numpy_practice_I

January 27, 2022

1 Practice with numpy

1.0.1 Group Members and Roles

- Jenny Lee (Proposer, Reviewer)
- Jiaxin Luo (Driver)
- Group Member 3 (Role)

1.1 Introduction

In this activity, we'll review some of our fundamental `numpy` skills, including another example of using `numpy` to significantly speed up a mathematical operation. Start by running the following cell to load in our star package.

[1]:
```
import numpy as np
```

As introduced in the pre-recorded lectures, `numpy` is a fast numerical computation module for Python. The basic tools that `numpy` offers are the *array* data structure and *vectorized* implementations of functions on arrays.

1.1.1 Note

It's ok to skip around in this worksheet. If you're having trouble solving a problem in a given section, we encourage you to move on to a different problem or even a different section. You may wish to ask for help on the problem (on the Discussion Tracker), and work on something else while you wait for help.

If you find yourself spending more than 10 minutes on a given problem, then go ahead and move on for now.

Additionally, we suggest **please moving on to Section §2** around the 30-minute mark, even if you haven't made it all the way through Section §1.

1.2 §0. Array Performance

A `numpy` array is implemented as a C array: it is a contiguous block of memory that stores data all of the same type. This is in contrast with native Python *lists*, which are implemented as C arrays that store pointers to their contents. These contents can in turn be scattered throughout memory. The much simpler structure of `numpy` arrays implies that operations performed on them can be *much* faster. Let's take a look. We'll start by building some experimental data. Run the block below.

[6]:
```
# run this cell -- do not modify
n = 10000

a = np.random.rand(n) # n random numbers
b = np.random.rand(n) # n more random numbers

# list versions
a_list = list(a)
b_list = list(b)
```

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

Suppose we'd like to multiply `a` and `b` entrywise. That is, we'd like to create a new set of data `c` such that `c[i] = a[i]b[i]`. There are several approaches to doing this. Let's compare three!

Approach 1: In the next cell, write a list comprehension that makes a new list `c_list` whose elements are products of corresponding elements of `a_list` and `b_list`.

Note: Leave the `%%timeit` decorator in the cell; this will let you measure the speed of execution. 1 ms = 1000 µs, 1 µs = 1000 ns

[5]:
```
%%timeit
# write your code here

c_list = [a * b for a, b in zip(a_list, b_list)]
```

```
1.49 ms ± 281 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Approach 2: Make a new `numpy` array `c` filled with 10000 zeros (hint: `np.zeros()`). Write a `for` loop that iterates over the elements of arrays `a` and `b` and assigns their product to the corresponding entry of `c`. Compare to Approach 1.

[12]:
```
%%timeit
# write your code here
c = np.zeros(10000)
for i in range(len(c)):
    c[i] = a[i]*b[i]
```

```
5.51 ms ± 586 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Now make `c` again, this time by multiplying the corresponding pairs of elements of `a` and `b` using `numpy` vectorized multiplication.

[13]:
```
%%timeit
# write your code here
c = a*b
```

```
6.05 µs ± 367 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

What do you observe about the performance of each approach? Discuss with your group and try to explain your observations.

write your answer here The third one of vectorized multiplication is the fastest among these approaches, then the first one and the second one. Since memory is contiguous in an array, it was able to perform the same task the fastest.

1.3 §1. Building Arrays

In the following exercises, build the indicated arrays using `numpy` functions. "You should complete the problems in this section **without using `for`-loops**. Instead use the functions that come with the numpy module such as `np.ones()` and similar tools.

Try not to store anything into variables – you don't need to yet. Each problem can be completed in a single line of code of under 40 characters.

These problems can be completed in any order, so feel free to skip around if you're feeling stuck.

Example:

```
array([ 5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
       22, 23, 24])
```

[ ]:
```
# Example solution
np.arange(5, 25)
```

1.3.1 (A)

```
array([[ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
```

[14]:
```
# write your code here
np.arange(5,25).reshape(4,5)
```

[14]:
```
array([[ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
```

1.3.2 (B)

```
array([[ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16],
       [17, 18, 19, 20],
       [21, 22, 23, 24]])
```

[15]:
```
np.arange(5,25).reshape(5,4)
```

[15]:
```
array([[ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16],
```

```
       [17, 18, 19, 20],
       [21, 22, 23, 24]])
```

1.3.3 (C)

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

[20]:
```
np.ones(5)
```

[20]:
```
array([1., 1., 1., 1., 1.])
```

1.3.4 (D)

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

[21]:
```
np.ones(6).reshape(3,2)
```

[21]:
```
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
```

1.3.5 (E)

```
array([[7., 7.],
       [7., 7.],
       [7., 7.]])
```

[24]:
```
(np.ones(6)*7).reshape(3,2)
```

[24]:
```
array([[7., 7.],
       [7., 7.],
       [7., 7.]])
```

1.3.6 (F)

```
array([ 1,  4,  7, 10, 13])
```

[25]:
```
np.arange(1,14,3)
```

[25]:
```
array([ 1,  4,  7, 10, 13])
```

1.3.7 (G)

```
array([  1.,   2.,   4.,   8.,  16.,  32.,  64., 128., 256., 512.])
```

[33]:
```
2**np.arange(10,dtype=float)
```

[33]: array([1.,   2.,   4.,   8., 16., 32., 64., 128., 256., 512.])

1.3.8 (H)

```
array([ 0.,  2.,  4.,  6.,  8., 10.])
```

[37]:
```
np.linspace(0,10,6)
```

[37]: array([0., 2., 4., 6., 8., 10.])

1.3.9 (I)

```
array([20., 15., 10.,  5.,  0.])
```

[39]:
```
np.linspace(20,0,5)
```

[39]: array([20., 15., 10., 5., 0.])

1.3.10 (J)

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

       [[10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]],

       [[20, 21, 22, 23, 24],
        [25, 26, 27, 28, 29]]])
```

[41]:
```
np.arange(0,30).reshape(3,2,5)
```

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

       [[10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]],

       [[20, 21, 22, 23, 24],
        [25, 26, 27, 28, 29]]])

1.3.11 (K)

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

[43]:
```
# there are 20 entries
np.arange(0,20) > 7
```

[43]:
```
array([False, False, False, False, False, False, False, False,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True])
```

1.3.12 (L)

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

[46]:
```
np.arange(0,20)%2 == 0
```

[46]:
```
array([ True, False,  True, False,  True, False,  True, False,  True,
       False,  True, False,  True, False,  True, False,  True, False,
        True, False])
```

1.3.13 (M)

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

[48]:
```
(np.arange(0,20) >7) & (np.arange(0,20) %2==0)
```

[48]:
```
array([False, False, False, False, False, False, False, False,  True,
       False,  True, False,  True, False,  True, False,  True, False,
        True, False])
```

1.4 §2. Manipulating Arrays

From this point on it might help to store the arrays to variables. You should complete the problems in this section **without using for-loops**.

1.4.1 (A)

```
array([ 0,  1,  2,  0,  4,  5,  0,  7,  8,  0, 10, 11,  0, 13, 14,  0, 16,
       17,  0, 19])
```

[55]:
```
a = np.arange(0,20)
a[a %3 == 0] = 0
a
```

[55]:
```
array([ 0,  1,  2,  0,  4,  5,  0,  7,  8,  0, 10, 11,  0, 13, 14,  0, 16,
       17,  0, 19])
```

1.4.2 (B)

```
array([[0, 1, 2, 3],
       [4, 5, 6, 0],
```

```
       [0, 0, 0, 0]])
```

[62]:
```
a = np.arange(12).reshape(3, 4)
a[a > 6] = 0
a
```

[62]:
```
array([[0, 1, 2, 3],
       [4, 5, 6, 0],
       [0, 0, 0, 0]])
```

1.4.3 (C)

```
array([[ 0,  1,  2, 13],
       [14, 15, 16, 17],
       [18, 19, 10, 11]])
```

[80]:
```
a = np.arange(12).reshape(3, 4)
a [(a > 2) & (a < 10)] += 10
a
```

[80]:
```
array([[ 0,  1,  2, 13],
       [14, 15, 16, 17],
       [18, 19, 10, 11]])
```

1.4.4 §2. Slicing multidimensional arrays

Run the next cell first.

[ ]:
```
A = np.arange(28).reshape(4, 7)
A
```

Obtain the following arrays by slicing `A`.

1.4.5 (A)

```
array([ 7,  8,  9, 10, 11, 12, 13])
```

[ ]:
```

```

1.4.6 (B)

```
array([ 4, 11, 18, 25])
```

[ ]:
```

```

1.4.7 (C)

```
array([[ 9, 10, 11],
       [16, 17, 18],
```

```
       [23, 24, 25]])
```

[ ]:

1.4.8 (D)

```
array([[ 1,  2,  3,  4],
       [ 8,  9, 10, 11],
       [15, 16, 17, 18],
       [22, 23, 24, 25]])
```

[ ]:

1.4.9 (E)

```
array([[ 0,  2,  4,  6],
       [ 7,  9, 11, 13],
       [14, 16, 18, 20],
       [21, 23, 25, 27]])
```

[ ]:

1.4.10 (F)

```
array([[ 0,  3,  5,  6],
       [ 7, 10, 12, 13]])
```

[ ]:

1.4.11 §3. Bonus

This is a bonus problem for students who have some familiarity with linear algebra, specifically, matrix-vector multiplication. If you are not familiar with linear algebra, no worries! That's why this one is a bonus.

Run the following cells and observe the output.

[ ]:
```
import numpy as np
A = np.arange(20).reshape(4,5)

print(A, A.sum(), A.sum(0), A.sum(1), sep='\n\n')
```

[ ]:
```
v = np.arange(1, 6)
v
```

In one line, using the above functions, write an expression for the matrix product `Av`, considering `v` as a column vector. Then, explain your approach.

[ ]:
```
# your solution here
```

Explain your approach here.