Exercise3 NumPy

April 15, 2018

1 NumPy

NumPy is the fundamental package for scientific computing with Python. It contains among other things:

- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

Library documentation: http://www.numpy.org/

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt

        %matplotlib inline
```

2 Task 1: declare a vector using a list as the argument

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

Out[2]: array([1, 2, 3, 4, 5])
```

3 Task 2: declare a matrix using a nested list as the argument

```
In [3]: np.array([[1,2,3],[4,5,6]])

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

4 Task 3: initialize x or x and y using the following functions: arange, linspace, logspace, mgrid

In [4]:
```
x = np.arange(1,10)
x
```

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

In [5]:
```
x = np.linspace(1,10,10)
x
```

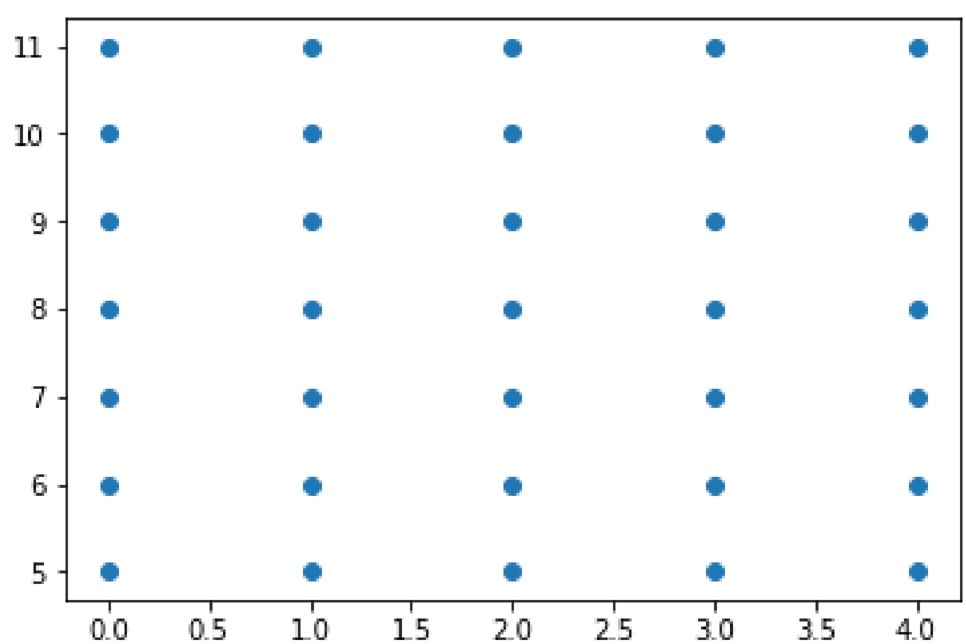
Out[5]:
```
array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
```

In [6]:
```
x = np.logspace(0,1,10)
x
```

Out[6]:
```
array([ 1.        ,  1.29154967,  1.66810054,  2.15443469,  2.7825594 ,
        3.59381366,  4.64158883,  5.9948425 ,  7.74263683, 10.        ])
```

In [7]:
```
x = np.arange(0, 5, 1)
y = np.arange(5, 12, 1)
xx, yy = np.meshgrid(x, y)

plt.scatter(xx,yy)
plt.show()
```



In [31]:
```
from numpy import random, array, arange
```

5 Task 4: what is difference between random.rand and random.randn

- `random.rand` samples from an uniform distribution.
- `random.randn` returns samples from the standard normal distribuion

6 Task 5: what are the funciotns diag, itemsize, nbytes and ndim about?

- `np.diag` constructs an 1D array with diagonal elements from a 2D array.
- `numpy.ndarray.itemsize` gives length of one array element in bytes
- `numpy.matrix.nbytes` returns total bytes consumed by all the elements of an array
- `np.ndim` returns the number of dimensions of an array passed as argument

In [15]:
```
M= random.randn(2,2)
# assign new value
M[0,0] = 7
M
```

Out[15]:
```
array([[ 7.        , -0.38783607],
       [ 0.36806094, -0.57687918]])
```

In [17]:
```
M[0,:] = 0
M
```

Out[17]:
```
array([[ 0.        ,  0.        ],
       [ 0.36806094, -0.57687918]])
```

In [70]:
```
# slicing works just like with lists
A = array([1,2,3,4,5])
A[1:3]
```

Out[70]:
```
array([2, 3])
```

7 Task 6: Using list comprehensions create the following matrix

array([[0, 1, 2, 3, 4], [10, 11, 12, 13, 14], [20, 21, 22, 23, 24], [30, 31, 32, 33, 34], [40, 41, 42, 43, 44]])

In [26]:
```
[[i+j*10 for i in np.arange(0,5)] for j in np.arange(0,5)]
```

Out[26]:
```
[[0, 1, 2, 3, 4],
 [10, 11, 12, 13, 14],
 [20, 21, 22, 23, 24],
 [30, 31, 32, 33, 34],
 [40, 41, 42, 43, 44]]
```

In [27]:
```
row_indices = [1, 2, 3]
A[row_indices]
```

```
Out[27]: array([2, 3, 4])
```

```
In [28]: # index masking
         B = array([n for n in range(5)])
         row_mask = array([True, False, True, False, False])
         B[row_mask]
```

```
Out[28]: array([0, 2])
```

7.0.1 Linear Algebra

```
In [34]: v1 = arange(0, 5)
         v1
```

```
Out[34]: array([0, 1, 2, 3, 4])
```

```
In [35]: v1 + 2
```

```
Out[35]: array([2, 3, 4, 5, 6])
```

```
In [36]: v1 * 2
```

```
Out[36]: array([0, 2, 4, 6, 8])
```

```
In [37]: v1 * v1
```

```
Out[37]: array([ 0,  1,  4,  9, 16])
```

```
In [39]: np.dot(v1, v1)
```

```
Out[39]: 30
```

```
In [41]: np.dot(A, v1)
```

```
Out[41]: 40
```

```
In [42]: A
```

```
Out[42]: array([1, 2, 3, 4, 5])
```

```
In [45]: # cast changes behavior of + - * etc. to use matrix algebra
         M = np.matrix(A)
         M.T * M
```

```
Out[45]: matrix([[ 1,  2,  3,  4,  5],
                 [ 2,  4,  6,  8, 10],
                 [ 3,  6,  9, 12, 15],
                 [ 4,  8, 12, 16, 20],
                 [ 5, 10, 15, 20, 25]])
```

```
In [47]: # inner product
         v1.T * v1
```

```
Out[47]: array([ 0,  1,  4,  9, 16])
```

```
In [49]: C = np.matrix([[1j, 2j], [3j, 4j]])
```

```
In [51]: np.conjugate(C)
```

```
Out[51]: matrix([[0.-1.j, 0.-2.j],
                 [0.-3.j, 0.-4.j]])
```

```
In [52]: # inverse
         C.I
```

```
Out[52]: matrix([[0.+2. j, 0.-1. j],
                 [0.-1.5j, 0.+0.5j]])
```

7.0.2 Statistics

```
In [53]: from numpy import *
```

```
In [74]: A = np.matrix(A)
         mean(A[:,3])
```

```
Out[74]: 4.0
```

```
In [75]: std(A[:,3]), var(A[:,3])
```

```
Out[75]: (0.0, 0.0)
```

```
In [76]: A[:,3].min(), A[:,3].max()
```

```
Out[76]: (4, 4)
```

```
In [77]: d = arange(1, 10)
         sum(d), prod(d)
```

```
Out[77]: (45, 362880)
```

```
In [78]: cumsum(d)
```

```
Out[78]: array([ 1,  3,  6, 10, 15, 21, 28, 36, 45])
```

```
In [79]: cumprod(d)
```

```
Out[79]: array([     1,      2,      6,     24,    120,    720,   5040,  40320,
               362880])
```

```
In [80]: # sum of diagonal
         trace(A)
```

```
Out[80]: 1
```

```
In [81]: m = random.rand(3, 3)
```

```
In [82]: # use axis parameter to specify how function behaves
         m.max(), m.max(axis=0)

Out[82]: (0.7260107697054486, array([0.72601077, 0.71723635, 0.34385472]))

In [83]: # reshape without copying underlying data
         n, m = A.shape
         B = A.reshape((1,n*m))

In [84]: # modify the array
         B[0,0:5] = 5

In [85]: # also changed
         A

Out[85]: matrix([[5, 5, 5, 5, 5]])

In [86]: # creates a copy
         B = A.flatten()

In [87]: # can insert a dimension in an array
         v = array([1,2,3])
         v[:, newaxis], v[:,newaxis].shape, v[newaxis,:].shape

Out[87]: (array([[1],
                 [2],
                 [3]]), (3, 1), (1, 3))

In [89]: repeat(v, 3)

Out[89]: array([1, 1, 1, 2, 2, 2, 3, 3, 3])

In [88]: tile(v, 3)

Out[88]: array([1, 2, 3, 1, 2, 3, 1, 2, 3])

In [90]: w = array([5, 6])

In [91]: concatenate((v, w), axis=0)

Out[91]: array([1, 2, 3, 5, 6])

In [92]: # deep copy
         B = copy(A)
```