0705 Python / AI Programming Practice

Tutor: Nanyang Ye
Note Taker: Y. Qiu

NumPy

n-dim array object (ndarray) index range:0..(n-1)

* NumPy Source Code -- realized by C

- a pointer showing its address -- quotation
- dtype -- the room required for each element, e.g. int -- 4 bytes
- shape (a tuple) -- the size of all dimensions
- stride (a tuple) -- e.g.[i][j] -> [i+1][j]

int[num] -- num=8/16/32/64 num-bit integer
uint[num] -- num = 8/16/32/64 unsigned num-bit integer
complex[num] -- num =64/128 real: 32/64-bit imaginal: 32/63-bit

In [38]:

```
import numpy as np

x = [1,2,3]
a = np.asarray(x)       # convert a list into an array
print (a)

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

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


print( *a, sep = ',')
```

```
[1 2 3]
[[1 2]
 [3 4]
 [5 6]]
[[1 2]
 [3 4]
 [5 6]]
[1 2],[3 4],[5 6]
```

print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)

```
sep: 分隔符
end
file
flush = True 时会强制刷新数据缓存（舍弃缓存内的数据）
```

In [13]:

```
import numpy as np
y = np.zeros((5,), dtype = np.int, order = 'C')
print (y)
```

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

In [15]:

```
import numpy as np
x = np.arange(0,102,2.0)
print (x)

'''x = np.asarray(range(0,102,2))
print (x)'''

x = np.linspace(0,100,51)
print (x)
```

```
[  0.   2.   4.   6.   8.  10.  12.  14.  16.  18.  20.  22.  24.  26.
  28.  30.  32.  34.  36.  38.  40.  42.  44.  46.  48.  50.  52.  54.
  56.  58.  60.  62.  64.  66.  68.  70.  72.  74.  76.  78.  80.  82.
  84.  86.  88.  90.  92.  94.  96.  98. 100.]
[  0.   2.   4.   6.   8.  10.  12.  14.  16.  18.  20.  22.  24.  26.
  28.  30.  32.  34.  36.  38.  40.  42.  44.  46.  48.  50.  52.  54.
  56.  58.  60.  62.  64.  66.  68.  70.  72.  74.  76.  78.  80.  82.
  84.  86.  88.  90.  92.  94.  96.  98. 100.]
```

In [37]:

```
x = np.arange(10)
s = slice(2,7,2)  # from index 2 to index 7, with stride = 2
print(x[s])

print(x[2:7:2])

# compare with list:

li = list(range(0,10))
print(li[2:7:2], end = '\n\n')

print(x[2:-1:2])
print(li[2:-1:2])

s = slice(2,-1,2)
print(x[s], end = '\n\n')

print(li[-5:-1:1])
print(x[-5:-1:1])
print(x[slice(-5,-1,1)])
```

```
[2 4 6]
[2 4 6]
[2, 4, 6]

[2 4 6 8]
[2, 4, 6, 8]
[2 4 6 8]

[5, 6, 7, 8]
[5 6 7 8]
[5 6 7 8]
```

In [70]:

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

print(a[...,1])
print(a[1,...])
print(a[...,1:])
print()

print(a[[1,0]])
print(a[[1,0],[2,2]])    # a[1,2] and a[0,2]
```

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

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

In [78]:

```
print([a>5])
print(a[ a>5 ])

b = ((False, False, True), (False, True, True))
print(a[b])
```

```
[array([[False, False, False],
       [False, False,  True],
       [ True,  True,  True]])]
[6 7 8 9]
[8 9]
```

In [84]:

```
a = np.arange(8)
print("original array:\n",a,end = "\n\n")

print("2 x 4:\n",a.reshape(2,4),end = '\n\n')

print("1 x 8:\n",a.reshape(1,-1),end = '\n\n')

print("8 x 1:\n",a[:, None],end = '\n\n')

print("4 x 2:\n",a.reshape(4,2), end = '\n\n')
```

```
original array:
 [0 1 2 3 4 5 6 7]

2 x 4:
 [[0 1 2 3]
 [4 5 6 7]]

1 x 8:
 [[0 1 2 3 4 5 6 7]]

8 x 1:
 [[0]
 [1]
 [2]
 [3]
 [4]
 [5]
 [6]
 [7]]

4 x 2:
 [[0 1]
 [2 3]
 [4 5]
 [6 7]]
```

In [90]:

```
a = np.array([[1,2,3,4,5],[6,7,8,9,10],[11,12,13,14,15]])
print (np.transpose(a))

b = np.array([1,2,3,4,5,6,7,8,9])
print (b)
print (np.transpose(b))

b = np.array([[1,2,3,4,5,6,7,8,9]])
print (b)
print (np.transpose(b))
```

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

In [8]:

```
import numpy as np

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

print(a, end = '\n\n')

print("未传递Axis参数。 插入前数组被强制展开。")
print(np.insert(a,3,[11,12]),end = '\n\n')

print("传递了Axis参数，广播值数组以匹配插入数组\nAxis = 0")
print(np.insert(a,1,[11],axis = 0), end = '\n\n')
print("Axis = 1")
print(np.insert(a,1,[11],axis = 1), end = '\n\n')
```

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

未传递Axis参数。 插入前数组被强制展开。
[ 1  2  3 11 12  4  5  6]

传递了Axis参数，广播值数组以匹配插入数组
Axis = 0
[[ 1  2]
 [11 11]
 [ 3  4]
 [ 5  6]]

Axis = 1
[[ 1 11  2]
 [ 3 11  4]
 [ 5 11  6]]
```

In [7]:

```
a = np.arange(12).reshape(3,4)

print(np.delete(a, 5))
print(a)
```

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

Concatenate

Used in ResNet, etc.

In [47]:

```
a = np.array([[1,2,3],[6,7,8],[11,12,13]])
a_r = np.array([[16,17,18]])
a_c = np.array([[4,5],[9,10],[14,15]])
a_d = np.array([[101,102,103],[106,107,108],[111,112,113]])

print(np.vstack((a,a_r)))     # the (only) parameter should be a tuple
print(np.concatenate((a,a_r),axis = 0), end = '\n\n')

print(np.hstack((a,a_c)))
print(np.concatenate((a,a_c),axis = 1), end = '\n\n')

print(np.dstack((a,a_d)))
# print(np.concatenate(([a,a_d]),axis = 2))
```

```
[[ 1  2  3]
 [ 6  7  8]
 [11 12 13]
 [16 17 18]]
[[ 1  2  3]
 [ 6  7  8]
 [11 12 13]
 [16 17 18]]

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

[[[  1 101]
  [  2 102]
  [  3 103]]

 [[  6 106]
  [  7 107]
  [  8 108]]

 [[ 11 111]
  [ 12 112]
  [ 13 113]]]
```

Broadcasting

Used to make matrix of a same size.

Normalization; Cartesian Product -- Linear Algebra

tensor -- 张量 (vector, matrix, etc.)

In [18]:

```
import numpy as np

a = np.array([[0,0,0],[10,10,10],[20,20,20],[30,30,30]])
b = np.array([1,2,3])

print(a + b)

bb = np.tile(b,(4,1))

print(a + bb, end='\n\n')
print(bb)

c = np.array([[1,2,3],[4,5,6]])
cc = np.tile(c,(3,2))

print(cc)
```

```
[[ 1  2  3]
 [11 12 13]
 [21 22 23]
 [31 32 33]]
[[ 1  2  3]
 [11 12 13]
 [21 22 23]
 [31 32 33]]

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

In [69]:

```
import numpy as np

a = np.array([[1,2,3],[4,5,6],[7,8,9]])
b = np.array([[1,1,1],[5,5,5],[9,9,9]])

print(np.add(a,b), end = ’\n\n’)              # print(a+b)
print(np.subtract(a,b), end = ’\n\n’)         # print(a-b)
print(np.multiply(a,b), end = ’\n\n’)         # print(a*b)
print(np.divide(a,b), end = ’\n\n’)           # print(a/b)

print(np.dot(a,b))                            # matrix production
print(np.dot([1,2,3],[4,5,6]))                # dot production
print(np.vdot([1,2,3],[4,5,6]))               # dot production

print()

print(np.linalg.det([[6,1,1], [4, -2, 5], [2,8,7]]))
print(np.linalg.solve(a,[[1],[4],[8]]))       # solve ax = b and return vector x
print(np.linalg.inv(a))                       # inverse matrix
```

```
[[ 2  3  4]
 [ 9 10 11]
 [16 17 18]]

[[ 0  1  2]
 [-1  0  1]
 [-2 -1  0]]

[[ 1  2  3]
 [20 25 30]
 [63 72 81]]

[[1.         2.         3.        ]
 [0.8        1.         1.2       ]
 [0.77777778 0.88888889 1.        ]]

[[ 38  38  38]
 [ 83  83  83]
 [128 128 128]]
32
32

-306.0
[[ 3.15251974e+15]
 [-6.30503948e+15]
 [ 3.15251974e+15]]
[[ 3.15251974e+15 -6.30503948e+15  3.15251974e+15]
 [-6.30503948e+15  1.26100790e+16 -6.30503948e+15]
 [ 3.15251974e+15 -6.30503948e+15  3.15251974e+15]]
```

In [20]:

```
import numpy as np

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

print(np.amin(a,1))   # row min
print(np.amax(a,0))   # column max
print(np.amin(a))     # universal min
```

```
[ 1  4  7 10]
[11 10 12]
1
```

In [70]:

```
import numpy as np

a = np.array([[3,1,2],[6,5,4],[7,8,9],[11,10,12]])
print(np.sort(a))                    # row-wise sort
print(np.sort(a,axis = 0))           # column-wise sort
print(np.sort(a,axis = 1))           # row-wise sort
```

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

File I/O

In [25]:

```
import numpy as np

a = np.array([1,2,3,4,5])

np.save("trial0705",a)
```

In [26]:

```
b = np.load("trial0705.npy")
print(b)
```

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

Example

In [4]:

```
import numpy as np

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

a = np.transpose(a)
b = np.concatenate((b,c),axis = 0)

print(a)
print(b)
c = np.matmul(b,a)

np.save("trial0705",c)
```

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

Pandas

```
Used to introduce Excel/SQL data
```

Data Structures

Series

```
a 1-dim array with index.
```

DataFrame

```
"Excel"-like table.
```

In [37]:

```
import pandas as pd
import numpy  as np

s = pd.Series([1, 3, 5, np.nan, 6, 8])
print(s)

excel1 = pd.DataFrame({'A': 1.,
                       'B': pd.Timestamp('20210705'),
                       'D': pd.array([3] * 4, dtype = 'int32'),
                       'C': pd.Categorical(['test','train','train','test'])
                      })

excel1
```

```
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

Out[37]:

	A	B	D	C
0	1.0	2021-07-05	3	test
1	1.0	2021-07-05	3	train
2	1.0	2021-07-05	3	train
3	1.0	2021-07-05	3	test

In [2]:

```
import pandas as pd
import numpy

data = [{'a':1, 'b':2},{'a':3,'b':4,'5':6}]
print(pd.DataFrame(data,index = ['1st','2nd']))
```

```
     a  b    5
1st  1  2  NaN
2nd  3  4  6.0
```

In [9]:

```
import pandas as pd

# read in Comma-Separated Values (CSV)

np.random.seed(0)
dates = pd.date_range('1/1/2000',periods = 8)
df = pd.DataFrame(np.random.randn(8,4),index = dates, columns = ['A','B','C','D'])

df
```

Out[9]:

	A	B	C	D
2000-01-01	1.764052	0.400157	0.978738	2.240893
2000-01-02	1.867558	-0.977278	0.950088	-0.151357
2000-01-03	-0.103219	0.410599	0.144044	1.454274
2000-01-04	0.761038	0.121675	0.443863	0.333674
2000-01-05	1.494079	-0.205158	0.313068	-0.854096
2000-01-06	-2.552990	0.653619	0.864436	-0.742165
2000-01-07	2.269755	-1.454366	0.045759	-0.187184
2000-01-08	1.532779	1.469359	0.154947	0.378163

In [16]:

```
print(df[[’B’,’A’]])
print(df.loc[:,[’B’,’A’]])
print(df.iloc[:,0:2])
```

```
                   B         A
2000-01-01  0.400157  1.764052
2000-01-02 -0.977278  1.867558
2000-01-03  0.410599 -0.103219
2000-01-04  0.121675  0.761038
2000-01-05 -0.205158  1.494079
2000-01-06  0.653619 -2.552990
2000-01-07 -1.454366  2.269755
2000-01-08  1.469359  1.532779
                   B         A
2000-01-01  0.400157  1.764052
2000-01-02 -0.977278  1.867558
2000-01-03  0.410599 -0.103219
2000-01-04  0.121675  0.761038
2000-01-05 -0.205158  1.494079
2000-01-06  0.653619 -2.552990
2000-01-07 -1.454366  2.269755
2000-01-08  1.469359  1.532779
                   A         B
2000-01-01  1.764052  0.400157
2000-01-02  1.867558 -0.977278
2000-01-03 -0.103219  0.410599
2000-01-04  0.761038  0.121675
2000-01-05  1.494079 -0.205158
2000-01-06 -2.552990  0.653619
2000-01-07  2.269755 -1.454366
2000-01-08  1.532779  1.469359
```

In [21]:

```
import pandas as pd

# Successfully swap ’A’ column and ’B’ column

np.random.seed(0)
dates = pd.date_range(’1/1/2000’,periods = 8)
df = pd.DataFrame(np.random.randn(8,4),index = dates, columns = [’A’,’B’,’C’,’D’])

df[[’B’,’A’]] = df[[’A’,’B’]]
df
```

Out[21]:

	A	B	C	D
2000-01-01	0.400157	1.764052	0.978738	2.240893
2000-01-02	-0.977278	1.867558	0.950088	-0.151357
2000-01-03	0.410599	-0.103219	0.144044	1.454274
2000-01-04	0.121675	0.761038	0.443863	0.333674
2000-01-05	-0.205158	1.494079	0.313068	-0.854096
2000-01-06	0.653619	-2.552990	0.864436	-0.742165
2000-01-07	-1.454366	2.269755	0.045759	-0.187184
2000-01-08	1.469359	1.532779	0.154947	0.378163

In [22]:

```
import pandas as pd

# Successfully swap ’A’ column and ’B’ column

np.random.seed(0)
dates = pd.date_range(’1/1/2000’,periods = 8)
df = pd.DataFrame(np.random.randn(8,4),index = dates, columns = [’A’,’B’,’C’,’D’])

df.loc[:,[’B’,’A’]] = df[[’A’,’B’]].values
df
```

Out[22]:

	A	B	C	D
2000-01-01	0.400157	1.764052	0.978738	2.240893
2000-01-02	-0.977278	1.867558	0.950088	-0.151357
2000-01-03	0.410599	-0.103219	0.144044	1.454274
2000-01-04	0.121675	0.761038	0.443863	0.333674
2000-01-05	-0.205158	1.494079	0.313068	-0.854096
2000-01-06	0.653619	-2.552990	0.864436	-0.742165
2000-01-07	-1.454366	2.269755	0.045759	-0.187184
2000-01-08	1.469359	1.532779	0.154947	0.378163

In [23]:

```
import pandas as pd

# Failure in swapping ’A’ column and ’B’ column

np.random.seed(0)
dates = pd.date_range(’1/1/2000’,periods = 8)
df = pd.DataFrame(np.random.randn(8,4),index = dates, columns = [’A’,’B’,’C’,’D’])

df.loc[:, [’B’,’A’]] = df[[’A’,’B’]]
df
```

Out[23]:

	A	B	C	D
2000-01-01	1.764052	0.400157	0.978738	2.240893
2000-01-02	1.867558	-0.977278	0.950088	-0.151357
2000-01-03	-0.103219	0.410599	0.144044	1.454274
2000-01-04	0.761038	0.121675	0.443863	0.333674
2000-01-05	1.494079	-0.205158	0.313068	-0.854096
2000-01-06	-2.552990	0.653619	0.864436	-0.742165
2000-01-07	2.269755	-1.454366	0.045759	-0.187184
2000-01-08	1.532779	1.469359	0.154947	0.378163

In [24]:

```
import pandas as pd
import numpy  as np

np.random.seed(0)
df = pd.DataFrame(np.random.randn(6,4),index = list(range(0,12,2)), columns = list(range(0,8,2)))\

df
```

Out[24]:

	0	2	4	6
0	1.764052	0.400157	0.978738	2.240893
2	1.867558	-0.977278	0.950088	-0.151357
4	-0.103219	0.410599	0.144044	1.454274
6	0.761038	0.121675	0.443863	0.333674
8	1.494079	-0.205158	0.313068	-0.854096
10	-2.552990	0.653619	0.864436	-0.742165

In [32]:

```
df.loc[4:9,2:5]
```

Out[32]:

```
          2         4
4  0.410599  0.144044
6  0.121675  0.443863
8 -0.205158  0.313068
```

In [35]:

```
df.iloc[2:5,1:3]
```

Out[35]:

```
          2         4
4  0.410599  0.144044
6  0.121675  0.443863
8 -0.205158  0.313068
```

Concat, Merge, Append

In [12]:

```
import pandas as pd
df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                    'B': ['B0', 'B1', 'B2', 'B3'],
                    'C': ['C0', 'C1', 'C2', 'C3'],
                    'D': ['D0', 'D1', 'D2', 'D3']},
                   index=[0, 1, 2, 3])
df2 = pd.DataFrame({'A': ['A4', 'A5', 'A6', 'A7'],
                    'B': ['B4', 'B5', 'B6', 'B7'],
                    'C': ['C4', 'C5', 'C6', 'C7'],
                    'D': ['D4', 'D5', 'D6', 'D7']},
                   index=[4, 5, 6, 7])
df3 = pd.DataFrame({'A': ['A8', 'A9', 'A10', 'A11'],
                    'B': ['B8', 'B9', 'B10', 'B11'],
                    'C': ['C8', 'C9', 'C10', 'C11'],
                    'D': ['D8', 'D9', 'D10', 'D11']},
                   index=[8, 9, 10, 11])
df4 = pd.DataFrame({'B': ['B2', 'B3', 'B6', 'B7'],
                    'D': ['D2', 'D3', 'D6', 'D7'],
                    'F': ['F2', 'F3', 'F6', 'F7']},
                   index=[2, 3, 6, 7])

df = pd.concat([df1,df2,df3])

df
```

Out[12]:

	A	**B**	**C**	**D**
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

In [7]:

```
df = pd.concat([df1,df2,df3], keys = ['x','y','z'])

df
```

Out[7]:

		A	B	C	D
x	**0**	A0	B0	C0	D0
	1	A1	B1	C1	D1
	2	A2	B2	C2	D2
	3	A3	B3	C3	D3
y	**4**	A4	B4	C4	D4
	5	A5	B5	C5	D5
	6	A6	B6	C6	D6
	7	A7	B7	C7	D7
z	**8**	A8	B8	C8	D8
	9	A9	B9	C9	D9
	10	A10	B10	C10	D10
	11	A11	B11	C11	D11

In [16]:

```
df = pd.concat([df1,df4.reindex(df1.index)],axis = 1)

df
```

Out[16]:

	A	B	C	D	B	D	F
0	A0	B0	C0	D0	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3

In [38]:

```
np.nan == np.nan
```

Out[38]:

```
False
```

Use "np.isnan()" or "xxx is np.nan" instead.