IC152 Lec 16

Feb 2021

Using numpy

```python
import numpy as np
np.random.seed(0)  # 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 ar
```

```
In [5]: x1
Out[5]: array([5, 0, 3, 3, 7, 9])

In [6]: x2
Out[6]:
array([[3, 5, 2, 4],
       [7, 6, 8, 8],
       [1, 6, 7, 7]])

In [7]: x3
Out[7]:
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 [8]:
```

```
In [9]: x3.ndim
Out[9]: 3

In [10]: x3.shape
Out[10]: (3, 4, 5)

In [11]: x3.size
Out[11]: 60

In [12]: x3.dtype
Out[12]: dtype('int64')
```

```
In [14]: x1.all()
Out[14]: False

In [15]: x1.argmax()
Out[15]: 5

In [16]: x2.all()
Out[16]: True
```

Creating ndarrays

```
x4 = np.array([[1, 2, 3], [4, 5, 6]], np.int32)
```

```
In [26]: x4
Out[26]:
array([[1, 2, 3],
       [4, 5, 6]], dtype=int32)
```

```
In [19]: type(x4)
Out[19]: numpy.ndarray

In [20]: x4.shape
Out[20]: (2, 3)

In [21]: x4.ndim
Out[21]: 2

In [22]: x4.dtype
Out[22]: dtype('int32')

In [23]: x4[1,2]
Out[23]: 6
```

```
In [38]: np.eye(3)
Out[38]:
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])

In [39]: np.ones((2,2,4))
Out[39]:
array([[[1., 1., 1., 1.],
        [1., 1., 1., 1.]],

       [[1., 1., 1., 1.],
        [1., 1., 1., 1.]]])
```

```
In [52]: np.zeros((2,2))
Out[52]:
array([[0., 0.],
       [0., 0.]])
```

There are many other ways to create np arrays.

`eye, ones, zeros`

```
In [46]: l = [2,3,1,0]

In [47]: np.array(l)
Out[47]: array([2, 3, 1, 0])

In [48]: np.arange(2, 3, 0.1)
Out[48]: array([2. , 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9])

In [49]: np.linspace(1., 4., 6)
Out[49]: array([1. , 1.6, 2.2, 2.8, 3.4, 4. ])
```

From a list

`arange` function

`linspace` function

Indexing and slicing

```
In [59]: x = np.arange(10)

In [60]: x[:5]
Out[60]: array([0, 1, 2, 3, 4])

In [61]: x[4:7]
Out[61]: array([4, 5, 6])

In [62]: x[::2]
Out[62]: array([0, 2, 4, 6, 8])

In [63]: x[::-1]
Out[63]: array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

`x[start:stop:step]`

Every other element

All elements, reversed

```
In [66]: x2
Out[66]:
array([[3, 5, 2, 4],
       [7, 6, 8, 8],
       [1, 6, 7, 7]])

In [67]: x2[0,0] = 12

In [68]: x2
Out[68]:
array([[12,  5,  2,  4],
       [ 7,  6,  8,  8],
       [ 1,  6,  7,  7]])

In [69]: x2[:2,:3]
Out[69]:
array([[12,  5,  2],
       [ 7,  6,  8]])
```

Change the value (arrays are mutable)

First two rows, first three columns

```
In [71]: x2[:, ::2]
Out[71]:
array([[12,  2],
       [ 7,  8],
       [ 1,  7]])
```

All rows, alternate columns

Subarrays are no-copy views

```
In [73]: x2
Out[73]:
array([[12,  5,  2,  4],
       [ 7,  6,  8,  8],
       [ 1,  6,  7,  7]])

In [74]: x2_sub = x2[:2, :2]

In [75]: x2_sub
Out[75]:
array([[12,  5],
       [ 7,  6]])

In [76]: x2_sub[0, 0] = 99

In [77]: x2_sub
Out[77]:
array([[99,  5],
       [ 7,  6]])

In [78]: x2
Out[78]:
array([[99,  5,  2,  4],
       [ 7,  6,  8,  8],
       [ 1,  6,  7,  7]])
```

Take a subarray

Modify the subarray

The original has changed!

This is an important feature.
Helps in manipulating data without making copies.
But be aware of this behaviour!

Copies can be made

```
In [80]: x2_sub_copy = x2[:2, :2].copy()

In [81]: x2_sub_copy[0, 0] = 42

In [82]: x2
Out[82]:
array([[99,  5,  2,  4],
       [ 7,  6,  8,  8],
       [ 1,  6,  7,  7]])
```

`copy()` method

Original is untouched

Reshaping, splitting and concatenating

```
In [96]: grid = np.arange(1, 10).reshape((3, 3))

In [97]: grid
Out[97]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Reshaping arrays with `reshape()`
Size of initial array must match the reshaped array

```
In [89]: x = np.array([1, 2, 3])

In [90]: y = np.array([3, 2, 1])

In [91]: np.concatenate([x, y])
Out[91]: array([1, 2, 3, 3, 2, 1])

In [92]: x = [1, 2, 3, 99, 99, 3, 2, 1]

In [93]: x1, x2, x3 = np.split(x, [3, 5])

In [94]: print(x1, x2, x3)
[1 2 3] [99 99] [3 2 1]
```

`np.concatenate()`

`np.split()`

Ufuncs to speed up loops

Avoid loops as much as possible
Operations can be vectorized using ufuncs

```
In [6]: np.arange(5) / np.arange(1, 6)
Out[6]: array([0.        , 0.5       , 0.66666667, 0.75      , 0.8       ])

In [7]: 2 ** (np.arange(9).reshape((3, 3)))
Out[7]:
array([[  1,   2,   4],
       [  8,  16,  32],
       [ 64, 128, 256]])
```

```python
import numpy as np
import time

np.random.seed(0)

def compute_reciprocals(values):
    output = np.empty(len(values))
    for i in range(len(values)):
        output[i] = 1.0 / values[i]
    return output


values = np.random.randint(1, 10, size=100000)

tic = time.perf_counter()
print(compute_reciprocals(values))
toc = time.perf_counter()
print(f'Time taken loop: {toc - tic:0.4f} seconds')

tic = time.perf_counter()
print(1/(values))
toc = time.perf_counter()
print(f'Time taken ufunc: {toc - tic:0.4f} seconds')
```

```
[0.16666667 1.         0.25       ... 0.14285714 0.16666667 0.16666667]
Time taken loop: 0.1405 seconds
[0.16666667 1.         0.25       ... 0.14285714 0.16666667 0.16666667]
Time taken ufunc: 0.0004 seconds
```

Aggregation functions

```python
import numpy as np

big_array = np.random.rand(1000000)
print(min(big_array), max(big_array))
print(np.min(big_array),np.max(big_array))
print(big_array.min(),big_array.max())
```

built-in
numpy’s built-in
ndarray’s method

```
2.516783892403396e-08 0.9999990679943628
2.516783892403396e-08 0.9999990679943628
2.516783892403396e-08 0.9999990679943628
```

```python
M = np.random.random((3, 4))
print(M)
```

```
[[0.38706171 0.12735852 0.31013613 0.97664639]
 [0.179092   0.07422295 0.45766201 0.42744923]
 [0.57553659 0.50942114 0.41053727 0.71563338]]
```

```
In [12]: M.sum()
Out[12]: 5.150757325329906

In [13]: M.min(axis=0)
Out[13]: array([0.179092  , 0.07422295, 0.31013613, 0.42744923])

In [14]: M.min(axis=1)
Out[14]: array([0.12735852, 0.07422295, 0.41053727])
```

Aggregations across various axes

Examples of numpy in action

$$\begin{aligned} x_1 - 2x_2 + x_3 &= 0 \\ 2x_2 - 8x_3 &= 8 \\ 5x_1 \phantom{{}-2x_2} - 5x_3 &= 10 \end{aligned}$$

system of equations

$$\begin{bmatrix} 1 & -2 & 1 \\ 0 & 2 & -8 \\ 5 & 0 & -5 \end{bmatrix}$$

Coefficient matrix

```
[[ 1.00000000e+00]
 [ 2.22044605e-16]
 [-1.00000000e+00]]

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

```python
import numpy as np
import numpy.linalg as la

A = np.array([[1,-2,1],[0,2,-8],[5,0,-5]])
b = np.array([[0],[8],[10]])
print('Rank of coeff matrix =', la.matrix_rank(A))

print()

sol = np.matmul(la.inv(A), b)
print(sol)

print()

sol2 = np.linalg.solve(A,b)
print(sol2)

```

Eigenvalues and eigenvectors

$$A = \begin{bmatrix} 1 & 6 \\ 5 & 2 \end{bmatrix}$$

Eigenvalues = 7,4

eigenvectors = $[1,1]^T$, $[3/2, -5/4]^T$

```python
import numpy as np
import numpy.linalg as la


A = np.array([[1,6],[5,2]])
w,v = la.eig(A)
print(w)
print(v)

```

```
[-4.  7.]
[[-0.76822128 -0.70710678]
 [ 0.6401844  -0.70710678]]
```