrays are copied by reference; array assignments are shallow

```
a= np.arange(16).reshape(4,4)
print(a)
b = a
b[0,0] = 2302
print(a)
```

We can make a deep copy of an array using the numpy **copy** function.

```
a= np.arange(16).reshape(4,4)
print(a)
b = np.copy(a)
b[0,0] = 2302
print(a)
print(b)
```

Numpy provides LOTS of array operations. See: https://numpy.org/doc/stable/reference/routines.math.html

Commonly used built-in functions:

min, max, mean, argmin, argmax

```
a = np.array([60,  0, 70, 50, 10, 30, 90, 40, 80, 20])
```

```
print(np.min(a))
```

```
0
```

```
print(np.max(a))
```

```
90
```

```
print(np.mean(a))
```

```
45.0
```

```
print(np.argmax(a)) # Returns the index of the maximum element in a
```

```
6
```

```
print(np.argmin(a)) # Returns the index of the minimum element in a
```

```
1
```

Since Python is an interpreted language, loops are slow. See the comparative running time of the same operation with and without loops.

```
import time

def sum_array_loops(a,b):
  c = np.zeros_like(a)
  for i in range(a.shape[0]):
    for j in range(a.shape[1]):
      c[i,j] =  a[i,j] + b[i,j]
  return c

def sum_array(a,b):
  c = a + b
  return c
```

```
size = 2000

a = np.random.random((size,size))
b = np.random.random((size,size))

start = time.time()
c = sum_array_loops(a,b)
elapsed_time1 = time.time() - start
print('elapsed time using loops', elapsed_time1,'secs')

start = time.time()
c = sum_array(a,b)
elapsed_time2 = time.time() - start
print('elapsed time without loops', elapsed_time2,'secs')

print('ratio',elapsed_time1/elapsed_time2)
```

```
elapsed time using loops 2.237966537475586 secs
elapsed time without loops 0.016918659210205078 secs
ratio 132.2780079479158
```

Arrays as indices

[ ] ↳ *3 cells hidden*

Boolean arrays as indices

[ ] ↳ *8 cells hidden*